Get More Refcardz! Visit refcardz.com
www.dzone.com
Java Contexts & Dependency Injection

CONTENTS INCLUDE:

- About CDI
- About Weld
- The Bean
- Using Injection
- Extending Beans
- Events and more...

Contexts and Dependency Injection for the Java EE Platform

By Norman Richards

ABOUT CDI

Contexts and Dependency Injection for the Java EE Platform (CDI) introduces a standard set of component management services to the Java EE platform. CDI manages the lifecycle and interactions of stateful components bound to well-defined contexts. CDI provides typesafe dependency injection between components. CDI provides interceptors and decorators to extend the behavior of components, an event model for loosely coupled components, and an SPI allowing portable extensions to integrate cleanly with the Java EE environment.

CDI and Java EE

The Java EE 6 platform includes CDI as part of both the full profile and the web profile. The EE 6 platform was designed to make sure all EE components make use of CDI services, putting CDI directly at the heart of the platform, welding together the various EE technologies.

CDI and Seam

CDI was influenced by many technologies, with Seam in particular pioneering many of the concepts that have been formalized in CDI. However, Seam is not CDI, and not all of Seam's innovations are part of the CDI specification. As such, the latest version of Seam, Seam 3, will be based on CDI and will provide a collection of portable extensions to CDI that can be integrated cleanly into any CDI environment. Seam 3 will also provide a legacy Seam 2 mode, giving existing Seam applications a migration path onto CDI.

ABOUT WELD

Weld is the CDI reference implementation. It is a great place to get started with CDI. Weld can be downloaded from http://seamframework.org/Download.

Running the examples

Weld comes with an extensive library of examples that is a great starting point to learn CDI. These examples run in Java EE containers such as JBoss AS and Glassfish as well as in servlet containers like Tomcat and Jetty.

Example name	Description
examples/jsf/login	A simple example demonstrating a user login component
examples/jsf/numberguess	A number guessing game
examples/jsf/permalink	A blog example
examples/jsf/translator	A text translation example using EJBs
examples/wicket/numberguess	The number guessing game implemented with Wicket
examples/wicket/gae	An alternative version of the Wicket number guess that runs on Google App Engine

Building the examples

Each example can be built using Maven 2.

```
$ mvn clean install
```

The target directory contains the built archive, which can be deployed to EE containers like JBoss AS and Glassfish. The examples that build to WAR files can also be deployed to Tomcat and Jetty.

For convenience, each example contains an Ant file with targets to build and deploy directly to JBoss AS and Tomcat.

```
ant -Djboss.home=/path/to/jboss-6.0 deploy
ant -Djboss.home=/path/to/jboss-6.0 undeploy

ant –Dtomcat.home=/path/to/tomcat tomcat.deploy
ant –Dtomcat.home=/path/to/tomcat tomcat.undeploy
```

Creating a Weld project

New project stubs can be created using Maven 2 archetypes.

Interactive Mode

```
mvn archetype:generate
```

Non-interactive mode

```
mvn archetype:generate
    -DinteractiveMode=n
    -DarchetypeArtifactId=weld-jsf-servlet-minimal
    -DarchetypeGroupId=org.jboss.weld.archetypes
    -Dversion=1.0.01
    -DgroupId=com.mycompany
    -DartifactId=myproject
```

Weld archetypes

Weld archetype	Description
weld-jsf-servlet-minimal	A Weld web application using JavaServer Faces (JSF)
weld-jsf-jee-minimal	A Weld application using EJB and JSF but no persistence
weld-jsf-jee	A Weld application using EJB and JSF with a persistence context

Need to integrate Spring, Hibernate, or Seam with JBoss AS?

We've already done it for you. JBoss Enterprise Application Platform 5.0 includes pre-integrated frameworks for building all sorts of Java apps.

- Customize your app server footprint
- Simply your configurations
- Bring the power of full text search to your app with Hibernate Search
- Use our pre-integrated Apache CXF web services stack
- Seam, Hibernate, and Hibernate Search tooling through JBoss Developer Studio Hibernate queries

Download today: jboss.com/download

Building a Weld project

```
mvn clean install
```

THE BEAN

CDI, at the most basic level, revolves around the notion of beans. Beans are instantiated by CDI, and their lifecycle is managed according to the stateful context to which they belong. So, it's natural to start by asking what is a bean? In CDI, almost any object can be a bean. When using CDI in a Java EE environment, any Java EE component is a bean if it is defined to be a managed bean by its EE specification. This includes session beans, message-driven beans, servlets, filters, etc.

