



CONTENTS INCLUDE:

- › What is Camel?
- › Why use Camel?
- › Configuring Camel Components
- › Essential Components
- › Hot Tips and more...

Camel Essential Components

By Christian Posta

WHAT IS CAMEL?

Camel is an open-source, lightweight, integration library that allows your applications to accomplish intelligent routing, message transformation, and protocol mediation using the established Enterprise Integration Patterns and out-of-the-box adapters with a highly expressive Domain Specific Language (Java, XML, or Scala). With Camel you can implement integration solutions as part of an overarching ESB solution, or as integration routes deployed to any container such as Tomcat, OSGI (FuseESB), or even a stand-alone java process.

WHY USE CAMEL?

Camel simplifies systems integrations with an easy-to-use DSL to create routes that clearly identify the integration intentions and endpoints. Camel's out of the box integration components are modeled after the Enterprise Integration Patterns cataloged in Gregor Hohpe and Bobby Wolf's book (<http://www.eaipatterns.com>). You can use these EIPs as pre-packaged units, along with any custom processors or external adapters you may need, to easily assemble otherwise complex routing and transformation routes.

```
from( "jms:incomingQueue" )
    .convertBodyTo( String.class )
    .unmarshal( jaxb )
    .process(new Processor() {
        @Override
        public void process( Exchange exchange ) throws Exception {
            .... Custom Logic If Needed ....
        }
    })
    .marshal( jaxb )
    .convertBodyTo( String.class )
    .to( "jms:outgoingQueue" );
```

Camel allows you to integrate with quite a few protocols out of the box and the community is constantly adding more. Each component is highly flexible and can be easily configured using Camel's consistent URI syntax. This Refcard introduces you to the more popular and widely used components and their configurations:

- Log
- JMS/ActiveMQ
- CXF
- File
- SED/ADirect
- Mock

CONFIGURING CAMEL COMPONENTS

All Camel components can be configured with a familiar URI syntax. Usually, components are specified in the from() clause (beginning of a route) or the to() clause (typically termination points in the route, but not always). A component "creates" specific "endpoint" instances. Think of a component as an "endpoint factory" where an endpoint is the actual object listening for a message (in the "from" scenario) or sending a message (in the "to" scenario).

An endpoint URI usually has the form:

```
<component>:<endpoint-local-name>:<config=value&config=value>
```

The component is listed first, followed by some local name specific to the endpoint, and finally some endpoint-specific configuration

Example:

```
jms:test-queue?preserveMessageQos=true
```

This URI creates a JMS endpoint that listens to the "test-queue" and sets the preserveMessageQos option to "true".

There are over 100 Camel components and each one has configuration options to finely tune its behavior. See the Camel components list to see if a component is available for the types of integrations you might need:

<http://camel.apache.org/components.html>

ESSENTIAL COMPONENTS

Bean Component

Camel implements the Service Activator pattern (from the EIPs) which allows you to plug a service into your route through a thin layer that's responsible for mediating between the messaging system (or integration route) and the service. Camel implements the Service Activator with its "bean" component and allows you to bind to and invoke POJO beans within your route.

Basic usage

Define your bean in the Camel registry (usually the spring context if you are instantiating from a Spring application context).

```
<bean id="beanName" class="com.some.class.ClassName"> ... </bean>
```

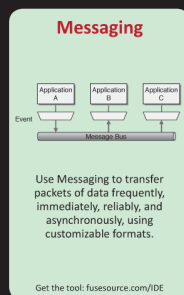
Then from your route, you can invoke a method on the bean:

```
from("direct:incoming").beanRef("beanName", "methodName")
```

FuseSource



Enterprise Integration Patterns



You can print out and use **The Enterprise Integration Pattern** electronic flashcards to visually piece together an integration solution.

[Free Download](#)

Instead of relying on the registry to have the bean, you could just let Camel instantiate and manage the bean by supplying the class:

```
from("direct:incoming").bean(InvokeCustomLogicService.class, "methodName")
```

How is a method chosen?

You don't always have to supply an explicit method name, but if you do it makes it easier on both Camel and reading the route. If you don't supply an explicit method name, Camel will do its best to resolve the correct method using the following rules:

- Is there a header with the key "CamelBeanMethodName" with a non-null value?
- If there is only a single method, Camel will try to use that.
- If there are multiple methods, camel will try to select one that has only one parameter.
- If there is an @Handler annotation, Camel will use that one.
- Lastly, Camel will try to match a method's parameters by type to the incoming message body.

