Beautiful code:

code with better type

@PeterHilton

http://hilton.org.uk/

Not type as in
type system, but
type as in…

1961 IBM Selectric had a choice of 45 typefaces including Courier

Before Courier became widespread, we printed COBOL on line printers…

```
    2 HEADER-A; SIZE IS 68; CLASS IS AN.
    2 PAGE; SIZE IS 7; CLASS IS NUMERIC; SIGNED; ZERO SUPPRESS; LEAVING 1 PLACES.
    2 FILL-7; SIZE IS 4; CLASS IS AN.
 1 SECOND-HEADING; SIZE IS 120; CLASS IS AN.
WORKING-STORAGE SECTION.
 77 DEFFICIENCY; SIZE IS 8; CLASS IS NUMERIC; SIGNED.
 77 PRODUCTION-QUOTA; SIZE IS 8; CLASS IS NUMERIC; SIGNED.
 77 DIFFERENCE-FACTOR; SIZE IS 8; CLASS IS NUMERIC; SIGNED.
 77 PAGE-COUNT; SIZE IS 7; CLASS IS NUMERIC; SIGNED.
 77 LINE-COUNT; SIZE IS 4; CLASS IS NUMERIC; SIGNED.
 1 COMPILATION-DATE; SIZE IS 17; CLASS IS AN.
    2 SW; SIZE IS 1.
    2 DATE; SIZE IS 15.
    2 EM; SIZE IS 1.
CONSTANT SECTION.
 77 TWENTY-PERCENT; SIZE IS 2; CLASS IS NUMERIC; SIGNED; POINT LOCATION IS LEFT 1 PLACE; VALUE IS "2 ".
 77 TEN-PERCENT; SIZE IS 2; CLASS IS NUMERIC; SIGNED; POINT LOCATION IS LEFT 1 PLACE; VALUE IS "1 ".
 77 HEADER-B; SIZE IS 120; CLASS IS AN; VALUE IS "      PRODUCT NUMBER          PRODUCT      MEASURE      QUANTITY ON-
    HAND   EXPECTED TURNOVER    PRODUCTION QUOTA      ".
PROCEDURE DIVISION.
    00001. OPEN INPUT MASTER, TRANSACTION; OUTPUT NEW-MASTER, REPORT.
    00002. ACCEPT COMPILATION-DATE FROM PAPER-TAPE-READER.
    00003. MOVE ZEROES TO PAGE-COUNT, LINE-COUNT.
    00004. PERFORM 35 THRU 42.
    00005. READ MASTER; AT END GO TO 46.
```

Windows 3.1 introduced Courier New, still widely used for source code…

```
          beq scroll

          lda speed
          eor #$01
          sta speed

; ---------------------------------------
; Ordinary scroll... (he, he - beat me
; it is probably the shortest 1x1 scroll
; routine on the world ;)

scroll    lda roll
          sec
          sbc speed
          bpl nzero
          and #$07
          sta roll

          ldy #$00
rewrite   lda scrline+1,y
          sta scrline,y
          iny
          cpy #$27
          bne rewrite
```

Source code typography innovation hasn’t been limited to new typefaces

Fast inverse square root algorithm, with syntax colour, bold, italic, and a new typeface…

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i = * ( long * ) &y;                       // evil floating point
                                               // bit level hacking

