

Spring4 源码学习笔记

何宗桂 2017.09 深圳

01、Spring 4.3.10.RELEASE

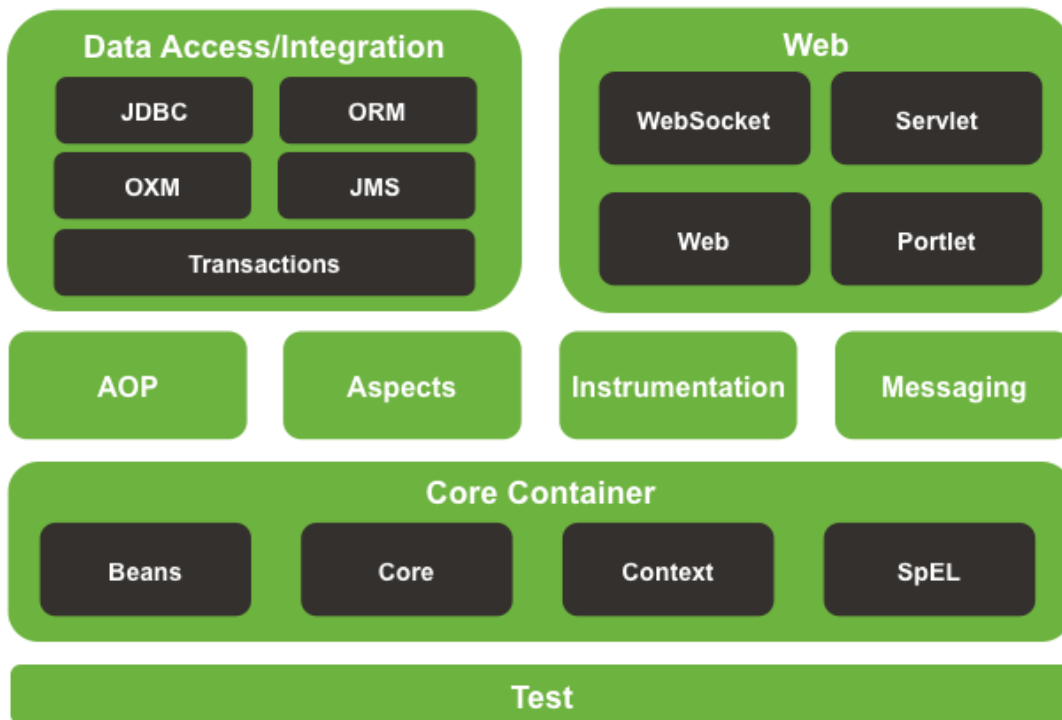
1.1、Spring jar 包

GroupId	ArtifactId	Description
org.springframework	spring-aop	Proxy-based AOP support
org.springframework	spring-aspects	AspectJ based aspects
org.springframework	spring-beans	Beans support, including Groovy
org.springframework	spring-context	Application context runtime, including scheduling and remoting abstractions
org.springframework	spring-context-support	Support classes for integrating common third-party libraries into a Spring application context
org.springframework	spring-core	Core utilities, used by many other Spring modules
org.springframework	spring-expression	Spring Expression Language (SpEL)
org.springframework	spring-instrument	Instrumentation agent for JVM bootstrapping
org.springframework	spring-instrument-tomcat	Instrumentation agent for Tomcat
org.springframework	spring-jdbc	JDBC support package, including DataSource setup and JDBC access support
org.springframework	spring-jms	JMS support package, including helper classes to send/receive JMS messages
org.springframework	spring-messaging	Support for messaging architectures and protocols
org.springframework	spring-orm	Object/Relational Mapping, including JPA and Hibernate support
org.springframework	spring-oxm	Object/XML Mapping
org.springframework	spring-test	Support for unit testing and integration testing Spring components
org.springframework	spring-tx	Transaction infrastructure, including DAO support and JCA integration
org.springframework	spring-web	Foundational web support, including web client and web-based remoting
org.springframework	spring-webmvc	HTTP-based Model-View-Controller and REST endpoints for Servlet stacks
org.springframework	spring-webmvc-portlet	MVC implementation to be used in a Portlet environment
org.springframework	spring-websocket	WebSocket and SockJS infrastructure, including STOMP messaging support

1.2、 Overview of the Spring Framework



Spring Framework Runtime



02、 Spring-core

01、 public class TypeDescriptor


implements Serializable

Context about a type to convert from or to.



TypeDescriptor

```
EMPTY_ANNOTATION_ARRAY : Annotation[]
streamAvailable : boolean
commonTypesCache : Map<Class<?>, TypeDescriptor>
CACHED_COMMON_TYPES : Class<?>[]
type : Class<?>
resolvableType : ResolvableType
annotatedElement : AnnotatedElementAdapter
TypeDescriptor(MethodParameter)
TypeDescriptor(Field)
TypeDescriptor(Property)
TypeDescriptor(ResolvableType, Class<?>, Annotation[])
```

- getObjectType() : Class<?>
- getType() : Class<?>
- getResolvableType() : ResolvableType
- getSource() : Object
- narrow(Object) : TypeDescriptor
- upcast(Class<?>) : TypeDescriptor
- getName() : String
- isPrimitive() : boolean
- getAnnotations() : Annotation[]
- hasAnnotation(Class<? extends Annotation>) : boolean
- getAnnotation(Class<T>) <T extends Annotation> : T
- isAssignableTo(TypeDescriptor) : boolean
- isNestedAssignable(TypeDescriptor, TypeDescriptor) : boolean
- isCollection() : boolean
- isArray() : boolean
- getElementTypeDescriptor() : TypeDescriptor
- elementTypeDescriptor(Object) : TypeDescriptor
- isMap() : boolean
- getMapKeyTypeDescriptor() : TypeDescriptor
- getMapKeyTypeDescriptor(Object) : TypeDescriptor
- getMapValueTypeDescriptor() : TypeDescriptor
- getMapValueTypeDescriptor(Object) : TypeDescriptor
- narrow(Object, TypeDescriptor) : TypeDescriptor
- equals(Object) : boolean
- annotationsMatch(TypeDescriptor) : boolean
- annotationEquals(Annotation, Annotation) : boolean
- hashCode() : int
- toString() : String
- ^S forObject(Object) : TypeDescriptor
- ^S valueOf(Class<?>) : TypeDescriptor
- ^S collection(Class<?>, TypeDescriptor) : TypeDescriptor
- ^S map(Class<?>, TypeDescriptor, TypeDescriptor) : TypeDescriptor
- ^S array(TypeDescriptor) : TypeDescriptor
- ^S nested(MethodParameter, int) : TypeDescriptor
- ^S nested(Field, int) : TypeDescriptor
- ^S nested(Property, int) : TypeDescriptor
- ^S nested(TypeDescriptor, int) : TypeDescriptor
- ^S getRelatedIfResolvable(TypeDescriptor, ResolvableType) : TypeDescriptor
- >  AnnotatedElementAdapter
- >  ^S StreamDelegate

02、public final class Property

A description of a JavaBeans Property that allows us to avoid a dependency on `java.beans.PropertyDescriptor`. The `java.beans` package is not available in a number of environments (e.g. Android, Java ME), so this is desirable for portability of Spring's core conversion facility. Used to build a `TypeDescriptor` from a property location. The built `TypeDescriptor` can then be used to convert from/to the property type.

Property





























- ▣ `annotationCache : Map<Property, Annotation[]>`
- ▣ `objectType : Class<?>`
- ▣ `readMethod : Method`
- ▣ `writeMethod : Method`
- ▣ `name : String`
- ▣ `methodParameter : MethodParameter`
- ▣ `annotations : Annotation[]`
- `Property(Class<?>, Method, Method)`
- `Property(Class<?>, Method, Method, String)`
- `getObjectType() : Class<?>`
- `getName() : String`
- `getType() : Class<?>`
- `getReadMethod() : Method`
- `getWriteMethod() : Method`
- ▲ `getMethodParameter() : MethodParameter`
- ▲ `getAnnotations() : Annotation[]`
- `resolveName() : String`
- `resolveMethodParameter() : MethodParameter`
- `resolveReadMethodParameter() : MethodParameter`
- `resolveWriteMethodParameter() : MethodParameter`
- `resolveParameterType(MethodParameter) : MethodParameter`
- `resolveAnnotations() : Annotation[]`
- `addAnnotationsToMap(Map<Class<? extends Annotation>, Annotation>, AnnotatedElement) : void`
- `getField() : Field`
- `declaringClass() : Class<?>`
- ▲ `equals(Object) : boolean`
- ▲ `hashCode() : int`

03、public class MethodParameter

Helper class that encapsulates the specification of a method parameter, i.e. a `@link Method` or `@link Constructor` plus a parameter index and a nested type index for a declared generic type. Useful as a specification object to pass along.

As of 4.2, there is a `org.springframework.core.annotation.SynthesizingMethodParameter` subclass available which synthesizes annotations with attribute aliases. That subclass is used for web and message endpoint processing, in particular.

MethodParameter

-  `javaUtilOptionalClass : Class<?>`
-  `method : Method`
-  `constructor : Constructor<?>`
-  `parameterIndex : int`
-  `nestingLevel : int`
-  `typeIndexesPerLevel : Map<Integer, Integer>`
-  `containingClass : Class<?>`
-  `parameterType : Class<?>`
-  `genericParameterType : Type`
-  `parameterAnnotations : Annotation[]`
-  `parameterNameDiscoverer : ParameterNameDiscoverer`
-  `parameterName : String`
-  `nestedMethodParameter : MethodParameter`
-  `MethodParameter(Method, int)`
-  `MethodParameter(Method, int, int)`
-  `MethodParameter(Constructor<?>, int)`
-  `MethodParameter(Constructor<?>, int, int)`
-  `MethodParameter(MethodParameter)`
-  `getMethod() : Method`
-  `getConstructor() : Constructor<?>`
-  `getDeclaringClass() : Class<?>`
-  `getMember() : Member`
-  `getAnnotatedElement() : AnnotatedElement`
-  `getParameterIndex() : int`
-  `increaseNestingLevel() : void`
-  `decreaseNestingLevel() : void`
-  `getNestingLevel() : int`
-  `setTypeIndexForCurrentLevel(int) : void`

- `getTypeIndexForCurrentLevel() : Integer`
- `getTypeIndexForLevel(int) : Integer`
- `getTypeIndexesPerLevel() : Map<Integer, Integer>`
- `nested() : MethodParameter`
- `isOptional() : boolean`
- `nestedIfOptional() : MethodParameter`
- ▲ `setContainingClass(Class<?>) : void`
- `getContainingClass() : Class<?>`
- ▲ `setParameterType(Class<?>) : void`
- `getParameterType() : Class<?>`
- `getGenericParameterType() : Type`
- `getNestedParameterType() : Class<?>`
- `getNestedGenericParameterType() : Type`
- `getMethodAnnotations() : Annotation[]`
- `getMethodAnnotation(Class<A>) <A extends Annotation> : A`
- `hasMethodAnnotation(Class<A>) <A extends Annotation> : boolean`
- `getParameterAnnotations() : Annotation[]`
- `hasParameterAnnotations() : boolean`
- `getParameterAnnotation(Class<A>) <A extends Annotation> : A`
- `hasParameterAnnotation(Class<A>) <A extends Annotation> : boolean`
- `initParameterNameDiscovery(ParameterNameDiscoverer) : void`
- `getParameterName() : String`
- ◆ `adaptAnnotation(A) <A extends Annotation> : A`
- ◆ `adaptAnnotationArray(Annotation[]) : Annotation[]`
- ▲ `equals(Object) : boolean`
- ▲ `hashCode() : int`
- ▲ `toString() : String`
- ▲ `clone() : MethodParameter`
- ^S `forMethodOrConstructor(Object, int) : MethodParameter`

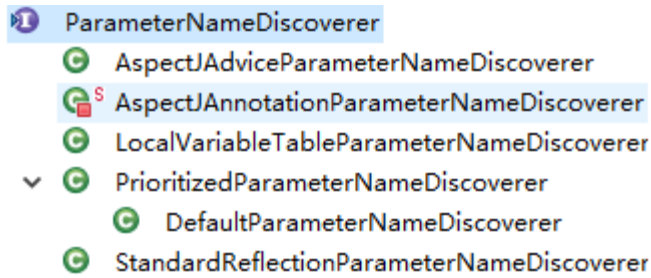
04、public interface ParameterNameDiscoverer

Interface to discover parameter names for methods and constructors.

Parameter name discovery is not always possible, but various strategies are available to try, such as looking for debug information that may have been emitted at compile time, and looking for argname annotation values optionally accompanying AspectJ annotated methods.

I ParameterNameDiscoverer

- ^A `getParameterNames(Method) : String[]`
- ^A `getParameterNames(Constructor<?>) : String[]`



05、 public class InheritableThreadLocal<T>

This class extends `ThreadLocal` to provide inheritance of values from parent thread to child thread: when a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values. Normally the child's values will be identical to the parent's; however, the child's value can be made an arbitrary function of the parent's by overriding the `childValue` method in this class.

Inheritable thread-local variables are used in preference to ordinary thread-local variables when the per-thread-attribute being maintained in the variable (e.g., User ID, Transaction ID) must be automatically transmitted to any child threads that are created.

```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {  
    protected T childValue(T parentValue) {  
        return parentValue;  
    }  
    ThreadLocalMap getMap(Thread t) {  
        return t.inheritableThreadLocals;  
    }  
    void createMap(Thread t, T firstValue) {  
        t.inheritableThreadLocals = new ThreadLocalMap(this, firstValue);  
    }  
}
```

06、 public class NamedInheritableThreadLocal<T>

{@link InheritableThreadLocal} subclass that exposes a specified name as {@link #toString()} result (allowing for introspection).

```
public class NamedInheritableThreadLocal<T> extends InheritableThreadLocal<T> {  
    private final String name;  
    public NamedInheritableThreadLocal(String name) {  
        Assert.hasText(name, "Name must not be empty");  
        this.name = name;  
    }  
    @Override  
    public String toString() {
```

```

        return this.name;
    }
}

```

07、 public class NamedThreadLocal<T>

{@link ThreadLocal} subclass that exposes a specified name as {@link #toString()} result (allowing for introspection).

```

public class NamedThreadLocal<T> extends ThreadLocal<T> {
    private final String name;

    public NamedThreadLocal(String name) {
        Assert.hasText(name, "Name must not be empty");
        this.name = name;
    }

    @Override
    public String toString() {
        return this.name;
    }
}

```

08、 public class AnnotationAttributes

{@link LinkedHashMap} subclass representing annotation attribute key-value pairs as read by {@link AnnotationUtils}, {@link AnnotatedElementUtils}, and Spring's reflection- and ASM-based {@link org.springframework.core.type.AnnotationMetadata} implementations.

Provides 'pseudo-reification' to avoid noisy Map generics in the calling code as well as convenience methods for looking up annotation attributes in a type-safe fashion.

```

@SuppressWarnings("serial")
public class AnnotationAttributes extends LinkedHashMap<String, Object> {

    private static final String UNKNOWN = "unknown";

    private final Class<? extends Annotation> annotationType;

    private final String displayName;

    boolean validated = false;
}

```


09、 public class ResolvableType

10、 public abstract class GenericTypeResolver

Helper class for resolving generic types against type variables. Mainly intended for usage within the framework, resolving method parameter types even when they are declared generically.

11、 public interface InfrastructureProxy

Interface to be implemented by transparent resource proxies that need to be considered as equal to the underlying resource, for example for consistent lookup key comparisons. Note that this interface does imply such special semantics and does not constitute a general-purpose mixin!

Such wrappers will automatically be unwrapped for key comparisons in `{@link org.springframework.transaction.support.TransactionSynchronizationManager}`. Only fully transparent proxies, e.g. for redirection or service lookups, are supposed to implement this interface. Proxies that decorate the target object with new behavior, such as AOP proxies, do *not* qualify here!

12、 public abstract class MethodIntrospector

Defines the algorithm for searching for metadata-associated methods exhaustively including interfaces and parent classes while also dealing with parameterized methods as well as common scenarios encountered with interface and class-based proxies.

Typically, but not necessarily, used for finding annotated handler methods.

13、 public class MethodParameter

Helper class that encapsulates the specification of a method parameter, i.e. a `{@link Method}` or `{@link Constructor}` plus a parameter index and a nested type index for a declared generic type. Useful as a specification object to pass along.

As of 4.2, there is a `{@link org.springframework.core.annotation.SynthesizingMethodParameter}` subclass available which synthesizes annotations with attribute aliases. That subclass is used for web and message endpoint processing, in particular.

14、 public class AnnotatedElementUtils

▼ ⓘ AnnotatedElement

- ^D isAnnotationPresent(Class<? extends Annotation>) : boolean
- ^A getAnnotation(Class<T>) <T extends Annotation> : T
- ^A getAnnotations() : Annotation[]
- ^D getAnnotationsByType(Class<T>) <T extends Annotation> : T[]
- ^D getDeclaredAnnotation(Class<T>) <T extends Annotation> : T
- ^D getDeclaredAnnotationsByType(Class<T>) <T extends Annotation> : T[]
- ^A getDeclaredAnnotations() : Annotation[]

```
@SuppressWarnings("serial")
public class AnnotationAttributes extends LinkedHashMap<String, Object> {

    private static final String UNKNOWN = "unknown";

    private final Class<? extends Annotation> annotationType;

    private final String displayName;

    boolean validated = false;
```






























15、 public abstract class AnnotationUtils

16、 public abstract class StreamUtils

Simple utility methods for dealing with streams. The copy methods of this class are similar to those defined in [{@link FileCopyUtils}](#) except that all affected streams are left open when done. All copy methods use a block size of 4096 bytes.

Mainly for use within the framework, but also useful for application code.

StreamUtils






















-  StreamUtils()
-   BUFFER_SIZE : int
-   EMPTY_CONTENT : byte[]
-   copyToByteArray(InputStream) : byte[]
-   copyToString(InputStream, Charset) : String
-   copy(byte[], OutputStream) : void
-   copy(String, Charset, OutputStream) : void
-   copy(InputStream, OutputStream) : int
-   copyRange(InputStream, OutputStream, long, long) : long
-   drain(InputStream) : int
-   emptyInput() : InputStream
-   nonClosing(InputStream) : InputStream
-   nonClosing(OutputStream) : OutputStream
- >   NonClosingInputStream
- >   NonClosingOutputStream

17、 public abstract class FileCopyUtils

Simple utility methods for file and stream copying. All copy methods use a block size of 4096 bytes, and close all affected streams when done. A variation of the copy methods from this class that leave streams open can be found in [{@link StreamUtils}](#).

Mainly for use within the framework, but also useful for application code.

FileCopyUtils

















-  FileCopyUtils()
-   BUFFER_SIZE : int
-   copy(File, File) : int
-   copy(byte[], File) : void
-   copyToByteArray(File) : byte[]
-   copy(InputStream, OutputStream) : int
-   copy(byte[], OutputStream) : void
-   copyToByteArray(InputStream) : byte[]
-   copy(Reader, Writer) : int
-   copy(String, Writer) : void
-   copyToString(Reader) : String

18、 public abstract class FileSystemUtils

Utility methods for working with the file system.
