How does Camel bind parameters?

Camel will automatically try to bind the first parameter to the body of the incoming message. Otherwise, if you have parameters of these types, they will automatically be bound, regardless of order:

- **CamelContext** – Use when you need access to the full context, components defined in the context, or to create objects that require the context.
- **Registry** – The registry is the object that holds all of the beans that might be used in a CamelContext; usually the Spring BeanFactory/ ApplicationFactory (but not always).
- **Exchange** – Contains headers, body, properties, and overall state for processing a message unit including a reply.
- **Message** – The incoming message.
- **TypeConverter** – Part of Camel's type conversion internals; can be used to explicitly convert types.

You can also use annotations to specifically bind certain parts of the Exchange to parameters in your methods. See the Camel documentation (<http://camel.apache.org/parameter-binding-annotations.html>) for more.

Further Reading

For more information on the bean component, see <http://camel.apache.org/bean.html>

Log Component

We always advise using logging in your routes to make it clear what steps in the processing have completed successfully. One way to do that is through the Log Component. The logging mechanism that's used is slf4j. Slf4j allows you to configure different types of logger implementations including log4j, logback, and the JDK's built in logging.

Logging

To log an exchange at debug level:

```
from("direct:start").to("log:com.fusesource.examples?level=DEBUG")
```

This will output the exchange type (InOnly/InOut) and the body of the In message. You some control over what is logged; for example, to see the headers along with the exchange:

```
from("direct:start")
    .to("log:com.fusesource.examples?showHeaders=true")
```

To make the Exchange logging easier to read, consider enabling multiline printing:

```
from("direct:start") .to("log:com.fusesource.examples?showHeaders=true&multiline=true")
```

For logging streams, you must determine whether you want to cache the stream first before logging it. Streams are read-once entities, and if you don't cache them, they won't be available to processors further down the chain. To cache streams and log them:

```
from("direct:start").streamCaching() .to("log:com.fusesource.examples?showHeaders=true&showStreams=true")
```

Formatting

You can add configurations to fine-tune exactly what's logged. Note that formatting options don't apply to groups when using groupSize, groupInterval, etc.

Option	Description
showAll	Turns all options on, such as headers, body, out, stackTrace, etc.
showExchangeId	Prints out the exchange ID.
showHeaders	Shows all headers for in the in message.
showBodyType	Shows the Java type for the body.
showBody	Shows the actual contents of the body.
showOut	Shows the out message.
showException	If there's an exception in the exchange, shows the exception. Doesn't show the full stacktrace.
showStackTrace	Prints the stacktrace from the exception.
multiline	Logs each part of the exchange and its details on separate lines for better readability.
maxChars	The maximum number of characters logged per line.

Further Reading

For more information on the log component, see <http://camel.apache.org/log.html>

JMS and ActiveMQ Component

The Camel JMS component allows you to write Camel routes with endpoints producing to or consuming from a JMS provider. Most commonly, JMS and associated routes are used for asynchronous inOnly style messaging. However, Camel-JMS understands request-reply or inOut style messaging and uses temporary queues to implement under the covers. Use the activemq-camel component for JMS endpoints that rely on an ActiveMQ provider as it provides some simple optimizations.

Consume from a queue

Note that you can specify "jms:queue:incomingOrders" but "queue" is default, so it can be left off:

```
from("jms:incomingOrders")
    .process(<some processing>).to("jms:inventoryCheck");
```

Consume from a topic

```
from("jms:topic:incomingOrders")
    .process(<some processing>).to("jms:inventoryCheck");
```

Set up pooled resources

JMS resources are often expensive to create. Connections, Sessions, Producers, Consumers should be cached where it makes sense, and setting up the appropriate caching starts with a pooled connection factory. If you're using ActiveMQ, you can use the org.apache.activemq.pool.PooledConnectionFactory. Alternatively for general purpose JMS connection pooling, you can use the Spring's CachingConnectionFactory. See the documentation for using the CachingConnectionFactory.

