

# Introduction to Web Beans

The new Java standard for dependency injection and contextual state management

**Gavin King**

Web Beans (JSR-299) specification lead

Red Hat Middleware LLC

---

# Table of Contents

I. Using contextual objects .....	1
<b>1. Getting started with Web Beans</b> .....	3
1.1. Your first Web Bean .....	3
1.2. What is a Web Bean? .....	4
1.2.1. API types, binding types and dependency injection .....	5
1.2.2. Deployment types .....	5
1.2.3. Scope .....	6
1.2.4. Web Bean names and Unified EL .....	6
1.2.5. Interceptor binding types .....	6
1.3. What kinds of objects can be Web Beans? .....	7
1.3.1. Simple Web Beans .....	7
1.3.2. Enterprise Web Beans .....	7
1.3.3. Producer methods .....	8
1.3.4. JMS endpoints .....	8
<b>2. JSF web application example</b> .....	9
<b>3. Dependency injection</b> .....	11
3.1. Binding annotations .....	12
3.1.1. Binding annotations with members .....	12
3.1.2. Combinations of binding annotations .....	13
3.1.3. Binding annotations and producer methods .....	13
3.1.4. The default binding type .....	13
3.2. Deployment types .....	13
3.2.1. Enabling deployment types .....	14
3.2.2. Deployment type precedence .....	14
3.2.3. Example deployment types .....	15
3.3. Fixing unsatisfied dependencies .....	15
3.4. Client proxies .....	15
3.5. Obtaining a Web Bean by programatic lookup .....	16
<b>4. Scopes and contexts</b> .....	17
4.1. Scope types .....	17
4.2. Built-in scopes .....	17
4.3. The conversation scope .....	18
4.3.1. Conversation demarcation .....	18
4.3.2. Conversation propagation .....	19
4.3.3. Conversation timeout .....	19
4.4. The dependent pseudo-scope .....	19
4.4.1. The @New annotation .....	19
<b>5. Producer methods</b> .....	21
5.1. Scope of a producer method .....	21
5.2. Injection into producer methods .....	22
5.3. Use of @New with producer methods .....	22
II. Developing loosely-coupled code .....	23
<b>6. Interceptors</b> .....	24
6.1. Interceptor bindings .....	24
6.2. Implementing interceptors .....	24
6.3. Enabling interceptors .....	24
6.4. Interceptor bindings with members .....	25
6.5. Multiple interceptor binding annotations .....	25
6.6. Interceptor binding type inheritance .....	26
<b>7. Decorators</b> .....	27
7.1. Delegate attributes .....	27
7.2. Enabling decorators .....	28
<b>8. Events</b> .....	29
8.1. Event observers .....	29
8.2. Event producers .....	29
8.3. Registering observers dynamically .....	30
8.4. Event bindings with members .....	30

8.5. Multiple event bindings .....	31
8.6. Transactional observers .....	31
III. Making the most of strong typing .....	33
<b>9. Stereotypes</b> .....	34
9.1. Default scope and deployment type for a stereotype .....	34
9.2. Restricting scope and type with a stereotype .....	34
9.3. Interceptor bindings for stereotypes .....	35
9.4. Name defaulting with stereotypes .....	35
9.5. Standard stereotypes .....	35
<b>10. Specialization</b> .....	36
10.1. Using specialization .....	36
10.2. Advantages of specialization .....	37
<b>11. Defining Web Beans using XML</b> .....	38
11.1. Declaring Web Bean classes .....	38
11.2. Declaring Web Bean metadata .....	38
11.3. Declaring Web Bean members .....	39
11.4. Declaring inline Web Beans .....	39
11.5. Using a schema .....	39
IV. Web Beans and the Java EE ecosystem .....	41
<b>12. Java EE integration</b> .....	42
12.1. Injecting Java EE resources into a Web Bean .....	42
12.2. Calling a Web Bean from a Servlet .....	42
12.3. Calling a Web Bean from a Message-Driven Bean .....	42
12.4. JMS endpoints .....	43
<b>13. Extending Web Beans</b> .....	44
13.1. The Manager object .....	44
13.2. The Bean class .....	45
13.3. The Context interface .....	45
<b>14. Next steps</b> .....	46

---

## Part I. Using contextual objects

The Web Beans (JSR-299) specification defines a set of services for the Java EE environment that makes applications much easier to develop. Web Beans layers an enhanced lifecycle and interaction model over existing Java component types including JavaBeans and Enterprise Java Beans. As a complement to the traditional Java EE programming model, the Web Beans services provide:

- an improved lifecycle for stateful components, bound to well-defined *contexts*,
- a typesafe approach to *dependency injection*,
- interaction via an *event notification* facility, and
- a better approach to binding *interceptors* to components, along with a new kind of interceptor, called a *decorator*, that is more appropriate for use in solving business problems.

Dependency injection, together with contextual lifecycle management, saves the user of an unfamiliar API from having to ask and answer the following questions:

- what is the lifecycle of this object?
- how many simultaneous clients can it have?
- is it multithreaded?
- where can I get one from?
- do I need to explicitly destroy it?
- where should I keep my reference to it when I'm not using it directly?
- how can I add an indirection layer, so that the implementation of this object can vary at deployment time?
- how should I go about sharing this object between other objects?

A Web Bean specifies only the type and semantics of other Web Beans it depends upon. It need not be aware of the actual lifecycle, concrete implementation, threading model or other clients of any Web Bean it depends upon. Even better, the concrete implementation, lifecycle and threading model of a Web Bean it depends upon may vary according to the deployment scenario, without affecting any client.

Events, interceptors and decorators enhance the *loose-coupling* that is inherent in this model:

- *event notifications* decouple event producers from event consumers,
- *interceptors* decouple technical concerns from business logic, and
- *decorators* allow business concerns to be compartmentalized.

Most importantly, Web Beans provides all these facilities in a *typesafe* way. Web Beans never uses string-based identifiers to determine how collaborating objects fit together. And XML, though it remains an option, is rarely used. Instead, Web Beans uses the typing information that is already available in the Java object model, together with a new pattern, called *binding annotations*, to wire together Web Beans, their dependencies, their interceptors and decorators and their event consumers.

The Web Beans services are general and apply to the following types of components that exist in the Java EE environment:

- all JavaBeans,
  - all EJBs, and
  - all Servlets.
-

---

Web Beans even provides the necessary integration points so that other kinds of components defined by future Java EE specifications or by non-standard frameworks may be cleanly integrated with Web Beans, take advantage of the Web Beans services, and interact with any other kind of Web Bean.

Web Beans was influenced by a number of existing Java frameworks, including Seam, Guice and Spring. However, Web Beans has its own very distinct character: more typesafe than Seam, more stateful and Java-centric than Spring, more web and enterprise-application capable than Guice.

Most importantly, Web Beans is a JCP standard that integrates cleanly with Java EE, and with any Java SE environment where embeddable EJB Lite is available.

---

---

## Chapter 1. Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

### 1.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session or singleton bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans—injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them—without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a pre-existing class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence in sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available—if the `SentenceTranslator` EJB was not deployed—the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

## 1.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecycle is*. Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs
- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in Section 3.2, "Deployment types".

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans—which are by nature not injectable, contextual objects—may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

A Web Bean comprises:

- A (nonempty) set of API types
- A (nonempty) set of binding annotation types
- A scope
- A deployment type
- Optionally, a Web Bean name
- A set of interceptor binding types
- A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

### 1.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in Chapter 3, *Dependency injection*.

### 1.2.2. Deployment types

*Deployment types* let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":



```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in Section 3.2, “Deployment types”.

### 1.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in Chapter 4, *Scopes and contexts*.

### 1.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in a JSF page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart`.

### 1.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in Chapter 6, *Interceptors* and Chapter 7, *Decorators*.

## 1.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

### 1.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- if it is an inner class, it is a static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

### 1.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans—since they are not intended to be injected into other objects—but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {
    ...
    @Remove
    public void destroy() {}
}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless/@Stateful/@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

### 1.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
public class Generator {  
    private Random random = new Random( System.currentTimeMillis() );  
    @Produces @Random int next() {  
        return random.nextInt(100);  
    }  
}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {  
    return createConnection( user.getId(), user.getPassword() );  
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {  
    connection.close();  
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in Chapter 5, *Producer methods*.

### 1.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the life-cycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in Section 12.4, "JMS endpoints".

---

## Chapter 2. JSF web application example

Let's illustrate these ideas by fleshing out the previous example. We're going to implement user login/logout. First, we'll define a Web Bean to hold the username and password entered during login:

```
@Named
public class Credentials {

    private String username;
    private String password;

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }

}
```

This Web Bean is bound to the login prompt in the following JSF form:

```
<f:form>
  <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
    <h:outputLabel for="username">Username:</h:outputLabel>
    <h:inputText id="username" value="#{credentials.username}" />
    <h:outputLabel for="password">Password:</h:outputLabel>
    <h:inputText id="password" value="#{credentials.password}" />
  </h:panelGrid>
  <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}" />
  <h:commandButton value="Logout" action="#{login.logout}" rendered="#{login.loggedIn}" />
</f:form>
```

The actual work is done by a session scoped Web Bean that maintains information about the currently logged-in user and exposes the `User` entity to other Web Beans:

```
@SessionScoped @Named
public class Login {

    @Current Credentials credentials;
    @PersistenceContext EntityManager userDatabase;

    private User user;

    public void login() {

        List<User> results = userDatabase.createQuery(
            "select u from User u where u.username=:username and u.password=:password")
            .setParameter("username", credentials.getUsername())
            .setParameter("password", credentials.getPassword())
            .getResultList();

        if ( !results.isEmpty() ) {
            user = results.get(0);
        }

    }

    public void logout() {
        user = null;
    }

    public boolean isLoggedIn() {
        return user!=null;
    }

    @Produces @LoggedIn User getCurrentUser() {
        return user;
    }

}
```

Of course, `@LoggedIn` is a binding annotation:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD})
@BindingType
public @interface LoggedIn {}
```

Now, any other Web Bean can easily inject the current user:

```
public class DocumentEditor {  
  
    @Current Document document;  
    @LoggedIn User currentUser;  
    @PersistenceContext EntityManager docDatabase;  
  
    public void save() {  
        document.setCreatedBy(currentUser);  
        docDatabase.persist(document);  
    }  
}
```

Hopefully, this example gives a flavor of the Web Bean programming model. In the next chapter, we'll explore Web Beans dependency injection in greater depth.