19、 public abstract class ResourceUtils

ResourceUtils

-  ResourceUtils()
-  CLASSPATH_URL_PREFIX : String
-  FILE_URL_PREFIX : String
-  JAR_URL_PREFIX : String
-  WAR_URL_PREFIX : String
-  URL_PROTOCOL_FILE : String
-  URL_PROTOCOL_JAR : String
-  URL_PROTOCOL_WAR : String
-  URL_PROTOCOL_ZIP : String
-  URL_PROTOCOL_WSJAR : String
-  URL_PROTOCOL_VFSZIP : String
-  URL_PROTOCOL_VFSFILE : String
-  URL_PROTOCOL_VFS : String
-  JAR_FILE_EXTENSION : String
-  JAR_URL_SEPARATOR : String
-  WAR_URL_SEPARATOR : String

Utility methods for resolving resource locations to files in the file system. Mainly for internal use within the framework.

Consider using Spring's Resource abstraction in the core package for handling all kinds of file resources in a uniform manner. [@link org.springframework.core.io.ResourceLoader](#)'s `getResource()` method can resolve any location to a [@link org.springframework.core.io.Resource](#) object, which in turn allows one to obtain a `java.io.File` in the file system through its `getFile()` method.

-  `isUrl(String) : boolean`
-  `getURL(String) : URL`
-  `getFile(String) : File`
-  `getFile(URL) : File`
-  `getFile(URL, String) : File`
-  `getFile(URI) : File`
-  `getFile(URI, String) : File`
-  `isFileURL(URL) : boolean`
-  `isJarURL(URL) : boolean`
-  `isJarFileURL(URL) : boolean`
-  `extractJarFileURL(URL) : URL`
-  `extractArchiveURL(URL) : URL`
-  `toURI(URL) : URI`
-  `toURI(String) : URI`
-  `useCachesIfNecessary(URLConnection) : void`

20、 public abstract class ClassUtils

Miscellaneous class utility methods. Mainly for internal use within the framework.

21、 public abstract class CollectionUtils

22、 public abstract class SerializationUtils

Static utilities for serialization and deserialization.

23、 public class SocketUtils

Simple utility methods for working with network sockets — for example, for finding available ports on `localhost`.

Within this class, a TCP port refers to a port for a [ServerSocket](#); whereas, a UDP port refers to a port for a [DatagramSocket](#).

24、 public class ConcurrentReferenceHashMap<K, V>

A [ConcurrentHashMap](#) that uses [ReferenceType#SOFT soft](#) or [ReferenceType#WEAK weak](#) references for both `keys` and `values`.

This class can be used as an alternative to `Collections.synchronizedMap(new WeakHashMap<K, Reference<V>>())` in order to support better performance when accessed concurrently. This implementation follows the same design constraints as [ConcurrentHashMap](#) with the exception that `null` values and `null` keys are supported.

NOTE: The use of references means that there is no guarantee that items placed into the map will be subsequently available. The garbage collector may discard references at any time, so it may appear that an unknown thread is silently removing entries.

If not explicitly specified, this implementation will use [SoftReference soft](#) entry references).

```
public class ConcurrentReferenceHashMap<K, V> extends AbstractMap<K, V> implements
ConcurrentMap<K, V> {
```

25、 public abstract class DigestUtils

Miscellaneous methods for calculating digests.

26、 public class MimeType

public class MimeType **implements** Comparable<MimeType>, Serializable

Represents a MIME Type, as originally defined in RFC 2046 and subsequently used in other Internet protocols including HTTP.

This class, however, does not contain support for the q-parameters used in HTTP content negotiation. Those can be found in the sub-class

{@code org.springframework.http.MediaType} in the {@code spring-web} module.

Consists of a {@linkplain #getType() type} and a {@linkplain #getSubtype() subtype}.

Also has functionality to parse media types from a string using

{@link #valueOf(String)}. For more parsing options see {@link MimeTypeUtils}.

27、 public abstract class MimeTypeUtils

Miscellaneous {@link MimeType} utility methods.

28、 public abstract class NumberUtils

Miscellaneous utility methods for number conversion and parsing.

Mainly for internal use within the framework; consider Apache's Commons Lang for a more comprehensive suite of number utilities.

29、 public abstract class StringUtils

This class delivers some simple functionality that should really be provided by the core Java {@link String} and {@link StringBuilder} classes. It also provides easy-to-use methods to convert between delimited strings, such as CSV strings, and collections and arrays.

30、 public abstract class ObjectUtils

Miscellaneous object utility methods.

Mainly for internal use within the framework.

Thanks to Alex Ruiz for contributing several enhancements to this class!

31、 public abstract class PatternMatchUtils

Utility methods for simple pattern matching, in particular for Spring's typical "xxx*", "*xxx" and "*xxx*" pattern styles.

32、 public abstract class ReflectionUtils

Simple utility class for working with the reflection API and handling reflection exceptions.

33、 public abstract class SystemPropertyUtils

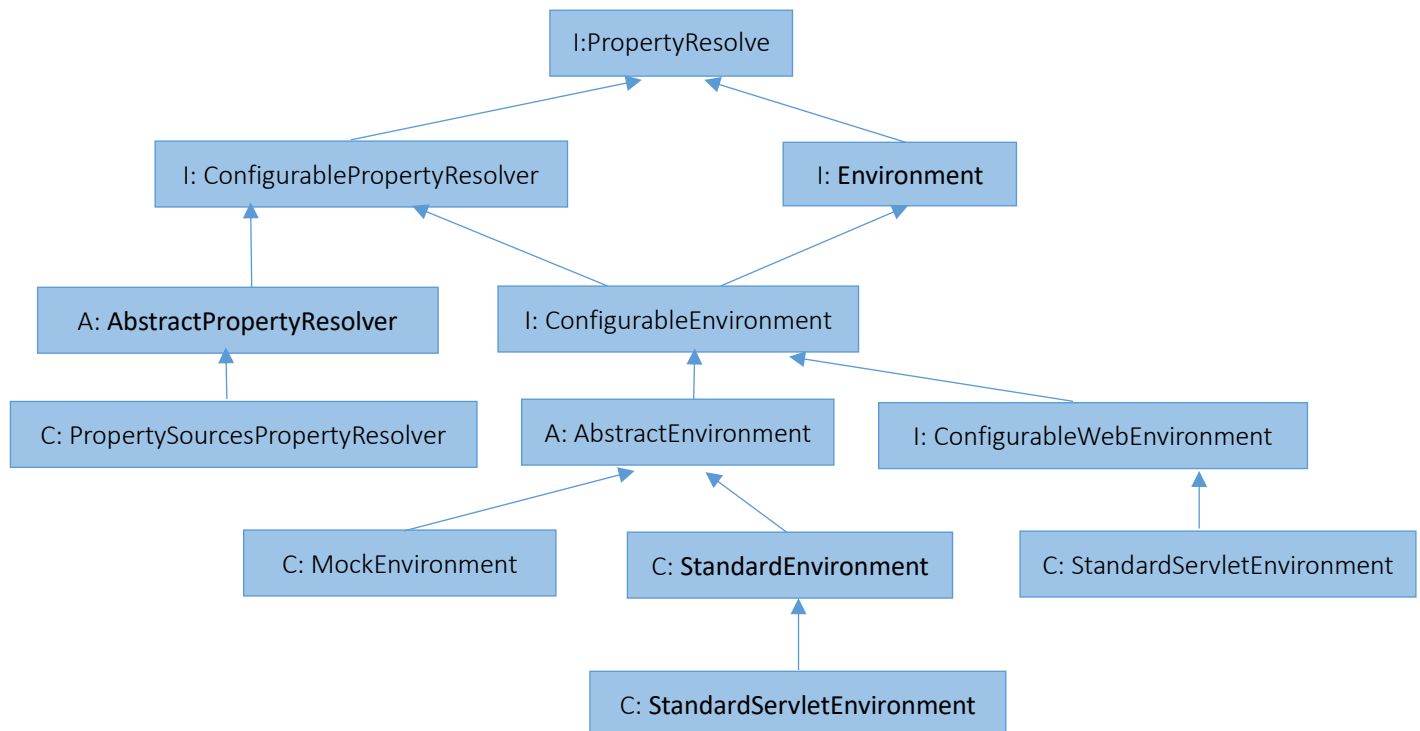
Helper class for resolving placeholders in texts. Usually applied to file paths.

34、 public abstract class TypeUtils

Utility to work with Java 5 generic type parameters. Mainly for internal use within the framework.

35、 public interface PropertyResolver

Interface for resolving properties against any underlying source.



- ▼ ⓘ PropertyResolver
 - ▼ ⓘ ConfigurablePropertyResolver
 - ▼ ⓘ AbstractPropertyResolver
 - PropertySourcesPropertyResolver
 - ▼ ⓘ ConfigurableEnvironment
 - ▼ ⓘ AbstractEnvironment
 - MockEnvironment
 - ▼ StandardEnvironment
 - StandardServletEnvironment
 - ▼ ⓘ ConfigurableWebEnvironment
 - StandardServletEnvironment
 - ▼ ⓘ Environment
 - ▼ ⓘ ConfigurableEnvironment
 - ▼ ⓘ AbstractEnvironment
 - MockEnvironment
 - ▼ StandardEnvironment
 - StandardServletEnvironment
 - ▼ ⓘ ConfigurableWebEnvironment
 - StandardServletEnvironment

- ▼ ⓘ PropertyResolver
 - containsProperty(String) : boolean
 - getProperty(String) : String
 - getProperty(String, String) : String
 - getProperty(String, Class<T>) <T> : T
 - getProperty(String, Class<T>, T) <T> : T
 - getPropertyAsClass(String, Class<T>) <T> : Class<T>
 - getRequiredProperty(String) : String
 - getRequiredProperty(String, Class<T>) <T> : T
 - resolvePlaceholders(String) : String
 - resolveRequiredPlaceholders(String) : String

- ▼ ⓘ ConfigurablePropertyResolver
 - getConversionService() : ConfigurableConversionService
 - setConversionService(ConfigurableConversionService) : void
 - setPlaceholderPrefix(String) : void
 - setPlaceholderSuffix(String) : void
 - setValueSeparator(String) : void
 - setIgnoreUnresolvableNestedPlaceholders(boolean) : void
 - setRequiredProperties(String...) : void
 - validateRequiredProperties() : void

Configuration interface to be implemented by most if not all Environment types. Provides facilities for setting active and default profiles and manipulating underlying property sources. Allows clients to set and validate required properties, customize the conversion service and more through the ConfigurablePropertyResolver superinterface.

- ▼ ⓘ ConfigurableEnvironment
 - ^A setActiveProfiles(String...) : void
 - ^A addActiveProfile(String) : void
 - ^A setDefaultProfiles(String...) : void
 - ^A getPropertySources() : MutablePropertySources
 - ^A getSystemEnvironment() : Map<String, Object>
 - ^A getSystemProperties() : Map<String, Object>
 - ^A merge(ConfigurableEnvironment) : void
- ▼ ⓘ MockEnvironment
 - ▢ propertySource : MockPropertySource
 - ^C MockEnvironment()
 - setProperty(String, String) : void
 - withProperty(String, String) : MockEnvironment
- ▼ ⓘ ConfigurableWebEnvironment
 - ^A initPropertySources(ServletContext, ServletConfig) : void

36、public interface Environment

public interface Environment extends PropertyResolver

Interface representing the environment in which the current application is running.

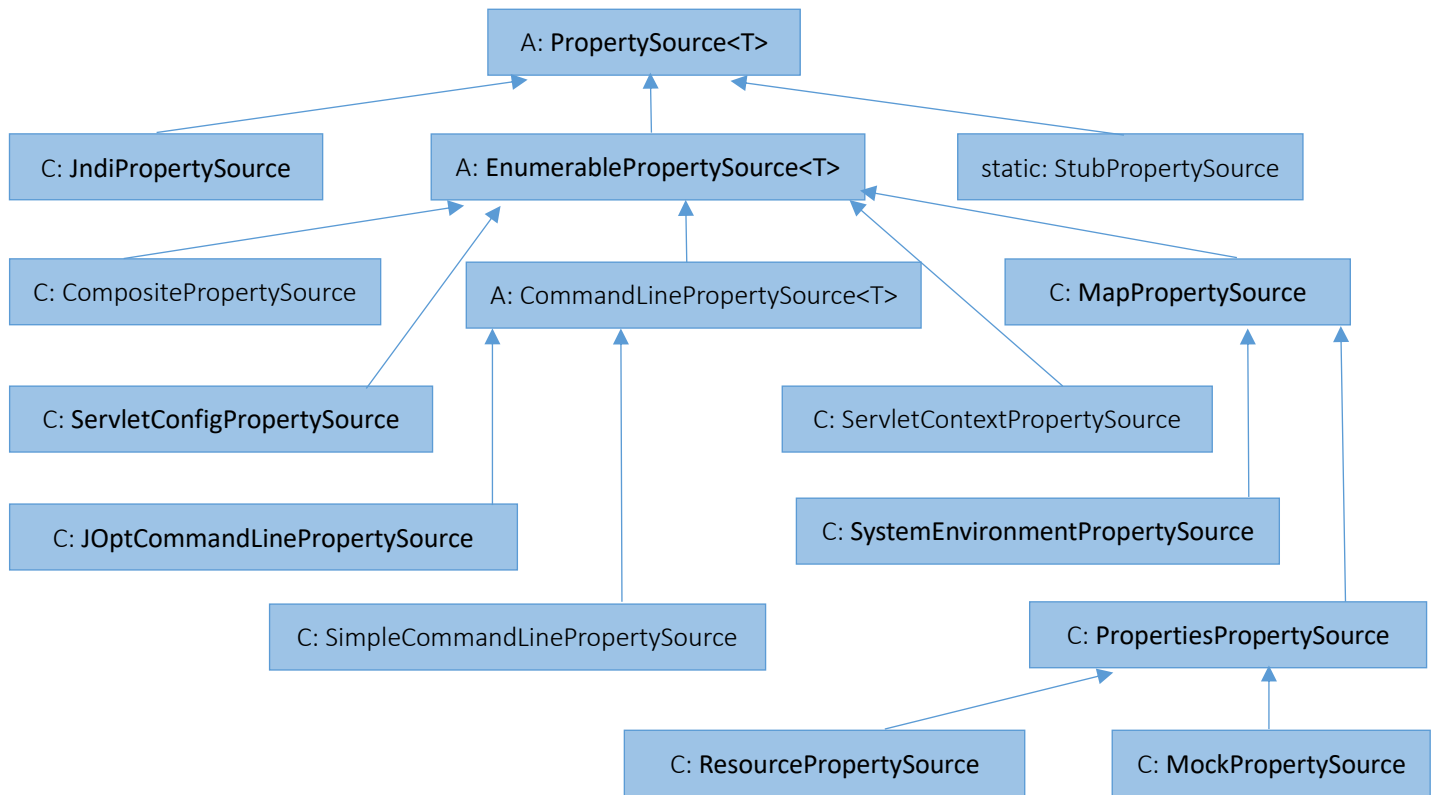
- ▼ ⓘ Environment
 - ^A getActiveProfiles() : String[]
 - ^A getDefaultProfiles() : String[]
 - ^A acceptsProfiles(String...) : boolean

Interface indicating a component that contains and exposes an Environment reference.

```
public interface EnvironmentCapable {  
    Environment getEnvironment();  
}
```

37、 public abstract class PropertySource<T>

Abstract base class representing a source of name/value property pairs.



```
▼ EnumerablePropertySource<T>
  • EnumerablePropertySource(String, T)
  • EnumerablePropertySource(String)
  • containsProperty(String) : boolean
  • getPropertyNames() : String[]




▼ JndiPropertySource
  • JndiPropertySource(String)
  • JndiPropertySource(String, JndiLocatorDelegate)
  • getProperty(String) : Object
```

Holder containing one or more PropertySource objects.

```
public interface PropertySources extends Iterable<PropertySource<?>> {
    boolean contains(String name);
    PropertySource<?> get(String name);
}
```

Default implementation of the PropertySources interface. Allows manipulation of contained property sources and provides a constructor for copying an existing PropertySources instance.

```
public class MutablePropertySources implements PropertySources
```

```
▼  PropertySource<T>
  ◊ F logger : Log
  ◊ F name : String
  ◊ F source : T
  ● C PropertySource(String, T)
  ● C PropertySource(String)
  ● getName() : String
  ● getSource() : T
  ● containsProperty(String) : boolean
  ● A getProperty(String) : Object
  ● ▲ equals(Object) : boolean
  ● ▲ hashCode() : int
  ● ▲ toString() : String
  ● S named(String) : PropertySource<?>
  >  S StubPropertySource
  >  S ComparisonPropertySource
```

Strategy interface for creating resource-based {@link PropertySource} wrappers.

38、public interface PropertySourceFactory

```
PropertySource<?> createPropertySource(String name, EncodedResource resource)
    throws IOException;
}

public class DefaultPropertySourceFactory implements PropertySourceFactory {
    @Override
    public PropertySource<?> createPropertySource(String name, EncodedResource
resource) throws IOException {
        return (name != null ? new ResourcePropertySource(name, resource) : new
ResourcePropertySource(resource));
    }
}
```

39、 public interface Converter<S, T>

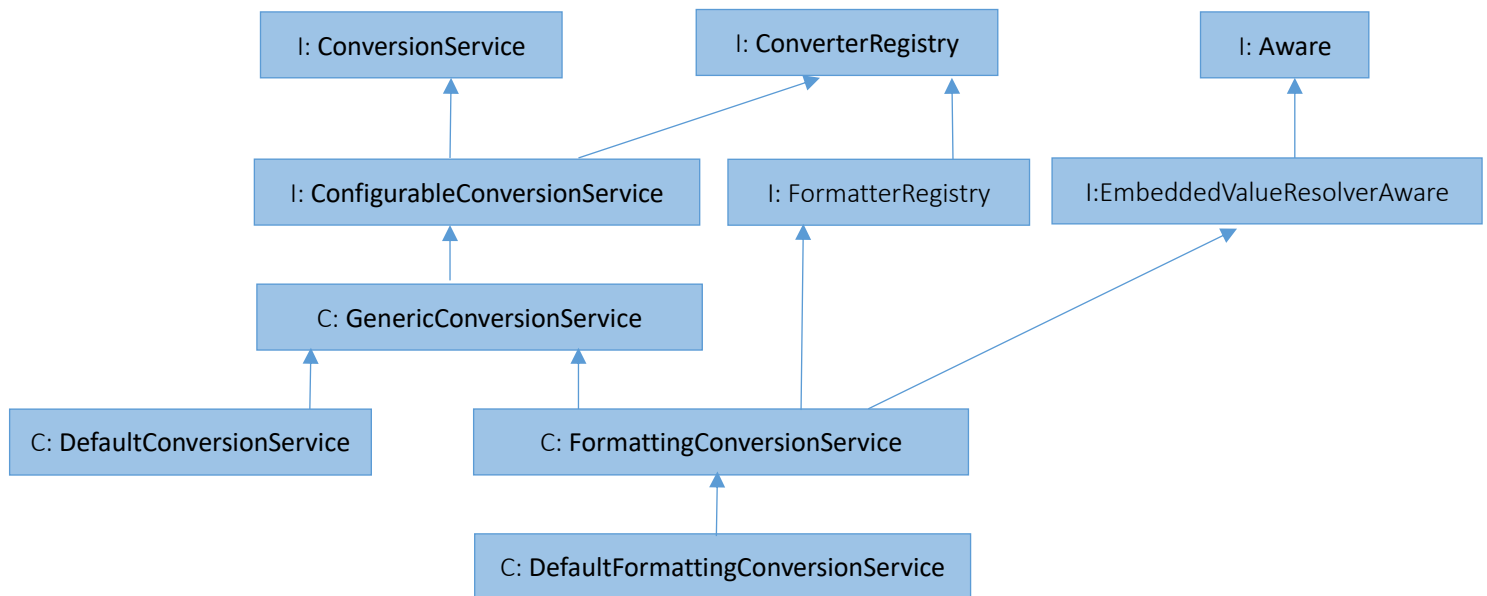
```
T convert(S source);  
}
```

A converter converts a source object of type S to a target of type code T.

