This section was originally published in the first edition of the book, Learning Java, as part of the Java Beans chapter. Although the BeanContext and BeanContextServices remain viable APIs, they have not been widely used and so we have moved this information to the CD pending their wider application.

BeanContext and BeanContextServices

To build advanced, extensible applications we'd like a way for Java Beans to find each other or "rendezvous" at runtime. The `java.beans.beancontext` package provides this kind of container environment. It also provides a generic "services" lookup mechanism for Beans that wish to advertise their capabilities. These mechanisms have existed for some time, but haven't found much use in the standard Java packages. Still, they are interesting and important facilities that you can use in your own applications.

To use the bean context, a Bean must implement the `BeanContextChild` interface (or a subinterface). Beans that implement the interface are passed a reference to the container's `BeanContext`, which is the source for all container-related information. The `BeanContext` implements interfaces that describe the Bean environment. (To use it, simply cast it appropriately.) The two interfaces that we will talk about here are `Collection` and `BeanContextServices`.

A *bean collection* is a `java.util.Collection` that can be used to iterate through the Bean instances in the container. A corresponding listener interface lets Beans find out when Beans are added or removed. The `BeanContextServices` interface provides a means for looking up Beans that provide services, either by the Bean's class type or through a list of the currently available Beans. There is a corresponding listener interface for being informed of new services and revoked services.

Doing much with these classes is a bit beyond the scope of this chapter, but we'll show a simple example that lists the beans in the current container and the available services. Here's the code; Figure 19-11 shows what the bean looks like.

```
//file: ShowContext.java
package magicbeans;
import javax.swing.*;
import java.beans.beancontext.*;
import java.util.*;

public class ShowContext extends JTabbedPane
                         implements BeanContextProxy {
  BeanContext context;
  BeanContextServices services;
  JList servicesList = new JList(), beansList = new JList( );

  public ShowContext( ) {
    addTab( "Beans", new JScrollPane( beansList ) );
    addTab( "Services", new JScrollPane( servicesList ) );
  }

  private BeanContextChildSupport beanContextChild =
    new BeanContextChildSupport( ) {

    public void initializeBeanContextResources( )  {
      context= getBeanContext( );
      try  {
        services = (BeanContextServices)context;
      } catch (ClassCastException ex){/*No BeanContextServices*/}
      updateBeanList( );
      updateServicesList( );

      context.addBeanContextMembershipListener(
        new BeanContextMembershipListener( ) {
          public void childrenAdded(
            BeanContextMembershipEvent e){
            updateBeanList( );
          }
          public void childrenRemoved(
            BeanContextMembershipEvent e){
            updateBeanList( );
          }
        } );
      services.addBeanContextServicesListener(
        new BeanContextServicesListener( ) {
          public void serviceAvailable(
                BeanContextServiceAvailableEvent e ) {
            updateServicesList( );
          }
          public void serviceRevoked(
                BeanContextServiceRevokedEvent e ) {
            updateServicesList( );
          }
        } );
    }
  };

  void updateServicesList( ) {
    if ( services == null )
      return;
    Iterator it = services.getCurrentServiceClasses( );
    Vector v = new Vector( );
    while( it.hasNext( ) )
      v.addElement( it.next( ) );
    servicesList.setListData( v );
  }
  void updateBeanList( ) {
    Iterator it = context.iterator( );
    Vector v = new Vector( );
    while( it.hasNext( ) )
      v.addElement( it.next( ) );
    beansList.setListData( v );
  }

  public BeanContextChild getBeanContextProxy( ) {
    return beanContextChild;
  }
}
```

Reporting on a bean collection

The purpose of the `BeanContextChild` interface is to flag the bean as a child of the container and allow it to receive a reference to the `BeanContext`. But if you look at the code, you won't see any `BeanContextChild`. Instead, you see that our example implements something called `BeanContextProxy`. What's the story? Where is our `BeanContextChild`?

The easiest way to answer this question is to think about some of the design decisions that must have gone into the `beancontext` package. `BeanContextChild` isn't an extremely complicated interface, but it still has a half-dozen or so methods, and Sun's engineers didn't want to oblige Bean authors to implement all of those methods whenever they wanted an object that implements `BeanContextChild`. There's an obvious solution to this problem that has been used successfully elsewhere in Java: creating an adapter class that implements `BeanContextChild`, and letting the Bean author override only those methods that he or she is interested in. This adapter class is called `BeanContextChildSupport`—but it solves one problem only to create another.

If we want an object that implements `BeanContextChild`, we can certainly extend `BeanContextChildSupport`, but that's ugly: it forces all of our Beans that need to know about the context to extend a particular class, which severely limits your design flexibility.

A better solution is to introduce another interface, `BeanContextProxy`, with a single method, `getBeanContextProxy( )`, that returns a `BeanContextChild`. If we implement `BeanContextProxy`, the Bean's container will call `getBeanContextProxy( )` to get a child object for us. We can then implement `getBeanContextProxy( )` easily, using a private anonymous inner class that extends `BeanContextChildSupport`, and override only the methods that are important to us. In our private instance, we override one method: `initializeBeanContextResources( )`. This method gives us a hook on which to hang any code that needs to run after the container has given us our reference to the `BeanContext`. Within this method we can safely ask for the `BeanContext` with the `getBeanContext( )` method and use it to look up Beans or services. There are other ways around this problem, but the one described here is as effective as any and requires minimal work.

Once we have a `BeanContext`, we can use it to look at the Beans and services that are available in our container. Getting the list of Beans available is simple: we just call the `iterator( )` method of the bean context to get an `Iterator` that lets us extract references to the Beans. We do this in a helper method called `updateBeanList( )`. Likewise, getting a list of the services available requires casting the `BeanContext` to a `BeanContextServices` object, calling `getCurrent-ServiceClasses( )` to retrieve an `Iterator` that lists the services, and extracting the services from the `Iterator`. Again, we use a helper method called `updateServicesList( )` to do most of this work. Finally, the Beans and services available can change at any time, so we register with the `BeanContext` as a `BeanContextMembershipListener` to find out about changes to the Beans, and as a `BeanContextServicesListener` to find out about changes to the services. `BeanContextMembershipListener` requires two methods, `childrenAdded( )` and `childrenRemoved( )`, both of which handle a `BeanContextMembershipEvent`. Likewise, `BeanContextServicesListener` requires the methods `serviceAvailable( )` and `serviceRevoked( )`, which handle (respectively) `BeanContextServiceAvailableEvent`s and `BeanContextServiceRevokedEvent`s.

Whew! That was a dense example. But there's really not much to it when you pick it apart. Next we'll move on a bit and talk about some additional JavaBeans-related APIs.