public
abstract class

AbstractBeanFactory

package org.springframework.beans.factory.support

extends FactoryBeanRegistrySupport

implements ConfigurableBeanFactory

```
{
/** Parent bean factory, for bean inheritance support */
@Nullable
private BeanFactory parentBeanFactory;

/** ClassLoader to resolve bean class names with, if necessary */
@Nullable
private ClassLoader beanClassLoader = ClassUtils.getDefaultClassLoader();

/** ClassLoader to temporarily resolve bean class names with, if necessary */
@Nullable
private ClassLoader tempClassLoader;

/** Whether to cache bean metadata or rather reobtain it for every access */
private boolean cacheBeanMetadata = true;

/** Resolution strategy for expressions in bean definition values */
@Nullable
private BeanExpressionResolver beanExpressionResolver;

/** Spring ConversionService to use instead of PropertyEditors */
@Nullable
private ConversionService conversionService;

/** Custom PropertyEditorRegistrars to apply to the beans of this factory */
private final Set<PropertyEditorRegistrar> propertyEditorRegistrars = new
 LinkedHashSet<>(4);

/** Custom PropertyEditors to apply to the beans of this factory */
private final Map<Class<?>, Class<? extends PropertyEditor>> customEditors = new
 HashMap<>(4);

/** A custom TypeConverter to use, overriding the default PropertyEditor mechanism
 */
@Nullable
private TypeConverter typeConverter;

/** String resolvers to apply e.g. to annotation attribute values */
private final List<StringValueResolver> embeddedValueResolvers = new
 LinkedList<>();

/** BeanPostProcessors to apply in createBean */
private final List<BeanPostProcessor> beanPostProcessors = new ArrayList<>();

/** Indicates whether any InstantiationAwareBeanPostProcessors have been
 registered */
private boolean hasInstantiationAwareBeanPostProcessors;

/** Indicates whether any DestructionAwareBeanPostProcessors have been registered
 */
private boolean hasDestructionAwareBeanPostProcessors;

/** Map from scope identifier String to corresponding Scope */
private final Map<String, Scope> scopes = new LinkedHashMap<>(8);

/** Security context used when running with a SecurityManager */
@Nullable
private SecurityContextProvider securityContextProvider;

/** Map from bean name to merged RootBeanDefinition */
private final Map<String, RootBeanDefinition> mergedBeanDefinitions = new
 ConcurrentHashMap<>(256);

/** Names of beans that have already been created at least once */
private final Set<String> alreadyCreated = Collections.newSetFromMap(new
 ConcurrentHashMap<>(256));

/** Names of beans that are currently in creation */
private final ThreadLocal<Object> prototypesCurrentlyInCreation =
  new NamedThreadLocal<>("Prototype beans currently in creation");
```

```
/**
 * Create a new AbstractBeanFactory.
 */
public AbstractBeanFactory
  () {

}
```

```
/**
 * Create a new AbstractBeanFactory with the given parent.
 * @param parentBeanFactory parent bean factory, or {@code null} if none
 * @see #getBean
 */
public AbstractBeanFactory
  (@Nullable BeanFactory parentBeanFactory) {

 this.parentBeanFactory = parentBeanFactory;
}
```