---

## Chapter 3. Dependency injection

Web Beans supports three primary mechanisms for dependency injection:

Constructor parameter injection:

```
public class Checkout {  
    private final ShoppingCart cart;  
  
    @Initializer  
    public Checkout(ShoppingCart cart) {  
        this.cart = cart;  
    }  
}
```

Initializer method parameter injection:

```
public class Checkout {  
    private ShoppingCart cart;  
  
    @Initializer  
    void setShoppingCart(ShoppingCart cart) {  
        this.cart = cart;  
    }  
}
```

And direct field injection:

```
public class Checkout {  
    private @Current ShoppingCart cart;  
}
```

Dependency injection always occurs when the Web Bean instance is first instantiated.

- First, the Web Bean manager calls the Web Bean constructor, to obtain an instance of the Web Bean.
- Next, the Web Bean manager initializes the values of all injected fields of the Web Bean.
- Next, the Web Bean manager calls all initializer methods of Web Bean.
- Finally, the `@PostConstruct` method of the Web Bean, if any, is called.

Constructor parameter injection is not supported for EJB beans, since the EJB is instantiated by the EJB container, not the Web Bean manager.

Parameters of constructors and initializer methods need not be explicitly annotated when the default binding type `@Current` applies. Injected fields, however, *must* specify a binding type, even when the default binding type applies. If the field does not specify a binding type, it will not be injected.

Producer methods also support parameter injection:

```
@Produces  
Checkout createCheckout(ShoppingCart cart) {  
    return new Checkout(cart);  
}
```

Finally, observer methods (which we'll meet in Chapter 8, *Events*), disposal methods and destructor methods all support parameter injection.

The Web Beans specification defines a procedure, called the *typesafe resolution algorithm*, that the Web Bean manager follows when identifying the Web Bean to inject to an injection point. This algorithm looks complex at first, but once you understand it, it's really quite intuitive. Typesafe resolution is performed at system initialization time, which means that the manager will inform the user immediately if a Web Bean's dependencies cannot be satisfied, by throwing a `UnsatisfiedDependencyException` or `AmbiguousDependencyException`.

The purpose of this algorithm is to allow multiple Web Beans to implement the same API type and either:

- allow the client to select which implementation it requires using *binding annotations*,
- allow the application deployer to select which implementation is appropriate for a particular deployment, without changes to the client, by enabling or disabling *deployment types*, or
- allow one implementation of an API to override another implementation of the same API at deployment time, without changes to the client, using *deployment type precedence*.

Let's explore how the Web Beans manager determines a Web Bean to be injected.

### 3.1. Binding annotations

If we have more than one Web Bean that implements a particular API type, the injection point can specify exactly which Web Bean should be injected using a binding annotation. For example, there might be two implementations of `PaymentProcessor`:

```
@PayByCheque
public class ChequePaymentProcessor implements PaymentProcessor {
    public void process(Payment payment) { ... }
}
```

```
@PayByCreditCard
public class CreditCardPaymentProcessor implements PaymentProcessor {
    public void process(Payment payment) { ... }
}
```

Where `@PayByCheque` and `@PayByCreditCard` are binding annotations:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayByCheque {}
```

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayByCreditCard {}
```

A client Web Bean developer uses the binding annotation to specify exactly which Web Bean should be injected.

Using field injection:

```
@PayByCheque PaymentProcessor chequePaymentProcessor;
@PayByCreditCard PaymentProcessor creditCardPaymentProcessor;
```

Using initializer method injection:

```
@Initializer
public void setPaymentProcessors(@PayByCheque PaymentProcessor chequePaymentProcessor,
                                @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
    this.chequePaymentProcessor = chequePaymentProcessor;
    this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

Or using constructor injection:

```
@Initializer
public Checkout(@PayByCheque PaymentProcessor chequePaymentProcessor,
               @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
    this.chequePaymentProcessor = chequePaymentProcessor;
    this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

#### 3.1.1. Binding annotations with members

Binding annotations may have members:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayBy {
    PaymentType value();
}
```

In which case, the member value is significant:

```
@PayBy(CHEQUE) PaymentProcessor chequePaymentProcessor;
@PayBy(CREDIT_CARD) PaymentProcessor creditCardPaymentProcessor;
```

You can tell the Web Bean manager to ignore a member of a binding annotation type by annotating the member `@NonBinding`.

### 3.1.2. Combinations of binding annotations

An injection point may even specify multiple binding annotations:

```
@Asynchronous @PayByCheque PaymentProcessor paymentProcessor
```

In this case, only a Web Bean which has *both* binding annotations would be eligible for injection.

### 3.1.3. Binding annotations and producer methods

Even producer methods may specify binding annotations:

```
@Produces
@Asynchronous @PayByCheque
PaymentProcessor createAsyncPaymentProcessor(@PayByCheque PaymentProcessor processor) {
    return new AsynchronousPaymentProcessor(processor);
}
```

### 3.1.4. The default binding type

Web Beans defines a binding type `@Current` that is the default binding type for any injection point or Web Bean that does not explicitly specify a binding type.

There are two common circumstances in which it is necessary to explicitly specify `@Current`:

- on a field, in order to declare it as an injected field with the default binding type, and
- on a Web Bean which has another binding type in addition to the default binding type.

## 3.2. Deployment types

All Web Beans have a *deployment type*. Each deployment type identifies a set of Web Beans that should be conditionally installed in some deployments of the system.

For example, we could define a deployment type named `@Mock`, which would identify Web Beans that should only be installed when the system executes inside an integration testing environment:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@DeploymentType
public @interface Mock {}
```

Suppose we had some Web Bean that interacted with an external system to process payments:

```
public class ExternalPaymentProcessor {

    public void process(Payment p) {
        ...
    }
}
```



```
}
}
```

Since this Web Bean does not explicitly specify a deployment type, it has the default deployment type `@Production`.

For integration or unit testing, the external system is slow or unavailable. So we would create a mock object:

```
@Mock
public class MockPaymentProcessor implements PaymentProcessor {

    @Override
    public void process(Payment p) {
        p.setSuccessful(true);
    }
}
```

But how does the Web Bean manager determine which implementation to use in a particular deployment?

### 3.2.1. Enabling deployment types

Web Beans defines two built-in deployment types: `@Production` and `@Standard`. By default, only Web Beans with the built-in deployment types are enabled when the system is deployed. We can identify additional deployment types to be enabled in a particular deployment by listing them in `web-beans.xml`.

Going back to our example, when we deploy our integration tests, we want all our `@Mock` objects to be installed:

```
<WebBeans>
  <Deploy>
    <Standard/>
    <Production/>
    <test:Mock/>
  </Deploy>
</WebBeans>
```

Now the Web Bean manager will identify and install all Web Beans annotated `@Production`, `@Standard` or `@Mock` at deployment time.

The deployment type `@Standard` is used only for certain special Web Beans defined by the Web Beans specification. We can't use it for our own Web Beans, and we can't disable it.

The deployment type `@Production` is the default deployment type for Web Beans which don't explicitly declare a deployment type, and may be disabled.

### 3.2.2. Deployment type precedence

If you've been paying attention, you're probably wondering how the Web Bean manager decides which implementation—`ExternalPaymentProcessor` or `MockPaymentProcessor`—to choose. Consider what happens when the manager encounters this injection point:

```
@Current PaymentProcessor paymentProcessor
```

There are now two Web Beans which satisfy the `PaymentProcessor` contract. Of course, we can't use a binding annotation to disambiguate, since binding annotations are hard-coded into the source at the injection point, and we want the manager to be able to decide at deployment time!

The solution to this problem is that each deployment type has a different *precedence*. The precedence of the deployment types is determined by the order in which they appear in `web-beans.xml`. In our example, `@Mock` appears later than `@Production` so it has a higher precedence.