Implementations of this interface are thread-safe and can be shared.

Implementations may additionally implement ConditionalConverter.

40、 public interface ConverterRegistry











```
▼ ⓘ ConverterRegistry  
  ●A addConverter(Converter<?, ?>) : void  
  ●A addConverter(Class<S>, Class<T>, Converter<? super S, ? extends T>) <S, T> : void  
  ●A addConverter(GenericConverter) : void  
  ●A addConverterFactory(ConverterFactory<?, ?>) : void  
  ●A removeConvertible(Class<?>, Class<?>) : void  
  
▼ ⓘ FormatterRegistry  
  ●A addFormatter(Formatter<?>) : void  
  ●A addFormatterForFieldType(Class<?>, Formatter<?>) : void  
  ●A addFormatterForFieldType(Class<?>, Printer<?>, Parser<?>) : void  
  ●A addFormatterForFieldAnnotation(AnnotationFormatterFactory<? extends Annotation>) : void
```





















```
public interface Formatter<T> extends Printer<T>, Parser<T> {

}
```

ConversionService

-   `canConvert(Class<?>, Class<?>) : boolean`
-   `canConvert(TypeDescriptor, TypeDescriptor) : boolean`
-   `convert(Object, Class<T>) <T> : T`
-   `convert(Object, TypeDescriptor, TypeDescriptor) : Object`

▼ DefaultConversionService

-   `javaUtilOptionalClassAvailable : boolean`
-   `jsr310Available : boolean`
-   `streamAvailable : boolean`
-   `sharedInstance : DefaultConversionService`
-   `getSharedInstance() : ConversionService`
-   `DefaultConversionService()`
-   `addDefaultConverters(ConverterRegistry) : void`
-   `addCollectionConverters(ConverterRegistry) : void`
-   `addScalarConverters(ConverterRegistry) : void`
- >   `Jsr310ConverterRegistrar`

```
public static void addCollectionConverters(ConverterRegistry converterRegistry) {
    ConversionService conversionService = (ConversionService) converterRegistry;

    converterRegistry.addConverter(new ArrayToCollectionConverter(conversionService));
    converterRegistry.addConverter(new CollectionToArrayConverter(conversionService));

    converterRegistry.addConverter(new ArrayToArrayConverter(conversionService));
    converterRegistry.addConverter(new CollectionToCollectionConverter(conversionService));
    converterRegistry.addConverter(new MapToMapConverter(conversionService));

    converterRegistry.addConverter(new ArrayToStringConverter(conversionService));
    converterRegistry.addConverter(new StringToArrayConverter(conversionService));

    converterRegistry.addConverter(new ArrayToObjectConverter(conversionService));
    converterRegistry.addConverter(new ObjectToArrayConverter(conversionService));

    converterRegistry.addConverter(new CollectionToStringConverter(conversionService));
    converterRegistry.addConverter(new StringToCollectionConverter(conversionService));

    converterRegistry.addConverter(new CollectionToObjectConverter(conversionService));
    converterRegistry.addConverter(new ObjectToCollectionConverter(conversionService));

    if (streamAvailable) {
        converterRegistry.addConverter(new StreamConverter(conversionService));
    }
}
```

```

private static void addScalarConverters(ConverterRegistry converterRegistry) {
    converterRegistry.addConverterFactory(new NumberToNumberConverterFactory());

    converterRegistry.addConverterFactory(new StringToNumberConverterFactory());
    converterRegistry.addConverter(Number.class, String.class, new ObjectToStringConverter());

    converterRegistry.addConverter(new StringToCharacterConverter());
    converterRegistry.addConverter(Character.class, String.class, new ObjectToStringConverter());

    converterRegistry.addConverter(new NumberToCharacterConverter());
    converterRegistry.addConverterFactory(new CharacterToNumberFactory());

    converterRegistry.addConverter(new StringToBooleanConverter());
    converterRegistry.addConverter(Boolean.class, String.class, new ObjectToStringConverter());

    converterRegistry.addConverterFactory(new StringToEnumConverterFactory());
    converterRegistry.addConverter(new EnumToStringConverter((ConversionService) converterRegistry));

    converterRegistry.addConverterFactory(new IntegerToEnumConverterFactory());
    converterRegistry.addConverter(new EnumToIntegerConverter((ConversionService) converterRegistry));

    converterRegistry.addConverter(new StringToLocaleConverter());
    converterRegistry.addConverter(Locale.class, String.class, new ObjectToStringConverter());

    converterRegistry.addConverter(new StringToCharsetConverter());
    converterRegistry.addConverter(Charset.class, String.class, new ObjectToStringConverter());

    converterRegistry.addConverter(new StringToCurrencyConverter());
    converterRegistry.addConverter(Currency.class, String.class, new ObjectToStringConverter());

    converterRegistry.addConverter(new StringToPropertiesConverter());
    converterRegistry.addConverter(new PropertiesToStringConverter());

    converterRegistry.addConverter(new StringToUUIDConverter());
    converterRegistry.addConverter(UUID.class, String.class, new ObjectToStringConverter());
}

private static final class Jsr310ConverterRegistrar {

    public static void registerJsr310Converters(ConverterRegistry converterRegistry) {
        converterRegistry.addConverter(new StringToTimeZoneConverter());
        converterRegistry.addConverter(new ZoneIdToTimeZoneConverter());
        converterRegistry.addConverter(new ZonedDateTimeToCalendarConverter());
    }
}

```

41、 public interface ConditionalConverter

```

    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}

public interface GenericConverter







public interface ConditionalGenericConverter extends GenericConverter,
ConditionalConverter {

























}

```






















45、public interface ConverterFactory<S, R>

```
<T extends R> Converter<S, T> getConverter(Class<T> targetType);  
}
```

- ▼  ConverterFactory<S, R>
 -  CharacterToNumberFactory
 -  IntegerToEnumConverterFactory
 -  NumberToNumberConverterFactory
 -  StringToEnumConverterFactory
 -  StringToNumberConverterFactory

- ▼  ConditionalGenericConverter
 -  AnnotationParserConverter
 -  AnnotationPrinterConverter
 -  ArrayToArrayConverter
 -  ArrayToCollectionConverter
 -  ArrayToObjectConverter
 -  ArrayToStringConverter
 -  ByteBufferConverter
 -  CollectionToArrayConverter
 -  CollectionToCollectionConverter
 -  CollectionToObjectConverter
 -  CollectionToStringConverter
 -  ConverterAdapter
 -  ConverterFactoryAdapter
 -  FallbackObjectToStringConverter
 -  IdToEntityConverter
 -  MapToMapConverter
 -  ObjectToArrayConverter
 -  ObjectToCollectionConverter
 -  ObjectToObjectConverter
 -  ObjectToOptionalConverter
 -  StreamConverter
 -  StringToArrayConverter
 -  StringToCollectionConverter

46、interface AnnotationAttributeExtractor<S>

- ▼  AnnotationAttributeExtractor<S>
 - ▼  AbstractAliasAwareAnnotationAttributeExtractor<S>
 -  DefaultAnnotationAttributeExtractor
 -  MapAnnotationAttributeExtractor
- ▼  AnnotationAttributeExtractor<S>
 -  `getAnnotationType() : Class<? extends Annotation>`
 -  `getAnnotatedElement() : Object`
 -  `getSource() : S`
 -  `getAttributeValue(Method) : Object`
- ▼  AbstractAliasAwareAnnotationAttributeExtractor<S>
 -  `annotationType : Class<? extends Annotation>`
 -  `annotatedElement : Object`
 -  `source : S`
 -  `attributeAliasMap : Map<String, List<String>>`
 -  `AbstractAliasAwareAnnotationAttributeExtractor(Class<? extends Annotation>, Object, S)`
 -  `getAnnotationType() : Class<? extends Annotation>`
 -  `getAnnotatedElement() : Object`
 -  `getSource() : S`
 -  `getAttributeValue(Method) : Object`
 -  `getRawAttributeValue(Method) : Object`
 -  `getRawAttributeValue(String) : Object`

47、 public class ConvertingPropertyEditorAdapter

Adapter that exposes a `java.beans.PropertyEditor` for any given `org.springframework.core.convert.ConversionService` and specific target type.

```
public class ConvertingPropertyEditorAdapter extends PropertyEditorSupport {  
  
    private final ConversionService conversionService;  
  
    private final TypeDescriptor targetDescriptor;  
  
    private final boolean canConvertToString;
```

```
public class DefaultConversionService extends  
GenericConversionService {  
  
    public static ConversionService getSharedInstance() {  
        if (sharedInstance == null) {  
            synchronized (DefaultConversionService.class) {  
                if (sharedInstance == null) {  
                    sharedInstance = new DefaultConversionService();  
                }  
            }  
        }  
        return sharedInstance;  
    }  
}
```

48、 public interface InputStreamSource

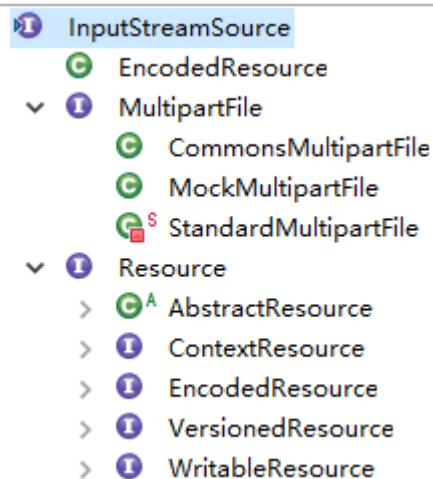
`InputStream getInputStream()` throws `IOException`;






































































Simple interface for objects that are sources for an `InputStream`.

This is the base interface for Spring's more extensive `Resource` interface.

For single-use streams, `InputStreamResource` can be used for any given `InputStream`. Spring's `ByteArrayResource` or any file-based `Resource` implementation can be used as a concrete instance, allowing one to read the underlying content stream multiple times.

This makes this interface useful as an abstract content source for mail attachments, for example.



-  InputStreamSource
 -  EncodedResource
 - 
 MultipartFile
 -  CommonsMultipartFile
 -  MockMultipartFile
 -   StandardMultipartFile
 - 
 Resource
 -   AbstractResource
 - 
 AbstractFileResolvingResource
 - 
 ClassPathResource
 -    ClassPathContextResource
 -   ClassRelativeContextResource
 -  ServletContextResource
 -  UrlResource
 -  BeanDefinitionResource
 - 
 ByteArrayResource
 -  TransformedResource
 -  DescriptiveResource
 -   FileNameVersionedResource
 - 
 FileSystemResource
 -   FileSystemContextResource
 -     GzippedResource
 -  InputStreamResource
 -  PathResource
 -  VfsResource
 - 
 ContextResource
 -    ClassPathContextResource
 -   ClassRelativeContextResource
 -   FileSystemContextResource
 -  ServletContextResource
 - 
 EncodedResource
 -     GzippedResource
 - 
 VersionedResource
 -   FileNameVersionedResource
 - 
 WritableResource
 - 
 FileSystemResource
 -    FileSystemContextResource
 -  PathResource

Holder that combines a {@link Resource} descriptor with a specific encoding or {@code Charset} to be used for reading from the resource.

Used as an argument for operations that support reading content with a specific encoding, typically via a {@code java.io.Reader}.

EncodedResource

- ▣ ^F resource : Resource
- ▣ ^F encoding : String
- ▣ ^F charset : Charset
- ^C EncodedResource(Resource)
- ^C EncodedResource(Resource, String)
- ^C EncodedResource(Resource, Charset)
- ^C EncodedResource(Resource, String, Charset)
- ^F getResource() : Resource
- ^F getEncoding() : String
- ^F getCharset() : Charset
- requiresReader() : boolean
- getReader() : Reader
- [▲] getInputStream() : InputStream
- [▲] equals(Object) : boolean
- [▲] hashCode() : int
- [▲] toString() : String

A representation of an uploaded file received in a multipart request.

The file contents are either stored in memory or temporarily on disk.

In either case, the user is responsible for copying file contents to a session-level or persistent store as and if desired. The temporary storage will be cleared at the end of request processing.

MultipartFile

- ^A getName() : String
- ^A getOriginalFilename() : String
- ^A getContentType() : String
- ^A isEmpty() : boolean
- ^A getSize() : long
- ^A getBytes() : byte[]
- ^A [▲] getInputStream() : InputStream
- ^A transferTo(File) : void

FileItem

- ^A getInputStream() : InputStream
- ^A getContentType() : String
- ^A getName() : String
- ^A isInMemory() : boolean
- ^A getSize() : long
- ^A get() : byte[]
- ^A getString(String) : String
- ^A getString() : String
- ^A write(File) : void
- ^A delete() : void
- ^A getFieldName() : String
- ^A setFieldName(String) : void
- ^A isFormField() : boolean
- ^A setFormField(boolean) : void
- ^A getOutputStream() : OutputStream

public interface FileItem extends
Serializable, FileItemHeadersSupport

```

public interface FileItemHeadersSupport {
    FileItemHeaders getHeaders();
    void setHeaders(FileItemHeaders headers);
}

public interface FileItemHeaders {
    String getHeader(String name);
    Iterator<String> getHeaders(String name);
    Iterator<String> getHeaderNames();
}

```

49、 public interface Resource extends InputStreamSource

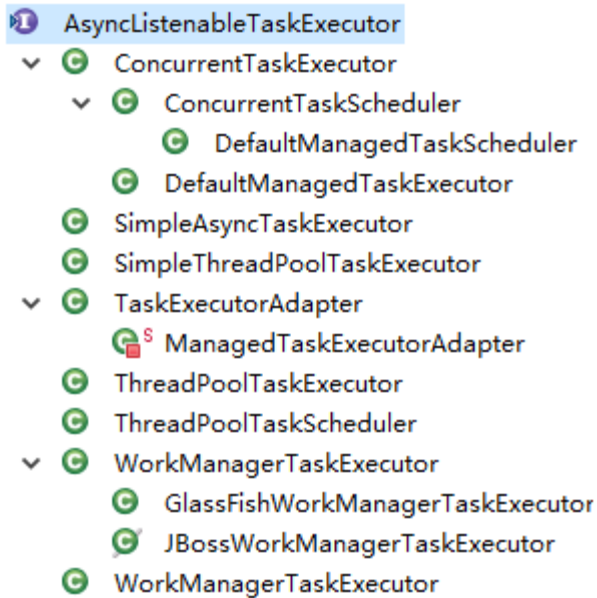
Interface for a resource descriptor that abstracts from the actual type of underlying resource, such as a file or class path resource.

An InputStream can be opened for every resource if it exists in physical form, but a URL or File handle can just be returned for certain resources. The actual behavior is implementation-specific.

Resource

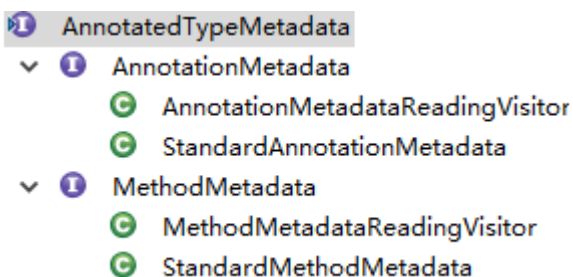
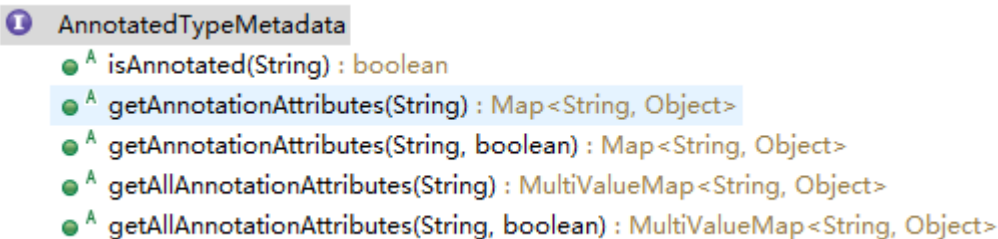
- ^A exists() : boolean
- ^A isReadable() : boolean
- ^A isOpen() : boolean
- ^A getURL() : URL
- ^A getURI() : URI
- ^A getFile() : File
- ^A contentLength() : long
- ^A lastModified() : long
- ^A createRelative(String) : Resource
- ^A getFilename() : String
- ^A getDescription() : String

50、 public interface AsyncListenableTaskExecutor extends AsyncTaskExecutor



51、 public interface AnnotatedTypeMetadata

Defines access to the annotations of a specific type (`{@link AnnotationMetadata class}` or `{@link MethodMetadata method}`), in a form that does not necessarily require the class-loading.



52、 public interface MethodMetadata

Interface that defines abstract access to the annotations of a specific class, in a form that does not require that class to be loaded yet.

