Appendix B. Classic Spring AOP Usage

In this appendix we discuss the lower-level Spring AOP APIs and the AOP support used in Spring 1.2 applications. For new applications, we recommend the use of the Spring 2.0 AOP support described in the AOP chapter, but when working with existing applications, or when reading books and articles, you may come across Spring 1.2 style examples. Spring 2.0 is fully backwards compatible with Spring 1.2 and everything described in this appendix is fully supported in Spring 2.0.

B.1 Pointcut API in Spring

Let's look at how Spring handles the crucial pointcut concept.

B.1.1 Concepts

Spring's pointcut model enables pointcut reuse independent of advice types. It's possible to target different advice using the same pointcut.

The `org.springframework.aop.Pointcut` interface is the central interface, used to target advices to particular classes and methods. The complete interface is shown below:

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();
```

```
}
```

Splitting the `Pointcut` interface into two parts allows reuse of class and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The `ClassFilter` interface is used to restrict the pointcut to a given set of target classes. If the `matches()` method always returns true, all target classes will be matched:

```
public interface ClassFilter {

    boolean matches(Class clazz);
}
```

The `MethodMatcher` interface is normally more important. The complete interface is shown below:

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

    boolean matches(Method m, Class targetClass, Object[] args);
}
```

The `matches(Method, Class)` method is used to test whether this pointcut will ever match a given method on a target class. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument matches method returns true for a given method, and the `isRuntime()` method for the MethodMatcher returns true, the 3-argument matches method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most MethodMatchers are static, meaning that their `isRuntime()` method returns false. In this case, the 3-argument matches method will never be invoked.

If possible, try to make pointcuts static, allowing the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

B.1.2 Operations on pointcuts

Spring supports operations on pointcuts: notably, *union* and *intersection*.

- Union means the methods that either pointcut matches.
- Intersection means the methods that both pointcuts match.
- Union is usually more useful.
- Pointcuts can be composed using the static methods in the *org.springframework.aop.support.Pointcuts* class, or using the *ComposablePointcut* class in the same package. However, using AspectJ pointcut expressions is usually a simpler approach.

B.1.3 AspectJ expression pointcuts

Since 2.0, the most important type of pointcut used by Spring is `org.springframework.aop.aspectj.AspectJExpressionPointcut`. This is a pointcut that uses an AspectJ supplied library to parse an AspectJ pointcut expression string.

See the previous chapter for a discussion of supported AspectJ pointcut primitives.

B.1.4 Convenience pointcut implementations

Spring provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient - *and best* - for most usages. It's possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let's consider some static pointcut implementations included with Spring.

Regular expression pointcuts

One obvious way to specify static pointcuts is regular expressions. Several AOP frameworks besides Spring make this possible. `org.springframework.aop.support.Perl5RegexpMethodPointcut` is a generic regular expression pointcut, using Perl 5 regular expression syntax. The `Perl5RegexpMethodPointcut` class depends on Jakarta ORO for regular expression matching. Spring also provides the `JdkRegexpMethodPointcut` class that uses the regular expression support in JDK 1.4+.

Using the `Perl5RegexpMethodPointcut` class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true. (So the result is effectively the union of these pointcuts.)

The usage is shown below:

```
<bean id="settersAndAbsquatulatePointcut"
    class="org.springframework.aop.support.Perl5RegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

Spring provides a convenience class, `RegexpMethodPointcutAdvisor`, that allows us to also reference an Advice (remember that an Advice can be an interceptor, before advice, throws advice etc.). Behind the scenes, Spring will use a `JdkRegexpMethodPointcut`. Using `RegexpMethodPointcutAdvisor` simplifies wiring, as the one bean encapsulates both pointcut and advice, as shown below:

```
<bean id="settersAndAbsquatulateAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

RegexpMethodPointcutAdvisor can be used with any Advice type.

Attribute-driven pointcuts

An important type of static pointcut is a *metadata-driven* pointcut. This uses the values of metadata attributes: typically, source-level metadata.

Dynamic pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method *arguments*, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the `control flow` pointcut.

Control flow pointcuts

Spring control flow pointcuts are conceptually similar to AspectJ *cflow* pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below a join point matched by another pointcut.) A control flow pointcut matches the

current call stack. For example, it might fire if the join point was invoked by a method in the `com.mycompany.web` package, or by the `SomeCaller` class. Control flow pointcuts are specified using the `org.springframework.aop.support.ControlFlowPointcut` class.

Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts. In Java 1.4, the cost is about 5 times that of other dynamic pointcuts.