Whenever the manager discovers that more than one Web Bean could satisfy the contract (API type plus binding annotations) specified by an injection point, it considers the relative precedence of the Web Beans. If one has a higher precedence than the others, it chooses the higher precedence Web Bean to inject. So, in our example, the Web Bean manager will inject `MockPaymentProcessor` when executing in our integration testing environment (which is exactly what we want).

It's interesting to compare this facility to today's popular manager architectures. Various "lightweight" containers also allow conditional deployment of classes that exist in the classpath, but the classes that are to be deployed must be explicitly, individually, listed in configuration code or in some XML configuration file. Web Beans does support Web Bean definition and configuration via XML, but in the common case where no complex configuration is required, deployment types allow a whole set of Web Beans to be enabled with a single line of XML. Meanwhile, a developer browsing the code can easily identify what deployment scenarios the Web Bean will be used in.

### 3.2.3. Example deployment types

Deployment types are useful for all kinds of things, here's some examples:

- `@Mock` and `@Staging` deployment types for testing
- `@AustralianTaxLaw` for site-specific Web Beans
- `@SeamFramework`, `@Guice` for third-party frameworks which build on Web Beans
- `@Standard` for standard Web Beans defined by the Web Beans specification

I'm sure you can think of more applications...

## 3.3. Fixing unsatisfied dependencies

The typesafe resolution algorithm fails when, after considering the binding annotations and deployment types of all Web Beans that implement the API type of an injection point, the Web Bean manager is unable to identify exactly one Web Bean to inject.

It's usually easy to fix an `UnsatisfiedDependencyException` or `AmbiguousDependencyException`.

To fix an `UnsatisfiedDependencyException`, simply provide a Web Bean which implements the API type and has the binding types of the injection point—or enable the deployment type of a Web Bean that already implements the API type and has the binding types.

To fix an `AmbiguousDependencyException`, introduce a binding type to distinguish between the two implementations of the API type, or change the deployment type of one of the implementations so that the Web Bean manager can use deployment type precedence to choose between them. An `AmbiguousDependencyException` can only occur if two Web Beans share a binding type and have exactly the same deployment type.

There's one more issue you need to be aware of when using dependency injection in Web Beans.

## 3.4. Client proxies

Clients of an injected Web Bean do not usually hold a direct reference to a Web Bean instance.

Imagine that a Web Bean bound to the application scope held a direct reference to a Web Bean bound to the request scope. The application scoped Web Bean is shared between many different requests. However, each request should see a different instance of the request scoped Web bean!

Now imagine that a Web Bean bound to the session scope held a direct reference to a Web Bean bound to the application scope. From time to time, the session context is serialized to disk in order to use memory more efficiently. However, the application scoped Web Bean instance should not be serialized along with the session scoped Web Bean!

Therefore, unless a Web Bean has the default scope `@Dependent`, the Web Bean manager must indirect all injected references to the Web Bean through a proxy object. This *client proxy* is responsible for ensuring that the Web Bean instance that receives a method invocation is the instance that is associated with the current context. The client proxy also allows Web Beans bound to contexts such as the session context to be serialized to disk without recursively serializing other injected Web Beans.

Unfortunately, due to limitations of the Java language, some Java types cannot be proxied by the Web Bean manager. Therefore, the Web Bean manager throws an `UnproxyableDependencyException` if the type of an injection point cannot be proxied.

The following Java types cannot be proxied by the Web Bean manager:

- classes which are declared `final` or have a `final` method,
- classes which have no non-private constructor with no parameters, and
- arrays and primitive types.

It's usually very easy to fix an `UnproxyableDependencyException`. Simply add a constructor with no parameters to the injected class, introduce an interface, or change the scope of the injected Web Bean to `@Dependent`.

### 3.5. Obtaining a Web Bean by programatic lookup

The application may obtain an instance of the interface `Manager` by injection:

```
@Current Manager manager;
```

The `Manager` object provides a set of methods for obtaining a Web Bean instance programatically.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class);
```

Binding annotations may be specified by subclassing the helper class `AnnotationLiteral`, since it is otherwise difficult to instantiate an annotation type in Java.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,  
                                                new AnnotationLiteral<CreditCard>(){});
```

If the binding type has an annotation member, we can't use an anonymous subclass of `AnnotationLiteral`—instead we'll need to create a named subclass:

```
abstract class CreditCardBinding  
    extends AnnotationLiteral<CreditCard>  
    implements CreditCard {}
```

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,  
                                                new CreditCardBinding() {  
            public void value() { return paymentType; }  
        } );
```

---

## Chapter 4. Scopes and contexts

So far, we've seen a few examples of *scope type annotations*. The scope of a Web Bean determines the lifecycle of instances of the Web Bean. The scope also determines which clients refer to which instances of the Web Bean. According to the Web Beans specification, a scope determines:

- When a new instance of any Web Bean with that scope is created
- When an existing instance of any Web Bean with that scope is destroyed
- Which injected references refer to any instance of a Web Bean with that scope

For example, if we have a session scoped Web Bean, `CurrentUser`, all Web Beans that are called in the context of the same `HttpSession` will see the same instance of `CurrentUser`. This instance will be automatically created the first time a `CurrentUser` is needed in that session, and automatically destroyed when the session ends.

### 4.1. Scope types

Web Beans features an *extensible context model*. It is possible to define new scopes by creating a new scope type annotation:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@ScopeType
public @interface ClusterScoped {}
```

Of course, that's the easy part of the job. For this scope type to be useful, we will also need to define a `Context` object that implements the scope! Implementing a `Context` is usually a very technical task, intended for framework development only.

We can apply a scope type annotation to a Web Bean implementation class to specify the scope of the Web Bean:

```
@ClusterScoped
public class SecondLevelCache { ... }
```

Usually, you'll use one of Web Beans' built-in scopes.

### 4.2. Built-in scopes

Web Beans defines four built-in scopes:

- `@RequestScoped`
- `@SessionScoped`
- `@ApplicationScoped`
- `@ConversationScoped`

For a web application that uses Web Beans:

- any servlet request has access to active request, session and application scopes, and, additionally
- any JSF request has access to an active conversation scope.

The request and application scopes are also active:

- during invocations of EJB remote methods,
- during EJB timeouts,

- during message delivery to a message-driven bean, and
- during web service invocations.

If the application tries to invoke a Web Bean with a scope that does not have an active context, a `ContextNotActiveException` is thrown by the Web Bean manager at runtime.

Three of the four built-in scopes should be extremely familiar to every Java EE developer, so let's not waste time discussing them here. One of the scopes, however, is new.

### 4.3. The conversation scope

The Web Beans conversation scope is a bit like the traditional session scope in that it holds state associated with a user of the system, and spans multiple requests to the server. However, unlike the session scope, the conversation scope:

- is demarcated explicitly by the application, and
- holds state associated with a particular web browser tab in a JSF application.

A conversation represents a task, a unit of work from the point of view of the user. The conversation context holds state associated with what the user is currently working on. If the user is doing multiple things at the same time, there are multiple conversations.

The conversation context is active during any JSF request. However, most conversations are destroyed at the end of the request. If a conversation should hold state across multiple requests, it must be explicitly promoted to a *long-running conversation*.

#### 4.3.1. Conversation demarcation

Web Beans provides a built-in Web Bean for controlling the lifecycle of conversations in a JSF application. This Web Bean may be obtained by injection:

```
@Current Conversation conversation;
```

To promote the conversation associated with the current request to a long-running conversation, call the `begin()` method from application code. To schedule the current long-running conversation context for destruction at the end of the current request, call `end()`.

In the following example, a conversation-scoped Web Bean controls the conversation with which it is associated:

```
@ConversationScoped @Stateful
public class OrderBuilder {

    private Order order;
    private @Current Conversation conversation;
    private @PersistenceContext(type=EXTENDED) EntityManager em;

    @Produces public Order getOrder() {
        return order;
    }

    public Order createOrder() {
        order = new Order();
        conversation.begin();
        return order;
    }

    public void addLineItem(Product product, int quantity) {
        order.add( new LineItem(product, quantity) );
    }

    public void saveOrder(Order order) {
        em.persist(order);
        conversation.end();
    }

    @Remove
    public void destroy() {}
}
```

This Web Bean is able to control its own lifecycle through use of the `Conversation` API. But some other Web Beans have a lifecycle which depends completely upon another object.

### 4.3.2. Conversation propagation

The conversation context automatically propagates with any JSF faces request (JSF form submission). It does not automatically propagate with non-faces requests, for example, navigation via a link.

We can force the conversation to propagate with a non-faces request by including the unique identifier of the conversation as a request parameter. The Web Beans specification reserves the request parameter named `cid` for this use. The unique identifier of the conversation may be obtained from the `Conversation` object, which has the Web Beans name `conversation`.

Therefore, the following link propagates the conversation:

```
<a href="/addProduct.jsp?cid=#{conversation.id}">Add Product</a>
```

The Web Bean manager is also required to propagate conversations across any redirect, even if the conversation is not marked long-running. This makes it very easy to implement the common POST-then-redirect pattern, without resort to fragile constructs such as a "flash" object. In this case, the Web Bean manager automatically adds a request parameter to the redirect URL.