// Implementation of BeanFactory interface

```
@Override
public Object getBean
    (String name) throws BeansException {

 return doGetBean(name, null, null, false);
}
```

```
@Override
public <T> T getBean
    (String name, @Nullable Class<T> requiredType) throws BeansException {

 return doGetBean(name, requiredType, null, false);
}
```

```
@Override
public Object getBean
    (String name, Object... args) throws BeansException {
 return doGetBean(name, null, args, false);
}
```

```
/**
 * Return an instance, which may be shared or independent, of the specified bean.
 * @param name the name of the bean to retrieve
 * @param requiredType the required type of the bean to retrieve
 * @param args arguments to use when creating a bean instance using explicit
 * arguments (only applied when creating a new instance)
 * @return an instance of the bean
 * @throws BeansException if the bean could not be created
 */
public <T> T getBean
    (String name, @Nullable Class<T> requiredType, @Nullable Object... args)
    throws BeansException {

 return doGetBean(name, requiredType, args, false);
}
```

```
/**
 * Return an instance, which may be shared or independent, of the specified bean.
 * @param name the name of the bean to retrieve
 * @param requiredType the required type of the bean to retrieve
 * @param args arguments to use when creating a bean instance using explicit
 * arguments (only applied when creating a new instance)
 * @param typeCheckOnly whether the instance is obtained for a type check,
 * not for actual use
 * @return an instance of the bean
 * @throws BeansException if the bean could not be created
 */
@SuppressWarnings("unchecked")
protected <T> T doGetBean
    (final String name, @Nullable final Class<T> requiredType,
     @Nullable final Object[] args, boolean typeCheckOnly) throws BeansException {

 final String beanName = transformedBeanName(name);
 Object bean;

 // Eagerly check singleton cache for manually registered singletons.
 Object sharedInstance = getSingleton(beanName);
 if (sharedInstance != null && args == null) {
  if (logger.isDebugEnabled()) {
   if (isSingletonCurrentlyInCreation(beanName)) {
    logger.debug("Returning eagerly cached instance of singleton bean '" +
      beanName + "' that is not fully initialized yet - circular reference");
   }
   else {
    logger.debug("Returning cached instance of bean '" + beanName + "'");
   }
  }
  bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
 }

 else {
  // Fail if we're already creating this bean instance:
  // We're assumably within a circular reference.
  if (isPrototypeCurrentlyInCreation(beanName)) {
   throw new BeanCurrentlyInCreationException(beanName);
  }

  // Check if bean definition exists in this factory.
  BeanFactory parentBeanFactory = getParentBeanFactory();
  if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
   // Not found -> check parent.
   String nameToLookup = originalBeanName(name);
   if (parentBeanFactory instanceof AbstractBeanFactory) {
    return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
      nameToLookup, requiredType, args, typeCheckOnly);
   }
   else if (args != null) {
    // Delegation to parent with explicit args.
    return (T) parentBeanFactory.getBean(nameToLookup, args);
   }
   else {
    // No args -> delegate to standard getBean method.
    return parentBeanFactory.getBean(nameToLookup, requiredType);
   }
  }

  if (!typeCheckOnly) {
   markBeanAsCreated(beanName);
  }

  try {
   final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
   checkMergedBeanDefinition(mbd, beanName, args);

   // Guarantee initialization of beans that the current bean depends on.
   String[] dependsOn = mbd.getDependsOn();
   if (dependsOn != null) {
    for (String dep : dependsOn) {
     if (isDependent(beanName, dep)) {
      throw new BeanCreationException(mbd.getResourceDescription(), beanName,
        "Circular relationship between '" + beanName + "' and '" + dep + "'");
     }
     registerDependentBean(dep, beanName);
     getBean(dep);
    }
   }

   // Create bean instance.
   if (mbd.isSingleton()) {
    sharedInstance = getSingleton(beanName, () -> {
     try {
      return createBean(beanName, mbd, args);
     }
     catch (BeansException ex) {
      // Explicitly remove instance from singleton cache: It might have been put
      // there eagerly by the creation process, to allow for circular reference
      // resolution. Also remove any beans that received a temporary reference to
      // the bean.
      destroySingleton(beanName);
      throw ex;
     }
    });
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
   }

   else if (mbd.isPrototype()) {
    // It's a prototype -> create a new instance.
    Object prototypeInstance = null;
    try {
     beforePrototypeCreation(beanName);
     prototypeInstance = createBean(beanName, mbd, args);
    }
    finally {
     afterPrototypeCreation(beanName);
    }
    bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
   }

   else {
    String scopeName = mbd.getScope();
    final Scope scope = this.scopes.get(scopeName);
    if (scope == null) {
     throw new IllegalStateException("No Scope for '" + scopeName + "'");
    }
    try {
     Object scopedInstance = scope.get(beanName, () -> {
      beforePrototypeCreation(beanName);
      try {
       return createBean(beanName, mbd, args);
      }
      finally {
       afterPrototypeCreation(beanName);
      }
     });
     bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
    }
    catch (IllegalStateException ex) {
     throw new BeanCreationException(beanName,
       "Scope '" + scopeName + "' is not active for the current thread; " +
       "define a scoped proxy for this bean to refer to it from a singleton", ex);
    }
   }
  }
  catch (BeansException ex) {
   cleanupAfterBeanCreationFailure(beanName);
   throw ex;
  }
 }

 // Check if required type matches the type of the actual bean instance. Note that the following
 // return declarations are technically violating the non-null policy for the getBean methods:
 // However, these will only result in null under very specific circumstances: such as a user-declared
 factory method returning null or a user-provided FactoryBean.getObject() returning null, without
 any custom post-processing of such null values. We will pass them on as null to corresponding