Most non-EE POJOs are also automatically managed beans and no special declarations are required to define a managed bean. A bean definition consists of the following:

Bean Type	The bean type is the set of Java types that the bean provides. CDI injection always uses type as the primary identifier for determining the bean that will provide an instance. Unless otherwise restricted, a bean's type includes all the superclasses and interfaces in the Java type hierarchy.
Qualifier	There may be multiple beans that implement a desired Java type. It is possible to distinguish between multiple types using qualifier annotations. Qualifiers are developer-defined and provide a type-safe way to distinguish between multiple beans implementations.
Scope	All beans have a well-defined scope that determines the lifecycle and visibility of instances. The set of scopes is fully extensible, with built-in scopes including request scope, conversation scope, session scope and application scope. Beans can also be dependent and inherit the scope of their injection point.
EL name	Although it's completely optional, beans may define an EL name that can be used for non-type safe access. One common usage of EL name is for binding components to JavaServer Faces (JSF) views.
Interceptors	A bean's behavior can be extended or overridden by adding interceptors and decorators to the bean.
Implementation	All beans must of course provide an implementation of the types they provide. This is normally the Java class that defines the bean.

USING INJECTION

Not all instances of a bean type are managed instances. Instances created with the Java new operator are not managed. Only instances provided by the CDI BeanManager or through an injection point are managed instances.

Managed instances can have injection performed on them when they are created. Injection points are declared using the `@javax.inject.Inject` annotation.

Injection is performed at creation time for any fields annotated `@Inject`.

```
public class Foo
{
    @Inject Bar bar;
}
```

Injection is also performed for any methods annotated `@Inject`.

```
public class Foo
{
    @Inject
    public void setBar(Bar bar) {
        // …
    }
}
```

Method injection points can be thought of as initializer methods. Method injection points can support multiple injected arguments.

```
public class Foo
{
    @Inject
    public void initializeMe(Bar bar, Baz baz) {
        // …
    }
}
```

Beans can have any number of field or method injection points. Additionally, a bean may have a constructor annotated `@Inject`. Any declared parameters will be injected when the instance is created.

```
public class Foo
{
    @Inject
    public Foo(Bar bar, Baz baz) {
        // …
    }
}
```

If a bean doesn't designate a constructor using `@Inject`, the no argument constructor will be used.

Producers

If CDI needs to instantiate a new instance, it will normally call the designated constructor. However, it's possible to directly control the creation of new instances with producer methods. Producer methods are annotated `@javax.enterprise.inject.Produces` and return the object to be produced. Any arguments to a producer method will be injected by CDI.

```
public class FooProducer {
    @Produces
    public Foo makeFooFromBar(Bar bar) {
        return new Foo(bar);
    }
}
```

An alternative form of producer is the producer field. The value of the producer field is the value to be injected.

```
public class AnotherFooProducer {
    @Produces Foo foo;

    @Inject
    public void initalizeMe(Bar bar) {
        foo = new Foo(bar);
    }
}
```

Producer fields can be combined with Java EE injection annotations such as `@Resource`, `@EJB` and `@PersistenceContext`.

```
@Produces
@WebServiceRef(lookup="java:app/service/PaymentService")
PaymentService paymentService;

@Produces
@EJB(ejbLink="../their.jar#PaymentService")
PaymentService paymentService;
```

New instances

CDI injects the contextual, and thus possibly shared, instance of a bean. When this is not desired, the `@javax.enterprise.inject.New` annotation can be used to force a new instance to be created and injected.

```
public class FooProducer {
    @Produces public Foo makeFoo(@New Bar bar) {
        return new Foo(bar);
    }
}
```

Programmatic lookup

Injection occurs when a component is created and initialized by CDI. There are many cases where this is not desirable. In those cases, programmatic lookup is available by injecting the corresponding `javax.enterprise.inject.Instance` for a bean.

```
public class Foo {
    @Inject Instance<Bar> bar;

    public Bar getBar() {
        return bar.get();
    }
}
```

The get method performs the actual lookup defined by the injection point.