B.1.5 Pointcut superclasses

Spring provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are most useful, you'll probably subclass StaticMethodMatcherPointcut, as shown below. This requires implementing just one abstract method (although it's possible to override other methods to customize behavior):

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }
}
```

There are also superclasses for dynamic pointcuts.

You can use custom pointcuts with any advice type in Spring 1.0 RC2 and above.

B.1.6 Custom pointcuts

Because pointcuts in Spring AOP are Java classes, rather than language features (as in AspectJ) it's possible to declare custom pointcuts, whether static or dynamic. Custom pointcuts in Spring can be arbitrarily complex. However, using the AspectJ pointcut

expression language is recommended if possible.

Later versions of Spring may offer support for "semantic pointcuts" as offered by JAC: for example, "all methods that change instance variables in the target object."

B.2 Advice API in Spring

Let's now look at how Spring AOP handles advice.

B.2.1 Advice lifecycles

Each advice is a Spring bean. An advice instance can be shared across all advised objects, or unique to each advised object. This corresponds to *per-class* or *per-instance* advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It's possible to use a mix of shared and per-instance advice in the same AOP proxy.

B.2.2 Advice types in Spring

Spring provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

Interception around advice

The most fundamental advice type in Spring is *interception around advice*.

Spring is compliant with the AOP Alliance interface for around advice using method interception. MethodInterceptors implementing around advice should implement the following interface:

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

The `MethodInvocation` argument to the `invoke()` method exposes the method being invoked; the target join point; the AOP proxy; and the arguments to the method. The `invoke()` method should return the invocation's result: the return value of the join point.

A simple `MethodInterceptor` implementation looks as follows:

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

Note the call to the MethodInvocation's `proceed()` method. This proceeds down the interceptor chain towards the join point. Most interceptors will invoke this method, and return its return value. However, a MethodInterceptor, like any around advice, can return a different value or throw an exception rather than invoke the proceed method. However, you don't want to do this without good reason!

MethodInterceptors offer interoperability with other AOP Alliance-compliant AOP implementations. The other advice

> types discussed in the remainder of this section implement common AOP concepts, but in a Spring-specific way. While there is an advantage in using the most specific advice type, stick with MethodInterceptor around advice if you are likely to want to run the aspect in another AOP framework. Note that pointcuts are not currently interoperable between frameworks, and the AOP Alliance does not currently define pointcut interfaces.

Before advice

A simpler advice type is a **before advice**. This does not need a `MethodInvocation` object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the `proceed()` method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The `MethodBeforeAdvice` interface is shown below. (Spring's API design would allow for field before advice, although the usual objects apply to field interception and it's unlikely that Spring will ever implement it).

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

Note the return type is `void`. Before advice can insert custom behavior before the join point executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring, which counts all method invocations:

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {

    private int count;
```

```
    public void before(Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

Before advice can be used with any pointcut.

Throws advice

Throws advice is invoked after the return of the join point if the join point threw an exception. Spring offers typed throws advice. Note that this means that the `org.springframework.aop.ThrowsAdvice` interface does not contain any methods: It is a tag interface identifying that the given object implements one or more typed throws advice methods. These should be in the form of:

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

Only the last argument is required. The method signatures may have either one or four arguments, depending on whether the advice method is interested in the method and arguments. The following classes are examples of throws advice.

The advice below is invoked if a `RemoteException` is thrown (including subclasses):

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

The following advice is invoked if a `ServletException` is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
    }
}
```

The final example illustrates how these two methods could be used in a single class, which handles both `RemoteException` and `ServletException`. Any number of throws advice methods can be combined in a single class.

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
    }
}
```

Note: If a throws-advice method throws an exception itself, it will override the original exception (i.e. change the exception thrown to the user). The overriding exception will typically be a RuntimeException; this is compatible with any method signature. However, if a throws-advice method throws a checked exception, it will have to match the declared exceptions of the target method and is hence to some degree coupled to specific target method signatures. *Do not throw an undeclared checked exception that is incompatible with the target method's signature!*

Throws advice can be used with any pointcut.

After Returning advice

An after returning advice in Spring must implement the *org.springframework.aop.AfterReturningAdvice* interface, shown below:

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable;
}
```

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {

    private int count;

    public void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

This advice doesn't change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.

After returning advice can be used with any pointcut.

Introduction advice

Spring treats introduction advice as a special kind of interception advice.

Introduction requires an `IntroductionAdvisor`, and an `IntroductionInterceptor`, implementing the following interface:

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

The `invoke()` method inherited from the AOP Alliance `MethodInterceptor` interface must implement the introduction: that is, if the invoked method is on an introduced interface, the introduction interceptor is responsible for handling the method call - it cannot invoke `proceed()`.

Introduction advice cannot be used with any pointcut, as it applies only at class, rather than method, level. You can only use introduction advice with the `IntroductionAdvisor`, which has the following methods:

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

    Class[] getInterfaces();
}
```

There is no `MethodMatcher`, and hence no `Pointcut`, associated with introduction advice. Only class filtering is logical.

The `getInterfaces()` method returns the interfaces introduced by this advisor.

The `validateInterfaces()` method is used internally to see whether or not the introduced interfaces can be implemented by the configured `IntroductionInterceptor`.

Let's look at a simple example from the Spring test suite. Let's suppose we want to introduce the following interface to one or more objects:

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

This illustrates a **mixin**. We want to be able to cast advised objects to Lockable, whatever their type, and call lock and unlock methods. If we call the lock() method, we want all setter methods to throw a `LockedException`. Thus we can add an aspect that provides the ability to make objects immutable, without them having any knowledge of it: a good example of AOP.

Firstly, we'll need an `IntroductionInterceptor` that does the heavy lifting. In this case, we extend the `org.springframework.aop.support.DelegatingIntroductionInterceptor` convenience class. We could implement IntroductionInterceptor directly, but using `DelegatingIntroductionInterceptor` is best for most cases.

The `DelegatingIntroductionInterceptor` is designed to delegate an introduction to an actual implementation of the introduced interface(s), concealing the use of interception to do so. The delegate can be set to any object using a constructor argument; the default delegate (when the no-arg constructor is used) is this. Thus in the example below, the delegate is the `LockMixin` subclass of `DelegatingIntroductionInterceptor`. Given a delegate (by default itself), a `DelegatingIntroductionInterceptor` instance looks for all interfaces implemented by the delegate (other than IntroductionInterceptor), and will support introductions against any of them. It's possible for subclasses such as `LockMixin` to call the `suppressInterface(Class intf)` method to suppress interfaces that should not be exposed. However, no matter how many interfaces an `IntroductionInterceptor` is prepared to support, the `IntroductionAdvisor` used will control which interfaces are actually exposed. An introduced interface will conceal any implementation of the same interface by the target.

Thus LockMixin subclasses `DelegatingIntroductionInterceptor` and implements Lockable itself. The superclass automatically picks up that Lockable can be supported for introduction, so we don't need to specify that. We could introduce any

number of interfaces in this way.

Note the use of the `locked` instance variable. This effectively adds additional state to that held in the target object.