 injection points in that exceptional case but do not expect user-level getBean callers to deal with
 such null values. In the end, regular getBean callers should be able to assign the outcome to non-null
 variables/arguments without being compromised by rather esoteric corner cases, in particular in
 functional configuration and Kotlin scenarios. A future Spring generation might eventually forbid
 null values completely and throw IllegalStateExceptions instead of leniently passing them through.
 if (requiredType != null && bean != null && !requiredType.isInstance(bean)) {
  try {
   return getTypeConverter().convertIfNecessary(bean, requiredType);
  }
  catch (TypeMismatchException ex) {
   if (logger.isDebugEnabled()) {
    logger.debug("Failed to convert bean '" + name + "' to required type '" +
      ClassUtils.getQualifiedName(requiredType) + "'", ex);
   }
   throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
  }
 }
 // For the nullability warning, see the elaboration in the comment above;
 // in short: This is never going to be null unless user-declared code enforces
 null.
 return (T) bean;
}
```

```
@Override
public boolean containsBean
    (String name) {

 String beanName = transformedBeanName(name);
 if (containsSingleton(beanName) || containsBeanDefinition(beanName)) {
  return (!BeanFactoryUtils.isFactoryDereference(name) || isFactoryBean(name));
 }
 // Not found -> check parent.
 BeanFactory parentBeanFactory = getParentBeanFactory();
 return (parentBeanFactory != null && parentBeanFactory.
   containsBean(originalBeanName(name)));
}
```

```
@Override
public boolean isSingleton
    (String name) throws NoSuchBeanDefinitionException {

 String beanName = transformedBeanName(name);

 Object beanInstance = getSingleton(beanName, false);
 if (beanInstance != null) {
  if (beanInstance instanceof FactoryBean) {
   return (BeanFactoryUtils.isFactoryDereference(name) ||
     ((FactoryBean<?>) beanInstance).isSingleton());
  }
  else {
   return !BeanFactoryUtils.isFactoryDereference(name);
  }
 }
 else if (containsSingleton(beanName)) {
  return true;
 }

 // No singleton instance found -> check bean definition.
 BeanFactory parentBeanFactory = getParentBeanFactory();
 if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
  // No bean definition found in this factory -> delegate to parent.
  return parentBeanFactory.isSingleton(originalBeanName(name));
 }

 RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

 // In case of FactoryBean, return singleton status of created object if not a
 dereference.
 if (mbd.isSingleton()) {
  if (isFactoryBean(beanName, mbd)) {
   if (BeanFactoryUtils.isFactoryDereference(name)) {
    return true;
   }
   FactoryBean<?> factoryBean =
     (FactoryBean<?>) getBean(FACTORY_BEAN_PREFIX + beanName);
   return factoryBean.isSingleton();
  }
  else {
   return !BeanFactoryUtils.isFactoryDereference(name);
  }
 }
 else {
  return false;
 }
}
```

```
@Override
public boolean isPrototype
    (String name) throws NoSuchBeanDefinitionException {

 String beanName = transformedBeanName(name);

 BeanFactory parentBeanFactory = getParentBeanFactory();
 if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
  // No bean definition found in this factory -> delegate to parent.
  return parentBeanFactory.isPrototype(originalBeanName(name));
 }

 RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
 if (mbd.isPrototype()) {
  // In case of FactoryBean, return singleton status of created object if not a
  // dereference.
  return (!BeanFactoryUtils.isFactoryDereference(name) ||
   isFactoryBean(beanName, mbd));
 }

 // Singleton or scoped - not a prototype.
 // However, FactoryBean may still produce a prototype object...
 if (BeanFactoryUtils.isFactoryDereference(name)) {
  return false;
 }
 if (isFactoryBean(beanName, mbd)) {
  final FactoryBean<?> fb =
    (FactoryBean<?>) getBean(FACTORY_BEAN_PREFIX + beanName);
  if (System.getSecurityManager() != null) {
   return AccessController.doPrivileged((PrivilegedAction<Boolean>) () ->
     ((fb instanceof SmartFactoryBean && ((SmartFactoryBean<?>) fb).isPrototype())
       || !fb.isSingleton()),
     getAccessControlContext());
  }
  else {
   return ((fb instanceof SmartFactoryBean && ((SmartFactoryBean<?>) fb).
 isPrototype()) ||
     !fb.isSingleton());
  }
 }
 else {
  return false;
 }
}
```

```
@Override
public boolean isTypeMatch
    (String name, ResolvableType typeToMatch)
    throws NoSuchBeanDefinitionException {

 String beanName = transformedBeanName(name);

 // Check manually registered singletons.
 Object beanInstance = getSingleton(beanName, false);
 if (beanInstance != null) {
  if (beanInstance instanceof FactoryBean) {
   if (!BeanFactoryUtils.isFactoryDereference(name)) {
    Class<?> type = getTypeForFactoryBean((FactoryBean<?>) beanInstance);
    return (type != null && typeToMatch.isAssignableFrom(type));
   }
   else {
    return typeToMatch.isInstance(beanInstance);
   }
  }
  else if (!BeanFactoryUtils.isFactoryDereference(name)) {
   if (typeToMatch.isInstance(beanInstance)) {
    // Direct match for exposed instance?
    return true;
   }
   else if (typeToMatch.hasGenerics() && containsBeanDefinition(beanName)) {
    // Generics potentially only match on the target class, not on the proxy...
    RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
    Class<?> targetType = mbd.getTargetType();
    if (targetType != null && targetType != ClassUtils.getUserClass(beanInstance)
       && typeToMatch.isAssignableFrom(targetType)) {
     // Check raw class match as well, making sure it's exposed on the proxy.
     Class<?> classToMatch = typeToMatch.resolve();
     return (classToMatch == null || classToMatch.isInstance(beanInstance));
    }
   }
  }
  return false;
 }
 else if (containsSingleton(beanName) && !containsBeanDefinition(beanName)) {
  // null instance registered
  return false;
 }

 // No singleton instance found -> check bean definition.
 BeanFactory parentBeanFactory = getParentBeanFactory();
 if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
  // No bean definition found in this factory -> delegate to parent.
  return parentBeanFactory.isTypeMatch(originalBeanName(name), typeToMatch);
 }

 // Retrieve corresponding bean definition.
 RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

 Class<?> classToMatch = typeToMatch.resolve();
```