    i = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration,
                                               // this can be removed
  return y;
}
```

The most recent innovation in source code is as old as typography itself

Ligatures combine letter combinations

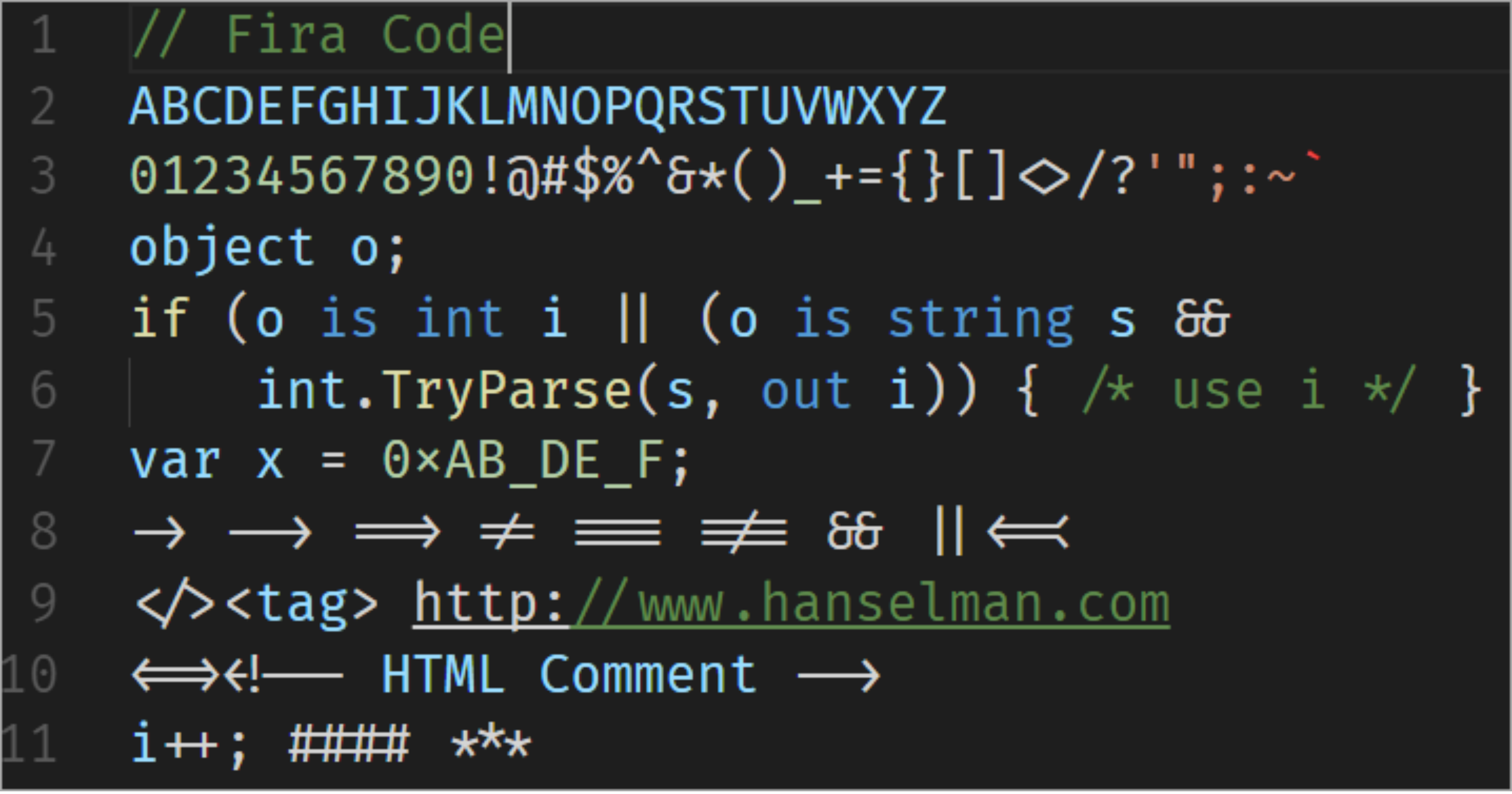
T h ~ Th, f j ~ fj

Source Code Pro

```
(>>=) :: (Num a) => a -> a -> a
a /= b ++ a <- c << [1..2] -<<
a == w <- rng -< ()
```

Hasklig

```
(>>=) :: (Num a) ⇒ a → a → a
a ≠= b ++ a ← c ≪ [1..2] -≪
a == w ← rng ≺ ()
```



https://www.hanselman.com/blog/MonospacedProgrammingFontsWithLigatures.aspx

Innovations in source code typography

Courier typeface, commissioned by IBM (1955)

Syntax highlighting - colour/bold (1985)

Courier New, introduced with Windows 3.1 (1992)

Consolas font, commissioned by Microsoft (2004)

Hasklig code font with ligatures, Ian Tuomi (2012)

Multiple wide monitors don’t help with long Java classes…

```
package org.springframework.beans.factory.support;

import org.springframework.beans.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.config.*;
import org.springframework.core.*;
import org.springframework.core.convert.ConversionService;
import org.springframework.lang.Nullable;
import org.springframework.util.*;

public abstract class AbstractBeanFactory 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();
  if (classToMatch == null) {
   classToMatch = FactoryBean.class;
  }
  Class<?>[] typesToMatch = (FactoryBean.class == classToMatch ?
    new Class<?>[] {classToMatch} : new Class<?>[] {FactoryBean.class, classToMatch});