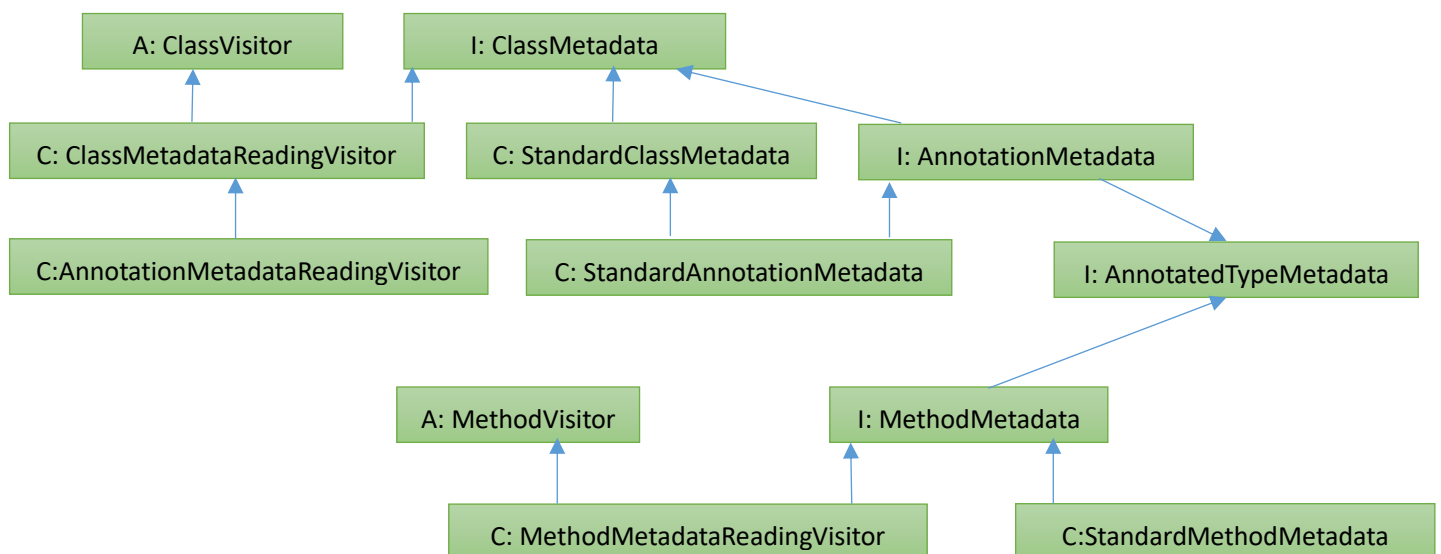
```
public interface MethodMetadata extends AnnotatedTypeMetadata {  
    String getMethodName();  
    String getDeclaringClassName();  
    String getReturnTypeName();  
    boolean isAbstract();  
    boolean isStatic();  
    boolean isFinal();  
    boolean isOverridable();  
}
```

53、 public interface AnnotationMetadata

Interface that defines abstract access to the annotations of a specific class, in a form that does not require that class to be loaded yet.

```
public interface AnnotationMetadata extends ClassMetadata, AnnotatedTypeMetadata {  
    Set<String> getAnnotationTypes();  
    Set<String> getMetaAnnotationTypes(String annotationName);  
    boolean hasAnnotation(String annotationName);  
    boolean hasMetaAnnotation(String metaAnnotationName);  
    boolean hasAnnotatedMethods(String annotationName);  
    Set<MethodMetadata> getAnnotatedMethods(String annotationName);  
}
```

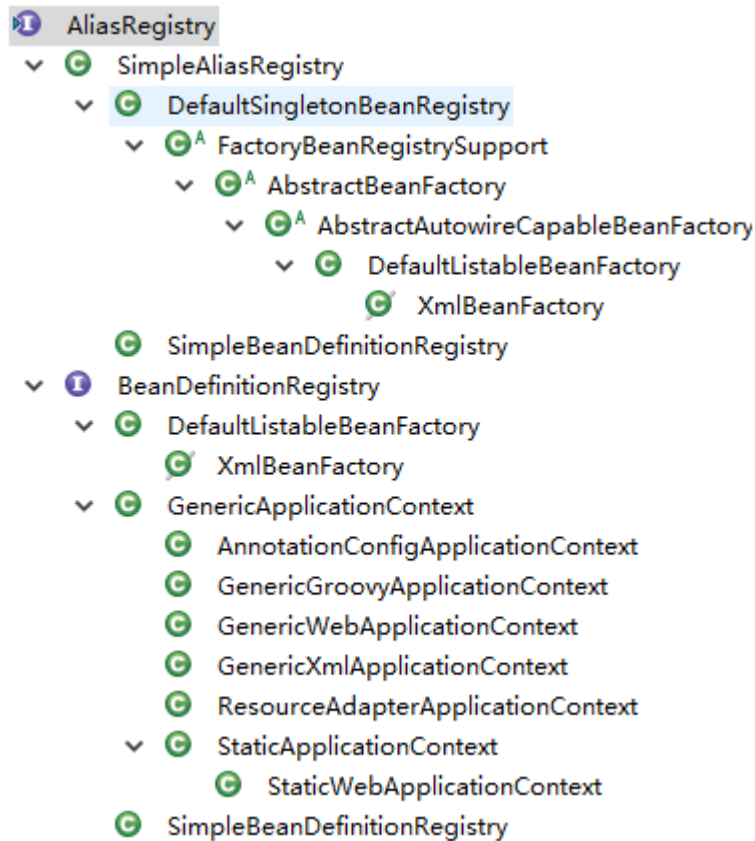
52、 public interface ClassMetadata



53、 public interface AliasRegistry

Common interface for managing aliases. Serves as super-interface for
{@link org.springframework.beans.factory.support.BeanDefinitionRegistry}.

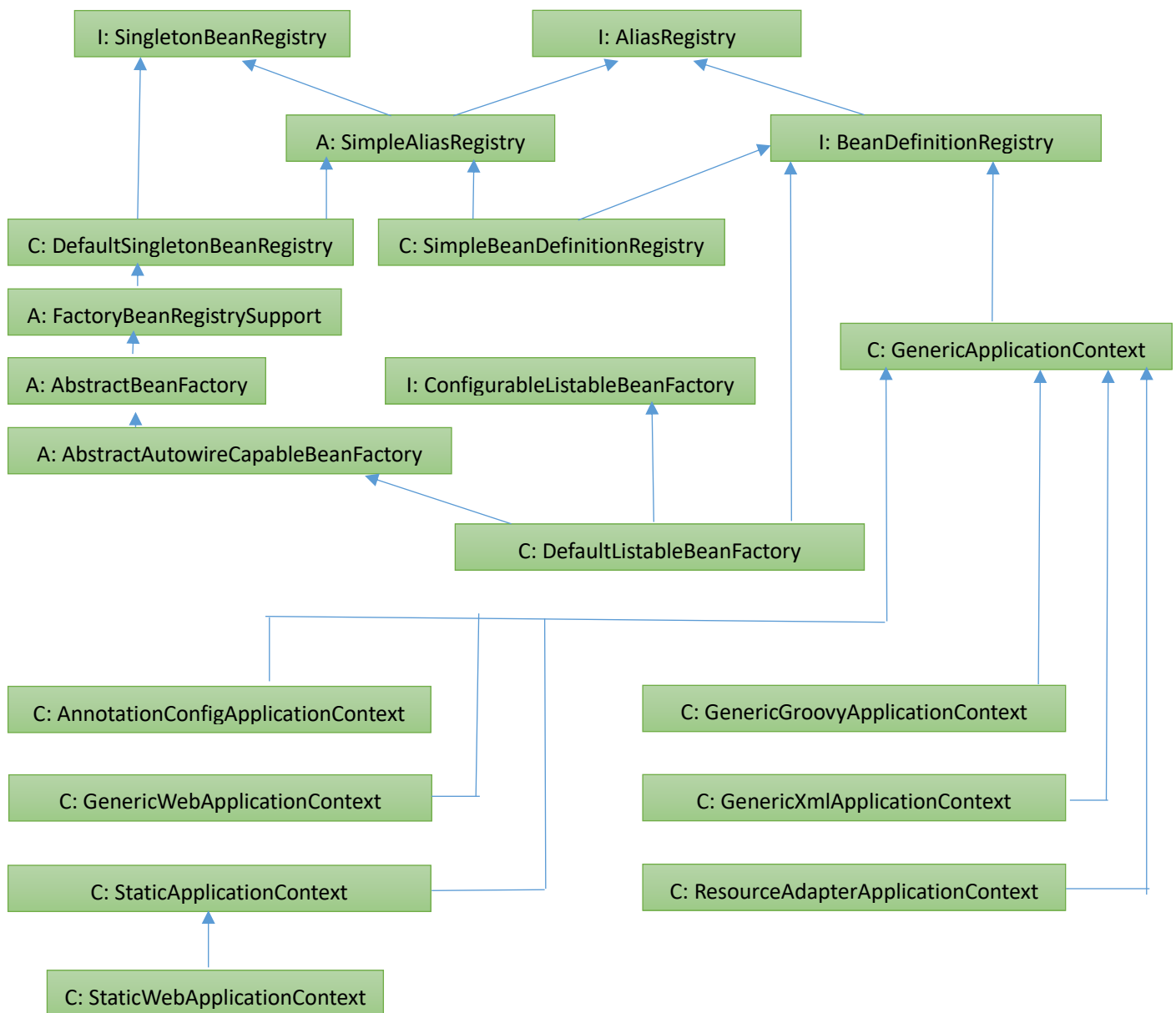
```
public interface AliasRegistry {  
    void registerAlias(String name, String alias);  
    void removeAlias(String alias);  
    boolean isAlias(String name);  
    String[] getAliases(String name);  
}
```



54、 public interface SingletonBeanRegistry

Interface that defines a registry for shared bean instances. Can be implemented by `{@link org.springframework.beans.factory.BeanFactory}` implementations in order to expose their singleton management facility in a uniform manner.

```
public interface SingletonBeanRegistry {  
    void registerSingleton(String beanName, Object singletonObject);  
    Object getSingleton(String beanName);  
    boolean containsSingleton(String beanName);  
    String[] getSingletonNames();  
    int getSingletonCount();  
    Object getSingletonMutex();  
}
```

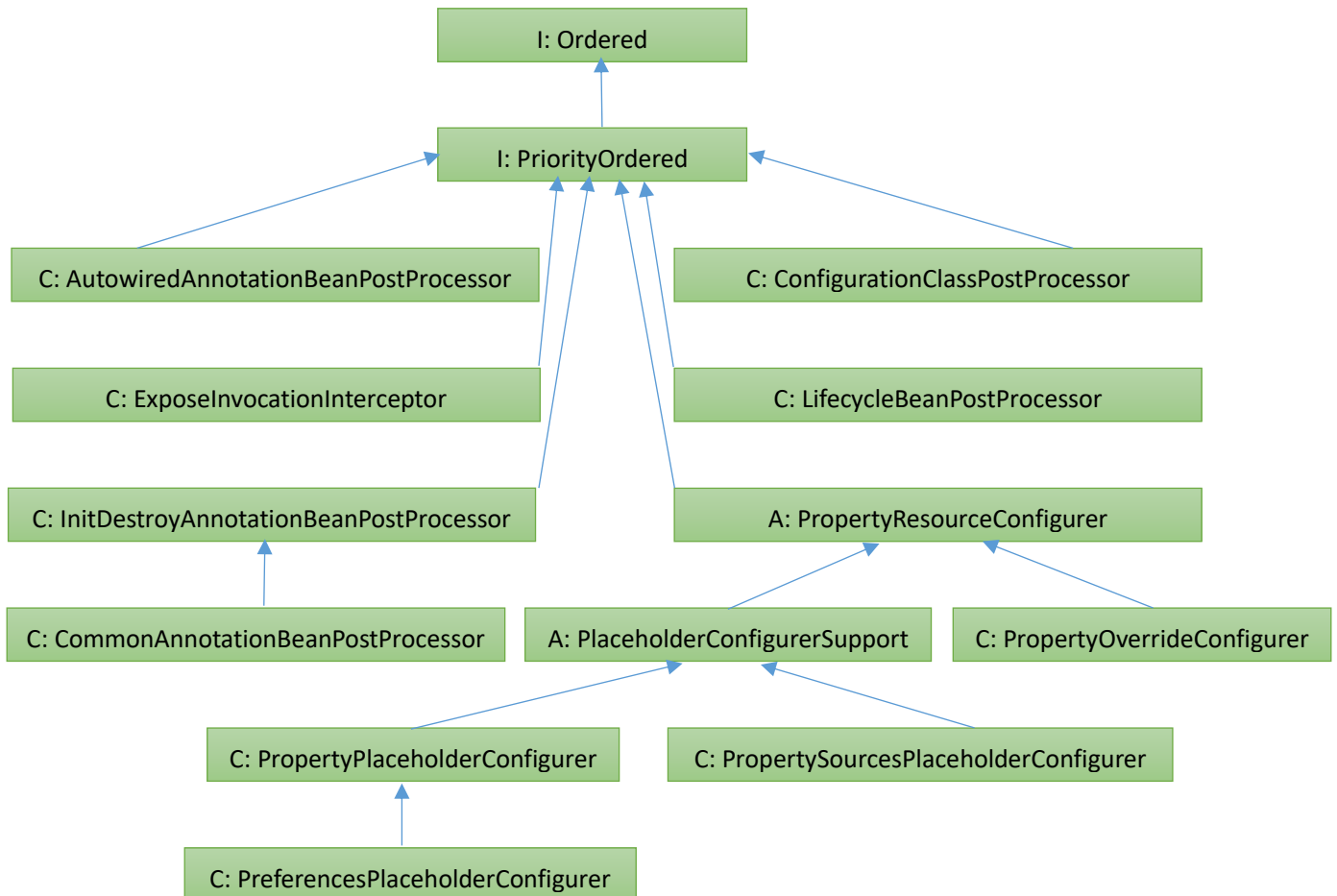


55、 public interface AttributeAccessor

Interface defining a generic contract for attaching and accessing metadata to/from arbitrary objects.

```
public interface AttributeAccessor {  
    void setAttribute(String name, Object value);  
    Object getAttribute(String name);  
    Object removeAttribute(String name);  
    boolean hasAttribute(String name);  
    String[] attributeNames();  
}  
  
public abstract class AttributeAccessorSupport implements AttributeAccessor,  
Serializable
```

56、 public interface PriorityOrdered



57、public interface ErrorHandler

A strategy for handling errors. This is especially useful for handling errors that occur during asynchronous execution of tasks that have been submitted to a `TaskScheduler`. In such cases, it may not be possible to throw the error to the original caller.

58、public class FastByteArrayOutputStream

A speedy alternative to `{@link java.io.ByteArrayOutputStream}`. Note that this variant does *not* extend `{@code ByteArrayOutputStream}`, unlike its sibling `{@link ResizableByteArrayOutputStream}`.

Unlike `{@link java.io.ByteArrayOutputStream}`, this implementation is backed by a `{@link java.util.LinkedList}` of `{@code byte[]}` instead of 1 constantly resizing `{@code byte[]}`. It does not copy buffers when it gets expanded.

The initial buffer is only created when the stream is first written. There is also no copying of the internal buffer if its contents is extracted with the `{@link #writeTo(OutputStream)}` method.

```
public class FastByteArrayOutputStream extends OutputStream
```

59、public class ResizableByteArrayOutputStream

```
public class ResizableByteArrayOutputStream extends ByteArrayOutputStream
```

An extension of `{@link java.io.ByteArrayOutputStream}` that has public `{@link org.springframework.util.ResizableByteArrayOutputStream#grow(int)}` and `{@link org.springframework.util.ResizableByteArrayOutputStream#resize(int)}` methods to get more control over the size of the internal buffer has a higher initial capacity (256) by default

As of 4.2, this class has been superseded by `{@link FastByteArrayOutputStream}` for Spring's internal use where no assignability to `{@link ByteArrayOutputStream}` is needed (since `{@link FastByteArrayOutputStream}` is more efficient with buffer resize management but doesn't extend the standard `{@link ByteArrayOutputStream}`).

60、public class InstanceFilter<T>

A simple instance filter that checks if a given instance match based on a collection of includes and excludes element.

Subclasses may want to override `{@link #match(Object, Object)}` to provide a custom matching algorithm.

61、 public class MethodInvocationer

Helper class that allows for specifying a method to invoke in a declarative fashion, be it static or non-static.

Usage: Specify "targetClass"/"targetMethod" or "targetObject"/"targetMethod", optionally specify arguments, prepare the invoker. Afterwards, you may invoke the method any number of times, obtaining the invocation result.

62、 public class PropertyPlaceholderHelper

Utility class for working with Strings that have placeholder values in them. A placeholder takes the form {@code \${name}}. Using {@code PropertyPlaceholderHelper} these placeholders can be substituted for user-supplied values. <p> Values for substitution can be supplied using a {@link Properties} instance or using a {@link PlaceholderResolver}.

63、 public interface Ordered

{@code Ordered} is an interface that can be implemented by objects that should be orderable, for example in a {@code Collection}.

The actual {@link #getOrder() order} can be interpreted as prioritization, with the first object (with the lowest order value) having the highest priority.

Note that there is also a priority marker for this interface: {@link PriorityOrdered}. Order values expressed by {@code PriorityOrdered} objects always apply before same order values expressed by plain {@link Ordered} objects.

Consult the Javadoc for {@link OrderComparator} for details on the sort semantics for non-ordered objects.

```
public interface Ordered {  
    int HIGHEST_PRECEDENCE = Integer.MIN_VALUE;  
    int LOWEST_PRECEDENCE = Integer.MAX_VALUE;  
    int getOrder();  
}
```

03、Spring-beans

01、 public abstract class AnnotationBeanUtils

General utility methods for working with annotations in JavaBeans style.

```
public static void copyPropertiesToBean(Annotation ann, Object bean, StringValueResolver valueResolver, String... excludedProperties) {
    Set<String> excluded = new HashSet<String>(Arrays.asList(excludedProperties));
    Method[] annotationProperties = ann.annotationType().getDeclaredMethods();
    BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(bean);
    for (Method annotationProperty : annotationProperties) {
        String propertyName = annotationProperty.getName();
        if (!excluded.contains(propertyName) && bw.isWritableProperty(propertyName)) {
            Object value = ReflectionUtils.invokeMethod(annotationProperty, ann);
            if (valueResolver != null && value instanceof String) {
                value = valueResolver.resolveStringValue((String) value);
            }
            bw.setPropertyValue(propertyName, value);
        }
    }
}
```

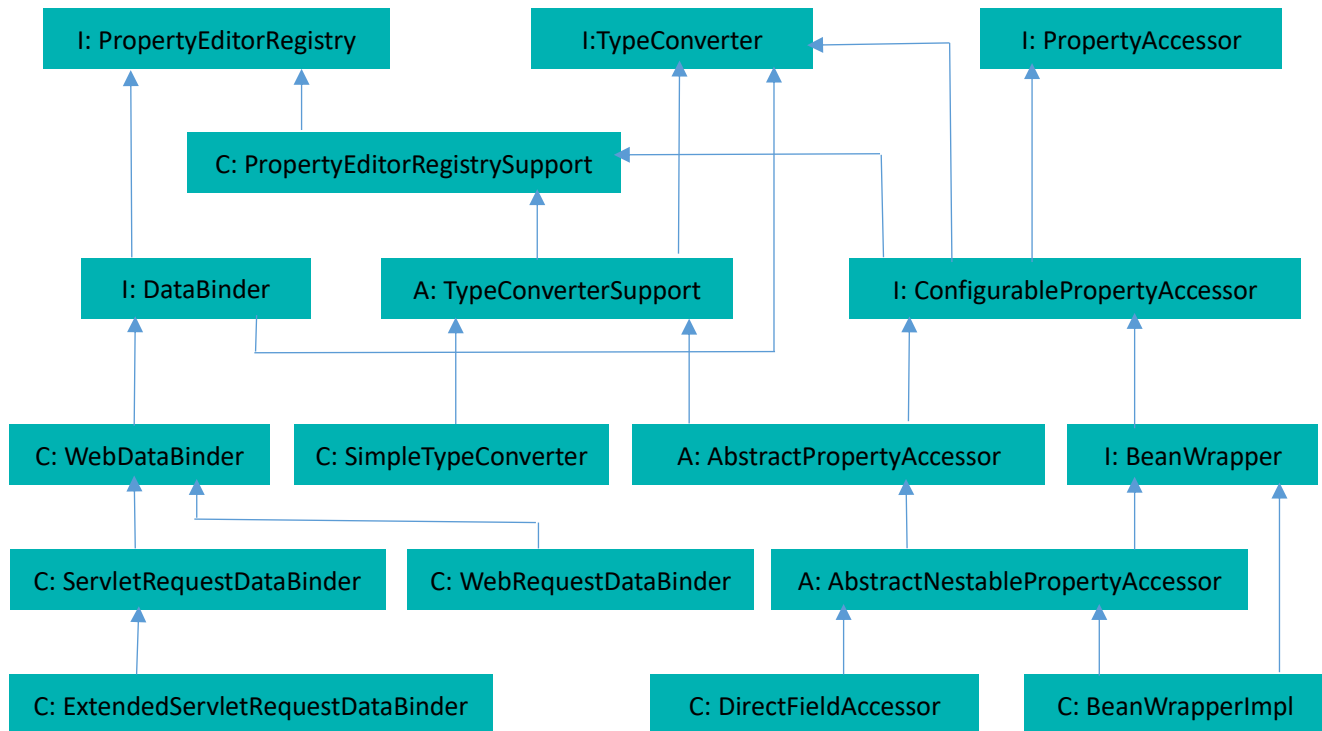
02、 public interface TypeConverter

- ❶ TypeConverter
 - ▼ ③ DataBinder
 - ▼ ③ WebDataBinder
 - > ③ ServletRequestDataBinder
 - ③ WebRequestDataBinder
 - ▼ ③^A TypeConverterSupport
 - ▼ ③^A AbstractPropertyAccessor
 - ▼ ③^A AbstractNestablePropertyAccessor
 - ③ BeanWrapperImpl
 - ③ DirectFieldAccessor
 - ③ SimpleTypeConverter
 - ▼ ❶ ConfigurablePropertyAccessor
 - ▼ ③^A AbstractPropertyAccessor
 - ▼ ③^A AbstractNestablePropertyAccessor
 - ③ BeanWrapperImpl
 - ③ DirectFieldAccessor
 - ▼ ❶ BeanWrapper
 - ③ BeanWrapperImpl

Interface that defines type conversion methods. Typically (but not necessarily) implemented in conjunction with the {@link PropertyEditorRegistry} interface.