Qualifiers

CDI does injection by type, but most systems have the need for more than one instance of a given type. Rather than giving them unique names or identifiers, CDI handles this through qualifiers. Qualifiers are annotations on bean types and injection points that differentiate between types. If a type is a noun, then a qualifier is an adjective that can be used to describe and distinguish the nouns.

A qualifier is an annotation that itself is annotated with the `@javax.inject.Qualifier` meta-annotation.

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

Qualifiers can be added to a class to define a qualified bean type. Typically this will be a subclass of a parent type or the implementation of an interface.

```
@Synchronous
public class SynchronousPaymentProcessor
    implements PaymentProcessor
{
    // ...
}
```

Producer methods and fields can also use qualifiers to distinguish between the types.

```
@Produces
@Asynchronous
public PaymentProcessor createAsynchronousProcessor() {
    return new AsynchronousPaymentProcessor();
}
```

It's now possible to distinguish between the two types by adding the qualifier to any injection point. It's not necessary to know the specific subtype or implementation class.

```
@Inject @Asynchronous PaymentProcessor processor;

@Inject
public void processSynchronously(@Synchronous PaymentProcessor) {
    // …
}
```

Hot Tip A bean can have multiple qualifiers. Injection points only need to specify enough qualifiers to uniquely match a bean.

Qualifiers annotations may have members.

```
@Target({FIELD, PARAMETER})
@Retention(RUNTIME)
@Qualifier
public @interface Currency {
    public String code();
}
```

All members must be equal, so the following injection point and producer field would not match.

```
@Inject @Currency(code="USD") PaymentProcessor processor;
@Produces @Currency(code="EUR") PaymentProcessor processor;
```

However, if the qualifier annotation member is marked as `@javax.enterprise.util.Nonbinding`, then the member values would not be considered and the injection point would match the producer field.

```
@Target({FIELD, PARAMETER})
@Retention(RUNTIME)
@Qualifier
public @interface Currency {
    @Nonbinding public String code();
}
```

Nonbinding values can provide useful metadata to a method. These nonbinding values can be retrieved by injecting the `javax.enterprise.inject.spi.InjectionPoint` and querying the qualifiers.

```
@Produces
@Currency(code="USD")
public PaymentProcessor processor(InjectionPoint injectionPoint) {
    PaymentProcessor processor = new PaymentProcessor();

    for (Annotation qualifier:injectionPoint.getQualifiers()) {
        if (qualifier instanceof Currency) {
            Currency currency = (Currency) qualifier;
            processor.setBaseCurrency(currency.code());
            break;
        }
    }

    return processor;
}
```

The built-in qualifiers

Qualifier	Description
@javax.enterprise.inject.Any	Every bean and every injection point has the @Any qualifier, even if it is not specified, unless the @New annotation is specified.
@javax.enterprise.inject.New	The @New annotation forces a new instance to be created by the container instead of using the contextual instance.
@javax.enterprise.inject.Default	Any bean or injection point that does not declare the @Named qualifier has the default qualifier added.
@javax.enterprise.inject.Named	This qualifier declares a text name that can be used to reference the bean. This is used for un-typed access such as through EL.

Bean names

The `@javax.inject.Named` qualifier specifies a textual that can be used in places where CDI's type-based lookup is not possible. The primary use case is EL access in places such as JSF views.

```
@Inject @Named("foo") Foo foo;
```

The String value of the annotation is the name of the bean. If the `@Named` annotation doesn't explicitly specify a name, the container will derive a default name. The default name is the unqualified class name of the bean class, after converting the first character to lower case. The following two injection points use the exact same name.

```
@Inject @Named Bar namedBar1;
@Inject @Named("bar") Bar namedBar2;
```

Bean Scopes

CDI manages contextual objects. Contextual objects are stateful and have a distinct lifecycle determined by the scope they belong to. When a contextual bean is needed, CDI looks in the appropriate shared context for the instance to inject. If

there is no shared instance, CDI creates one and stores it in the context for future use. When the enclosing scope is destroyed, the beans inside will be destroyed.