  // Check decorated bean definition, if any: We assume it'll be easier
  // to determine the decorated bean's type than the proxy's type.
  BeanDefinitionHolder dbd = mbd.getDecoratedDefinition();
  if (dbd != null && !BeanFactoryUtils.isFactoryDereference(name)) {
   RootBeanDefinition tbd = getMergedBeanDefinition(dbd.getBeanName(),
           dbd.getBeanDefinition(), mbd);
   Class<?> targetClass = predictBeanType(dbd.getBeanName(), tbd, typesToMatch);
   if (targetClass != null && !FactoryBean.class.isAssignableFrom(targetClass)) {
    return typeToMatch.isAssignableFrom(targetClass);
   }
  }

  Class<?> beanType = predictBeanType(beanName, mbd, typesToMatch);
  if (beanType == null) {
   return false;
  }

  // Check bean class whether we're dealing with a FactoryBean.
  if (FactoryBean.class.isAssignableFrom(beanType)) {
   if (!BeanFactoryUtils.isFactoryDereference(name)) {
    // If it's a FactoryBean, we want to look at what it creates, not the factory class.
    beanType = getTypeForFactoryBean(beanName, mbd);
    if (beanType == null) {
     return false;
    }
   }
  }
  else if (BeanFactoryUtils.isFactoryDereference(name)) {
   // Special case: A SmartInstantiationAwareBeanPostProcessor returned a non-FactoryBean
   // type but we nevertheless are being asked to dereference a FactoryBean...
   // Let's check the original bean class and proceed with it if it is a FactoryBean.
   beanType = predictBeanType(beanName, mbd, FactoryBean.class);
   if (beanType == null || !FactoryBean.class.isAssignableFrom(beanType)) {
    return false;
   }
  }

  ResolvableType resolvableType = mbd.targetType;
  if (resolvableType == null) {
   resolvableType = mbd.factoryMethodReturnType;
  }
  if (resolvableType != null && resolvableType.resolve() == beanType) {
   return typeToMatch.isAssignableFrom(resolvableType);
  }
  return typeToMatch.isAssignableFrom(beanType);
 }

 @Override

 public boolean isTypeMatch(String name, @Nullable
    Class<?> typeToMatch) throws NoSuchBeanDefinitionException {
  return isTypeMatch(name, ResolvableType.forRawClass(typeToMatch));
 }

 @Override
 public Class<?> getType(String name) throws NoSuchBeanDefinitionException {
  String beanName = transformedBeanName(name);

  // Check manually registered singletons.
  Object beanInstance = getSingleton(beanName, false);
  if (beanInstance != null) {
   if (beanInstance instanceof FactoryBean && !BeanFactoryUtils.isFactoryDereference(name)) {
    return getTypeForFactoryBean((FactoryBean<?>) beanInstance);
   }
   else {
```

```
package org.springframework.beans.factory.support;

import org.springframework.beans.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.config.*;
import org.springframework.core.*;
import org.springframework.core.convert.ConversionService;
import org.springframework.lang.Nullable;
import org.springframework.util.*;

public abstract class AbstractBeanFactory 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).isProto-
type()) ||
     !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();
 if (classToMatch == null) {
  classToMatch = FactoryBean.class;
 }
 Class<?>[] typesToMatch = (FactoryBean.class == classToMatch ?
   new Class<?>[] {classToMatch} : new Class<?>[] {FactoryBean.class, classToM-
atch});

 // Check decorated bean definition, if any: We assume it'll be easier
 // to determine the decorated bean's type than the proxy's type.
 BeanDefinitionHolder dbd = mbd.getDecoratedDefinition();
 if (dbd != null && !BeanFactoryUtils.isFactoryDereference(name)) {
  RootBeanDefinition tbd = getMergedBeanDefinition(dbd.getBeanName(), dbd.getBean-
Definition(), mbd);
  Class<?> targetClass = predictBeanType(dbd.getBeanName(), tbd, typesToMatch);
  if (targetClass != null && !FactoryBean.class.isAssignableFrom(targetClass)) {
   return typeToMatch.isAssignableFrom(targetClass);
  }
 }

 Class<?> beanType = predictBeanType(beanName, mbd, typesToMatch);
 if (beanType == null) {
  return false;
 }