When using ActiveMQ::

```
<bean id="connectionFactory" class="org.apache.activemq.
ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
<bean id="jmsPooledConnectionFactory" class="org.apache.activemq.pool.
PooledConnectionFactory">
  <property name="connectionFactory" ref="connectionFactory">
</property>
</bean>
```

Request/Reply

By default, Camel will use a Message Exchange Pattern of InOnly when dealing with JMS. However, it will be changed to InOut if there is a JMSReplyTo header or if explicitly set:

```
from("direct:incoming").inOut().to("jms:outgoingOrders").process(process the
jms response here and continue your route)
```

In the previous example, the MEP was set to InOut which means before sending to the outgoingOrders queue, Camel will create a temporary destination, listen to it, and set the JMSReplyTo header to the temporary destination before it sends the JMS message. Some other consumer will need to listen to outgoingOrders queue and send a reply to the destination named in the JMSReplyTo header. The response received over the temporary destination will be used in further processing in the route. Camel will listen for a default 20s on the temporary destination before it times out. If you prefer to use named queues instead of temporary ones, you can set the replyTo configuration option:

```
from("direct:incoming").inOut().to("jms:outgoingOrders?replyTo=outgoingOrdersR
eply").process(process the jms response here and continue your route)
```

Transactions

To use transactions with Camel-JMS, you should set up a pooled connection factory, a transaction manager, and configure the Camel-JMS configuration in a JMSConfiguration object:

```
<!-- JmsConfiguration object -->
<bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration" >
  <property name="connectionFactory" ref="jmsPooledConnectionFactory" />
  </bean>
  <property name="transacted" value="true" />
  <property name="transactionManager" ref="jmsTransactionManager" />
  <property name="cacheLevelName" value="CACHE_CONSUMER" />
</bean>

<!-- Spring Transaction Manager -->
<bean id="jmsTransactionManager" class="org.springframework.jms.connection.
JmsTransactionManager">
  <property name="connectionFactory" ref="jmsPo
oledConnectionFactory" />
</bean>

<!-- Set up Pooled Connection Factory -->
<bean id="jmsPooledConnectionFactory" class="org.apache.activemq.pool.
PooledConnectionFactory">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>
  </property>
</bean>
```

Make sure to set the cache level to CACHE_CONSUMER so that your consumers, sessions, and connections are cached between messages.

Updated for Camel 2.10, you can now use "local" transactions with the Camel-JMS consumer (you don't have to specify an external transaction manager):

```
<bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration" >
  <property name="connectionFactory" ref="jmsPooledConnectionFacto
ry" />
  <property name="transacted" value="true" />
  <property name="cacheLevelName" value="CACHE_CONSUMER" />
</bean>
```

Common configuration options:

Option	Default	Description
concurrentConsumers	1	Specifies the number of consumers to create to listen on the destination.
disableReplyTo	false	Sets this endpoint to not do inOut messaging.
durableSubscriptionName	null	Explicitly sets the name of a durable subscription (per JMS, must be used with clientId).
clientId	null	JMS client id for this endpoint. Must be unique.

Option	Default	Description
replyTo	null	Specifies the name of a destination to be included as a JMSReplyTo header on outgoing messages.
selector	null	JMS selector to specify a predicate for what messages the broker should deliver to this endpoint consumer.
timeToLive	null	Time to live (in milliseconds) for the message as it travels through a broker (possible network of brokers).
transacted	false	Specifies whether to do send and receives in a transaction.
acknowledgementModeName	AUTO_ACKNOWLEDGE	JMS acknowledgement mode.
asyncConsumer	false	Whether to process a message from a destination asynchronously. By default this is false.
cacheLevelName	CACHE_AUTO	Caches consumers, sessions, and connections. For example, CACHE_AUTO, and CACHE_CONSUMER.

Further Reading

For more information on the jms component, see <http://camel.apache.org/jms.html>

CXF Component

When you want to expose or consume a SOAP or REST style web service for use in an integration route, sometimes a solution you're working on requires a web-service endpoint to kick off a larger process, or enrich data before calling another web service, or just act as a proxy to another service. Camel works nicely with CXF to allow you to configure a web-service endpoint that hooks into and leverages the power of Camel's routing and mediation engine.