### 4.3.3. Conversation timeout

The Web Bean manager is permitted to destroy a conversation and all state held in its context at any time in order to preserve resources. A Web Bean manager implementation will normally do this on the basis of some kind of timeout—though this is not required by the Web Beans specification. The timeout is the period of inactivity before the conversation is destroyed.

The `Conversation` object provides a method to set the timeout. This is a hint to the Web Bean manager, which is free to ignore the setting.

```
conversation.setTimeout(timeoutInMillis);
```

## 4.4. The dependent pseudo-scope

In addition to the four built-in scopes, Web Beans features the so-called *dependent pseudo-scope*. This is the default scope for a Web Bean which does not explicitly declare a scope type.

For example, this Web Bean has the scope type `@Dependent`:

```
public class Calculator { ... }
```

When an injection point of a Web Bean resolves to a dependent Web Bean, a new instance of the dependent Web Bean is created every time the first Web Bean is instantiated. Instances of dependent Web Beans are never shared between different Web Beans or different injection points. They are *dependent objects* of some other Web Bean instance.

Dependent Web Bean instances are destroyed when the instance they depend upon is destroyed.

Web Beans makes it easy to obtain a dependent instance of a Java class or EJB bean, even if the class or EJB bean is already declared as a Web Bean with some other scope type.

### 4.4.1. The `@New` annotation

The built-in `@New` binding annotation allows *implicit* definition of a dependent Web Bean at an injection point. Suppose we declare the following injected field:

```
@New Calculator calculator;
```

Then a Web Bean with scope `@Dependent`, binding type `@New`, API type `Calculator`, implementation class `Calculator`

and deployment type `@Standard` is implicitly defined.

This is true even if `Calculator` is *already* declared with a different scope type, for example:

```
@ConversationScoped
public class Calculator { ... }
```

So the following injected attributes each get a different instance of `Calculator`:

```
public class PaymentCalc {

    @Current Calculator calculator;
    @New Calculator newCalculator;

}
```

The `calculator` field has a conversation-scoped instance of `Calculator` injected. The `newCalculator` field has a new instance of `Calculator` injected, with a lifecycle that is bound to the owning `PaymentCalc`.

This feature is particularly useful with producer methods, as we'll see in the next chapter.

---

## Chapter 5. Producer methods

Producer methods let us overcome certain limitations that arise when the Web Bean manager, instead of the application, is responsible for instantiating objects. They're also the easiest way to integrate objects which are not Web Beans into the Web Beans environment. (We'll meet a second approach in Chapter 11, *Defining Web Beans using XML*.)

According to the spec:

A Web Beans producer method acts as a source of objects to be injected, where:

- the objects to be injected are not required to be instances of Web Beans,
- the concrete type of the objects to be injected may vary at runtime or
- the objects require some custom initialization that is not performed by the Web Bean constructor

For example, producer methods let us:

- expose a JPA entity as a Web Bean,
- expose any JDK class as a Web Bean,
- define multiple Web Beans, with different scopes or initialization, for the same implementation class, or
- vary the implementation of an API type at runtime.

In particular, producer methods let us use runtime polymorphism with Web Beans. As we've seen, deployment types are a powerful solution to the problem of deployment-time polymorphism. But once the system is deployed, the Web Bean implementation is fixed. A producer method has no such limitation:

```
@SessionScoped
public class Preferences {

    private PaymentStrategyType paymentStrategy;

    ...

    @Produces @Preferred
    public PaymentStrategy getPaymentStrategy() {
        switch (paymentStrategy) {
            case CREDIT_CARD: return new CreditCardPaymentStrategy();
            case CHEQUE: return new ChequePaymentStrategy();
            case PAYPAL: return new PayPalPaymentStrategy();
            default: return null;
        }
    }
}
```

Consider an injection point:

```
@Preferred PaymentStrategy paymentStrat;
```

This injection point has the same type and binding annotations as the producer method, so it resolves to the producer method using the usual Web Beans injection rules. The producer method will be called by the Web Bean manager to obtain an instance to service this injection point.

### 5.1. Scope of a producer method

The scope of the producer method defaults to `@Dependent`, and so it will be called *every time* the Web Bean manager injects this field or any other field that resolves to the same producer method. Thus, there could be multiple instances of the `PaymentStrategy` object for each user session.

To change this behavior, we can add a `@SessionScoped` annotation to the method.

```
@Produces @Preferred @SessionScoped
```



```
public PaymentStrategy getPaymentStrategy() {
    ...
}
```

Now, when the producer method is called, the returned `PaymentStrategy` will be bound to the session context. The producer method won't be called again in the same session.

## 5.2. Injection into producer methods

There's one potential problem with the code above. The implementations of `CreditCardPaymentStrategy` are instantiated using the Java `new` operator. Objects instantiated directly by the application can't take advantage of dependency injection and don't have interceptors.

If this isn't what we want we can use dependency injection into the producer method to obtain Web Bean instances:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                          ChequePaymentStrategy cps,
                                          PayPalPaymentStrategy ppps) {

    switch (paymentStrategy) {
        case CREDIT_CARD: return ccps;
        case CHEQUE: return cps;
        case PAYPAL: return ppps;
        default: return null;
    }
}
```

Wait, what if `CreditCardPaymentStrategy` is a request scoped Web Bean? Then the producer method has the effect of "promoting" the current request scoped instance into session scope. This is almost certainly a bug! The request scoped object will be destroyed by the Web Bean manager before the session ends, but the reference to the object will be left "hanging" in the session scope. This error will *not* be detected by the Web Bean manager, so please take extra care when returning Web Bean instances from producer methods!

There's at least three ways we could go about fixing this bug. We could change the scope of the `CreditCardPaymentStrategy` implementation, but this would affect other clients of that Web Bean. A better option would be to change the scope of the producer method to `@Dependent` or `@RequestScoped`.

But a more common solution is to use the special `@New` binding annotation.

## 5.3. Use of `@New` with producer methods

Consider the following producer method:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(@New CreditCardPaymentStrategy ccps,
                                          @New ChequePaymentStrategy cps,
                                          @New PayPalPaymentStrategy ppps) {

    switch (paymentStrategy) {
        case CREDIT_CARD: return ccps;
        case CHEQUE: return cps;
        case PAYPAL: return ppps;
        default: return null;
    }
}
```

Then a new *dependent* instance of `CreditCardPaymentStrategy` will be created, passed to the producer method, returned by the producer method and finally bound to the session context. The dependent object won't be destroyed until the `Preferences` object is destroyed, at the end of the session.

---

## Part II. Developing loosely-coupled code

The first major theme of Web Beans is *loose coupling*. We've already seen three means of achieving loose coupling:

- *deployment types* enable deployment time polymorphism,
- *producer methods* enable runtime polymorphism, and
- *contextual lifecycle management* decouples Web Bean lifecycles.

These techniques serve to enable loose coupling of client and server. The client is no longer tightly bound to an implementation of an API, nor is it required to manage the lifecycle of the server object. This approach lets *stateful objects interact as if they were services*.

Loose coupling makes a system more *dynamic*. The system can respond to change in a well-defined manner. In the past, frameworks that attempted to provide the facilities listed above invariably did it by sacrificing type safety. Web Beans is the first technology that achieves this level of loose coupling in a typesafe way.

Web Beans provides three extra important facilities that further the goal of loose coupling:

- *interceptors* decouple technical concerns from business logic,
- *decorators* may be used to decouple some business concerns, and
- *event notifications* decouple event producers from event consumers.

Let's explore interceptors first.

---

---

## Chapter 6. Interceptors

Web Beans re-uses the basic interceptor architecture of EJB 3.0, extending the functionality in two directions:

- Any Web Bean may have interceptors, not just session beans.
- Web Beans features a more sophisticated annotation-based approach to binding interceptors to Web Beans.

### 6.1. Interceptor bindings

Suppose we want to declare that some of our Web Beans are transactional. The first thing we need is an *interceptor binding annotation* to specify exactly which Web Beans we're interested in:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {}
```

Now we can easily specify that our `ShoppingCart` is a transactional object:

```
@Transactional
public class ShoppingCart { ... }
```

Or, if we prefer, we can specify that just one method is transactional:

```
public class ShoppingCart {
    @Transactional public void checkout() { ... }
}
```

### 6.2. Implementing interceptors

That's great, but somewhere along the line we're going to have to actually implement the interceptor that provides this transaction management aspect. All we need to do is create a standard EJB interceptor, and annotate it `@Interceptor` and `@Transactional`.

```
@Transactional @Interceptor
public class TransactionInterceptor {
    @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

All Web Beans interceptors are simple Web Beans, and can take advantage of dependency injection and contextual lifecycle management.

```
@ApplicationScoped @Transactional @Interceptor
public class TransactionInterceptor {

    @Resource Transaction transaction;

    @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Multiple interceptors may use the same interceptor binding type.

### 6.3. Enabling interceptors

Finally, we need to *enable* our interceptor in `web-beans.xml`.