 // Check bean class whether we're dealing with a FactoryBean.
 if (FactoryBean.class.isAssignableFrom(beanType)) {
  if (!BeanFactoryUtils.isFactoryDereference(name)) {
   // If it's a FactoryBean, we want to look at what it creates, not the factory
 class.
   beanType = getTypeForFactoryBean(beanName, mbd);
   if (beanType == null) {
    return false;
   }
  }
 }
 else if (BeanFactoryUtils.isFactoryDereference(name)) {
  // Special case: A SmartInstantiationAwareBeanPostProcessor returned a non-Fac-
toryBean
  // type but we nevertheless are being asked to dereference a FactoryBean...
  // Let's check the original bean class and proceed with it if it is a Factory-
Bean.
  beanType = predictBeanType(beanName, mbd, FactoryBean.class);
  if (beanType == null || !FactoryBean.class.isAssignableFrom(beanType)) {
   return false;
  }
 }

 ResolvableType resolvableType = mbd.targetType;
 if (resolvableType == null) {
  resolvableType = mbd.factoryMethodReturnType;
 }
 if (resolvableType != null && resolvableType.resolve() == beanType) {
  return typeToMatch.isAssignableFrom(resolvableType);
 }
 return typeToMatch.isAssignableFrom(beanType);
}

@Override

public boolean isTypeMatch(String name, @Nullable
   Class<?> typeToMatch) throws NoSuchBeanDefinitionException {
 return isTypeMatch(name, ResolvableType.forRawClass(typeToMatch));
}

@Override
public Class<?> getType(String name) throws NoSuchBeanDefinitionException {
 String beanName = transformedBeanName(name);

 // Check manually registered singletons.
 Object beanInstance = getSingleton(beanName, false);
 if (beanInstance != null) {
  if (beanInstance instanceof FactoryBean && !BeanFactoryUtils.
 isFactoryDereference(name)) {
   return getTypeForFactoryBean((FactoryBean<?>) beanInstance);
  }
  else {
   return beanInstance.getClass();
  }
 }
 else if (containsSingleton(beanName) && !containsBeanDefinition(beanName)) {
  // null instance registered
  return null;
 }

 // 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.getType(originalBeanName(name));
 }

 RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

 // Check decorated bean definition, if any: We assume it'll be easier
 // to determine the decorated bean's type than the proxy's type.
 BeanDefinitionHolder dbd = mbd.getDecoratedDefinition();
 if (dbd != null && !BeanFactoryUtils.isFactoryDereference(name)) {
  RootBeanDefinition tbd = getMergedBeanDefinition(dbd.getBeanName(), dbd.
 getBeanDefinition(), mbd);
  Class<?> targetClass = predictBeanType(dbd.getBeanName(), tbd);
  if (targetClass != null && !FactoryBean.class.isAssignableFrom(targetClass)) {
   return targetClass;
  }
 }
```

```
{ /** 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();
  if (classToMatch == null) {
   classToMatch = FactoryBean.class;
  }
  Class<?>[] typesToMatch = (FactoryBean.class == classToMatch ?
    new Class<?>[] {classToMatch} : new Class<?>[] {FactoryBean.class,
  classToMatch});

  // Check decorated bean definition, if any: We assume it'll be easier
  // to determine the decorated bean's type than the proxy's type.
  BeanDefinitionHolder dbd = mbd.getDecoratedDefinition();
  if (dbd != null && !BeanFactoryUtils.isFactoryDereference(name)) {
   RootBeanDefinition tbd = getMergedBeanDefinition(dbd.getBeanName(), dbd.
  getBeanDefinition(), mbd);
   Class<?> targetClass = predictBeanType(dbd.getBeanName(), tbd, typesToMatch);
   if (targetClass != null && !FactoryBean.class.isAssignableFrom(targetClass)) {
    return typeToMatch.isAssignableFrom(targetClass);
   }
  }

  Class<?> beanType = predictBeanType(beanName, mbd, typesToMatch);
  if (beanType == null) {
   return false;
  }

  // Check bean class whether we're dealing with a FactoryBean.
  if (FactoryBean.class.isAssignableFrom(beanType)) {
   if (!BeanFactoryUtils.isFactoryDereference(name)) {
    // If it's a FactoryBean, we want to look at what it creates, not the factory
  class.
    beanType = getTypeForFactoryBean(beanName, mbd);
    if (beanType == null) {
     return false;
    }
   }
  }
  else if (BeanFactoryUtils.isFactoryDereference(name)) {
   // Special case: A SmartInstantiationAwareBeanPostProcessor returned a non-
  FactoryBean
   // type but we nevertheless are being asked to dereference a FactoryBean...
   // Let's check the original bean class and proceed with it if it is a
  FactoryBean.
   beanType = predictBeanType(beanName, mbd, FactoryBean.class);
   if (beanType == null || !FactoryBean.class.isAssignableFrom(beanType)) {
    return false;
   }
  }