You can use Camel-CXF as a consumer (from() clause) which internally starts up a Jetty server to publish the web service. It can also be used as a producer, which will send SOAP or REST requests. You can alternatively configure your endpoint to deploy into an existing servlet container instead of relying on the built-in Jetty server.

JAX-WS (SOAP)

Using the JAX-WS front end for CXF with Camel requires you to set up your contract (WSDL) or Java classes first and follow the correct wsdl2java or java annotations to describe your web service. Once you've done this, you can use Camel to hook the endpoint into a route.

Configuration

As a bean (easier to read in the DSL)

To configure the JAX-WS component, specify a <cxfEndpoint> with an ID attribute that can be later referenced by a Camel route:

```
<cxf:cxfEndpoint id="helloWorld"
  wsdlURL="wsdl/HelloWorld.wsdl"
  serviceClass="org.apache.helloworld.HelloWorld"
  address="http://localhost:9090/helloworld" >
</cxf:cxfEndpoint>
```

Note that for this configuration, you'll need this namespace in your spring application context:

```
http://camel.apache.org/schema/cxf
http://camel.apache.org/schema/cxf/camel-cxf.xsd
```

Then, in your Camel route, you can refer to the CXF endpoint with:

```
from("cxf:bean:helloWorld").log("This is what was said: ${body}").
transform(constant("Hello to you good sir!"));
```

With the URI (more convenient, not as easy to read in the DSL)

You can leave out the bean config in the registry (spring XML) altogether and configure options directly on the endpoint:

```
from("cxf://http://localhost:9090/helloworld?serviceClass=org.apache.helloworld.HelloWorld&wsdlURL=wsdl/HelloWorld.wsdl&dataFormat=MESSAGE")
```

In this example, org.apache.helloworld.HelloWorld is the SEI (Service Endpoint Interface). Note, there is no *Impl class that implements the SEI, as camel takes care of that.

DataFormat

Camel allows you to work with the SOAP message in three different formats:

- **POJO (default)** – a list of Java objects that represent the parameters to the service being called
- **MESSAGE** – Raw message, no SOAP processing
- **PAYLOAD** – just deal with the SOAP body of the message

JAX-RS (REST)

Just like with JAX-WS, you'll need to set up your Java classes with the correct JAX-RS annotations and use Camel to expose the REST endpoint

Configuration

As a bean

Note that for this configuration, you'll need this namespace in your spring application context:

```
http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
<cxf:rsServer id="rsServer" address="http://localhost:9090/route"
serviceClass="com.fusesource.samples.CustomerServiceResource">
```

Then, within your route (just like the bean for the JAX-WS web service) you can refer to the bean like this:

```
from("cxfrs:bean:rsServer")
```

With the URI

```
from("cxfrs://http://localhost:9090/route?resourceClasses=com.fusesource.samples.CustomerServiceResource")
```

Further Reading

For more information on the CXF component, see <http://camel.apache.org/cxf.html>

For more about Camel, CXF, and JAX-RS see: <http://www.christianposta.com/blog/?p=229>

File Component

When reading or writing files (CSV, XML, fixed-format, for example) to or from the file system, use the Camel File component. File-based integrations are fairly common with legacy applications but can be tedious to interact with. With a file consumer endpoint, you can consume files from the file system and work with them as exchanges in your camel route. Conversely, you can also persist the content of your exchanges to disk.

Read Files

When used in a "from" DSL clause, the file component will be used in "consumer" mode or "read".

```
from("file:/location/of/files?configs").log("${body}")
```

Common Configuration Options for Reading Files

Option	Default	Description
delay	500ms	Sets how long (in milliseconds) to wait before polling the file system for the next file.

Option	Default	Description
delete	false	Sets whether to delete the file after it has been processed.
doneFileName	null	Names of files that must exist before Camel starts processing files from the folder.
exclude	null	Regex for file patterns to ignore.
filter	null	A custom filter class used to filter out files/folders you don't want to process (specify as a bean name: #beanName).
idempotent	false	Skips already-processed files following the Idempotent consumer pattern.
include	null	Regex for file patterns to include.
move	.camel	Sets the default place to move files after they've been processed.
noop	false	If true, the file is not moved or deleted after it has been processed.
readLock	markerFile	Camel will only process files that it can get a read-lock for. Specifies a strategy for how to determine whether it has read-lock access.
readLock Timeout	10000ms	Sets the timeout i(n milliseconds) for how long to wait to acquire a read-lock.
recursive	false	Recurs through sub-directories as well.