Bean Scope	Description
@javax.enterprise.context. RequestScoped	@RequestScoped beans are shared for the length of a single request. This could be an HTTP request, a remote EJB invocation, a web services invocation or message-delivery to an MDB. These beans are destroyed at the end of the request.
@javax.enterprise.context. ConversationScoped	@ConversationScoped beans are shared across multiple requests in the same HTTP session but only if there is an active conversation maintained. Conversations are supported for JSF requests through the javax.enterprise.context.Conversation bean.
@javax.enterprise.context. SessionScoped	@SessionScoped beans are shared between all requests that occur in the same HTTP session and are destroyed when the session is destroyed.
@javax.enterprise.context. ApplicationScoped	An @ApplicationScoped bean will live for as long as the application is running and is destroyed when the application is shut down.
@javax.enterprise.context. Dependent	@Dependent beans are never shared between injection points. Any injection of a dependent bean is a new instance whose lifecycle is bound to the lifecycle of the object it is being injected into.

The set of scopes is extensible. New scopes are declared with the `@javax.inject.Scope` or `@javax.enterprise.context.NormalScope` meta-annotation.

Bean Destruction

When a contextual bean goes out of scope, it is destroyed. To destroy a bean, the container calls any `@PreDestroy` callbacks for the bean and destroys any `@Dependent` objects before disposing of the object.

An application can perform custom cleanup of created objects by using a dispose method. A dispose method is the analog of a producer method and is designated by marking the parameter with `@javax.enterprise.inject.Disposes`.

```
@ApplicationScoped
public class BarProducer {
    ArrayList<Bar> allBars = new ArrayList<Bar>();

    @Produces
    public Bar createBar() {
        Bar newBar = new Bar();
        allBars.add(newBar);
        return newBar;
    }
    public void disposeBar(@Disposes Bar bar) {
        allBars.remove(bar);
    }
}
```

As with producer methods, disposer methods may take additional arguments to receive injected values. Only the one argument to be disposed is annotated `@Disposes`.

Alternatives

Alternatives allow for deployment-time selection of bean implementation. An alternative is a bean marked with the `@javax.enterprise.inject.Alternative` annotation. Alternatives provide an alternate implementation of a bean that is not enabled unless is specifically enabled in the beans.xml file, in which case it overrides the original bean.

```
@Alternative
public class AlternativeFoo
   extends Foo
{
    // alternative implementation
}
```

Alternatives are enabled only when activated in the `beans.xml` file.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee">
    <alternatives>
        <class>org.example.AlternativeFoo</class>
    </alternatives>
</beans>
```

If the `@Alternative` annotation is applied to a stereotype, all beans with the stereotype may be enabled as a group.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee">
    <alternatives>
        <stereotype>org.example.MyAlternatives</stereotype>
    </alternatives>
</beans>
```

An alternative that extends the object it replaces will normally want to directly inherit the metadata (qualifiers, name, etc.) of the parent. In that case, the alternative should include the `@javax.enterprise.inject.Specializes` annotation to ensure that the original class is completely replaced by the alternative.

```
@Alternative
@Specializes
public class AlternativeFoo
   extends Foo
{
    // alternative implementation
}
```

EXTENDING BEAN FUNCTIONALITY

CDI supports two mechanisms for dynamically adding or modifying the behavior of beans: interceptors and decorators.

Interceptors

Interceptors provide a mechanism for implementing functionality across multiple beans and bean methods that is orthogonal to the core function of those beans.

Interceptor

An interceptor is a bean declared with the `@javax.interceptor.Interceptor annotation`. Method interceptor should have a method annotated `@javax.interceptor.AroundInvoke` that takes the javax.interceptor.InvocationContext as a parameter.

```
@Interceptor
public class TransactionInterceptor {
    @AroundInvoke
    public Object manageTransaction(InvocationContext ctx) {
        // …
    }
}
```

Interceptor Binding Type

Interceptors are bound using an interceptor binding type. An interceptor binding type may be declared by specifying the `@javax.interceptor.InterceptorBinding` meta-annotation.

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

The interceptor binding is applied to both the interceptor and the interception point to bind the two together.

```
@Interceptor
@Transactional
public class TransactionInterceptor {
    // …
}

@Tranactional
public class Foo {
    //…
}
```

An interceptor bound to a class will intercept all methods. Alternatively, the interceptor can be bound to specific methods.

```
public class Foo {
    @Transactional
    public void someTransactionalWork() {
        // …
    }
}
```

As with qualifiers, binding types may declare members. For an interceptor binding to match, all members must be equal unless they are declared `@NonBinding`.