Note: Since TypeConverter implementations are typically based on {@link java.beans.PropertyEditor PropertyEditors} which aren't thread-safe, TypeConverters themselves are *not* to be considered as thread-safe either.

- ❶ TypeConverter
 - ③^A convertIfNecessary(Object, Class<T>) <T> : T
 - ③^A convertIfNecessary(Object, Class<T>, MethodParameter) <T> : T
 - ③^A convertIfNecessary(Object, Class<T>, Field) <T> : T



03、 public interface PropertyEditorRegistry

Encapsulates methods for registering JavaBeans {@link PropertyEditor PropertyEditors}.

This is the central interface that a {@link PropertyEditorRegistrar} operates on.
 Extended by {@link BeanWrapper}; implemented by {@link BeanWrapperImpl}
 and {@link org.springframework.validation.DataBinder}.

```

PropertyEditorRegistry
  ● A registerCustomEditor(Class<?>, PropertyEditor) : void
  ● A registerCustomEditor(Class<?>, String, PropertyEditor) : void
  ● A findCustomEditor(Class<?>, String) : PropertyEditor
  
```

04、 public class PropertyEditorRegistrySupport

Base implementation of the {@link PropertyEditorRegistry} interface.
 Provides management of default editors and custom editors.
 Mainly serves as base class for {@link BeanWrapperImpl}.

05、public abstract class TypeConverterSupport

Base implementation of the `{@link TypeConverter}` interface, using a package-private delegate. Mainly serves as base class for `{@link BeanWrapperImpl}`.

```
GA TypeConverterSupport
  ● C TypeConverterSupport()
  ▲ typeConverterDelegate : TypeConverterDelegate
  ● ▲ convertIfNecessary(Object, Class<T>) <T> : T
  ● ▲ convertIfNecessary(Object, Class<T>, MethodParameter) <T> : T
  ● ▲ convertIfNecessary(Object, Class<T>, Field) <T> : T
  ■ doConvert(Object, Class<T>, MethodParameter, Field) <T> : T
```

06、class TypeConverterDelegate

Internal helper class for converting property values to target types.


















Works on a given `{@link PropertyEditorRegistrySupport}` instance.

Used as a delegate by `{@link BeanWrapperImpl}` and `{@link SimpleTypeConverter}`.

```
G TypeConverterDelegate
  ■ SF logger : Log
  ■ S javaUtilOptionalEmpty : Object
  ■ F propertyEditorRegistry : PropertyEditorRegistrySupport
  ■ F targetObject : Object
  ● C TypeConverterDelegate(PropertyEditorRegistrySupport)
  ● C TypeConverterDelegate(PropertyEditorRegistrySupport, Object)
  ● convertIfNecessary(Object, Class<T>, MethodParameter) <T> : T
  ● convertIfNecessary(Object, Class<T>, Field) <T> : T
  ● convertIfNecessary(String, Object, Object, Class<T>) <T> : T
  ● convertIfNecessary(String, Object, Object, Class<T>, TypeDescriptor) <T> : T
  ■ attemptToConvertStringToEnum(Class<?>, String, Object) : Object
  ■ findDefaultEditor(Class<?>) : PropertyEditor
  ■ doConvertValue(Object, Object, Class<?>, PropertyEditor) : Object
  ■ doConvertTextValue(Object, String, PropertyEditor) : Object
  ■ convertToTypedArray(Object, String, Class<?>) : Object
  ■ convertToTypedCollection(Collection<?>, String, Class<?>, TypeDescriptor) : Collection<?>
  ■ convertToTypedMap(Map<?, ?>, String, Class<?>, TypeDescriptor) : Map<?, ?>
  ■ buildIndexedPropertyName(String, int) : String
  ■ buildKeyedPropertyName(String, Object) : String
  ■ canCreateCopy(Class<?>) : boolean
```

07、 public interface PropertyAccessor

PropertyAccessor

 NESTED_PROPERTY_SEPARATOR : String	Common interface for classes that can access named properties (such as bean properties of an object or fields in an object) Serves as base interface for {@link BeanWrapper}.
 NESTED_PROPERTY_SEPARATOR_CHAR : char	
 PROPERTY_KEY_PREFIX : String	
 PROPERTY_KEY_PREFIX_CHAR : char	
 PROPERTY_KEY_SUFFIX : String	
 PROPERTY_KEY_SUFFIX_CHAR : char	
 isReadableProperty(String) : boolean	
 isWritableProperty(String) : boolean	
 getPropertyType(String) : Class<?>	
 getPropertyTypeDescriptor(String) : TypeDes	
 getPropertyValue(String) : Object	
 setPropertyValue(String, Object) : void	
 setPropertyValue(PropertyValue) : void	
 setPropertyValues(Map<?, ?>) : void	
 setPropertyValues(PropertyValues) : void	
 setPropertyValues(PropertyValues, boolean) :	
 setPropertyValues(PropertyValues, boolean, l	







08、 public interface ConfigurablePropertyAccessor

Interface that encapsulates configuration methods for a PropertyAccessor.

Also extends the PropertyEditorRegistry interface, which defines methods for PropertyEditor management.

```
public interface ConfigurablePropertyAccessor extends PropertyAccessor,
PropertyEditorRegistry, TypeConverter {
    void setConversionService(ConversionService conversionService);
    ConversionService getConversionService();
    void setExtractOldValueForEditor(boolean extractOldValueForEditor);
    boolean isExtractOldValueForEditor();
    void setAutoGrowNestedPaths(boolean autoGrowNestedPaths);
    boolean isAutoGrowNestedPaths();
}
```

ConfigurablePropertyAccessor

 setConversionService(ConversionService) : void
 getConversionService() : ConversionService
 setExtractOldValueForEditor(boolean) : void
 isExtractOldValueForEditor() : boolean
 setAutoGrowNestedPaths(boolean) : void
 isAutoGrowNestedPaths() : boolean

09、 public abstract class AbstractPropertyAccessor

Abstract implementation of the `{@link PropertyAccessor}` interface.

Provides base implementations of all convenience methods, with the implementation of actual property access left to subclasses.

```
AbstractPropertyAccessor
  AbstractPropertyAccessor()
    extractOldValueForEditor : boolean
    autoGrowNestedPaths : boolean
    setExtractOldValueForEditor(boolean) : void
    isExtractOldValueForEditor() : boolean
    setAutoGrowNestedPaths(boolean) : void
    isAutoGrowNestedPaths() : boolean
    setPropertyValue(PropertyValue) : void
    setPropertyValues(Map<?, ?>) : void
    setPropertyValues(PropertyValues) : void
    setPropertyValues(PropertyValues, boolean) : void
    setPropertyValues(PropertyValues, boolean, boolean) : void
    getPropertyType(String) : Class<?>
    A getPropertyValue(String) : Object
    A setPropertyValue(String, Object) : void
```

10、 public abstract class AbstractNestablePropertyAccessor

A basic `{@link ConfigurablePropertyAccessor}` that provides the necessary infrastructure for all typical use cases.

This accessor will convert collection and array values to the corresponding target collections or arrays, if necessary. Custom property editors that deal with collections or arrays can either be written via `PropertyEditor`'s `{@code setValue}`, or against a comma-delimited String via `{@code setAsText}`, as String arrays are converted in such a format if the array itself is not assignable.

11、 public class DirectFieldAccessor

{@link ConfigurablePropertyAccessor} implementation that directly accesses instance fields. Allows for direct binding to fields instead of going through JavaBean setters.

As of Spring 4.2, the vast majority of the {@link BeanWrapper} features have been merged to {@link AbstractPropertyAccessor}, which means that property traversal as well as collections and map access is now supported here as well.

A DirectFieldAccessor's default for the "extractOldValueForEditor" setting is "true", since a field can always be read without side effects.

```
DirectFieldAccessor
  F fieldMap : Map<String, FieldPropertyHandler>
  DirectFieldAccessor(Object)
  DirectFieldAccessor(Object, String, DirectFieldAccessor)
  getLocalPropertyHandler(String) : FieldPropertyHandler
  newNestedPropertyAccessor(Object, String) : DirectFieldAccessor
  createNotWritablePropertyException(String) : NotWritablePropertyException
  FieldPropertyHandler
    F field : Field
    FieldPropertyHandler(DirectFieldAccessor, Field)
    toTypeDescriptor() : TypeDescriptor
    getResolvableType() : ResolvableType
    nested(int) : TypeDescriptor
    getValue() : Object
    setValue(Object, Object) : void
```

12、 public interface BeanWrapper

The central interface of Spring's low-level JavaBeans infrastructure. Typically not used directly but rather implicitly via a {@link org.springframework.beans.factory.BeanFactory} or a {@link org.springframework.validation.DataBinder}.

Provides operations to analyze and manipulate standard JavaBeans: the ability to get and set property values (individually or in bulk), get property descriptors, and query the readability/writability of properties.

This interface supports nested properties enabling the setting of properties on subproperties to an unlimited depth.

A BeanWrapper's default for the "extractOldValueForEditor" setting is "false", to avoid side effects caused by getter method invocations.

Turn this to "true" to expose present property values to custom editors.

```
public interface BeanWrapper extends ConfigurablePropertyAccessor {
    void setAutoGrowCollectionLimit(int autoGrowCollectionLimit);
    int getAutoGrowCollectionLimit();
    Object getWrappedInstance();
}
```

```

Class<?> getWrappedClass();

PropertyDescriptor[] getPropertyDescriptors();

PropertyDescriptor getPropertyDescriptor(String propertyName) throws InvalidPropertyException;

}

```

13、 public class BeanWrapperImpl

Default {@link BeanWrapper} implementation that should be sufficient for all typical use cases. Caches introspection results for efficiency.

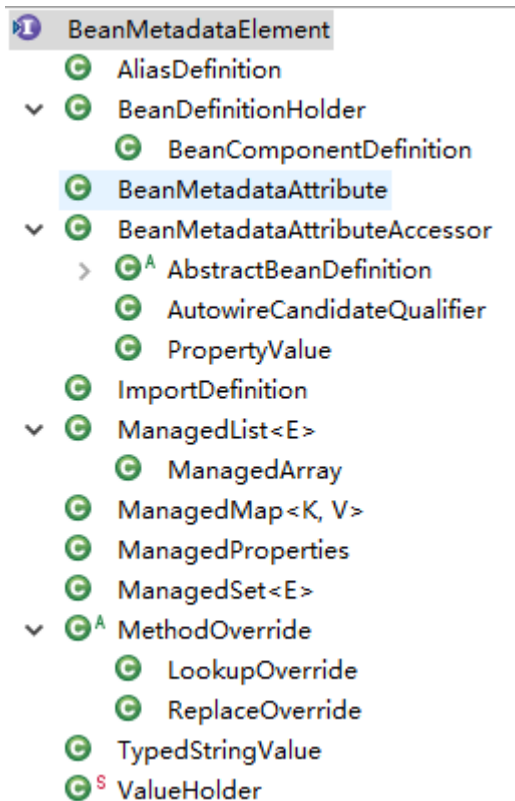
Note: Auto-registers default property editors from the `org.springframework.beans.propertyeditors` package, which apply in addition to the JDK's standard PropertyEditors. Applications can call the `registerCustomEditor(Class, java.beans.PropertyEditor)` method to register an editor for a particular instance (i.e. they are not shared across the application). See the base class `PropertyEditorRegistrySupport` for details.

NOTE: As of Spring 2.5, this is - for almost all purposes - an internal class. It is just public in order to allow for access from other framework packages. For standard application access purposes, use the

`PropertyAccessorFactory#forBeanPropertyAccess` factory method instead.

14、 public interface BeanMetadataElement

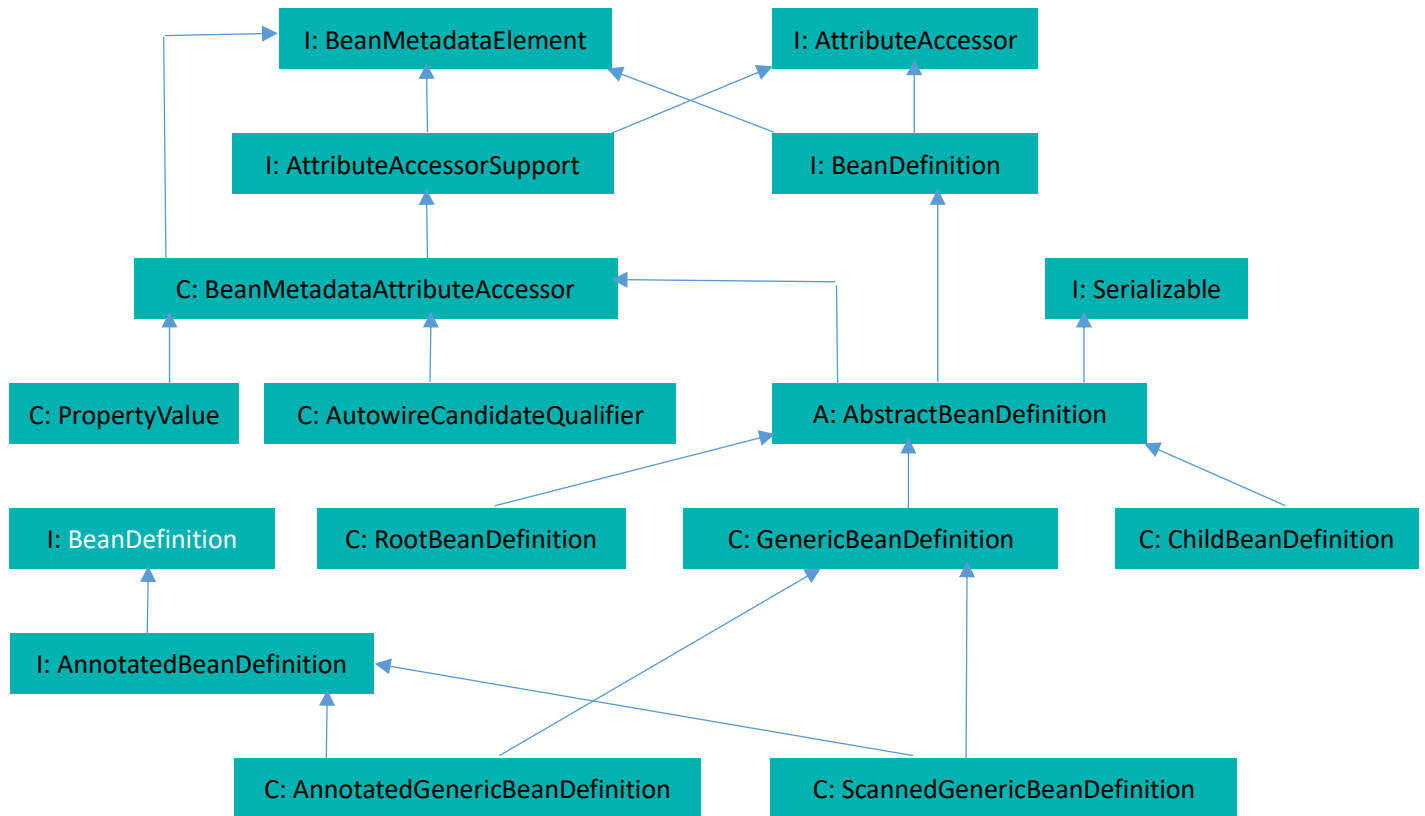
Interface to be implemented by bean metadata elements that carry a configuration source object.



- ▼ ⓘ BeanDefinition
 - ▼ ⓘ^A AbstractBeanDefinition
 - ↳ ChildBeanDefinition
 - ▼ ⓘ GenericBeanDefinition
 - ↳ AnnotatedGenericBeanDefinition
 - ↳ ScannedGenericBeanDefinition
 - ▼ ⓘ RootBeanDefinition
 - ↳ ⓘ^S ConfigurationClassBeanDefinition
 - ▼ ⓘ AnnotatedBeanDefinition
 - ↳ AnnotatedGenericBeanDefinition
 - ↳ ⓘ^S ConfigurationClassBeanDefinition
 - ↳ ScannedGenericBeanDefinition
- ▼ ⓘ BeanReference
 - ↳ RuntimeBeanNameReference
 - ▼ ⓘ RuntimeBeanReference
 - ↳ ⓘ GroovyRuntimeBeanReference
- ▼ ⓘ ComponentDefinition
 - ▼ ⓘ^A AbstractComponentDefinition
 - ↳ AdvisorComponentDefinition
 - ▼ ⓘ CompositeComponentDefinition
 - ↳ AspectComponentDefinition
 - ↳ PointcutComponentDefinition
 - ↳ BeanComponentDefinition
- ▼ ⓘ DefaultsDefinition
 - ↳ DocumentDefaultsDefinition

15、 public class BeanMetadataAttributeAccessor

Extension of {@link org.springframework.core.AttributeAccessorSupport}, holding attributes as {@link BeanMetadataAttribute} objects in order to keep track of the definition source.