```
public class LockMixin extends DelegatingIntroductionInterceptor
    implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0)
            throw new LockedException();
        return super.invoke(invocation);
    }

}
```

Often it isn't necessary to override the `invoke()` method: the `DelegatingIntroductionInterceptor` implementation - which calls the delegate method if the method is introduced, otherwise proceeds towards the join point - is usually sufficient. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

The introduction advisor required is simple. All it needs to do is hold a distinct `LockMixin` instance, and specify the introduced interfaces - in this case, just `Lockable`. A more complex example might take a reference to the introduction interceptor (which would be defined as a prototype): in this case, there's no configuration relevant for a `LockMixin`, so we simply create it using `new`.

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

We can apply this advisor very simply: it requires no configuration. (However, it *is* necessary: It's impossible to use an `IntroductionInterceptor` without an *IntroductionAdvisor*.) As usual with introductions, the advisor must be per-instance, as it is stateful. We need a different instance of `LockMixinAdvisor`, and hence `LockMixin`, for each advised object. The advisor comprises part of the advised object's state.

We can apply this advisor programmatically, using the `Advised.addAdvisor()` method, or (the recommended way) in XML configuration, like any other advisor. All proxy creation choices discussed below, including "auto proxy creators," correctly handle introductions and stateful mixins.

B.3 Advisor API in Spring

In Spring, an Advisor is an aspect that contains just a single advice object associated with a pointcut expression.

Apart from the special case of introductions, any advisor can be used with any advice. `org.springframework.aop.support.DefaultPointcutAdvisor` is the most commonly used advisor class. For example, it can be used with a `MethodInterceptor`, `BeforeAdvice` or `ThrowsAdvice`.

It is possible to mix advisor and advice types in Spring in the same AOP proxy. For example, you could use a interception around advice, throws advice and before advice in one proxy configuration: Spring will automatically create the necessary interceptor

chain.

B.4 Using the ProxyFactoryBean to create AOP proxies

If you're using the Spring IoC container (an ApplicationContext or BeanFactory) for your business objects - and you should be! - you will want to use one of Spring's AOP FactoryBeans. (Remember that a factory bean introduces a layer of indirection, enabling it to create objects of a different type.)

The Spring 2.0 AOP support also uses factory beans under the covers.

The basic way to create an AOP proxy in Spring is to use the *org.springframework.aop.framework.ProxyFactoryBean*. This gives complete control over the pointcuts and advice that will apply, and their ordering. However, there are simpler options that are preferable if you don't need such control.

B.4.1 Basics

The `ProxyFactoryBean`, like other Spring `FactoryBean` implementations, introduces a level of indirection. If you define a `ProxyFactoryBean` with name `foo`, what objects referencing `foo` see is not the `ProxyFactoryBean` instance itself, but an object created by the `ProxyFactoryBean`'s implementation of the `getObject()` method. This method will create an AOP proxy wrapping a target object.

One of the most important benefits of using a `ProxyFactoryBean` or another IoC-aware class to create AOP proxies, is that it means that advices and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

B.4.2 JavaBean properties

In common with most `FactoryBean` implementations provided with Spring, the `ProxyFactoryBean` class is itself a JavaBean. Its properties are used to:

- Specify the target you want to proxy.
- Specify whether to use CGLIB (see below and also Section 10.5.3, “JDK- and CGLIB-based proxies”).

Some key properties are inherited from `org.springframework.aop.framework.ProxyConfig` (the superclass for all AOP proxy factories in Spring). These key properties include:

- `proxyTargetClass`: `true` if the target class is to be proxied, rather than the target class' interfaces. If this property value is set to `true`, then CGLIB proxies will be created (but see also below Section 10.5.3, “JDK- and CGLIB-based proxies”).
- `optimize`: controls whether or not aggressive optimizations are applied to proxies *created via CGLIB*. One should not blithely use this setting unless one fully understands how the relevant AOP proxy handles optimization. This is currently used only for CGLIB proxies; it has no effect with JDK dynamic proxies.
- `frozen`: if a proxy configuration is `frozen`, then changes to the configuration are no longer allowed. This is useful both as a slight optimization and for those cases when you don't want callers to be able to manipulate the proxy (via the `Advised` interface) after the proxy has been created. The default value of this property is `false`, so changes such as adding additional advice are allowed.
- `exposeProxy`: determines whether or not the current proxy should be exposed in a `ThreadLocal` so that it can be accessed by the target. If a target needs to obtain the proxy and the `exposeProxy` property is set to `true`, the target can use the `AopContext.currentProxy()` method.
- `aopProxyFactory`: the implementation of `AopProxyFactory` to use. Offers a way of customizing whether to use dynamic proxies, CGLIB or any other proxy strategy. The default implementation will choose dynamic proxies or CGLIB appropriately. There should be no need to use this property; it is intended to allow the addition of new proxy types in Spring 1.1.