  ResolvableType resolvableType = mbd.targetType;
  if (resolvableType == null) {
   resolvableType = mbd.factoryMethodReturnType;
  }
  if (resolvableType != null && resolvableType.resolve() == beanType) {
   return typeToMatch.isAssignableFrom(resolvableType);
  }
  return typeToMatch.isAssignableFrom(beanType);
 }

 @Override
 public boolean isTypeMatch(String name, @Nullable Class<?> typeToMatch) throws
  NoSuchBeanDefinitionException {
  return isTypeMatch(name, ResolvableType.forRawClass(typeToMatch));
 }

 @Override
 public Class<?> getType(String name) throws NoSuchBeanDefinitionException {
  String beanName = transformedBeanName(name);

  // Check manually registered singletons.
  Object beanInstance = getSingleton(beanName, false);
  if (beanInstance != null) {
   if (beanInstance instanceof FactoryBean && !BeanFactoryUtils.
  isFactoryDereference(name)) {
    return getTypeForFactoryBean((FactoryBean<?>) beanInstance);
   }
   else {
    return beanInstance.getClass();
   }
  }
  else if (containsSingleton(beanName) && !containsBeanDefinition(beanName)) {
   // null instance registered
   return null;
  }

  // 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.getType(originalBeanName(name));
  }

  RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

  // Check decorated bean definition, if any: We assume it'll be easier
  // to determine the decorated bean's type than the proxy's type.
```

```
{ /** 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();
  if (classToMatch == null) {
   classToMatch = FactoryBean.class;
  }
  Class<?>[] typesToMatch = (FactoryBean.class == classToMatch ?
    new Class<?>[] {classToMatch} : new Class<?>[] {FactoryBean.class,
  classToMatch});

  // Check decorated bean definition, if any: We assume it'll be easier
  // to determine the decorated bean's type than the proxy's type.
  BeanDefinitionHolder dbd = mbd.getDecoratedDefinition();
  if (dbd != null && !BeanFactoryUtils.isFactoryDereference(name)) {
   RootBeanDefinition tbd = getMergedBeanDefinition(dbd.getBeanName(), dbd.
  getBeanDefinition(), mbd);
   Class<?> targetClass = predictBeanType(dbd.getBeanName(), tbd, typesToMatch);
   if (targetClass != null && !FactoryBean.class.isAssignableFrom(targetClass)) {
    return typeToMatch.isAssignableFrom(targetClass);
   }
  }

  Class<?> beanType = predictBeanType(beanName, mbd, typesToMatch);
  if (beanType == null) {
   return false;
  }

  // Check bean class whether we're dealing with a FactoryBean.
  if (FactoryBean.class.isAssignableFrom(beanType)) {
   if (!BeanFactoryUtils.isFactoryDereference(name)) {
    // If it's a FactoryBean, we want to look at what it creates, not the factory
  class.
    beanType = getTypeForFactoryBean(beanName, mbd);
    if (beanType == null) {
     return false;
    }
   }
  }
  else if (BeanFactoryUtils.isFactoryDereference(name)) {
   // Special case: A SmartInstantiationAwareBeanPostProcessor returned a non-
  FactoryBean
   // type but we nevertheless are being asked to dereference a FactoryBean...
   // Let's check the original bean class and proceed with it if it is a
  FactoryBean.
   beanType = predictBeanType(beanName, mbd, FactoryBean.class);
   if (beanType == null || !FactoryBean.class.isAssignableFrom(beanType)) {
    return false;
   }
  }

  ResolvableType resolvableType = mbd.targetType;
  if (resolvableType == null) {
   resolvableType = mbd.factoryMethodReturnType;
  }
  if (resolvableType != null && resolvableType.resolve() == beanType) {
   return typeToMatch.isAssignableFrom(resolvableType);
  }
  return typeToMatch.isAssignableFrom(beanType);
 }

 @Override
 public boolean isTypeMatch(String name, @Nullable Class<?> typeToMatch) throws
  NoSuchBeanDefinitionException {
  return isTypeMatch(name, ResolvableType.forRawClass(typeToMatch));
 }