Interceptors are not enabled unless they are declared in the beans.xml file.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee">
    <interceptors>
        <class>org.example.TransactionInterceptor</class>
        <class>org.example.LoggingInterceptor</class>
     </interceptors>
</beans>
```

If multiple interceptors are defined for a call, the interceptors calls are chained. The ordering is determined by the order they are listed in `beans.xml`.

Decorators

Decorators also dynamically extend beans but with a slightly different mechanism than interceptors. Where interceptors deliver functionality orthogonal to potentially many beans, decorators extend the functionality of a single bean type with functionality that is specific to that type.

A decorator is bean with the `@javax.decorator.Decorator` annotation. A decorator only decorates the interfaces that it implements.

```
@Decorator class TimestampLogger
    implements Logger
{
    @Inject @Delegate Logger logger;

    public void log(String message) {
        logger.log( timestamp() + ": " + message);
    }
}
```

A decorator must declare a single delegate injection point annoted `@javax.decorator.Delegate`. The delegate injection point is the object to be decorated. Any calls to the delegate object that correspond to a decorated type will be called on the decorator, which may in turn invoke the method directly on the delegate object.

The decorator bean does not need to implement all methods of the decorated types and may be abstract. Decorators are called after interceptors.

Decorators are not active unless they are explicitly enabled in beans.xml.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee">
    <decorators>
        <class>org.example.TimestampLogger</class>
        <class>org.example.IdentityLogger</class>
    </decorators>
</beans>
```

EVENTS

Events provide a mechanism for loosely coupled communication between components. An event consists of an event type, which may be any Java object, and optional event qualifiers.

The event object

Events are managed through instances of javax.enterprise.event.Event. Event objects are injected based on the event type.

```
@Inject Event<LoggedInEvent> normalEvent;
@Inject @Admin Event<LoggedInEvent> adminEvent;
```

Events are fired by calling fire() with an instance of the event type to be passed to the observer.

```
event.fire(new LoggedInEvent(username));
```

Observers

Observers listen for events with observer methods. The event type is annotated `@javax.enterprise.event.Observes`. Additional parameters to an observer method are normal CDI injection points.

```
public void afterLogin(@Observes LoggedInEvent event) {
    //...
}

public void afterAdminLogin(@Observes @Admin LoggedInEvent event) {
    // ...
}
```

If there are multiple observers for an event, the order that they are called in is not defined.

Conditional observers

If an instance of a component with an observer method doesn't exist when the corresponding event is fired, the container will instantiate a new instance to handle the event. This behavior is controllable using the receive value of `@Observes`.

```
public void refreshOnDocumentUpdate(@Observes(receive=IF_EXISTS)
                                     @Updated Document doc) {
    // ...
}
```

`javax.enterprise.event.Reception`

Reception value	Meaning
IF_EXISTS	The observer method is only called if an instance of the component already exists.
ALWAYS	The observer method is always called. If an instance doesn't exist, one will be created. This is the default value.

Transactional observer

Events are normally processed when the event is fired. For transactional methods, it is often desirable for the event at a certain point in the transaction lifecycle, such as after the transaction completes. This is specified with the during value of `@Observes`. If a transaction phase is specified but no transaction is active, the event is fired immediately.

TransactionPhase value	Meaning
IN_PROGRESS	The event is called when it is fired, without regard to the transaction phase. This is the default value.
BEFORE_COMPLETION	The event is called during the before completion phase of the transaction.
AFTER_COMPLETION	The event is called during the after completion phase of the transaction.
AFTER__FAILURE	The event is called during the after completion phase of the transaction, only when the transaction fails.
AFTER_SUCCESS	The event is called during the after completion phase of the transaction, only when the transaction completes successfully.

STEREOTYPES

A stereotype is a meta-annotation that bundles multiple annotations together for re-use. A stereotype may be declared by specifying the `@javax.enterprise.inject.Stereotype` meta-annotation.

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

Any bean that is annotated `@Action` will inherit all of the annotations of the stereotype.

CDI defines the stereotype, `@javax.enterprise.inject.Model` for declaring the model layer of a web application.