Other properties specific to `ProxyFactoryBean` include:

- `proxyInterfaces`: array of String interface names. If this isn't supplied, a CGLIB proxy for the target class will be used (but see also below Section 10.5.3, “JDK- and CGLIB-based proxies”).
- `interceptorNames`: String array of `Advisor`, interceptor or other advice names to apply. Ordering is significant, on a first come-first served basis. That is to say that the first interceptor in the list will be the first to be able to intercept the invocation.

The names are bean names in the current factory, including bean names from ancestor factories. You can't mention bean references here since doing so would result in the `ProxyFactoryBean` ignoring the singleton setting of the advice.
You can append an interceptor name with an asterisk (`*`). This will result in the application of all advisor beans with names starting with the part before the asterisk to be applied. An example of using this feature can be found in Section 10.5.6, “Using 'global' advisors”.

- singleton: whether or not the factory should return a single object, no matter how often the `getObject()` method is called. Several `FactoryBean` implementations offer such a method. The default value is `true`. If you want to use stateful advice - for example, for stateful mixins - use prototype advices along with a singleton value of `false`.

B.4.3 JDK- and CGLIB-based proxies

This section serves as the definitive documentation on how the `ProxyFactoryBean` chooses to create one of either a JDK- and CGLIB-based proxy for a particular target object (that is to be proxied).

The behavior of the `ProxyFactoryBean` with regard to creating JDK- or CGLIB-based proxies changed between versions 1.2.x and 2.0 of Spring. The `ProxyFactoryBean` now exhibits similar semantics with regard to auto-detecting interfaces as those of the `TransactionProxyFactoryBean` class.

If the class of a target object that is to be proxied (hereafter simply referred to as the target class) doesn't implement any interfaces, then a CGLIB-based proxy will be created. This is the easiest scenario, because JDK proxies are interface based, and no interfaces means JDK proxying isn't even possible. One simply plugs in the target bean, and specifies the list of interceptors via the `interceptorNames` property. Note that a CGLIB-based proxy will be created even if the `proxyTargetClass` property of the `ProxyFactoryBean` has been set to `false`. (Obviously this makes no sense, and is best removed from the bean definition because it is at best redundant, and at worst confusing.)

If the target class implements one (or more) interfaces, then the type of proxy that is created depends on the configuration of the `ProxyFactoryBean`.

If the `proxyTargetClass` property of the `ProxyFactoryBean` has been set to `true`, then a CGLIB-based proxy will be created. This makes sense, and is in keeping with the principle of least surprise. Even if the `proxyInterfaces` property of the `ProxyFactoryBean` has been set to one or more fully qualified interface names, the fact that the `proxyTargetClass` property is set to `true` *will* cause CGLIB-based proxying to be in effect.

If the `proxyInterfaces` property of the `ProxyFactoryBean` has been set to one or more fully qualified interface names, then a JDK-based proxy will be created. The created proxy will implement all of the interfaces that were specified in the `proxyInterfaces` property; if the target class happens to implement a whole lot more interfaces than those specified in the `proxyInterfaces` property, that is all well and good but those additional interfaces will not be implemented by the returned proxy.

If the `proxyInterfaces` property of the `ProxyFactoryBean` has *not* been set, but the target class *does implement one (or more)* interfaces, then the `ProxyFactoryBean` will auto-detect the fact that the target class does actually implement at least one interface, and a JDK-based proxy will be created. The interfaces that are actually proxied will be *all* of the interfaces that the target class implements; in effect, this is the same as simply supplying a list of each and every interface that the target class implements to the `proxyInterfaces` property. However, it is significantly less work, and less prone to typos.

B.4.4 Proxying interfaces

Let's look at a simple example of `ProxyFactoryBean` in action. This example involves:

- A *target bean* that will be proxied. This is the "personTarget" bean definition in the example below.
- An Advisor and an Interceptor used to provide advice.
- An AOP proxy bean definition specifying the target object (the personTarget bean) and the interfaces to proxy, along with the advices to apply.

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name"><value>Tony</value></property>
    <property name="age"><value>51</value></property>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
```

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty"><value>Custom string property value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>

    <property name="target"><ref local="personTarget"/></property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

Note that the `interceptorNames` property takes a list of String: the bean names of the interceptor or advisors in the current factory. Advisors, interceptors, before, after returning and throws advice objects can be used. The ordering of advisors is significant.

You might be wondering why the list doesn't hold bean references. The reason for this is that if the ProxyFactoryBean's singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it's necessary to be able to obtain an instance of the prototype from the factory; holding a reference isn't sufficient.