```
<Interceptors>
    <tx:TransactionInterceptor/>
</Interceptors>
```

Whoah! Why the angle bracket stew?

Well, the XML declaration solves two problems:

- it enables us to specify a total ordering for all the interceptors in our system, ensuring deterministic behavior, and
- it lets us enable or disable interceptor classes at deployment time.

For example, we could specify that our security interceptor runs before our `TransactionInterceptor`.

```
<Interceptors>
  <sx:SecurityInterceptor/>
  <tx:TransactionInterceptor/>
</Interceptors>
```

Or we could turn them both off in our test environment!

## 6.4. Interceptor bindings with members

Suppose we want to add some extra information to our `@Transactional` annotation:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {
    boolean requiresNew() default false;
}
```

Web Beans will use the value of `requiresNew` to choose between two different interceptors, `TransactionInterceptor` and `RequiresNewTransactionInterceptor`.

```
@Transactional(requiresNew=true) @Interceptor
public class RequiresNewTransactionInterceptor {
    @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Now we can use `RequiresNewTransactionInterceptor` like this:

```
@Transactional(requiresNew=true)
public class ShoppingCart { ... }
```

But what if we only have one interceptor and we want the manager to ignore the value of `requiresNew` when binding interceptors? We can use the `@NonBinding` annotation:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Secure {
    @NonBinding String[] rolesAllowed() default {};
}
```

## 6.5. Multiple interceptor binding annotations

Usually we use combinations of interceptor bindings types to bind multiple interceptors to a Web Bean. For example, the following declaration would be used to bind `TransactionInterceptor` and `SecurityInterceptor` to the same Web Bean:

```
@Secure(rolesAllowed="admin") @Transactional
public class ShoppingCart { ... }
```

However, in very complex cases, an interceptor itself may specify some combination of interceptor binding types:

```
@Transactional @Secure @Interceptor
public class TransactionalSecureInterceptor { ... }
```

Then this interceptor could be bound to the `checkout()` method using any one of the following combinations:

```
public class ShoppingCart {
    @Transactional @Secure public void checkout() { ... }
}
```

```
}
```

```
@Secure
public class ShoppingCart {
    @Transactional public void checkout() { ... }
}
```

```
@Transactional
public class ShoppingCart {
    @Secure public void checkout() { ... }
}
```

```
@Transactional @Secure
public class ShoppingCart {
    public void checkout() { ... }
}
```

## 6.6. Interceptor binding type inheritance

One limitation of the Java language support for annotations is the lack of annotation inheritance. Really, annotations should have reuse built in, to allow this kind of thing to work:

```
public @interface Action extends Transactional, Secure { ... }
```

Well, fortunately, Web Beans works around this missing feature of Java. We may annotate one interceptor binding type with other interceptor binding types. The interceptor bindings are transitive—any Web Bean with the first interceptor binding inherits the interceptor bindings declared as meta-annotations.

```
@Transactional @Secure
@InterceptorBindingType
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action { ... }
```

Any Web Bean annotated `@Action` will be bound to both `TransactionInterceptor` and `SecurityInterceptor`. (And even `TransactionalSecureInterceptor`, if it exists.)

---

## Chapter 7. Decorators

Interceptors are a powerful way to capture and separate concerns which are *orthogonal* to the type system. Any interceptor is able to intercept invocations of any Java type. This makes them perfect for solving technical concerns such as transaction management and security. However, by nature, interceptors are unaware of the actual semantics of the events they intercept. Thus, interceptors aren't an appropriate tool for separating business-related concerns.

The reverse is true of *decorators*. A decorator intercepts invocations only for a certain Java interface, and is therefore aware of all the semantics attached to that interface. This makes decorators a perfect tool for modeling some kinds of business concerns. It also means that a decorator doesn't have the generality of an interceptor. Decorators aren't able to solve technical concerns that cut across many disparate types.

Suppose we have an interface that represents accounts:

```
public interface Account {
    public BigDecimal getBalance();
    public User getOwner();
    public void withdraw(BigDecimal amount);
    public void deposit(BigDecimal amount);
}
```

Several different Web Beans in our system implement the `Account` interface. However, we have a common legal requirement that, for any kind of account, large transactions must be recorded by the system in a special log. This is a perfect job for a decorator.

A decorator is a simple Web Bean that implements the type it decorates and is annotated `@Decorator`.

```
@Decorator
public abstract class LargeTransactionDecorator
    implements Account {

    @Decorates Account account;

    @PersistenceContext EntityManager em;

    public void withdraw(BigDecimal amount) {
        account.withdraw(amount);
        if ( amount.compareTo(LARGE_AMOUNT)>0 ) {
            em.persist( new LoggedWithdrawal(amount) );
        }
    }

    public void deposit(BigDecimal amount);
        account.deposit(amount);
        if ( amount.compareTo(LARGE_AMOUNT)>0 ) {
            em.persist( new LoggedDeposit(amount) );
        }
    }
}
```

Unlike other simple Web Beans, a decorator may be an abstract class. If there's nothing special the decorator needs to do for a particular method of the decorated interface, you don't need to implement that method.

### 7.1. Delegate attributes

All decorators have a *delegate attribute*. The type and binding types of the delegate attribute determine which Web Beans the decorator is bound to. The delegate attribute type must implement or extend all interfaces implemented by the decorator.

This delegate attribute specifies that the decorator is bound to all Web Beans that implement `Account`:

```
@Decorates Account account;
```

A delegate attribute may specify a binding annotation. Then the decorator will only be bound to Web Beans with the same binding.

```
@Decorates @Foreign Account account;
```

A decorator is bound to any Web Bean which:

- has the type of the delegate attribute as an API type, and
- has all binding types that are declared by the delegate attribute.

The decorator may invoke the delegate attribute, which has much the same effect as calling `InvocationContext.proceed()` from an interceptor.

## 7.2. Enabling decorators

We need to *enable* our decorator in `web-beans.xml`.

```
<Decorators>
  <myapp:LargeTransactionDecorator/>
</Decorators>
```

This declaration serves the same purpose for decorators that the `<Interceptors>` declaration serves for interceptors:

- it enables us to specify a total ordering for all decorators in our system, ensuring deterministic behavior, and
- it lets us enable or disable decorator classes at deployment time.

Interceptors for a method are called before decorators that apply to that method.

---

## Chapter 8. Events

The Web Beans event notification facility allows Web Beans to interact in a totally decoupled manner. Event *producers* raise events that are then delivered to event *observers* by the Web Bean manager. This basic schema might sound like the familiar observer/observable pattern, but there are a couple of twists:

- not only are event producers decoupled from observers; observers are completely decoupled from producers,
- observers can specify a combination of "selectors" to narrow the set of event notifications they will receive, and
- observers can be notified immediately, or can specify that delivery of the event should be delayed until the end of the current transaction

### 8.1. Event observers

An *observer method* is a method of a Web Bean with a parameter annotated `@Observes`.

```
public void onAnyDocumentEvent(@Observes Document document) { ... }
```

The annotated parameter is called the *event parameter*. The type of the event parameter is the observed *event type*. Observer methods may also specify "selectors", which are just instances of Web Beans binding types. When a binding type is used as an event selector, it is called an *event binding type*.

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Updated { ... }
```

We specify the event bindings of the observer method by annotating the event parameter:

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

An observer method need not specify any event bindings—in this case it is interested in *all* events of a particular type. If it does specify event bindings, it is only interested in events which also have those event bindings.

The observer method may have additional parameters, which are injected according to the usual Web Beans method parameter injection semantics:

```
public void afterDocumentUpdate(@Observes @Updated Document document, User user) { ... }
```

### 8.2. Event producers

The event producer may obtain an *event notifier* object by injection:

```
@Observable Event<Document> documentEvent
```

The `@Observable` annotation implicitly defines a Web Bean with scope `@Dependent` and deployment type `@Standard`, with an implementation provided by the Web Bean manager.

A producer raises events by calling the `fire()` method of the `Event` interface, passing an *event object*:

```
documentEvent.fire(document);
```

An event object may be an instance of any Java class that has no type variables or wildcard type parameters. The event will be delivered to every observer method that:

- has an event parameter to which the event object is assignable, and
- specifies no event bindings.

The Web Bean manager simply calls all the observer methods, passing the event object as the value of the event paramet-



er. If any observer method throws an exception, the Web Bean manager stops calling observer methods, and the exception is rethrown by the `fire()` method.

To specify a "selector", the event producer may pass an instance of the event binding type to the `fire()` method:

```
documentEvent.fire( document, new AnnotationLiteral<Updated>(){} );
```

The helper class `AnnotationLiteral` makes it possible to instantiate binding types inline, since this is otherwise difficult to do in Java.

The event will be delivered to every observer method that:

- has an event parameter to which the event object is assignable, and
- does not specify any event binding *except* for the event bindings passed to `fire()`.

Alternatively, event bindings may be specified by annotating the event notifier injection point:

```
@Observable @Updated Event<Document> documentUpdatedEvent
```

Then every event fired via this instance of `Event` has the annotated event binding. The event will be delivered to every observer method that:

- has an event parameter to which the event object is assignable, and
- does not specify any event binding *except* for the event bindings passed to `fire()` or the annotated event bindings of the event notifier injection point.

### 8.3. Registering observers dynamically

It's often useful to register an event observer dynamically. The application may implement the `Observer` interface and register an instance with an event notifier by calling the `observe()` method.