 @Override
 public Class<?> getType(String name) throws NoSuchBeanDefinitionException {
  String beanName = transformedBeanName(name);

  // Check manually registered singletons.
  Object beanInstance = getSingleton(beanName, false);
  if (beanInstance != null) {
   if (beanInstance instanceof FactoryBean && !BeanFactoryUtils.
  isFactoryDereference(name)) {
    return getTypeForFactoryBean((FactoryBean<?>) beanInstance);
   }
   else {
    return beanInstance.getClass();
   }
  }
  else if (containsSingleton(beanName) && !containsBeanDefinition(beanName)) {
   // null instance registered
   return null;
  }

  // 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.getType(originalBeanName(name));
  }

  RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
```

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();
```

Numb. 1.

The Oxford Gazette.

Published by Authority.

Oxon, Nov. 7.

THis day the Reverend Dr. *Walter Blandford*, Warden of *Wadham Colledge* in this University, was Elected Lord Bishop of this See; vacant by the death of Dr. *Paul*, late Bishop here.

Oxon, Nov. 12. This day His Majesty in Councel, according to the usual custom, having the Roll of Sheriffs presented to him, pricked these persons following, to be Sheriffs for the succeeding year, in their respective Counties of *England* and *Wales*.

Berks.	Basil Brent, *Esquire.*
Bedford.	Tho. Snagge, *Esq;*
Buckingham.	Simon Bennet, *Esq;*
Cumberland.	Sir William Dalston, *Baronet.*
Chester.	Sir John Arderne, *Knight.*
Cambridge.	Sir Tho. Willis, Kt. *and Baronet.*
Cornwal	Tho. Porrel, *Esq;*

sieur *de Caniſſac* having been put to death by the Commissioners of the *Grands Jours*: It seems they had laid some new Taxes or Impositions on those parts. There are Troups marching against them, and it is thought they will soon be reduced. My Lord *Aubigny* Lord Almoner to her Majesty, having lain sick some time here of an Hydropsie attended with a Flux, is this week dead.

Paris, Nov. 18. The Mareschal *de Turenne* arrived here on Sunday last from the Frontiers, whence he brings account that the Succors intended against the Prince of *Munster* had passed in small parties, and that they had been received at *Maestricht* by Monsieur *Beverning* in the name of the States-General.

Guernzey, Octob. 20. Yesterday came into our Road the *Unity* Frigat, Captain *Trafford* Commander, who brought in a Prize Captain *John Gilson* of *Flushing*, being a Privateer of 7 Guns, and 45 Men.

Chatham, Nov. 4. Captain *Eliot* Commander of the *Sapphire* has taken a Busses, two of them our Scots the

Game source code deserves stylish layout & typography

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i = * ( long * ) &y;                       // evil floating point
                                               // bit level hacking

    i = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration,
                                               // this can be removed
  return y;
}
```

Improvements

1. Modern typeface
2. Colour scheme more appropriate for game code
3. Header style for function name
4. Two-column layout
5. Pull-out code comments
6. Magic number highlight and abbreviate to ‘WTF’
7. Special characters, superscripts and subscripts
8. Tighten up magic number kerning

Q_rsqrt

```
float
(float number)                                                    {
  long i;
  float x2, y;
  const float threehalfs = 1·5F;
  x2 = number * 0·5F;
  y  = number;
  i  = * ( long * ) &y;
  i  = 0x5f3759df - ( i ≫ 1 );
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );
  // y  = y * ( threehalfs - ( x2 * y * y ) );
  return y;                                                       }
```

evil floating point bit level hacking

WTF?

1st iteration

2nd iteration, this can be removed

Corporate Code

Scheduled Executor Factory Bean

package org.springframework.scheduling.concurrent

@author Juergen Hoeller
@since 2.0
@see #setPoolSize
@see #setRemoveOnCancelPolicy
@see #setThreadFactory
@see ScheduledExecutorTask
@see java.util.concurrent.ScheduledExecutorService
@see java.util.concurrent.ScheduledThreadPoolExecutor

License

`org.springframework.beans.factory.FactoryBean` that sets up a `java.util.concurrent.ScheduledExecutorService` (by default: a `java.util.concurrent.ScheduledThreadPoolExecutor`) and exposes it for bean references.

Allows for registration of `ScheduledExecutorTask` `ScheduledExecutorTasks`, automatically starting the `ScheduledExecutorService` on initialization and cancelling it on destruction of the context. In scenarios that only require static registration of tasks at startup, there is no need to access the `ScheduledExecutorService` instance itself in application code at all; `ScheduledExecutorFactoryBean` is then just being used for lifecycle integration.

For an alternative, you may set up a `ScheduledThreadPoolExecutor` instance directly using constructor injection, or use a factory method definition that points to the `java.util.concurrent.Executors`

Scheduled Executor Factory Bean