The "person" bean definition above can be used in place of a Person implementation, as follows:

```
Person person = (Person) factory.getBean("person");
```

Other beans in the same IoC context can express a strongly typed dependency on it, as with an ordinary Java object:

```
<bean id="personUser" class="com.mycompany.PersonUser">
  <property name="person"><ref local="person" /></property>
</bean>
```

The `PersonUser` class in this example would expose a property of type Person. As far as it's concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its class would be a dynamic proxy class. It would be possible to cast it to the `Advised` interface (discussed below).

It's possible to conceal the distinction between target and proxy using an anonymous *inner bean*, as follows. Only the `ProxyFactoryBean` definition is different; the advice is included only for completeness:

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
  <property name="someProperty"><value>Custom string property value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>
  <!-- Use inner bean, not local reference to target -->
  <property name="target">
    <bean class="com.mycompany.PersonImpl">
      <property name="name"><value>Tony</value></property>
      <property name="age"><value>51</value></property>
    </bean>
  </property>
  <property name="interceptorNames">
    <list>
      <value>myAdvisor</value>
      <value>debugInterceptor</value>
    </list>
  </property>
```

```
</bean>
```

This has the advantage that there's only one object of type `Person`: useful if we want to prevent users of the application context from obtaining a reference to the un-advised object, or need to avoid any ambiguity with Spring IoC *autowiring*. There's also arguably an advantage in that the ProxyFactoryBean definition is self-contained. However, there are times when being able to obtain the un-advised target from the factory might actually be an *advantage*: for example, in certain test scenarios.

B.4.5 Proxying classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our example above, there was no `Person` interface: we needed to advise a class called `Person` that didn't implement any business interface. In this case, you can configure Spring to use CGLIB proxying, rather than dynamic proxies. Simply set the `proxyTargetClass` property on the ProxyFactoryBean above to true. While it's best to program to interfaces, rather than classes, the ability to advise classes that don't implement interfaces can be useful when working with legacy code. (In general, Spring isn't prescriptive. While it makes it easy to apply good practices, it avoids forcing a particular approach.)

If you want to, you can force the use of CGLIB in any case, even if you do have interfaces.

CGLIB proxying works by generating a subclass of the target class at runtime. Spring configures this generated subclass to delegate method calls to the original target: the subclass is used to implement the *Decorator* pattern, weaving in the advice.

CGLIB proxying should generally be transparent to users. However, there are some issues to consider:

- `Final` methods can't be advised, as they can't be overridden.
- As of Spring 3.2 it is no longer required to add CGLIB to your project classpath. CGLIB classes have been repackaged under org.springframework and included directly in the spring-core JAR. This is both for user convenience as well as to avoid potential conflicts with other projects that have dependence on a differing version of CGLIB.

There's little performance difference between CGLIB proxying and dynamic proxies. As of Spring 1.0, dynamic proxies are slightly faster. However, this may change in the future. Performance should not be a decisive consideration in this case.

B.4.6 Using 'global' advisors

By appending an asterisk to an interceptor name, all advisors with bean names matching the part before the asterisk, will be added to the advisor chain. This can come in handy if you need to add a standard set of 'global' advisors:

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="service"/>
  <property name="interceptorNames">
    <list>
      <value>global*</value>
    </list>
  </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

B.5 Concise proxy definitions

Especially when defining transactional proxies, you may end up with many similar proxy definitions. The use of parent and child bean definitions, along with inner bean definitions, can result in much cleaner and more concise proxy definitions.

First a parent, *template*, bean definition is created for the proxy:

```
<bean id="txProxyTemplate" abstract="true"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
```

```
    </props>
  </property>
</bean>
```

This will never be instantiated itself, so may actually be incomplete. Then each proxy which needs to be created is just a child bean definition, which wraps the target of the proxy as an inner bean definition, since the target will never be used on its own anyway.

```
<bean id="myService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MyServiceImpl">
    </bean>
  </property>
</bean>
```

It is of course possible to override properties from the parent template, such as in this case, the transaction propagation settings:

```
<bean id="mySpecialService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MySpecialServiceImpl">
    </bean>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="store*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Note that in the example above, we have explicitly marked the parent bean definition as *abstract* by using the *abstract* attribute, as described previously, so that it may not actually ever be instantiated. Application contexts (but not simple bean factories) will by default pre-instantiate all singletons. It is therefore important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually try to pre-instantiate it.

B.6 Creating AOP proxies programmatically with the ProxyFactory

It's easy to create AOP proxies programmatically using Spring. This enables you to use Spring AOP without dependency on Spring IoC.

The following listing shows creation of a proxy for a target object, with one interceptor and one advisor. The interfaces implemented by the target object will automatically be proxied:

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addInterceptor(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

The first step is to construct an object of type `org.springframework.aop.framework.ProxyFactory`. You can create this with a target object, as in the above example, or specify the interfaces to be proxied in an alternate constructor.

You can add interceptors or advisors, and manipulate them for the life of the ProxyFactory. If you add an IntroductionInterceptionAroundAdvisor you can cause the proxy to implement additional interfaces.

There are also convenience methods on ProxyFactory (inherited from `AdvisedSupport`) which allow you to add other advice types such as before and throws advice. AdvisedSupport is the superclass of both ProxyFactory and ProxyFactoryBean.

Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from Java code with AOP, as in general.