```
documentEvent.observe( new Observer<Document>() { public void notify(Document doc) { ... } } );
```

Event binding types may be specified by the event notifier injection point or by passing event binding type instances to the `observe()` method:

```
documentEvent.observe( new Observer<Document>() { public void notify(Document doc) { ... } },
                      new AnnotationLiteral<Updated>(){} );
```

### 8.4. Event bindings with members

An event binding type may have annotation members:

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Role {
    RoleType value();
}
```

The member value is used to narrow the messages delivered to the observer:

```
public void adminLoggedIn(@Observes @Role(ADMIN) LoggedIn event) { ... }
```

Event binding type members may be specified statically by the event producer, via annotations at the event notifier injection point:

```
@Observable @Role(ADMIN) Event<LoggedIn> LoggedInEvent;}}
```

Alternatively, the value of the event binding type member may be determined dynamically by the event producer. We start

by writing an abstract subclass of `AnnotationLiteral`:

```
abstract class RoleBinding
    extends AnnotationLiteral<Role>
    implements Role {}
```

The event producer passes an instance of this class to `fire()`:

```
documentEvent.fire( document, new RoleBinding() { public void value() { return user.getRole(); } } );
```

## 8.5. Multiple event bindings

Event binding types may be combined, for example:

```
@Observable @Blog Event<Document> blogEvent;
...
if (document.isBlog()) blogEvent.fire(document, new AnnotationLiteral<Updated>({}));
```

When this event occurs, all of the following observer methods will be notified:

```
public void afterBlogUpdate(@Observes @Updated @Blog Document document) { ... }
```

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

```
public void onAnyBlogEvent(@Observes @Blog Document document) { ... }
```

```
public void onAnyDocumentEvent(@Observes Document document) { ... }}}
```

## 8.6. Transactional observers

Transactional observers receive their event notifications during the before or after completion phase of the transaction in which the event was raised. For example, the following observer method needs to refresh a query result set that is cached in the application context, but only when transactions that update the `Category` tree succeed:

```
public void refreshCategoryTree(@AfterTransactionSuccess @Observes CategoryUpdateEvent event) { ... }
```

There are three kinds of transactional observers:

- `@AfterTransactionSuccess` observers are called during the after completion phase of the transaction, but only if the transaction completes successfully
- `@AfterTransactionFailure` observers are called during the after completion phase of the transaction, but only if the transaction fails to complete successfully
- `@AfterTransactionCompletion` observers are called during the after completion phase of the transaction
- `@BeforeTransactionCompletion` observers are called during the before completion phase of the transaction

Transactional observers are very important in a stateful object model like Web Beans, because state is often held for longer than a single atomic transaction.

Imagine that we have cached a JPA query result set in the application scope:

```
@ApplicationScoped @Singleton
public class Catalog {

    @PersistenceContext EntityManager em;

    List<Product> products;

    @Produces @Catalog
    List<Product> getCatalog() {
        if (products==null) {
            products = em.createQuery("select p from Product p where p.deleted = false")
```

```
        .getResultList();
    }
    return products;
}
}
```

From time to time, a `Product` is created or deleted. When this occurs, we need to refresh the `Product` catalog. But we should wait until *after* the transaction completes successfully before performing this refresh!

The Web Bean that creates and deletes `Products` could raise events, for example:

```
@Stateless
public class ProductManager {

    @PersistenceContext EntityManager em;
    @Observable Event<Product> productEvent;

    public void delete(Product product) {
        em.delete(product);
        productEvent.fire(product, new AnnotationLiteral<Deleted>(){});
    }

    public void persist(Product product) {
        em.persist(product);
        productEvent.fire(product, new AnnotationLiteral<Created>(){});
    }

    ...
}
```

And now `Catalog` can observe the events after successful completion of the transaction:

```
@ApplicationScoped @Singleton
public class Catalog {

    ...

    void addProduct(@AfterTransactionSuccess @Observes @Created Product product) {
        products.add(product);
    }

    void addProduct(@AfterTransactionSuccess @Observes @Deleted Product product) {
        products.remove(product);
    }

}
```

---

## Part III. Making the most of strong typing

The second major theme of Web Beans is *strong typing*. The information about the dependencies, interceptors and decorators of a Web Bean, and the information about event consumers for an event producer, is contained in typesafe Java constructs that may be validated by the compiler.

You don't see string-based identifiers in Web Beans code, not because the framework is hiding them from you using clever defaulting rules—so-called "configuration by convention"—but because there are simply no strings there to begin with!

The obvious benefit of this approach is that *any* IDE can provide autocompletion, validation and refactoring without the need for special tooling. But there is a second, less-immediately-obvious, benefit. It turns out that when you start thinking of identifying objects, events or interceptors via annotations instead of names, you have an opportunity to lift the semantic level of your code.

Web Beans encourages you develop annotations that model concepts, for example,

- `@Asynchronous`,
- `@Mock`,
- `@Secure` **or**
- `@Updated`,

instead of using compound names like

- `asyncPaymentProcessor`,
- `mockPaymentProcessor`,
- `SecurityInterceptor` **or**
- `DocumentUpdatedEvent`.

The annotations are reusable. They help describe common qualities of disparate parts of the system. They help us categorize and understand our code. They help us deal with common concerns in a common way. They make our code more literate and more understandable.

Web Beans *stereotypes* take this idea a step further. A stereotype models a common *role* in your application architecture. It encapsulates various properties of the role, including scope, interceptor bindings, deployment type, etc, into a single reusable package.

Even Web Beans XML metadata is strongly typed! There's no compiler for XML, so Web Beans takes advantage of XML schemas to validate the Java types and attributes that appear in XML. This approach turns out to make the XML more literate, just like annotations made our Java code more literate.

We're now ready to meet some more advanced features of Web Beans. Bear in mind that these features exist to make our code both easier to validate and more understandable. Most of the time you don't ever really *need* to use these features, but if you use them wisely, you'll come to appreciate their power.

---

---

## Chapter 9. Stereotypes

According to the Web Beans specification:

In many systems, use of architectural patterns produces a set of recurring Web Bean roles. A stereotype allows a framework developer to identify such a role and declare some common metadata for Web Beans with that role in a central place.

A stereotype encapsulates any combination of:

- a default deployment type,
- a default scope type,
- a restriction upon the Web Bean scope,
- a requirement that the Web Bean implement or extend a certain type, and
- a set of interceptor binding annotations.

A stereotype may also specify that all Web Beans with the stereotype have defaulted Web Bean names.

A Web Bean may declare zero, one or multiple stereotypes.

A stereotype is a Java annotation type. This stereotype identifies action classes in some MVC framework:

```
@Retention(RUNTIME)
@Target(TYPE)
@Stereotype
public @interface Action {}
```

We use the stereotype by applying the annotation to a Web Bean.

```
@Action
public class LoginAction { ... }
```

### 9.1. Default scope and deployment type for a stereotype

A stereotype may specify a default scope and/or default deployment type for Web Beans with that stereotype. For example, if the deployment type `@WebTier` identifies Web Beans that should only be deployed when the system executes as a web application, we might specify the following defaults for action classes:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype
public @interface Action {}
```

Of course, a particular action may still override these defaults if necessary:

```
@Dependent @Mock @Action
public class MockLoginAction { ... }
```

If we want to force all actions to a particular scope, we can do that too.

### 9.2. Restricting scope and type with a stereotype

Suppose that we wish to prevent actions from declaring certain scopes. Web Beans lets us explicitly specify the set of allowed scopes for Web Beans with a certain stereotype. For example:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
```

```
@WebTier
@Stereotype(supportedScopes=RequestScoped.class)
public @interface Action {}
```

If a particular action class attempts to specify a scope other than the Web Beans request scope, an exception will be thrown by the Web Bean manager at initialization time.

We can also force all Web Bean with a certain stereotype to implement an interface or extend a class:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype(requiredTypes=AbstractAction.class)
public @interface Action {}
```

If a particular action class does not extend the class `AbstractAction`, an exception will be thrown by the Web Bean manager at initialization time.

### 9.3. Interceptor bindings for stereotypes

A stereotype may specify a set of interceptor bindings to be inherited by all Web Beans with that stereotype.

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@WebTier
@Stereotype
public @interface Action {}
```

This helps us get technical concerns even further away from the business code!

### 9.4. Name defaulting with stereotypes

Finally, we can specify that all Web Beans with a certain stereotype have a Web Bean name, defaulted by the Web Bean manager. Actions are often referenced in JSP pages, so they're a perfect use case for this feature. All we need to do is add an empty `@Named` annotation:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@Named
@WebTier
@Stereotype
public @interface Action {}
```

Now, `LoginAction` will have the name `loginAction`.

### 9.5. Standard stereotypes

We've already met two standard stereotypes defined by the Web Beans specification: `@Interceptor` and `@Decorator`.

Web Beans defines one further standard stereotype:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Model {}
```

This stereotype is intended for use with JSF. Instead of using JSF managed beans, just annotate a Web Bean `@Model`, and use it directly in your JSF page.