```
package org.springframework.scheduling.concurrent
```

@author Juergen Hoeller
@since 2.0
@see **#setPoolSize**
@see **#setRemoveOnCancelPolicy**
@see **#setThreadFactory**
@see **ScheduledExecutorTask**
@see **java.util.concurrent.ScheduledExecutorService**
@see **java.util.concurrent.ScheduledThreadPoolExecutor**

License

Import

```
java.util.concurrent.ExecutorService
java.util.concurrent.Executors
java.util.concurrent.RejectedExecutionHandler
java.util.concurrent.ScheduledExecutorService
java.util.concurrent.ScheduledThreadPoolExecutor
java.util.concurrent.ThreadFactory

org.springframework.beans.factory.FactoryBean
org.springframework.lang.Nullable
org.springframework.scheduling.support.
    DelegatingErrorHandlingRunnable
org.springframework.scheduling.support.TaskUtils
org.springframework.util.Assert
org.springframework.util.ObjectUtils
```

org.springframework.beans.factory.FactoryBean that sets up a **java.util.concurrent.ScheduledExecutorService** (by default: a **java.util.concurrent.ScheduledThreadPoolExecutor**) and exposes it for bean references.

Allows for registration of **ScheduledExecutorTask ScheduledExecutorTasks**, automatically starting the **ScheduledExecutorService** on initialization and cancelling it on destruction of the context. In scenarios that only require static registration of tasks at startup, there is no need to access the **ScheduledExecutorService** instance itself in application code at all; **ScheduledExecutorFactoryBean** is then just being used for lifecycle integration.

For an alternative, you may set up a **ScheduledThreadPoolExecutor** instance directly using constructor injection, or use a factory method definition that points to the **java.util.concurrent.Executors** class. **This is strongly recommended in particular for common @Bean methods in configuration classes, where this FactoryBean variant would force you to return the FactoryBean type instead of ScheduledExecutorService.**

Note that **java.util.concurrent.ScheduledExecutorService** uses a **Runnable** instance that is shared between repeated executions, in contrast to Quartz which instantiates a new Job for each execution.

WARNING: Runnable Runnables submitted via a native **ScheduledExecutorService** are removed from the execution schedule once they throw an exception. If you would prefer to continue execution after such an exception, switch this **FactoryBean**'s **continueScheduledExecutionAfterException** property to **true**.

```
@SuppressWarnings("serial") public class
ScheduledExecutorFactoryBean
 extends ExecutorConfigurationSupport implements FactoryBean<ScheduledExecutorService> {

    private int poolSize = 1;

    @Nullable
    private ScheduledExecutorTask[] scheduledExecutorTasks;

    private boolean removeOnCancelPolicy = false;
    private boolean continueScheduledExecutionAfterException = false;
    private boolean exposeUnconfigurableExecutor = false;

    @Nullable
    private ScheduledExecutorService exposedExecutor;
```

Set the **ScheduledExecutorService**'s pool size. Default is 1.

```
    public void
    setPoolSize
    (int poolSize) {
        Assert.isTrue(poolSize > 0, "'poolSize' must be 1 or higher");
        this.poolSize = poolSize;
    }
```

Register a list of **ScheduledExecutorTask** objects with the **ScheduledExecutorService** that this **FactoryBean** creates. Depending on each **ScheduledExecutorTask**'s settings, it will be registered via one of **ScheduledExecutorService**'s schedule methods.

```
    public void
    setScheduledExecutorTasks
    (ScheduledExecutorTask... scheduledExecutorTasks) {
        this.scheduledExecutorTasks = scheduledExecutorTasks;
    }
```

Set the remove-on-cancel mode on **ScheduledThreadPoolExecutor** (JDK 7+). Default is **false**. If set to **true**, the target executor will be switched into remove-on-cancel mode (if possible, with a soft fallback otherwise).