B.7 Manipulating advised objects

However you create AOP proxies, you can manipulate them using the `org.springframework.aop.framework.Advised` interface. Any AOP proxy can be cast to this interface, whichever other interfaces it implements. This interface includes the following methods:

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice)
        throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

The `getAdvisors()` method will return an Advisor for every advisor, interceptor or other advice type that has been added to the factory. If you added an Advisor, the returned advisor at this index will be the object that you added. If you added an interceptor or other advice type, Spring will have wrapped this in an advisor with a pointcut that always returns true. Thus if you added a `MethodInterceptor`, the advisor returned for this index will be an `DefaultPointcutAdvisor` returning your `MethodInterceptor` and a pointcut that matches all classes and methods.

The `addAdvisor()` methods can be used to add any Advisor. Usually the advisor holding pointcut and advice will be the generic `DefaultPointcutAdvisor`, which can be used with any advice or pointcut (but not for introductions).

By default, it's possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it's impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

A simple example of casting an AOP proxy to the `Advised` interface and examining and manipulating its advice:

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors",
     oldAdvisorCount + 2, advised.getAdvisors().length);
```

It's questionable whether it's advisable (no pun intended) to modify advice on a business object in production, although there are no doubt legitimate usage cases. However, it can be very useful in development: for example, in tests. I have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation I want to test. (For example, the advice can get inside a transaction created for that method: for example, to run SQL to check that a database was correctly updated, before marking the transaction for roll back.)

Depending on how you created the proxy, you can usually set a `frozen` flag, in which case the `Advised` `isFrozen()` method will return true, and any attempts to modify advice through addition or removal will result in an `AopConfigException`. The ability to freeze the state of an advised object is useful in some cases, for example, to prevent calling code removing a security interceptor. It may also be used in Spring 1.1 to allow aggressive optimization if runtime advice modification is known not to be required.

B.8 Using the "autoproxy" facility

So far we've considered explicit creation of AOP proxies using a `ProxyFactoryBean` or similar factory bean.

Spring also allows us to use "autoproxy" bean definitions, which can automatically proxy selected bean definitions. This is built on Spring "bean post processor" infrastructure, which enables modification of any bean definition as the container loads.

In this model, you set up some special bean definitions in your XML bean definition file to configure the auto proxy infrastructure. This allows you just to declare the targets eligible for autoproxying: you don't need to use `ProxyFactoryBean`.

There are two ways to do this:

- Using an autoproxy creator that refers to specific beans in the current context.
- A special case of autoproxy creation that deserves to be considered separately; autoproxy creation driven by source-level metadata attributes.

B.8.1 Autoproxy bean definitions

The `org.springframework.aop.framework.autoproxy` package provides the following standard autoproxy creators.

BeanNameAutoProxyCreator

The `BeanNameAutoProxyCreator` class is a `BeanPostProcessor` that automatically creates AOP proxies for beans with names matching literal values or wildcards.

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="beanNames"><value>jdk*,onlyJdk</value></property>
  <property name="interceptorNames">
    <list>
      <value>myInterceptor</value>
    </list>
  </property>
</bean>
```

As with `ProxyFactoryBean`, there is an `interceptorNames` property rather than a list of interceptors, to allow correct behavior for prototype advisors. Named "interceptors" can be advisors or any advice type.

As with auto proxying in general, the main point of using `BeanNameAutoProxyCreator` is to apply the same configuration consistently to multiple objects, with minimal volume of configuration. It is a popular choice for applying declarative transactions to multiple objects.

Bean definitions whose names match, such as "jdkMyBean" and "onlyJdk" in the above example, are plain old bean definitions with the target class. An AOP proxy will be created automatically by the `BeanNameAutoProxyCreator`. The same advice will be applied to all matching beans. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different beans.

DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is `DefaultAdvisorAutoProxyCreator`. This will automagically apply eligible advisors in the current context, without the need to include specific bean names in the autoproxy advisor's bean definition. It offers the same merit of consistent configuration and avoidance of duplication as `BeanNameAutoProxyCreator`.

Using this mechanism involves:

- Specifying a `DefaultAdvisorAutoProxyCreator` bean definition.
- Specifying any number of Advisors in the same or related contexts. Note that these *must* be Advisors, not just interceptors or other advices. This is necessary because there must be a pointcut to evaluate, to check the eligibility of each advice to

candidate bean definitions.

The `DefaultAdvisorAutoProxyCreator` will automatically evaluate the pointcut contained in each advisor, to see what (if any) advice it should apply to each business object (such as "businessObject1" and "businessObject2" in the example).

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object will not be proxied. As bean definitions are added for new business objects, they will automatically be proxied if necessary.

Autoproxying in general has the advantage of making it impossible for callers or dependencies to obtain an un-advised object. Calling getBean("businessObject1") on this ApplicationContext will return an AOP proxy, not the target business object. (The "inner bean" idiom shown earlier also offers this benefit.)

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
  <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

The `DefaultAdvisorAutoProxyCreator` is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can simply add new business objects without including specific proxy configuration. You can also drop in additional aspects very easily - for example, tracing or performance monitoring aspects - with minimal change to configuration.