Things to Watch Out For

- Locking of files that are being processed (on reads) until route is complete (by default).
- Will ignore files that start with "."
- By default, will move processed files to ".camel" directory.
- Moving/deleting files will happen after routing.

Write Files

```
from(<endpoint>).to("file:/location/of/files?fileName=<filename>")
```

Common Configuration Options for Writing Files

Option	Description
doneFileName	Name of file that must exist before Camel starts processing files from the folder.
fileExist	What to do if a file exists: override, append, ignore, or fail.
fileName	Name of file to be written.
tempFileName	A temporary name given to the file as it's being written. Will change to real name when writing is finished.

Big Files

Sometimes you need to process files that are too large to fit into memory, or would be too large to process efficiently if loaded into memory. A common approach to dealing with such files using the Camel-file component is to stream them and process them in chunks. Generally, the files are structured in such a way that it makes sense to process them in chunks and we can leverage the power of camel's EIP processors to "split" a file into those chunks. Here's an example route:

```
from("file:/location/of/files").split(body()).tokenize("\n").streaming().log("${body}").end()
```

This route splits on newlines in the file, streaming and processing one line at a time.

For more information on splitting large files, see <http://www.davsclaus.com/2011/11/splitting-big-xml-files-with-apache.html>

Headers

Headers are automatically inserted (consumer) for use in your routes or can be set in your route and used by the file component (producer). Here are some of the headers that are set automatically by the file consumer endpoint:

- **CamelFileName** - Name of the file consumed as a relative location with the offset from the directory configured in the endpoint.
- **CamelFileNameOnly** - File name with no location offset.
- **CamelFileAbsolute** - A Boolean that specifies whether the path is absolute or not.
- **CamelFileAbsolutePath** - Will be the absolute path if the **CamelFileAbsolute** header is true.
- **CamelFilePath** - Starting directory + relative filename.
- **CamelFileRelativePath** - Just the relative path.
- **CamelFileParent** - Just the parent path.
- **CamelFileLength** - Length of the file.
- **CamelFileLastModified** - Date of the last modified timestamp.

Headers that, if set, will be used by the file producer endpoint:

- **CamelFileName** - Name of the file to write to the output directory (usually an expression).
- **CamelFileNameProduced** - The actual filename produced (absolute file path).

Further Reading

For more information on the file component, see <http://camel.apache.org/file2.html>

SEDA and Direct Component

You can still use messaging as a means of communicating with or between your routes without having to use an external messaging broker. Camel allows you to do in-memory messaging via synchronous or asynchronous queues. Use the "direct" component for synchronous processing and use the "seda" for asynchronous, "staged," event-driven processing.

Direct

Use the direct component to break up routes using synchronous messaging:

```
from("direct:channelName").process(<process something>).
to("direct:outgoingChannel");
from("direct:outgoingChannel").transform(<transform something>).
to("jms:outgoingOrders")
```

SEDA

SEDA endpoints and their associated routes will be run in separate threads and process exchanges asynchronously. Although the pattern of usage is similar to the "direct" component, the semantics are quite different.

```
from("<any component>").choice().when(header("accountType"),
endsWith("Consumer")).to("seda:consumerAccounts")
    .when(header("accountType").endsWith("Business")).
to("seda:businessAccounts")
from("seda:consumerAccounts").process(<process logic>)
from("seda:businessAccounts").process(<process logic>)
```

You can set up the SEDA endpoint to use multiple threads to do its processing by adding the `concurrentConsumers` configuration:

```
from("seda:consumerAccounts?concurrentConsumers=20").process(<process logic>)
```

Keep in mind that using SEDA (or any asynchronous endpoint) will behave differently in a transaction, i.e., the consumers of the SEDA queue will not participate in the transaction as they are in different threads.

Common configuration options

The "direct" component does not have any configuration options.

SEDA commonly used options:

Option	Default	Description
size	Unbounded	Max capacity of the in-memory queue.
concurrentConsumers	1	The number of concurrent threads that can process exchanges.
multipleConsumers	false	Determines whether multiple consumers are allowed.
blockWhenFull	false	Blocks a thread that tries to write to a full SEDA queue instead of throwing an exception.