---

## Chapter 10. Specialization

We've already seen how the Web Beans dependency injection model lets us *override* the implementation of an API at deployment time. For example, the following enterprise Web Bean provides an implementation of the API `PaymentProcessor` in production:

```
@CreditCard @Stateless
public class CreditCardPaymentProcessor
    implements PaymentProcessor {
    ...
}
```

But in our staging environment, we override that implementation of `PaymentProcessor` with a different Web Bean:

```
@CreditCard @Stateless @Staging
public class StagingCreditCardPaymentProcessor
    implements PaymentProcessor {
    ...
}
```

What we've tried to do with `StagingCreditCardPaymentProcessor` is to completely replace `AsyncPaymentProcessor` in a particular deployment of the system. In that deployment, the deployment type `@Staging` would have a higher priority than the default deployment type `@Production`, and therefore clients with the following injection point:

```
@CreditCard PaymentProcessor ccpp
```

Would receive an instance of `StagingCreditCardPaymentProcessor`.

Unfortunately, there are several traps we can easily fall into:

- the higher-priority Web Bean may not implement all the API types of the Web Bean that it attempts to override,
- the higher-priority Web Bean may not declare all the binding types of the Web Bean that it attempts to override,
- the higher-priority Web Bean might not have the same name as the Web Bean that it attempts to override, or
- the Web Bean that it attempts to override might declare a producer method, disposal method or observer method.

In each of these cases, the Web Bean that we tried to override could still be called at runtime. Therefore, overriding is somewhat prone to developer error.

Web Beans provides a special feature, called *specialization*, that helps the developer avoid these traps. Specialization looks a little esoteric at first, but it's easy to use in practice, and you'll really appreciate the extra security it provides.

### 10.1. Using specialization

Specialization is a feature that is specific to simple and enterprise Web Beans. To make use of specialization, the higher-priority Web Bean must:

- be a direct subclass of the Web Bean it overrides, and
- be a simple Web Bean if the Web Bean it overrides is a simple Web Bean or an enterprise Web Bean if the Web Bean it overrides is an enterprise Web Bean, and
- be annotated `@Specializes`.

```
@Stateless @Staging @Specializes
public class StagingCreditCardPaymentProcessor
    extends CreditCardPaymentProcessor {
    ...
}
```

We say that the higher-priority Web Bean *specializes* its superclass.

## 10.2. Advantages of specialization

When specialization is used:

- the binding types of the superclass are automatically inherited by the Web Bean annotated `@Specializes`, and
- the Web Bean name of the superclass is automatically inherited by the Web Bean annotated `@Specializes`, and
- producer methods, disposal methods and observer methods declared by the superclass are called upon an instance of the Web Bean annotated `@Specializes`.

In our example, the binding type `@CreditCard` of `CreditCardPaymentProcessor` is inherited by `StagingCreditCardPaymentProcessor`.

Furthermore, the Web Bean manager will validate that:

- all API types of the superclass are API types of the Web Bean annotated `@Specializes` (all local interfaces of the superclass enterprise bean are also local interfaces of the subclass),
- the deployment type of the Web Bean annotated `@Specializes` has a higher precedence than the deployment type of the superclass, and
- there is no other enabled Web Bean that also specializes the superclass.

If any of these conditions are violated, the Web Bean manager throws an exception at initialization time.

Therefore, we can be certain that the superclass with *never* be called in any deployment of the system where the Web Bean annotated `@Specializes` is deployed and enabled.



---

## Chapter 11. Defining Web Beans using XML

So far, we've seen plenty of examples of Web Beans declared using annotations. However, there are a couple of occasions when we can't use annotations to define the Web Bean:

- when the implementation class comes from some pre-existing library, or
- when there should be multiple Web Beans with the same implementation class.

In either of these cases, Web Beans gives us two options:

- write a producer method, or
- declare the Web Bean using XML.

Many frameworks use XML to provide metadata relating to Java classes. However, Web Beans uses a very different approach to specifying the names of Java classes, fields or methods to most other frameworks. Instead of writing class and member names as the string values of XML elements and attributes, Web Beans lets you use the class or member name as the name of the XML element.

The advantage of this approach is that you can write an XML schema that prevents spelling errors in your XML document. It's even possible for a tool to generate the XML schema automatically from the compiled Java code. Or, an integrated development environment could perform the same validation without the need for the explicit intermediate generation step.

### 11.1. Declaring Web Bean classes

For each Java package, Web Beans defines a corresponding XML namespace. The namespace is formed by prepending `urn:java:` to the Java package name. For the package `com.mydomain.myapp`, the XML namespace is `urn:java:com.mydomain.myapp`.

Java types belonging to a package are referred to using an XML element in the namespace corresponding to the package. The name of the element is the name of the Java type. Fields and methods of the type are specified by child elements in the same namespace. If the type is an annotation, members are specified by attributes of the element.

For example, the element `<util:Date/>` in the following XML fragment refers to the class `java.util.Date`:

```
<WebBeans xmlns="urn:java:javax.webbeans"
          xmlns:util="urn:java:java.util">

  <util:Date/>

</WebBeans>
```

And this is all the code we need to declare that `Date` is a simple Web Bean! An instance of `Date` may now be injected by any other Web Bean:

```
@Current Date date
```

### 11.2. Declaring Web Bean metadata

We can declare the scope, deployment type and interceptor binding types using direct child elements of the Web Bean declaration:

```
<myapp:ShoppingCart>
  <SessionScoped/>
  <myfwk:Transactional requiresNew="true"/>
  <myfwk:Secure/>
</myapp:ShoppingCart>
```

We use exactly the same approach to specify names and binding type:

```
<util:Date>
  <Named>currentTime</Named>
</util:Date>
```

```

<util:Date>
  <SessionScoped/>
  <myapp:Login/>
  <Named>loginTime</Named>
</util:Date>

<util:Date>
  <ApplicationScoped/>
  <myapp:SystemStart/>
  <Named>systemStartTime</Named>
</util:Date>

```

Where @Login and @SystemStart are binding annotations types.

```

@Current Date currentTime;
@Login Date loginTime;
@SystemStart Date systemStartTime;

```

As usual, a Web Bean may support multiple binding types:

```

<myapp:AsynchronousChequePaymentProcessor>
  <myapp:PayByCheque/>
  <myapp:Asynchronous/>
</myapp:AsynchronousChequePaymentProcessor>

```

Interceptors and decorators are just simple Web Beans, so they may be declared just like any other simple Web Bean:

```

<myfwk:TransactionInterceptor>
  <Interceptor/>
  <myfwk:Transactional/>
</myfwk:TransactionInterceptor>

```

## 11.3. Declaring Web Bean members

TODO!

## 11.4. Declaring inline Web Beans

Web Beans lets us define a Web Bean at an injection point. For example:

```

<myapp:System>
  <ApplicationScoped/>
  <myapp:admin>
    <myapp:Name>
      <myapp:firstname>Gavin</myapp:firstname>
      <myapp:lastname>King</myapp:lastname>
      <myapp:email>gavin@hibernate.org</myapp:email>
    </myapp:Name>
  </myapp:admin>
</myapp:System>

```

The <Name> element declares a simple Web Bean of scope @Dependent and class Name, with a set of initial field values. This Web Bean has a special, container-generated binding and is therefore injectable only to the specific injection point at which it is declared.

This simple but powerful feature allows the Web Beans XML format to be used to specify whole graphs of Java objects. It's not quite a full databinding solution, but it's close!

## 11.5. Using a schema

If we want our XML document format to be authored by people who aren't Java developers, or who don't have access to our code, we need to provide a schema. There's nothing specific to Web Beans about writing or using the schema.

```

<WebBeans xmlns="urn:java:javax.webbeans"
  xmlns:myapp="urn:java:com.mydomain.myapp"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:java:javax.webbeans http://java.sun.com/jee/web-beans-1.0.xsd
    urn:java:com.mydomain.myapp http://mydomain.com/xsd/myapp-1.2.xsd">

```

```
<myapp:System>
  ...
</myapp:System>

</WebBeans>
```

Writing an XML schema is quite tedious. Therefore, the Web Beans RI project will provide a tool which automatically generates the XML schema from compiled Java code.

---

## Part IV. Web Beans and the Java EE ecosystem

The third theme of Web Beans is *integration*. Web Beans was designed to work in concert with other technologies, helping the application developer fit the other technologies together. Web Beans is an open technology. It forms a part of the Java EE ecosystem, and is itself the foundation for a new ecosystem of portable extensions and integration with existing frameworks and technologies.

We've already seen how Web Beans helps integrate EJB and JSF, allowing EJBs to be bound directly to JSF pages. That's just the beginning. Web Beans offers the same potential to diverse other technologies, such as Business Process Management engines, other Web Frameworks, and third-party component models. The Java EE platform will never be able to standardize all the interesting technologies that are used in the world of Java application development, but Web Beans makes it easier to use the technologies which are not yet part of the platform seamlessly within the Java EE environment.

We're about to see how to take full advantage of the Java EE platform in an application that uses Web Beans. We'll also briefly meet a set of SPIs that are provided to support portable extensions to Web Beans. You might not ever need to use these SPIs directly, but it's nice to know they are there if you need them. Most importantly, you'll take advantage of them indirectly, every time you use a third-party extension.