The DefaultAdvisorAutoProxyCreator offers support for filtering (using a naming convention so that only certain advisors are evaluated, allowing use of multiple, differently configured, AdvisorAutoProxyCreators in the same factory) and ordering. Advisors can implement the `org.springframework.core.Ordered` interface to ensure correct ordering if this is an issue. The TransactionAttributeSourceAdvisor used in the above example has a configurable order value; the default setting is unordered.

AbstractAdvisorAutoProxyCreator

This is the superclass of DefaultAdvisorAutoProxyCreator. You can create your own autoproxy creators by subclassing this class, in the unlikely event that advisor definitions offer insufficient customization to the behavior of the framework `DefaultAdvisorAutoProxyCreator`.

B.8.2 Using metadata-driven auto-proxying

A particularly important type of autoproxying is driven by metadata. This produces a similar programming model to .NET `ServicedComponents`. Instead of using XML deployment descriptors as in EJB, configuration for transaction management and other enterprise services is held in source-level attributes.

In this case, you use the `DefaultAdvisorAutoProxyCreator`, in combination with Advisors that understand metadata attributes. The metadata specifics are held in the pointcut part of the candidate advisors, rather than in the autoproxy creation class itself.

This is really a special case of the `DefaultAdvisorAutoProxyCreator`, but deserves consideration on its own. (The metadata-aware code is in the pointcuts contained in the advisors, not the AOP framework itself.)

The `/attributes` directory of the JPetStore sample application shows the use of attribute-driven autoproxying. In this case, there's no need to use the `TransactionProxyFactoryBean`. Simply defining transactional attributes on business objects is sufficient, because of the use of metadata-aware pointcuts. The bean definitions include the following code, in `/WEB-INF/declarativeServices.xml`. Note that this is generic, and can be used outside the JPetStore:

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

```
<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
      <property name="attributes" ref="attributes"/>
    </bean>
  </property>
</bean>

<bean id="attributes" class="org.springframework.metadata.commons.CommonsAttributes"/>
```

The `DefaultAdvisorAutoProxyCreator` bean definition (the name is not significant, hence it can even be omitted) will pick up all eligible pointcuts in the current application context. In this case, the "transactionAdvisor" bean definition, of type `TransactionAttributeSourceAdvisor`, will apply to classes or methods carrying a transaction attribute. The TransactionAttributeSourceAdvisor depends on a TransactionInterceptor, via constructor dependency. The example resolves this via autowiring. The `AttributesTransactionAttributeSource` depends on an implementation of the `org.springframework.metadata.Attributes` interface. In this fragment, the "attributes" bean satisfies this, using the Jakarta Commons Attributes API to obtain attribute information. (The application code must have been compiled using the Commons Attributes compilation task.)

The `/annotation` directory of the JPetStore sample application contains an analogous example for auto-proxying driven by JDK 1.5+ annotations. The following configuration enables automatic detection of Spring's `Transactional` annotation, leading to implicit proxies for beans containing that annotation:

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>
```

```
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
  </property>
</bean>
```

The `TransactionInterceptor` defined here depends on a `PlatformTransactionManager` definition, which is not included in this generic file (although it could be) because it will be specific to the application's transaction requirements (typically JTA, as in this example, or Hibernate, JDO or JDBC):

```
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

If you require only declarative transaction management, using these generic XML definitions will result in Spring automatically proxying all classes or methods with transaction attributes. You won't need to work directly with AOP, and the programming model is similar to that of .NET ServicedComponents.

This mechanism is extensible. It's possible to do autoproxying based on custom attributes. You need to:

- Define your custom attribute.
- Specify an Advisor with the necessary advice, including a pointcut that is triggered by the presence of the custom attribute on a class or method. You may be able to use an existing advice, merely implementing a static pointcut that picks up the custom attribute.

It's possible for such advisors to be unique to each advised class (for example, mixins): they simply need to be defined as prototype, rather than singleton, bean definitions. For example, the `LockMixin` introduction interceptor from the Spring test suite,

shown above, could be used in conjunction with an attribute-driven pointcut to target a mixin, as shown here. We use the generic `DefaultPointcutAdvisor`, configured using JavaBean properties:

```
<bean id="lockMixin" class="org.springframework.aop.LockMixin"
    scope="prototype"/>

<bean id="lockableAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor"
    scope="prototype">
  <property name="pointcut" ref="myAttributeAwarePointcut"/>

  <property name="advice" ref="lockMixin"/>
</bean>

<bean id="anyBean" class="anyclass" ...
```

If the attribute aware pointcut matches any methods in the `anyBean` or other bean definitions, the mixin will be applied. Note that both `lockMixin` and `lockableAdvisor` definitions are prototypes. The `myAttributeAwarePointcut` pointcut can be a singleton definition, as it doesn't hold state for individual advised objects.

B.9 Using TargetSources

Spring offers the concept of a *TargetSource*, expressed in the `org.springframework.aop.TargetSource` interface. This interface is responsible for returning the "target object" implementing the join point. The `TargetSource` implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers using Spring AOP don't normally need to work directly with TargetSources, but this provides a powerful means of supporting pooling, hot swappable and other sophisticated targets. For example, a pooling TargetSource can return a different target instance for each invocation, using a pool to manage instances.

If you do not specify a TargetSource, a default implementation is used that wraps a local object. The same target is returned for each invocation (as you would expect).

Let's look at the standard target sources provided with Spring, and how you can use them.