Further Reading

For more information on the direct and SEDA component, see <http://camel.apache.org/direct.html> and <http://camel.apache.org/seda.html> respectively

Mock Component

Testing your routes is an important aspect in integration development and Camel makes it easier with the mock component. The component can be used in your JUnit or TestNG tests. You can declare a set of expectations on a mock such as how many messages were processed, or that certain headers must be present at an endpoint, and then run your route. At the completion of the route, you can verify that the intended expectations were met and fail the test if they were not.

Mocks

You start by obtaining the mock endpoint from the route:

```
MockEndpoint mock = context.getEndpoint("mock:outgoing",
MockEndpoint.class);
```

Next, you declare expectations. Methods that declare expectations start with "expect", for example:

```
mock.expectedMessageCount( 1 )
```

Then you run your route. Finally, you assert that the declared expectations were met:

```
mock.assertIsSatisfied()
```

Expectation Methods:

- `expectedMessageCount(int)`
- `expectedMinimumMessageCount(int)`
- `expectedBodiesReceived()`
- `expectedHeadersReceived()`
- `expectsAscending(Expression)`
- `expectsDescending(Expression)`
- `expectsNoDuplicate(Expression)`

AssertionMethods:

- `assertIsSatisfied()`
- `assertIsNotSatisfied()`

Mocking existing endpoints

Sometimes, you'll have routes with live endpoints that you cannot change (or don't want to change) for testing. For such routes, you can insert mocks where the live mocks are to do some testing.

```
RouteDefinition route = context.getRouteDefinition("advice");
route.adviceWith(context, new AdviceWithRouteBuilder() {
    @Override
    public void configure() throws Exception {
        mockEndpoints(<pattern>);
    }
});
```

<pattern> allows you to specify which endpoints to mock. For example, to mock out only the JMS components, you would do `mockEndpoints("jms*")`. To mock all endpoints, leave off the pattern completely.

Note that inserting mocks at the location of the endpoints does not replace the endpoints, i.e., the live endpoints will still exist. Updated for Camel 2.10, you can now skip sending the exchange to the live endpoint:

```
RouteDefinition route = context.getRouteDefinition("advice");
route.adviceWith(context, new AdviceWithRouteBuilder() {
    @Override
    public void configure() throws Exception {
        mockEndpointsAndSkip(<pattern>);
    }
});
```

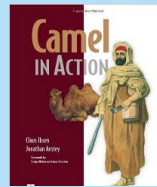
Further Reading

For more information on the mock component, see <http://camel.apache.org/mock.html>

ABOUT THE AUTHOR

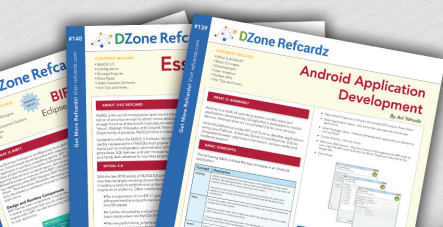
Based in Phoenix, AZ, I'm a Senior Consultant and Architect at Red Hat and I specialize in messaging-based enterprise integrations. I'm passionate about software development, love solving tough technical problems, and enjoy learning new languages and programming paradigms. Favorite languages include Python and Scala, but spend a lot of time writing

Java. I am a committer on Apache ActiveMQ and Apache Apollo projects and frequently blog at <http://www.christianposta.com/blog> as well as tweet about interesting technology @christianposta.

RECOMMENDED BOOK

[Buy Here.](#)

Camel in Action is a Camel tutorial full of small examples showing how to work with the integration patterns. It starts with core concepts like sending, receiving, routing, and transforming data. It then shows you the entire lifecycle and goes in depth on how to test, deal with errors, scale, deploy, and even monitor your app— details you can find only in the Camel code itself.



Browse our collection of over 150 Free Cheat Sheets

Free PDF

Upcoming Refcardz

Mongo DB
JSON
Cypher
Object-Oriented js



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, cheat sheets, blogs, feature articles, source code and more. **'DZone is a developer's dream,'** says PC Magazine.

Copyright © 2013 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.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513
888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-68-4
ISBN-10: 1-936502-68-2



\$7.95

Version 1.0