16、 public class PropertyValue

Object to hold information and value for an individual bean property. Using an object here, rather than just storing all properties in a map keyed by property name, allows for more flexibility, and the ability to handle indexed properties etc in an optimized way.

Note that the value doesn't need to be the final required type: A {@link BeanWrapper} implementation should handle any necessary conversion, as this object doesn't know anything about the objects it will be applied to.

17、 public class MutablePropertyValues

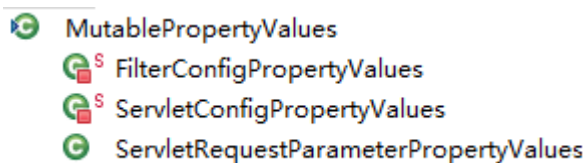
Default implementation of the [{@link PropertyValue}](#) interface.

Allows simple manipulation of properties, and provides constructors to support deep copy and construction from a Map.

```
public class MutablePropertyValues implements PropertyValue, Serializable
```

18、 public class AutowireCandidateQualifier

Qualifier for resolving autowire candidates. A bean definition that includes one or more such qualifiers enables fine-grained matching against annotations on a field or parameter to be autowired.



19、 public class BeanDefinitionHolder

```
public class BeanDefinitionHolder implements BeanMetadataElement {  
    private final BeanDefinition beanDefinition;  
    private final String beanName;  
    private final String[] aliases;
```

Holder for a BeanDefinition with name and aliases. Can be registered as a placeholder for an inner bean. Can also be used for programmatic registration of inner bean definitions. If you don't care about BeanNameAware and the like, registering RootBeanDefinition or ChildBeanDefinition is good enough.

20、 public interface BeanDefinition

A BeanDefinition describes a bean instance, which has property values, constructor argument values, and further information supplied by concrete implementations.

This is just a minimal interface: The main intention is to allow a [{@link BeanFactoryPostProcessor}](#) such as [{@link PropertyPlaceholderConfigurer}](#) to introspect and modify property values and other bean metadata.

BeanDefinition

- SCOPE_SINGLETON : String
- SCOPE_PROTOTYPE : String
- ROLE_APPLICATION : int
- ROLE_SUPPORT : int
- ROLE_INFRASTRUCTURE : int
- setParentName(String) : void
- getParentName() : String
- setBeanClassName(String) : void
- getBeanClassName() : String
- setScope(String) : void
- getScope() : String
- setLazyInit(boolean) : void
- isLazyInit() : boolean
- setDependsOn(String...) : void
- getDependsOn() : String[]
- setAutowireCandidate(boolean) : void
- isAutowireCandidate() : boolean
- setPrimary(boolean) : void
- isPrimary() : boolean
- setFactoryBeanName(String) : void
- getFactoryBeanName() : String
- setFactoryMethodName(String) : void
- getFactoryMethodName() : String
- getConstructorArgumentValues() : ConstructorArgumentValues
- getPropertyValues() : MutablePropertyValues
- isSingleton() : boolean
- isPrototype() : boolean
- isAbstract() : boolean
- getRole() : int
- getDescription() : String
- getResourceDescription() : String
- getOriginatingBeanDefinition() : BeanDefinition

21、 public interface AnnotatedBeanDefinition

Extended [{@link org.springframework.beans.factory.config.BeanDefinition}](#) interface that exposes [{@link org.springframework.core.type.AnnotationMetadata}](#) about its bean class - without requiring the class to be loaded yet.

```
public interface AnnotatedBeanDefinition extends BeanDefinition {
    AnnotationMetadata getMetadata();
    MethodMetadata getFactoryMethodMetadata();
}
```

22、 public abstract class AbstractBeanDefinition

Base class for concrete, full-fledged {@link BeanDefinition} classes, factoring out common properties of {@link GenericBeanDefinition}, {@link RootBeanDefinition}, and {@link ChildBeanDefinition}. The autowire constants match the ones defined in the {@link org.springframework.beans.factory.config.AutowireCapableBeanFactory} interface.

```
public abstract class AbstractBeanDefinition extends BeanMetadataAttributeAccessor
implements BeanDefinition, Cloneable
```

23、 public class ChildBeanDefinition

```
public class ChildBeanDefinition extends AbstractBeanDefinition {
```

Bean definition for beans which inherit settings from their parent. Child bean definitions have a fixed dependency on a parent bean definition.

A child bean definition will inherit constructor argument values, property values and method overrides from the parent, with the option to add new values. If init method, destroy method and/or static factory method are specified, they will override the corresponding parent settings.

The remaining settings will always be taken from the child definition: depends on, autowire mode, dependency check, singleton, lazy init.

NOTE: Since Spring 2.5, the preferred way to register bean definitions programmatically is the {@link GenericBeanDefinition} class, which allows to dynamically define parent dependencies through the {@link GenericBeanDefinition#setParentName} method. This effectively supersedes the ChildBeanDefinition class for most use cases.

24、 public class GenericBeanDefinition

```
public class GenericBeanDefinition extends AbstractBeanDefinition
```

GenericBeanDefinition is a one-stop shop for standard bean definition purposes. Like any bean definition, it allows for specifying a class plus optionally constructor argument values and property values. Additionally, deriving from a parent bean definition can be flexibly configured through the "parentName" property.

In general, use this {@code GenericBeanDefinition} class for the purpose of registering user-visible bean definitions (which a post-processor might operate on, potentially even reconfiguring the parent name).

Use {@code RootBeanDefinition} / {@code ChildBeanDefinition} where parent/child relationships happen to be pre-determined.

25、 public class AnnotatedGenericBeanDefinition

public class `AnnotatedGenericBeanDefinition` **extends** `GenericBeanDefinition` **implements** `AnnotatedBeanDefinition`

Extension of the [{@link org.springframework.beans.factory.support.GenericBeanDefinition}](#) class, adding support for annotation metadata exposed through the [{@link AnnotatedBeanDefinition}](#) interface.

This `GenericBeanDefinition` variant is mainly useful for testing code that expects to operate on an `AnnotatedBeanDefinition`, for example strategy implementations in Spring's component scanning support (where the default definition class is [{@link org.springframework.context.annotation.ScannedGenericBeanDefinition}](#), which also implements the `AnnotatedBeanDefinition` interface).

26、 public class ScannedGenericBeanDefinition

public class `ScannedGenericBeanDefinition` **extends** `GenericBeanDefinition` **implements** `AnnotatedBeanDefinition`

Extension of the [{@link org.springframework.beans.factory.support.GenericBeanDefinition}](#) class, based on an ASM `ClassReader`, with support for annotation metadata exposed through the [{@link AnnotatedBeanDefinition}](#) interface.

This class does *not* load the bean [{@code Class}](#) early. It rather retrieves all relevant metadata from the ".class" file itself, parsed with the ASM `ClassReader`.

It is functionally equivalent to [{@link AnnotatedGenericBeanDefinition#AnnotatedGenericBeanDefinition\(AnnotationMetadata\)}](#) but distinguishes by type beans that have been *scanned* vs those that have been otherwise registered or detected by other means.

27、 public abstract class PropertyMatches

Helper class for calculating property matches, according to a configurable distance. Provide the list of potential matches and an easy way to generate an error message. Works for both java bean properties and fields.

Mainly for use within the framework and in particular the binding facility.

28、 final class GenericTypeAwarePropertyDescriptor

final class `GenericTypeAwarePropertyDescriptor` **extends** `PropertyDescriptor`

Extension of the standard JavaBeans [{@link PropertyDescriptor}](#) class, overriding [{@code getPropertyType\(\)}](#) such that a generically declared type variable will be resolved against the containing bean class.

29、class ExtendedBeanInfo

Decorator for a standard {@link BeanInfo} object, e.g. as created by {@link Introspector#getBeanInfo(Class)}, designed to discover and register static and/or non-void returning setter methods.

The standard JavaBeans {@code Introspector} will discover the {@code getFoo} read method, but will bypass the {@code #setFoo(Foo)} write method, because its non-void returning signature does not comply with the JavaBeans specification.

{@code ExtendedBeanInfo}, on the other hand, will recognize and include it. This is designed to allow APIs with "builder" or method-chaining style setter signatures to be used within Spring {@code <beans>} XML. {@link #getPropertyDescriptors()} returns all existing property descriptors from the wrapped {@code BeanInfo} as well any added for non-void returning setters. Both standard ("non-indexed") and [indexed properties](http://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html) are fully supported.

30、public interface BeanInfoFactory

Strategy interface for creating {@link BeanInfo} instances for Spring beans.

Can be used to plug in custom bean property resolution strategies (e.g. for other languages on the JVM) or more efficient {@link BeanInfo} retrieval algorithms.

BeanInfoFactories are instantiated by the {@link CachedIntrospectionResults}, by using the {@link org.springframework.core.io.support.SpringFactoriesLoader} utility class.

When a {@link BeanInfo} is to be created, the {@code CachedIntrospectionResults} will iterate through the discovered factories, calling {@link #getBeanInfo(Class)} on each one. If {@code null} is returned, the next factory will be queried.

If none of the factories support the class, a standard {@link BeanInfo} will be created as a default.

Note that the {@link org.springframework.core.io.support.SpringFactoriesLoader} sorts the {@code BeanInfoFactory} instances by {@link org.springframework.core.annotation.Order @Order}, so that ones with a higher precedence come first.

```
public interface BeanInfoFactory {  
    BeanInfo getBeanInfo(Class<?> beanClass) throws IntrospectionException;  
}
```

31、public class ExtendedBeanInfoFactory

{@link BeanInfoFactory} implementation that evaluates whether bean classes have "non-standard" JavaBeans setter methods and are thus candidates for introspection by Spring's (package-visible) {@code ExtendedBeanInfo} implementation.

Ordered at {@link Ordered#LOWEST_PRECEDENCE} to allow other user-defined {@link BeanInfoFactory} types to take precedence.

```
public class ExtendedBeanInfoFactory implements BeanInfoFactory, Ordered {
    @Override
    public BeanInfo getBeanInfo(Class<?> beanClass) throws IntrospectionException {
        return (supports(beanClass) ? new ExtendedBeanInfo(Introspector.getBeanInfo(beanClass)) : null);
    }
    private boolean supports(Class<?> beanClass) {
        for (Method method : beanClass.getMethods()) {
            if (ExtendedBeanInfo.isCandidateWriteMethod(method)) {
                return true;
            }
        }
        return false;
    }
    @Override
    public int getOrder() {
        return Ordered.LOWEST_PRECEDENCE;
    }
}
```

32、public abstract class PropertyAccessorUtils

33、class PropertyDescriptorUtils

Common delegate methods for Spring's internal {@link PropertyDescriptor} implementations.

34、public interface PropertyEditorRegistrar

Interface for strategies that register custom {@link java.beans.PropertyEditor} property editors with a {@link org.springframework.beans.PropertyEditorRegistry} property editor registry.

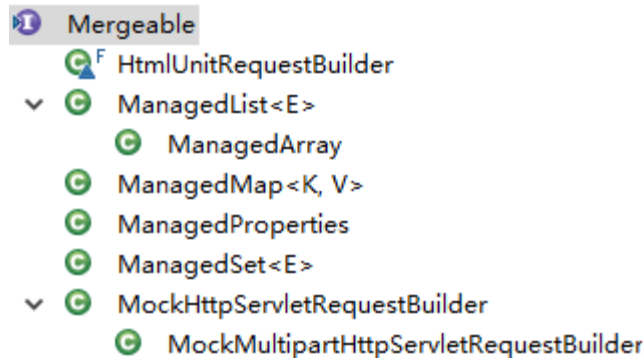
This is particularly useful when you need to use the same set of property editors in several different situations: write a corresponding registrar and reuse that in each case.

```
public interface PropertyEditorRegistrar {
    void registerCustomEditors(PropertyEditorRegistry registry);
}
```

35、public interface Mergeable

Interface representing an object whose value set can be merged with that of a parent object.

```
public interface Mergeable {  
    boolean isMergeEnabled();  
    Object merge(Object parent);  
}
```



36、public class ResourceEditorRegistrar

PropertyEditorRegistrar implementation that populates a given
{@link org.springframework.beans.PropertyEditorRegistry}
(typically a {@link org.springframework.beans.BeanWrapper} used for bean
creation within an {@link org.springframework.context.ApplicationContext})
with resource editors. Used by
{@link org.springframework.context.support.AbstractApplicationContext}.

```
public class ResourceEditorRegistrar implements PropertyEditorRegistrar {
```

37、public class PropertyValuesEditor

{@link java.beans.PropertyEditor Editor} for a {@link PropertyValues} object.
The required format is defined in the {@link java.util.Properties}
documentation. Each property must be on a new line.

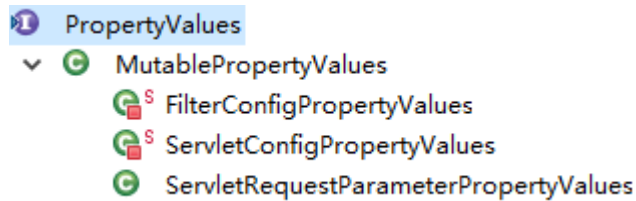
The present implementation relies on a
{@link org.springframework.beans.propertyeditors.PropertiesEditor} underneath.

```
public class PropertyValuesEditor extends PropertyEditorSupport {  
    private final PropertiesEditor propertiesEditor = new PropertiesEditor();  
    @Override  
    public void setAsText(String text) throws IllegalArgumentException {  
        this.propertiesEditor.setAsText(text);  
        Properties props = (Properties) this.propertiesEditor.getValue();  
        setValue(new MutablePropertyValues(props));  
    }  
}
```

38、 public interface PropertyValues

Holder containing one or more {@link PropertyValue} objects, typically comprising one update for a specific target bean.

```
public interface PropertyValues {  
    PropertyValue[] getPropertyValues();  
    PropertyValue getPropertyValue(String propertyName);  
    PropertyValues changesSince(PropertyValues old);  
    boolean contains(String propertyName);  
    boolean isEmpty();  
}
```



39、 public class ArgumentConvertingMethodInvoker

Subclass of {@link MethodInvocation} that tries to convert the given arguments for the actual target method via a {@link TypeConverter}.

Supports flexible argument conversions, in particular for invoking a specific overloaded method.

40、 public interface SortDefinition

Definition for sorting bean instances by a property.

```
public interface SortDefinition {  
    String getProperty();  
    boolean isIgnoreCase();  
    boolean isAscending();  
}
```

41、 public class PagedListHolder<E>

PagedListHolder is a simple state holder for handling lists of objects, separating them into pages. Page numbering starts with 0.

This is mainly targetted at usage in web UIs. Typically, an instance will be instantiated with a list of beans, put into the session, and exported as model. The properties can all be set/get programmatically, but the most common way will be data binding, i.e. populating the bean from request parameters. The getters will mainly be used by the view.

Supports sorting the underlying list via a [{@link SortDefinition}](#) implementation, available as property "sort". By default, a [{@link MutableSortDefinition}](#) instance will be used, toggling the ascending value on setting the same property again.

The data binding names have to be called "pageSize" and "sort.ascending", as expected by BeanWrapper. Note that the names and the nesting syntax match the respective JSTL EL expressions, like "myModelAttr.pageSize" and "myModelAttr.sort.ascending".

42、 public interface Aware

Marker superinterface indicating that a bean is eligible to be notified by the Spring container of a particular framework object through a callback-style method.

Actual method signature is determined by individual subinterfaces, but should typically consist of just one void-returning method that accepts a single argument.

Note that merely implementing [{@link Aware}](#) provides no default functionality. Rather, processing must be done explicitly, for example in a [{@link org.springframework.beans.factory.config.BeanPostProcessor BeanPostProcessor}](#). Refer to [{@link org.springframework.context.support.ApplicationContextAwareProcessor}](#) and [{@link org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory}](#) for examples of processing `*Aware` interface callbacks.

```
public interface Aware {  
}
```

43、 public interface BeanClassLoaderAware

Callback that allows a bean to be aware of the bean [{@link ClassLoader class loader}](#); that is, the class loader used by the present bean factory to load bean classes.

This is mainly intended to be implemented by framework classes which have to pick up application classes by name despite themselves potentially being loaded from a shared class loader.

```
public interface BeanClassLoaderAware extends Aware {  
    void setBeanClassLoader(ClassLoader classLoader);  
}
```

44、 public interface BeanFactory

The root interface for accessing a Spring bean container.

This is the basic client view of a bean container;further interfaces such as [{@link ListableBeanFactory}](#) And [{@link org.springframework.beans.factory.config.ConfigurableBeanFactory}](#) are available for specific purposes.

This interface is implemented by objects that hold a number of bean definitions, each uniquely identified by a String name. Depending on the bean definition,the factory will return either an independent instance of a contained object (the Prototype design pattern), or a single shared instance (a superior alternative to the Singleton design pattern, in which the instance is a singleton in the scope of the factory). Which type of instance will be returned depends on the bean factory configuration: the API is the same. Since Spring2.0, further scopes are available depending on the concrete application context (e.g. "request" and "session" scopes in a web environment).

The point of this approach is that the BeanFactory is a central registry of application components, and centralizes configuration of application components (no more do individual objects need to read properties files,for example). See chapters 4 and 11 of "Expert One-on-One J2EE Design and Development" for a discussion of the benefits of this approach.

Note that it is generally better to rely on Dependency Injection ("push" configuration) to configure application objects through setters or constructors, rather than use any form of "pull" configuration like a BeanFactory lookup. Spring's Dependency Injection functionality is implemented using this BeanFactory interface and its subinterfaces.

Normally a BeanFactory will load bean definitions stored in a configuration source (such as an XML document), and use the [{@code org.springframework.beans}](#) package to configure the beans. However, an implementation could simply return Java objects it creates as necessary directly in Java code. There are no constraints on how the definitions could be stored: LDAP, RDBMS, XML,properties file, etc. Implementations are encouraged to support references amongst beans (Dependency Injection).

In contrast to the methods in [{@link ListableBeanFactory}](#), all of the operations in this interface will also check parent factories if this is a [{@link HierarchicalBeanFactory}](#).