When using a custom target source, your target will usually need to be a prototype rather than a singleton bean definition. This allows Spring to create a new target instance when required.

B.9.1 Hot swappable target sources

The `org.springframework.aop.target.HotSwappableTargetSource` exists to allow the target of an AOP proxy to be switched while allowing callers to keep their references to it.

Changing the target source's target takes effect immediately. The `HotSwappableTargetSource` is threadsafe.

You can change the target via the `swap()` method on HotSwappableTargetSource as follows:

```
HotSwappableTargetSource swapper =
    (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

The XML definitions required look as follows:

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
  <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="swapper"/>
</bean>
```

The above `swap()` call changes the target of the swappable bean. Clients who hold a reference to that bean will be unaware of the change, but will immediately start hitting the new target.

Although this example doesn't add any advice - and it's not necessary to add advice to use a `TargetSource` - of course any `TargetSource` can be used in conjunction with arbitrary advice.

B.9.2 Pooling target sources

Using a pooling target source provides a similar programming model to stateless session EJBs, in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring pooling and SLSB pooling is that Spring pooling can be applied to any POJO. As with Spring in general, this service can be applied in a non-invasive way.

Spring provides out-of-the-box support for Jakarta Commons Pool 1.3, which provides a fairly efficient pooling implementation. You'll need the commons-pool Jar on your application's classpath to use this feature. It's also possible to subclass `org.springframework.aop.target.AbstractPoolingTargetSource` to support any other pooling API.

Sample configuration is shown below:

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"
    scope="prototype">
  ... properties omitted
</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
  <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="poolTargetSource"/>
  <property name="interceptorNames" value="myInterceptor"/>
```

```
</bean>
```

Note that the target object - "businessObjectTarget" in the example - *must* be a prototype. This allows the `PoolingTargetSource` implementation to create new instances of the target to grow the pool as necessary. See the Javadoc for `AbstractPoolingTargetSource` and the concrete subclass you wish to use for information about its properties: "maxSize" is the most basic, and always guaranteed to be present.

In this case, "myInterceptor" is the name of an interceptor that would need to be defined in the same IoC context. However, it isn't necessary to specify interceptors to use pooling. If you want only pooling, and no other advice, don't set the interceptorNames property at all.

It's possible to configure Spring so as to be able to cast any pooled object to the `org.springframework.aop.target.PoolingConfig` interface, which exposes information about the configuration and current size of the pool through an introduction. You'll need to define an advisor like this:

```
<bean id="poolConfigAdvisor" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetObject" ref="poolTargetSource"/>
  <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

This advisor is obtained by calling a convenience method on the `AbstractPoolingTargetSource` class, hence the use of MethodInvokingFactoryBean. This advisor's name ("poolConfigAdvisor" here) must be in the list of interceptors names in the ProxyFactoryBean exposing the pooled object.

The cast will look as follows:

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

Pooling stateless service objects is not usually necessary. We don't believe it should be the default choice, as most stateless objects are naturally thread safe, and instance pooling is problematic if resources are cached.

Simpler pooling is available using autoproxying. It's possible to set the TargetSources used by any autoproxy creator.

B.9.3 Prototype target sources

Setting up a "prototype" target source is similar to a pooling TargetSource. In this case, a new instance of the target will be created on every method invocation. Although the cost of creating a new object isn't high in a modern JVM, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus you shouldn't use this approach without very good reason.

To do this, you could modify the `poolTargetSource` definition shown above as follows. (I've also changed the name, for clarity.)

```
<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">
  <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

There's only one property: the name of the target bean. Inheritance is used in the TargetSource implementations to ensure consistent naming. As with the pooling target source, the target bean must be a prototype bean definition.

B.9.4 `ThreadLocal` target sources

`ThreadLocal` target sources are useful if you need an object to be created for each incoming request (per thread that is). The concept of a `ThreadLocal` provide a JDK-wide facility to transparently store resource alongside a thread. Setting up a `ThreadLocalTargetSource` is pretty much the same as was explained for the other types of target source:

```
<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

ThreadLocals come with serious issues (potentially resulting in memory leaks) when incorrectly using them in a multi-threaded and multi-classloader environments. One should always consider wrapping a threadlocal in some other class and never directly use the `ThreadLocal` itself (except of course in the wrapper class). Also, one should always remember to correctly set and unset (where the latter simply involved a call to `ThreadLocal.set(null)`) the resource local to the thread. Unsetting should be done in any case since not unsetting it might result in problematic behavior. Spring's ThreadLocal support does this for you and should always be considered in favor of using ThreadLocals without other proper handling code.

B.10 Defining new `Advice` types

Spring AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to the out-of-the-box interception around advice, before, throws advice and after returning advice.

The `org.springframework.aop.framework.adapter` package is an SPI package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom `Advice` type is that it must implement the `org.aopalliance.aop.Advice` tag interface.

Please refer to the `org.springframework.aop.framework.adapter` package's Javadocs for further information.

B.11 Further resources

Please refer to the Spring sample applications for further examples of Spring AOP:

- The JPetStore's default configuration illustrates the use of the `TransactionProxyFactoryBean` for declarative transaction management.
- The `/attributes` directory of the JPetStore illustrates the use of attribute-driven declarative transaction management.

Appendix A. Classic Spring Usage Home Appendix C. Migrating to Spring Framework 3.1