---

## Chapter 12. Java EE integration

Web Beans is fully integrated into the Java EE environment. Web Beans have access to Java EE resources and JPA persistence contexts. They may be used in Unified EL expressions in JSF and JSP pages. They may even be injected into some objects, such as Servlets and Message-Driven Beans, which are not Web Beans.

### 12.1. Injecting Java EE resources into a Web Bean

All simple and enterprise Web Beans may take advantage of Java EE dependency injection using `@Resource`, `@EJB` and `@PersistenceContext`. We've already seen a couple of examples of this, though we didn't pay much attention at the time:

```
@Transactional @Interceptor
public class TransactionInterceptor {

    @Resource Transaction transaction;

    @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

```
@SessionScoped
public class Login {

    @Current Credentials credentials;
    @PersistenceContext EntityManager userDatabase;

    ...

}
```

The Java EE `@PostConstruct` and `@PreDestroy` callbacks are also supported for all simple and enterprise Web Beans. The `@PostConstruct` method is called after *all* injection has been performed.

There is one restriction to be aware of here: `@PersistenceContext(type=EXTENDED)` is not supported for simple Web Beans.

### 12.2. Calling a Web Bean from a Servlet

It's easy to use a Web Bean from a Servlet in Java EE 6. Simply inject the Web Bean using Web Beans field or initializer method injection.

```
public class Login extends HttpServlet {

    @Current Credentials credentials;
    @Current Login login;

    @Override
    public void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        credentials.setUsername( request.getAttribute("username") );
        credentials.setPassword( request.getAttribute("password") );
        login.login();
        if ( login.isLoggedIn() ) {
            response.sendRedirect("/home.jsp");
        }
        else {
            response.sendRedirect("/loginError.jsp");
        }
    }

}
```

The Web Beans client proxy takes care of routing method invocations from the Servlet to the correct instances of `Credentials` and `Login` for the current request and HTTP session.

### 12.3. Calling a Web Bean from a Message-Driven Bean

Web Beans injection applies to all EJBs, even when they aren't under the control of the Web Bean manager (if they were

obtained by direct JNDI lookup, or injection using `@EJB`, for example. In particular, you can use Web Beans injection in Message-Driven Beans, which are not considered Web Beans because you can't inject them.

You can even use Web Beans interceptor bindings for Message-Driven Beans.

```
@Transactional @MessageDriven
public class ProcessOrder implements MessageListener {

    @Current Inventory inventory;
    @PersistenceContext EntityManager em;

    public void onMessage(Message message) {
        ...
    }
}
```

Thus, receiving messages is super-easy in a Web Beans environment. But beware that there is no session or conversation context available when a message is delivered to a Message-Driven Bean. Only `@RequestScoped` and `@ApplicationScoped` Web Beans are available.

It's also easy to send messages using Web Beans.

## 12.4. JMS endpoints

Sending messages using JMS can be quite complex, because of the number of different objects you need to deal with. For queues we have `Queue`, `QueueConnectionFactory`, `QueueConnection`, `QueueSession` and `QueueSender`. For topics we have `Topic`, `TopicConnectionFactory`, `TopicConnection`, `TopicSession` and `TopicPublisher`. Each of these objects has its own lifecycle and threading model that we need to worry about.

Web Beans takes care of all this for us. All we need to do is declare the queue or topic in `web-beans.xml`, specifying an associated binding type and connection factory.

```
<Queue>
  <destination>java:comp/env/jms/OrderQueue</destination>
  <connectionFactory>java:comp/env/jms/QueueConnectionFactory</connectionFactory>
  <myapp:OrderProcessor/>
</Queue>
```

```
<Topic>
  <destination>java:comp/env/jms/StockPrices</destination>
  <connectionFactory>java:comp/env/jms/TopicConnectionFactory</connectionFactory>
  <myapp:StockPrices/>
</Topic>
```

Now we can just inject the `Queue`, `QueueConnection`, `QueueSession` or `QueueSender` for a queue, or the `Topic`, `TopicConnection`, `TopicSession` or `TopicPublisher` for a topic.

```
@OrderProcessor QueueSender orderSender;
@OrderProcessor QueueSession orderSession;

public void sendMessage() {
    MapMessage msg = orderSession.createMapMessage();
    ...
    orderSender.send(msg);
}
```

```
@StockPrices TopicPublisher pricePublisher;
@StockPrices TopicSession priceSession;

public void sendMessage(String price) {
    pricePublisher.send( priceSession.createTextMessage(price) );
}
```

The lifecycle of the injected JMS objects are completely controlled by the Web Bean manager.

---

## Chapter 13. Extending Web Beans

Web Beans is intended to be a platform for frameworks, extensions and integration with other technologies. Therefore, Web Beans exposes a set of SPIs for the use of developers of portable extensions to Web Beans. For example, the following kinds of extensions were envisaged by the designers of Web Beans:

- integration with Business Process Management engines,
- integration with third-party frameworks such as Spring, Seam, GWT or Wicket, and
- new technology based upon the Web Beans programming model.

The nerve center for extending Web Beans is the `Manager` object.

### 13.1. The `Manager` object

The `Manager` interface lets us register and obtain Web Beans, interceptors, decorators, observers and contexts programmatically.

```
public interface Manager
{
    public <T> Set<Bean<T>> resolveByType(Class<T> type, Annotation... bindings);
    public <T> Set<Bean<T>> resolveByType(TypeLiteral<T> apiType,
        Annotation... bindingTypes);
    public <T> T getInstanceByType(Class<T> type, Annotation... bindingTypes);
    public <T> T getInstanceByType(TypeLiteral<T> type,
        Annotation... bindingTypes);
    public Set<Bean<?>> resolveByName(String name);
    public Object getInstanceByName(String name);
    public <T> T getInstance(Bea<T> bean);
    public void fireEvent(Object event, Annotation... bindings);
    public Context getContext(Class<? extends Annotation> scopeType);
    public Manager addContext(Context context);
    public Manager addBean(Bea<?> bean);
    public Manager addInterceptor(Interceptor interceptor);
    public Manager addDecorator(Decorator decorator);
    public <T> Manager addObserver(Observer<T> observer, Class<T> eventType,
        Annotation... bindings);
    public <T> Manager addObserver(Observer<T> observer, TypeLiteral<T> eventType,
        Annotation... bindings);
    public <T> Manager removeObserver(Observer<T> observer, Class<T> eventType,
        Annotation... bindings);
    public <T> Manager removeObserver(Observer<T> observer,
        TypeLiteral<T> eventType, Annotation... bindings);
    public <T> Set<Observer<T>> resolveObservers(T event, Annotation... bindings);
    public List<Interceptor> resolveInterceptors(InterceptionType type,
        Annotation... interceptorBindings);
    public List<Decorator> resolveDecorators(Set<Class<?>> types,
        Annotation... bindingTypes);
}
```

We can obtain an instance of `Manager` via injection:

```
@Current Manager manager
```

## 13.2. The `Bean` class

Instances of the abstract class `Bean` represent Web Beans. There is an instance of `Bean` registered with the `Manager` object for every Web Bean in the application.

```
public abstract class Bean<T> {  
    private final Manager manager;  
  
    protected Bean(Manager manager) {  
        this.manager=manager;  
    }  
  
    protected Manager getManager() {  
        return manager;  
    }  
  
    public abstract Set<Class> getTypes();  
    public abstract Set<Annotation> getBindingTypes();  
    public abstract Class<? extends Annotation> getScopeType();  
    public abstract Class<? extends Annotation> getDeploymentType();  
    public abstract String getName();  
  
    public abstract boolean isSerializable();  
    public abstract boolean isNullable();  
  
    public abstract T create();  
    public abstract void destroy(T instance);  
}
```

It's possible to extend the `Bean` class and register instances by calling `Manager.addBean()` to provide support for new kinds of Web Beans, beyond those defined by the Web Beans specification (simple and enterprise Web Beans, producer methods and JMS endpoints). For example, we could use the `Bean` class to allow objects managed by another framework to be injected into Web Beans.

There are two subclasses of `Bean` defined by the Web Beans specification: `Interceptor` and `Decorator`.

## 13.3. The `Context` interface

The `Context` interface supports addition of new scopes to Web Beans, or extension of the built-in scopes to new environments.

```
public interface Context {  
    public Class<? extends Annotation> getScopeType();  
  
    public <T> T get(Bean<T> bean, boolean create);  
  
    boolean isActive();  
}
```

For example, we might implement `Context` to add a business process scope to Web Beans, or to add support for the conversation scope to an application that uses Wicket.



---

## Chapter 14. Next steps

Because Web Beans is so new, there's not yet a lot of information available online.

Of course, the Web Beans specification is the best source of more information about Web Beans. The spec is about 100 pages long, only two and a half times the length of this article, and almost as readable. But of course, it covers many details that we've skipped over. The spec is available from <http://jcp.org/en/jsr/detail?id=299>.

The Web Beans Reference implementation is being developed at <http://seamframework.org/WebBeans>. The RI development team and the Web Beans spec lead blog at <http://in.relation.to>. This article is substantially based upon a series of blog entries published there.