If a bean is not found in this factory instance,the immediate parent factory will be asked. Beans in this factory instance are supposed to override beans of the same name in any parent factory.Bean factory implementations should support the standard bean lifecycle interfaces as far as possible. The full set of initialization methods and their standard order is:

BeanNameAware's [{@code setBeanName}](#) BeanClassLoaderAware's [{@code setBeanClassLoader}](#)
BeanFactoryAware's [{@code setBeanFactory}](#) EnvironmentAware's [{@code setEnvironment}](#)
EmbeddedValueResolverAware's [{@code setEmbeddedValueResolver}](#)
ResourceLoaderAware's [{@code setResourceLoader}](#)(only applicable when running in an

application context)

ApplicationEventPublisherAware's {@code setApplicationEventPublisher}

(only applicable when running in an application context)

MessageSourceAware's {@code setMessageSource} (only applicable when running in an application context)

ApplicationContextAware's {@code setApplicationContext}(only applicable when running in an application context)

ServletContextAware's {@code setServletContext}(only applicable when running in a web application context){@code postProcessBeforeInitialization} methods of BeanPostProcessors

InitializingBean's {@code afterPropertiesSet}a custom init-method definition

{@code postProcessAfterInitialization} methods of BeanPostProcessors

On shutdown of a bean factory, the following lifecycle methods apply:

{@code postProcessBeforeDestruction} methods of DestructionAwareBeanPostProcessors

DisposableBean's {@code destroy}a custom destroy-method definition

```
public interface BeanFactory {
    String FACTORY_BEAN_PREFIX = "&";
    Object getBean(String name) throws BeansException;
    <T> T getBean(String name, Class<T> requiredType) throws BeansException;
    <T> T getBean(Class<T> requiredType) throws BeansException;
    Object getBean(String name, Object... args) throws BeansException;
    <T> T getBean(Class<T> requiredType, Object... args) throws BeansException;
    boolean containsBean(String name);
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
    boolean isTypeMatch(String name, ResolvableType typeToMatch) throws NoSuchBeanDefinitionException;
    boolean isTypeMatch(String name, Class<?> typeToMatch) throws NoSuchBeanDefinitionException;
    Class<?> getType(String name) throws NoSuchBeanDefinitionException;
    String[] getAliases(String name);
}
```

45、 public interface HierarchicalBeanFactory

Sub-interface implemented by bean factories that can be part of a hierarchy.

The corresponding {@code setParentBeanFactory} method for bean factories that allow setting the parent in a configurable fashion can be found in the ConfigurableBeanFactory interface.

```
public interface HierarchicalBeanFactory extends BeanFactory {
    BeanFactory getParentBeanFactory();
    boolean containsLocalBean(String name);
}
```

46、public interface ConfigurableBeanFactory

Configuration interface to be implemented by most bean factories. Provides facilities to configure a bean factory, in addition to the bean factory client methods in the {@link org.springframework.beans.factory.BeanFactory} interface.

This bean factory interface is not meant to be used in normal application code: Stick to {@link org.springframework.beans.factory.BeanFactory} or

{@link org.springframework.beans.factory.ListableBeanFactory} for typical needs. This extended interface is just meant to allow for framework-internal plug'n'play and for special access to bean factory configuration methods.

47、public interface BeanFactoryAware

Interface to be implemented by beans that wish to be aware of their owning {@link BeanFactory}. For example, beans can look up collaborating beans via the factory (Dependency Lookup). Note that most beans will choose to receive references to collaborating beans via corresponding bean properties or constructor arguments (Dependency Injection).

```
public interface BeanFactoryAware extends Aware {  
    void setBeanFactory(BeanFactory beanFactory) throws BeansException;  
}
```

48、public abstract class BeanFactoryUtils

Convenience methods operating on bean factories, in particular on the {@link ListableBeanFactory} interface.

Returns bean counts, bean names or bean instances, taking into account the nesting hierarchy of a bean factory (which the methods defined on the ListableBeanFactory interface don't, in contrast to the methods defined on the BeanFactory interface).

49、public interface BeanNameAware

Interface to be implemented by beans that want to be aware of their bean name in a bean factory. Note that it is not usually recommended that an object depend on its bean name, as this represents a potentially brittle dependence on external configuration, as well as a possibly unnecessary dependence on a Spring API.

```
public interface BeanNameAware extends Aware {  
    void setBeanName(String name);  
}
```


50、public interface DisposableBean

```
void destroy() throws Exception;
}
```

Interface to be implemented by beans that want to release resources on destruction. A BeanFactory is supposed to invoke the destroy method if it disposes a cached singleton. An application context is supposed to dispose all of its singletons on close.

An alternative to implementing DisposableBean is specifying a custom destroy-method, for example in an XML bean definition.

51、public interface FactoryBean<T>

```
T getObject() throws Exception;
Class<?> getObjectType();
boolean isSingleton();
}
```

Interface to be implemented by objects used within a [{@link BeanFactory}](#) which are themselves factories for individual objects. If a bean implements this interface, it is used as a factory for an object to expose, not directly as a bean instance that will be exposed itself.

NB: A bean that implements this interface cannot be used as a normal bean.

A FactoryBean is defined in a bean style, but the object exposed for bean references ([{@link #getObject\(\)}](#)) is always the object that it creates.

FactoryBeans can support singletons and prototypes, and can either create objects lazily on demand or eagerly on startup. The [{@link SmartFactoryBean}](#) interface allows for exposing more fine-grained behavioral metadata.

This interface is heavily used within the framework itself, for example for the AOP [{@link org.springframework.aop.framework.ProxyFactoryBean}](#) or the [{@link org.springframework.jndi.JndiObjectFactoryBean}](#). It can be used for custom components as well; however, this is only common for infrastructure code.

[{@code FactoryBean}](#) is a programmatic contract. Implementations are not supposed to rely on annotation-driven injection or other reflective facilities.

[{@link #getObjectType\(\)}](#) [{@link #getObject\(\)}](#) invocations may arrive early in the bootstrap process, even ahead of any post-processor setup. If you need access other beans, implement [{@link BeanFactoryAware}](#) and obtain them programmatically.

Finally, FactoryBean objects participate in the containing BeanFactory's synchronization of bean creation. There is usually no need for internal synchronization other than for purposes of lazy initialization within the FactoryBean itself (or the like).

52、public interface SmartFactoryBean<T>

```
public interface SmartFactoryBean<T> extends FactoryBean<T> {  
    boolean isPrototype();  
    boolean isEagerInit();  
}
```

Extension of the [{@link FactoryBean}](#) interface. Implementations may indicate whether they always return independent instances, for the case where their [{@link #isSingleton\(\)}](#) implementation returning `false` does not clearly indicate independent instances.

Plain [{@link FactoryBean}](#) implementations which do not implement this extended interface are simply assumed to always return independent instances if their [{@link #isSingleton\(\)}](#) implementation returns `false`; the exposed object is only accessed on demand.

NOTE: This interface is a special purpose interface, mainly for internal use within the framework and within collaborating frameworks.

In general, application-provided [FactoryBeans](#) should simply implement the plain [{@link FactoryBean}](#) interface. New methods might be added to this extended interface even in point releases.

53、public interface InitializingBean

```
void afterPropertiesSet() throws Exception;  
}
```

Interface to be implemented by beans that need to react once all their properties have been set by a [BeanFactory](#): for example, to perform custom initialization, or merely to check that all mandatory properties have been set.

An alternative to implementing [InitializingBean](#) is specifying a custom `init`-method, for example in an XML bean definition.

54、public class InjectionPoint

A simple descriptor for an injection point, pointing to a method/constructor parameter or a field. Exposed by [{@link UnsatisfiedDependencyException}](#).

55、public interface ListableBeanFactory

Extension of the [{@link BeanFactory}](#) interface to be implemented by bean factories that can enumerate all their bean instances, rather than attempting bean lookup by name one by one as requested by clients. [BeanFactory](#) implementations that preload all their bean definitions (such as XML-based factories) may implement this interface.

If this is a [{@link HierarchicalBeanFactory}](#), the return values will not take any [BeanFactory](#) hierarchy into account, but will relate only to the beans defined in the current factory. Use the [{@link BeanFactoryUtils}](#) helper class to consider beans in

ancestor factories too.

The methods in this interface will just respect bean definitions of this factory. They will ignore any singleton beans that have been registered by other means like `{@link org.springframework.beans.factory.config.ConfigurableBeanFactory}`'s `{@code registerSingleton}` method, with the exception of `{@code getBeanNamesOfType}` and `{@code getBeansOfType}` which will check such manually registered singletons too. Of course, `BeanFactory`'s `{@code getBean}` does allow transparent access to such special beans as well. However, in typical scenarios, all beans will be defined by external bean definitions anyway, so most applications don't need to worry about this differentiation.

NOTE: With the exception of `{@code getBeanDefinitionCount}` and `{@code containsBeanDefinition}`, the methods in this interface are not designed for frequent invocation. Implementations may be slow.

```
public interface ListableBeanFactory extends BeanFactory {

    boolean containsBeanDefinition(String beanName);

    int getBeanDefinitionCount();

    String[] getBeanDefinitionNames();

    String[] getBeanNamesForType(ResolvableType type);

    String[] getBeanNamesForType(Class<?> type);

    String[] getBeanNamesForType(Class<?> type, boolean includeNonSingletons, boolean allowEagerInit);

    <T> Map<String, T> getBeansOfType(Class<T> type) throws BeansException;

    <T> Map<String, T> getBeansOfType(Class<T> type, boolean includeNonSingletons, boolean
allowEagerInit) throws BeansException;

    String[] getBeanNamesForAnnotation(Class<? extends Annotation> annotationType);

    Map<String, Object> getBeansWithAnnotation(Class<? extends Annotation> annotationType) throws
BeansException;

    <A extends Annotation> A findAnnotationOnBean(String beanName, Class<A> annotationType)
throws NoSuchBeanDefinitionException;

}
```

56、public interface NamedBean

Counterpart of `{@link BeanNameAware}`. Returns the bean name of an object.

This interface can be introduced to avoid a brittle dependence on bean name in objects used with Spring IoC and Spring AOP.

```
public interface NamedBean {

    String getBeanName();

}
```

57、 public interface ObjectFactory<T>

```
T getObject() throws BeansException;  
}
```

Defines a factory which can return an Object instance (possibly shared or independent) when invoked.

This interface is typically used to encapsulate a generic factory which returns a new instance (prototype) of some target object on each invocation.

This interface is similar to `FactoryBean`, but implementations of the latter are normally meant to be defined as SPI instances in a `BeanFactory`, while implementations of this class are normally meant to be fed as an API to other beans (through injection). As such, the `getObject()` method has different exception handling behavior.

58、 public interface ObjectProvider<T>

A variant of `ObjectFactory` designed specifically for injection points, allowing for programmatic optionality and lenient not-unique handling.

```
public interface ObjectProvider<T> extends ObjectFactory<T> {  
    T getObject(Object... args) throws BeansException;  
    T getIfAvailable() throws BeansException;  
    T getIfUnique() throws BeansException;  
}
```

59、 public interface SmartInitializingSingleton

```
void afterSingletonsInstantiated();  
}
```

Callback interface triggered at the end of the singleton pre-instantiation phase during `BeanFactory` bootstrap. This interface can be implemented by singleton beans in order to perform some initialization after the regular singleton instantiation algorithm, avoiding side effects with accidental early initialization (e.g. from `ListableBeanFactory#getBeansOfType` calls).

In that sense, it is an alternative to `InitializingBean` which gets triggered right at the end of a bean's local construction phase. This callback variant is somewhat similar to `org.springframework.context.event.ContextRefreshedEvent` but doesn't require an implementation of `org.springframework.context.ApplicationListener`, with no need to filter context references across a context hierarchy etc.

It also implies a more minimal dependency on just the `beans` package and is being honored by standalone `ListableBeanFactory` implementations, not just in an `org.springframework.context.ApplicationContext` environment.

NOTE: If you intend to start/manage asynchronous tasks, preferably implement `org.springframework.context.Lifecycle` instead which offers a richer model for runtime management and allows for phased startup/shutdown.

60、 public class BeanWiringInfo

Holder for bean wiring metadata information about a particular class. Used in conjunction with the `@link org.springframework.beans.factory.annotation.Configurable` annotation and the AspectJ `@code AnnotationBeanConfigurerAspect`.

61、 public interface BeanWiringInfoResolver

```
BeanWiringInfo resolveWiringInfo(Object beanInstance);
}
```

Strategy interface to be implemented by objects that can resolve bean name information, given a newly instantiated bean object. Invocations to the `@link #resolveWiringInfo` method on this interface will be driven by the AspectJ pointcut in the relevant concrete aspect.

Metadata resolution strategy can be pluggable. A good default is `@link ClassNameBeanWiringInfoResolver`, which uses the fully-qualified class name as bean name.

```
public class ClassNameBeanWiringInfoResolver implements BeanWiringInfoResolver {
    @Override
    public BeanWiringInfo resolveWiringInfo(Object beanInstance) {
        Assert.notNull(beanInstance, "Bean instance must not be null");
        return new BeanWiringInfo(ClassUtils.getUserClass(beanInstance).getName(), true);
    }
}
```

Simple default implementation of the `@link BeanWiringInfoResolver` interface, looking for a bean with the same name as the fully-qualified class name. This matches the default name of the bean in a Spring XML file if the bean tag's "id" attribute is not used.

62、 public class BeanConfigurerSupport

```
public class BeanConfigurerSupport implements BeanFactoryAware, InitializingBean,
DisposableBean
```

Convenient base class for bean configurers that can perform Dependency Injection on objects (however they may be created). Typically subclassed by AspectJ aspects.

Subclasses may also need a custom metadata resolution strategy, in the `@link BeanWiringInfoResolver` interface. The default implementation looks for a bean with the same name as the fully-qualified class name. (This is the default name of the bean in a Spring XML file if the '{@code id}' attribute is not used.)

63、 public interface BeanFactoryLocator

Defines a contract for the lookup, use, and release of a {@link org.springframework.beans.factory.BeanFactory}, or a {@code BeanFactory} subclass such as an {@link org.springframework.context.ApplicationContext}.

Where this interface is implemented as a singleton class such as {@link SingletonBeanFactoryLocator}, the Spring team strongly suggests that it be used sparingly and with caution. By far the vast majority of the code inside an application is best written in a Dependency Injection style, where that code is served out of a {@code BeanFactory}/{@code ApplicationContext} container, and has its own dependencies supplied by the container when it is created. However, even such a singleton implementation sometimes has its use in the small glue layers of code that is sometimes needed to tie other code together. For example, third party code may try to construct new objects directly, without the ability to force it to get these objects out of a {@code BeanFactory}.

If the object constructed by the third party code is just a small stub or proxy, which then uses an implementation of this class to get a {@code BeanFactory} from which it gets the real object, to which it delegates, then proper Dependency Injection has been achieved.

As another example, in a complex J2EE app with multiple layers, with each layer having its own {@code ApplicationContext} definition (in a hierarchy), a class like {@code SingletonBeanFactoryLocator} may be used to demand load these contexts.

```
public interface BeanFactoryLocator {  
    BeanFactoryReference useBeanFactory(String factoryKey) throws BeansException;  
}
```

64、 public interface BeanFactoryReference

```
    BeanFactory getFactory();  
    void release();  
}
```

Used to track a reference to a {@link BeanFactory} obtained through a {@link BeanFactoryLocator}.

65、 public class SingletonBeanFactoryLocator

```
public class SingletonBeanFactoryLocator implements BeanFactoryLocator
```

66、 public class AnnotationBeanWiringInfoResolver

public class AnnotationBeanWiringInfoResolver **implements** BeanWiringInfoResolver
{@link org.springframework.beans.factory.wiring.BeanWiringInfoResolver} that uses the Configurable annotation to identify which classes need autowiring.

The bean name to look up will be taken from the {@link Configurable} annotation if specified; otherwise the default will be the fully-qualified name of the class being configured.

67、 public @interface Autowired

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER,  
ElementType.FIELD, ElementType.ANNOTATION_TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Autowired {  
    boolean required() default true;  
}
```

68、 public class AutowiredAnnotationBeanPostProcessor

public class AutowiredAnnotationBeanPostProcessor **extends** InstantiationAwareBeanPostProcessorAdapter **implements** MergedBeanDefinitionPostProcessor, PriorityOrdered, BeanFactoryAware
{@link org.springframework.beans.factory.config.BeanPostProcessor} implementation that autowires annotated fields, setter methods and arbitrary config methods. Such members to be injected are detected through a Java 5 annotation: by default, Spring's {@link Autowired @Autowired} and {@link Value @Value} annotations.

Also supports JSR-330's {@link javax.inject.Inject @Inject} annotation, if available, as a direct alternative to Spring's own {@code @Autowired}.

Only one constructor (at max) of any given bean class may carry this annotation with the 'required' parameter set to {@code true}, indicating the constructor to autowire when used as a Spring bean. If multiple non-required constructors carry the annotation, they will be considered as candidates for autowiring. The constructor with the greatest number of dependencies that can be satisfied by matching beans in the Spring container will be chosen. If none of the candidates can be satisfied, then a default constructor (if present) will be used. An annotated constructor does not have to be public.

Fields are injected right after construction of a bean, before any config methods are invoked. Such a config field does not have to be public.

Config methods may have an arbitrary name and any number of arguments; each of those arguments will be autowired with a matching bean in the Spring container. Bean property setter methods are effectively just a special case of such a general config method. Config methods do not have to be public.

Note: A default AutowiredAnnotationBeanPostProcessor will be registered by the

"context:annotation-config" and "context:component-scan" XML tags.

Remove or turn off the default annotation configuration there if you intend to specify a custom `AutowiredAnnotationBeanPostProcessor` bean definition.

NOTE: Annotation injection will be performed before XML injection; thus the latter configuration will override the former for properties wired through both approaches.

In addition to regular injection points as discussed above, this post-processor also handles Spring's `@Lookup` annotation which identifies lookup methods to be replaced by the container at runtime.

69、 public abstract class BeanFactoryAnnotationUtils

Convenience methods performing bean lookups related to annotations, for example Spring's `@Qualifier` annotation.

70、 public @interface Configurable

Marks a class as being eligible for Spring-driven configuration.

Typically used with the AspectJ `@code AnnotationBeanConfigurerAspect`.

71、 public interface BeanFactoryPostProcessor

```
void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;  
}
```

Allows for custom modification of an application context's bean definitions, adapting the bean property values of the context's underlying bean factory.

Application contexts can auto-detect `BeanFactoryPostProcessor` beans in their bean definitions and apply them before any other beans get created.

Useful for custom config files targeted at system administrators that override bean properties configured in the application context.

See `PropertyResourceConfigurer` and its concrete implementations for out-of-the-box solutions that address such configuration needs.

A `BeanFactoryPostProcessor` may interact with and modify bean definitions, but never bean instances. Doing so may cause premature bean instantiation, violating the container and causing unintended side-effects. If bean instance interaction is required, consider implementing `@link BeanPostProcessor` instead.

72、 public class CustomAutowireConfigurer

```
public class CustomAutowireConfigurer implements BeanFactoryPostProcessor,  
BeanClassLoaderAware, Ordered
```

A `@link org.springframework.beans.factory.config.BeanFactoryPostProcessor` implementation that allows for convenient registration of custom autowire qualifier types.

73、public class InitDestroyAnnotationBeanPostProcessor