```
    public void
    setRemoveOnCancelPolicy
    (boolean removeOnCancelPolicy) {
        this.removeOnCancelPolicy = removeOnCancelPolicy;
    }
```

Specify whether to continue the execution of a scheduled task after it threw an exception. Default is **false**, matching the native behavior of a **java.util.concurrent.ScheduledExecutorService**. Switch this flag to **true** for exception-proof execution of each task, continuing scheduled execution as in the case of successful execution.

```
    public void
    setContinueScheduledExecutionAfterException
    (boolean continueScheduledExecutionAfterException) {
        this.continueScheduledExecutionAfterException = continueScheduledExecutionAfterException;
    }
```

Specify whether this **FactoryBean** should expose an unconfigurable decorator for the created executor. Default is **false**, exposing the raw executor as bean reference. Switch this flag to **true** to strictly prevent clients from modifying the executor's configuration.

```
    public void
    setExposeUnconfigurableExecutor
    (boolean exposeUnconfigurableExecutor) {
        this.exposeUnconfigurableExecutor = exposeUnconfigurableExecutor;
    }

    @Override
    protected ExecutorService
    initializeExecutor
    (ThreadFactory threadFactory, RejectedExecutionHandler rejectedExecutionHandler) {

        ScheduledExecutorService executor = createExecutor(this.poolSize, threadFactory, rejectedExecutionHandler);

        if (this.removeOnCancelPolicy) {
            if (executor instanceof ScheduledThreadPoolExecutor) {
                ((ScheduledThreadPoolExecutor) executor).setRemoveOnCancelPolicy(true);
            }
            else {
                logger.info("Could not apply remove-on-cancel policy - not a Java 7+ ScheduledThreadPoolExecutor");
            }
        }
```

Register specified **ScheduledExecutorTasks**, if necessary.

```
        if (!ObjectUtils.isEmpty(this.scheduledExecutorTasks)) {
            registerTasks(this.scheduledExecutorTasks, executor);
        }
```

Wrap executor with an unconfigurable decorator.

```
        this.exposedExecutor = (this.exposeUnconfigurableExecutor ?
            Executors.unconfigurableScheduledExecutorService(executor) : executor);

        return executor;
    }
```

Create a new **ScheduledExecutorService** instance. The default implementation creates a **ScheduledThreadPoolExecutor**. Can be overridden in subclasses to provide custom **ScheduledExecutorService** instances.

@param **poolSize** the specified pool size
@param **threadFactory** the **ThreadFactory** to use
@return a new **ScheduledExecutorService** instance

```
    protected ScheduledExecutorService
    createExecutor
    (int poolSize, ThreadFactory threadFactory, RejectedExecutionHandler rejectedExecutionHandler) {
        return new ScheduledThreadPoolExecutor(poolSize, threadFactory, rejectedExecutionHandler);
    }
```

Register the specified **ScheduledExecutorTask**s on the given ScheduledExecutorService.

@param **tasks** the specified **ScheduledExecutorTasks** (never empty)
@param **executor** the **ScheduledExecutorService** to register the tasks on.

```
    protected void
    registerTasks
    (ScheduledExecutorTask[] tasks, ScheduledExecutorService executor) {
        for (ScheduledExecutorTask task : tasks) {
            Runnable runnable = getRunnableToSchedule(task);
            if (task.isOneTimeTask()) {
                executor.schedule(runnable, task.getDelay(), task.getTimeUnit());
            }
            else {
                if (task.isFixedRate()) {
                    executor.scheduleAtFixedRate(runnable, task.getDelay(), task.getPeriod(), task.getTimeUnit());
                }
                else {
```

```
@Override
protected ExecutorService
initializeExecutor
(ThreadFactory threadFactory, RejectedExecutionHandler rejectedExecutionHandler) {

    ScheduledExecutorService executor = createExecutor(this.poolSize, threadFactory, re

    if (this.removeOnCancelPolicy) {
        if (executor instanceof ScheduledThreadPoolExecutor) {
            ((ScheduledThreadPoolExecutor) executor).setRemoveOnCancelPolicy(true);
        }
        else {
            logger.info("Could not apply remove-on-cancel policy - not a Java 7+ Scheduled
        }
    }
```

We no longer need

`fixed-width typefaces`

now that we have

high-res monitors

Interesting code deserves infographic-style design, and belongs on A0 posters

Every development team should have a designer who does the layout and typography for the code

@PeterHilton

http://hilton.org.uk/presentations/beautiful-code