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

CONVERSATIONS

Conversations are available in JSF only. Programmatically accessible through `javax.enterprise.context.Conversation` component.

```
public interface Conversation {
    public void begin();
    public void begin(String id);
    public void end();
    public String getId();
    public long getTimeout();
    public void setTimeout(long milliseconds);
    public boolean isTransient();
}
```

Any conversation is in one of two states: transient or long-running. Switch between states by calling begin/end. Long running conversations and their state will be maintained by requests in that conversation. Transient conversations are destroyed at the end of the request.

If a conversation is requested that is timed out or otherwise destroyed, a `javax.enterprise.context.NonexistentConversationException` is thrown.

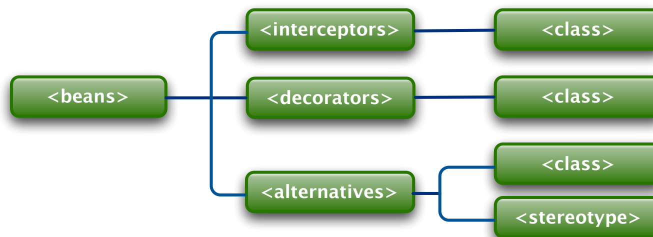
DEPLOYMENT

Bean archive

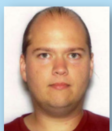
Bean classes of enabled beans must be deployed in bean deployment archives. A bean deployment archive is any JAR, EE archive, or directory on the classpath that contains a beans.xml file in the META-INF directory. For WAR files, the WEB-INF classes directory is also considered if there is a beans.xml file in the WEB-INF directory.

beans.xml structure

The beans.xml file is defined by the XSD at http://java.sun.com/xml/ns/javaee/beans_1_0.xsd

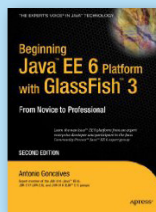


ABOUT THE AUTHOR



Norman Richards is a senior software engineer at Socialware in Austin, Texas. He is an independent contributor to the Seam and Weld projects and was formerly a core developer on Seam at Red Hat and JBoss. He is the author of numerous articles and several books, including JBoss: A Developer's Notebook, JBoss 4.0: The Official Guide and JBoss: A developer's Notebook. Norman is a graduate of the University of Texas at Austin. Norman can be contacted through his website at http://nostacktrace.com/

RECOMMENDED BOOK

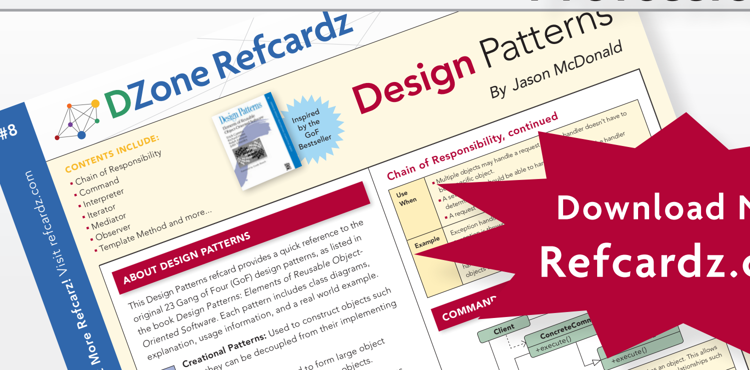


Step by step and easy to follow, this book describes many of the Java EE 6 specifications and reference implementations, and shows them in action using practical examples. This book uses the new version of GlassFish 3 to deploy and administer the code examples.

BUY NOW

books.dzone.com/books/javaee-glassfish

Professional Cheat Sheets You Can Trust



"Exactly what busy developers need: simple, short, and to the point."

James Ward, Adobe Systems

Upcoming Titles

Blaze DS
Domain Driven Design
Virtualization
Java Performance Tuning
Expression Web
Spring Web Flow
BPEL

Most Popular

Spring Configuration
jQuery Selectors
Windows Powershell
Dependency Injection with EJB 3
Netbeans IDE JavaEditor
Getting Started with Eclipse
Very First Steps in Flex

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-83-7
ISBN-10: 1-934238-83-X
50795
9 781934 238837
$7.95

Copyright © 2009 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Version 1.0