```
public class InitDestroyAnnotationBeanPostProcessor implements
DestructionAwareBeanPostProcessor, MergedBeanDefinitionPostProcessor, PriorityOrdered,
Serializable

protected transient Log logger = LoggerFactory.getLog(getClass());
private Class<? extends Annotation> initAnnotationType;
private Class<? extends Annotation> destroyAnnotationType;
private int order = Ordered.LOWEST_PRECEDENCE;
private transient final Map<Class<?>, LifecycleMetadata> lifecycleMetadataCache =
    new ConcurrentHashMap<Class<?>, LifecycleMetadata>(256);
```

74、public class InjectionMetadata

Internal class for managing injection metadata. Not intended for direct use in applications.

Used by {@link AutowiredAnnotationBeanPostProcessor},
{@link org.springframework.context.annotation.CommonAnnotationBeanPostProcessor} and
{@link org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor}.

75、public @interface Lookup

An annotation that indicates 'lookup' methods, to be overridden by the container to redirect them back to the {@link org.springframework.beans.factory.BeanFactory} for a {@code getBean} call. This is essentially an annotation-based version of the XML {@code lookup-method} attribute, resulting in the same runtime arrangement.

The resolution of the target bean can either be based on the return type ({@code getBean(Class)}) or on a suggested bean name ({@code getBean(String)}), in both cases passing the method's arguments to the {@code getBean} call for applying them as target factory method arguments or constructor arguments.

Such lookup methods can have default (stub) implementations that will simply get replaced by the container, or they can be declared as abstract - for the container to fill them in at runtime. In both cases, the container will generate runtime subclasses of the method's containing class via CGLIB, which is why such lookup methods can only work on beans that the container instantiates through regular constructors: i.e. lookup methods cannot get replaced on beans returned from factory methods where we cannot dynamically provide a subclass for them.

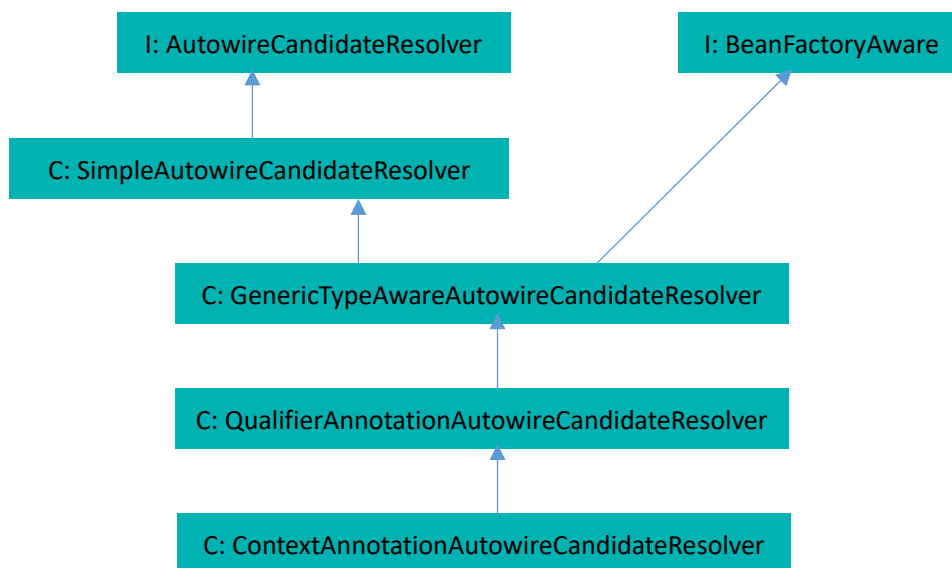
Concrete limitations in typical Spring configuration scenarios: When used with component scanning or any other mechanism that filters out abstract beans, provide stub implementations of your lookup methods to be able to declare them as concrete classes. And please remember that lookup methods won't work on beans returned from {@code @Bean} methods in configuration classes; you'll have to resort to {@code @Inject Provider<TargetBean>} or the like instead.

76、public @interface Qualifier

This annotation may be used on a field or parameter as a qualifier for candidate beans when autowiring. It may also be used to annotate other custom annotations that can then in turn be used as qualifiers.

77、public class QualifierAnnotationAutowireCandidateResolver

```
public class QualifierAnnotationAutowireCandidateResolver extends
GenericTypeAwareAutowireCandidateResolver {
    private final Set<Class<? extends Annotation>> qualifierTypes = new
    LinkedHashSet<Class<? extends Annotation>>(2);
    private Class<? extends Annotation> valueAnnotationType = Value.class;
    {@link AutowireCandidateResolver} implementation that matches bean definition
    qualifiers against {@link Qualifier} qualifier annotations on the field or parameter to
    be autowired. Also supports suggested expression values through a {@link Value} value
    annotation.
    Also supports JSR-330's {@link javax.inject.Qualifier} annotation, if available.
```



78、public interface AutowireCandidateResolver

Strategy interface for determining whether a specific bean definition qualifies as an autowire candidate for a specific dependency.

```
public interface AutowireCandidateResolver {
    boolean isAutowireCandidate(BeaDefinitionHolder bdHolder, DependencyDescriptor descriptor);
    Object getSuggestedValue(DependencyDescriptor descriptor);
    Object getLazyResolutionProxyIfNecessary(DependencyDescriptor descriptor, String beanName);
}
```

79、 public class SimpleAutowireCandidateResolver

{@link AutowireCandidateResolver} implementation to use when no annotation support is available. This implementation checks the bean definition only.

```
public class SimpleAutowireCandidateResolver implements AutowireCandidateResolver {

    @Override

    public boolean isAutowireCandidate(BeanDefinitionHolder bdHolder, DependencyDescriptor descriptor) {

        return bdHolder.getBeanDefinition().isAutowireCandidate();

    }

    public boolean isRequired(DependencyDescriptor descriptor) {

        return descriptor.isRequired();

    }

    @Override

    public Object getSuggestedValue(DependencyDescriptor descriptor) {

        return null;

    }

    @Override

    public Object getLazyResolutionProxyIfNecessary(DependencyDescriptor descriptor, String beanName) {

        return null;

    }

}
```

80、 public class RequiredAnnotationBeanPostProcessor

```
public class RequiredAnnotationBeanPostProcessor extends
InstantiationAwareBeanPostProcessorAdapter implements MergedBeanDefinitionPostProcessor,
PriorityOrdered, BeanFactoryAware
```

81、 public abstract class AbstractFactoryBean<T>

```
public abstract class AbstractFactoryBean<T> implements FactoryBean<T>,
BeanClassLoaderAware, BeanFactoryAware, InitializingBean, DisposableBean {

    protected final Log logger = LoggerFactory.getLog(getClass());

    private boolean singleton = true;

    private ClassLoader beanClassLoader = ClassUtils.getDefaultClassLoader();

    private BeanFactory beanFactory;

    private boolean initialized = false;

    private T singletonInstance;

    private T earlySingletonInstance;
```

Simple template superclass for {@link FactoryBean} implementations that creates a singleton or a prototype object, depending on a flag.

If the "singleton" flag is {@code true} (the default), this class will create the object that it creates exactly once on initialization and subsequently return said singleton instance on all calls to the {@link #getObject()} method.

Else, this class will create a new instance every time the {@link #getObject()} method is invoked. Subclasses are responsible for implementing the abstract {@link #createInstance()} template method to actually create the object(s) to expose.

82、public interface AutowireCapableBeanFactory

```
1 AutowireCapableBeanFactory
  SF AUTOWIRE_NO : int
  SF AUTOWIRE_BY_NAME : int
  SF AUTOWIRE_BY_TYPE : int
  SF AUTOWIRE_CONSTRUCTOR : int
  SF AUTOWIRE_AUTODETECT : int
  A createBean(Class<T>) <T> : T
  A autowireBean(Object) : void
  A configureBean(Object, String) : Object
  A createBean(Class<?>, int, boolean) : Object
  A autowire(Class<?>, int, boolean) : Object
  A autowireBeanProperties(Object, int, boolean) : void
  A applyBeanPropertyValues(Object, String) : void
  A initializeBean(Object, String) : Object
  A applyBeanPostProcessorsBeforeInitialization(Object, String) : Object
  A applyBeanPostProcessorsAfterInitialization(Object, String) : Object
  A destroyBean(Object) : void
  A resolveNamedBean(Class<T>) <T> : NamedBeanHolder<T>
  A resolveDependency(DependencyDescriptor, String) : Object
  A resolveDependency(DependencyDescriptor, String, Set<String>, TypeConverter) : Object
```

83、public interface BeanExpressionResolver

```
Object evaluate(String value, BeanExpressionContext evalContext) throws BeansException;
}
```

Strategy interface for resolving a value through evaluating it as an expression, if applicable.

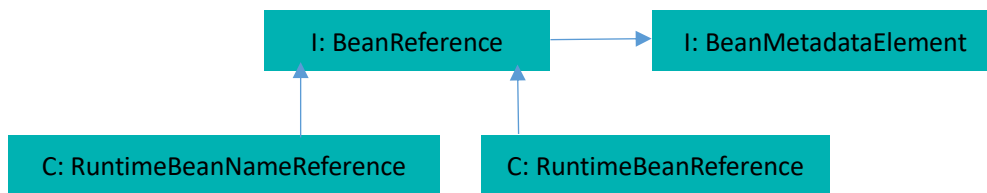
A raw {@link org.springframework.beans.factory.BeanFactory} does not contain a default implementation of this strategy.

However, {@link org.springframework.context.ApplicationContext} implementations will provide expression support out of the box.

84、 public interface BeanReference

Interface that exposes a reference to a bean name in an abstract fashion. This interface does not necessarily imply a reference to an actual bean instance; it just expresses a logical reference to the name of a bean. Serves as common interface implemented by any kind of bean reference holder, such as `{@link RuntimeBeanReference RuntimeBeanReference}` and `{@link RuntimeBeanNameReference RuntimeBeanNameReference}`.

```
public interface BeanReference extends BeanMetadataElement {  
    String getBeanName();  
}
```



85、 public interface ConfigurableListableBeanFactory

extends `ListableBeanFactory`, `AutowireCapableBeanFactory`, `ConfigurableBeanFactory`

Configuration interface to be implemented by most listable bean factories. In addition to `{@link ConfigurableBeanFactory}`, it provides facilities to analyze and modify bean definitions, and to pre-instantiate singletons.

This subinterface of `{@link org.springframework.beans.factory.BeanFactory}` is not meant to be used in normal application code: Stick to `{@link org.springframework.beans.factory.BeanFactory}` or `{@link org.springframework.beans.factory.ListableBeanFactory}` for typical use cases. This interface is just meant to allow for framework-internal plug'n'play even when needing access to bean factory configuration methods.

86、 public class CustomEditorConfigurer

public class `CustomEditorConfigurer` **implements** `BeanFactoryPostProcessor`, `Ordered`

`{@link BeanFactoryPostProcessor}` implementation that allows for convenient registration of custom `{@link PropertyEditor}` property editors).

In case you want to register `{@link PropertyEditor}` instances, the recommended usage as of Spring 2.0 is to use custom `{@link PropertyEditorRegistrar}` implementations that in turn register any desired editor instances on a given `{@link org.springframework.beans.PropertyEditorRegistry registry}`. Each `PropertyEditorRegistrar` can register any number of custom editors.

87、 public class CustomScopeConfigurer

public class CustomScopeConfigurer **implements** BeanFactoryPostProcessor,
BeanClassLoaderAware, Ordered

Simple {@link BeanFactoryPostProcessor} implementation that registers custom {@link Scope Scope(s)} with the containing {@link ConfigurableBeanFactory}.

Will register all of the supplied {@link #setScopes(java.util.Map) scopes} with the {@link ConfigurableListableBeanFactory} that is passed to the
{@link #postProcessBeanFactory(ConfigurableListableBeanFactory)} method.

This class allows for declarative registration of custom scopes. Alternatively, consider implementing a custom {@link BeanFactoryPostProcessor} that calls {@link ConfigurableBeanFactory#registerScope} programmatically.

88、 public class DependencyDescriptor

public class DependencyDescriptor **extends** InjectionPoint **implements** Serializable

Descriptor for a specific dependency that is about to be injected. Wraps a constructor parameter, a method parameter or a field, allowing unified access to their metadata.

89、 public class MethodInvokingBean

public class MethodInvokingBean **extends** ArgumentConvertingMethodInvoker
implements BeanClassLoaderAware, BeanFactoryAware, InitializingBean

Simple method invoker bean: just invoking a target method, not expecting a result to expose to the container (in contrast to {@link MethodInvokingFactoryBean}).

This invoker supports any kind of target method. A static method may be specified by setting the {@link #setTargetMethod targetMethod} property to a String representing the static method name, with {@link #setTargetClass targetClass} specifying the Class that the static method is defined on. Alternatively, a target instance method may be specified, by setting the {@link #setTargetObject targetObject} property as the target object, and the {@link #setTargetMethod targetMethod} property as the name of the method to call on that target object. Arguments for the method invocation may be specified by setting the {@link #setArguments arguments} property.

90、 public interface NamedBean

```
{  
    String getBeanName();  
}
```

Counterpart of {@link BeanNameAware}. Returns the bean name of an object. This interface can be introduced to avoid a brittle dependence on bean name in objects used with Spring IoC and Spring AOP.

91、public class MethodInvokingFactoryBean

```
public class MethodInvokingFactoryBean extends MethodInvokingBean implements
FactoryBean<Object>
```

`{@link FactoryBean}` which returns a value which is the result of a static or instance method invocation. For most use cases it is better to just use the container's built-in factory method support for the same purpose, since that is smarter at converting arguments. This factory bean is still useful though when you need to call a method which doesn't return any value (for example, a static class method to force some sort of initialization to happen). This use case is not supported by factory methods, since a return value is needed to obtain the bean instance.

92、public abstract class PlaceholderConfigurerSupport

```
public abstract class PlaceholderConfigurerSupport extends PropertyResourceConfigurer
implements BeanNameAware, BeanFactoryAware
```

Abstract base class for property resource configurers that resolve placeholders in bean definition property values. Implementations `pull` values from a properties file or other `{@link plain org.springframework.core.env.PropertySource property source}` into bean definitions.

93、public class PreferencesPlaceholderConfigurer

```
public class PreferencesPlaceholderConfigurer extends PropertyPlaceholderConfigurer
implements InitializingBean
```

Subclass of `PropertyPlaceholderConfigurer` that supports JDK 1.4's Preferences API (`{@code java.util.prefs}`).

Tries to resolve placeholders as keys first in the user preferences, then in the system preferences, then in this configurer's properties. Thus, behaves like `PropertyPlaceholderConfigurer` if no corresponding preferences defined.

Supports custom paths for the system and user preferences trees. Also supports custom paths specified in placeholders ("myPath/myPlaceholderKey"). Uses the respective root node if not specified.

94、public class PropertiesFactoryBean

```
public class PropertiesFactoryBean extends PropertiesLoaderSupport
implements FactoryBean<Properties>, InitializingBean
```

Allows for making a properties file from a classpath location available as `Properties` instance in a bean factory. Can be used to populate any bean property of type `Properties` via a bean reference.

Supports loading from a properties file and/or setting local properties on this `FactoryBean`. The created `Properties` instance will be merged from loaded and local values. If neither a location nor local properties are set, an exception will be thrown on initialization.

Can create a singleton or a new object on each request. Default is a singleton.

95、 public class PropertyOverrideConfigurer

```
public class PropertyOverrideConfigurer extends PropertyResourceConfigurer

    Property resource configurer that overrides bean property values in an application context
    definition. It <i>pushes</i> values from a properties file into bean definitions.
```

96、 public class PropertyPathFactoryBean

```
public class PropertyPathFactoryBean implements FactoryBean<Object>, BeanNameAware,
BeanFactoryAware

    {@link FactoryBean} that evaluates a property path on a given target object.The target object can be
    specified directly or via a bean name.
```

97、 public class PropertyPlaceholderConfigurer

```
public class PropertyPlaceholderConfigurer extends PlaceholderConfigurerSupport
```

98、 public abstract class PropertyResourceConfigurer

```
public abstract class PropertyResourceConfigurer extends PropertiesLoaderSupport
    implements BeanFactoryPostProcessor, PriorityOrdered

    Allows for configuration of individual bean property values from a property resource, i.e. a
    properties file. Useful for custom config files targeted at system administrators that override bean
    properties configured in the application context.
```

99、 public class ProviderCreatingFactoryBean

```
public class ProviderCreatingFactoryBean extends AbstractFactoryBean<Provider<Object>>

    A {@link org.springframework.beans.factory.FactoryBean} implementation that returns a value which is
    a JSR-330 {@link javax.inject.Provider} that in turn returns a bean sourced from a {@link
    org.springframework.beans.factory.BeanFactory}.
```

100、 public class RuntimeBeanNameReference

```
public class RuntimeBeanNameReference implements BeanReference

    Immutable placeholder class used for a property value object when it's a reference to another bean
    name in the factory, to be resolved at runtime.
```

101、 public class RuntimeBeanReference

```
public class RuntimeBeanReference implements BeanReference

    Immutable placeholder class used for a property value object when it's a reference to another bean in the
    factory, to be resolved at runtime.
```


Strategy interface used by a [{@link ConfigurableBeanFactory}](#), representing a target scope to hold bean instances in. This allows for extending the BeanFactory's standard scopes [{@link ConfigurableBeanFactory#SCOPE_SINGLETON "singleton"}](#) and [{@link ConfigurableBeanFactory#SCOPE_PROTOTYPE "prototype"}](#) with custom further scopes, registered for a [{@link ConfigurableBeanFactory#registerScope\(String, Scope\) specific key}](#).

102、 public interface Scope

```
{
    Object get(String name, ObjectFactory<?> objectFactory);
    Object remove(String name);
    void registerDestructionCallback(String name, Runnable callback);
    Object resolveContextualObject(String key);
    String getConversationId();
}
```

103、 public class ServiceLocatorFactoryBean

```
public class ServiceLocatorFactoryBean implements FactoryBean<Object>, BeanFactoryAware,
InitializingBean
```

104、 public class SetFactoryBean

```
public class SetFactoryBean extends AbstractFactoryBean<Set<Object>>
```

Simple factory for shared Set instances. Allows for central setup of Sets via the "set" element in XML bean definitions.

105、 public interface SmartInstantiationAwareBeanPostProcessor

```
public interface SmartInstantiationAwareBeanPostProcessor extends InstantiationAwareBeanPostProcessor {
    Class<?> predictBeanType(Class<?> beanClass, String beanName) throws BeansException;
    Constructor<?>[] determineCandidateConstructors(Class<?> beanClass, String beanName) throws
BeansException;
    Object getEarlyBeanReference(Object bean, String beanName) throws BeansException;
}
```

Extension of the [{@link InstantiationAwareBeanPostProcessor}](#) interface, adding a callback for predicting the eventual type of a processed bean.

NOTE: This interface is a special purpose interface, mainly for internal use within the framework. In general, application-provided post-processors should simply implement the plain [{@link BeanPostProcessor}](#) interface or derive from the [{@link InstantiationAwareBeanPostProcessorAdapter}](#) class. New methods might be added to this interface even in point releases.

Holder for a typed String value. Can be added to bean definitions in order to explicitly specify a target type for a String value,for example for collection elements.

This holder will just store the String value and the target type. The actual conversion will be performed by the bean factory.

106、 public class TypedStringValue

```
public class TypedStringValue implements BeanMetadataElement
```