RMI ARCHITECTURE

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

### stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

### skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

Java RMI Example

The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

### 1) create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

1. **import** java.rmi.\*;
2. **public** **interface** Adder **extends** Remote{
3. **public** **int** add(**int** x,**int** y)**throws** RemoteException;
4. }

### 2) Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

* Either extend the UnicastRemoteObject class,
* or use the exportObject() method of the UnicastRemoteObject class

In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.

1. **import** java.rmi.\*;
2. **import** java.rmi.server.\*;
3. **public** **class** AdderRemote **extends** UnicastRemoteObject **implements** Adder{
4. AdderRemote()**throws** RemoteException{
5. **super**();
6. }
7. **public** **int** add(**int** x,**int** y){**return** x+y;}
8. }

### 3) create the stub and skeleton objects using the rmic tool.

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

1. rmic AdderRemote

### 4) Start the registry service by the rmiregistry tool

Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

1. rmiregistry 5000

### 5) Create and run the server application

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. The Naming class provides 5 methods.

|  |  |
| --- | --- |
| public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException; | It returns the reference of the remote object. |
| public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException; | It binds the remote object with the given name. |
| public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException; | It destroys the remote object which is bound with the given name. |
| public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException; | It binds the remote object to the new name. |
| public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException; | It returns an array of the names of the remote objects bound in the registry. |

In this example, we are binding the remote object by the name sonoo.

1. **import** java.rmi.\*;
2. **import** java.rmi.registry.\*;
3. **public** **class** MyServer{
4. **public** **static** **void** main(String args[]){
5. **try**{
6. Adder stub=**new** AdderRemote();
7. Naming.rebind("rmi://localhost:5000/sonoo",stub);
8. }**catch**(Exception e){System.out.println(e);}
9. }
10. }

### 6) Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

1. **import** java.rmi.\*;
2. **public** **class** MyClient{
3. **public** **static** **void** main(String args[]){
4. **try**{
5. Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
6. System.out.println(stub.add(34,4));
7. }**catch**(Exception e){}
8. }
9. }
10. For running **this** rmi example,
12. 1) compile all the java files
14. javac \*.java
16. 2)create stub and skeleton object by rmic tool
18. rmic AdderRemote
20. 3)start rmi registry in one command prompt
22. rmiregistry 5000
24. 4)start the server in another command prompt
26. java MyServer
28. 5)start the client application in another command prompt
30. java MyClient

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

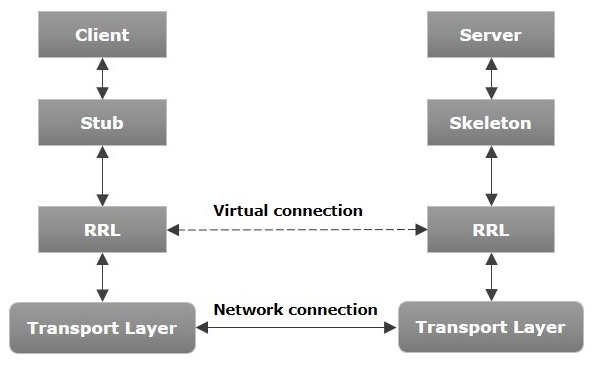
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

* Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
* The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

* **Transport Layer** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
* **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
* **Skeleton** − This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.
* **RRL(Remote Reference Layer)** − It is the layer which manages the references made by the client to the remote object.

## Working of an RMI Application

The following points summarize how an RMI application works −

* When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
* When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
* The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
* The result is passed all the way back to the client.

## Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as **marshalling**.

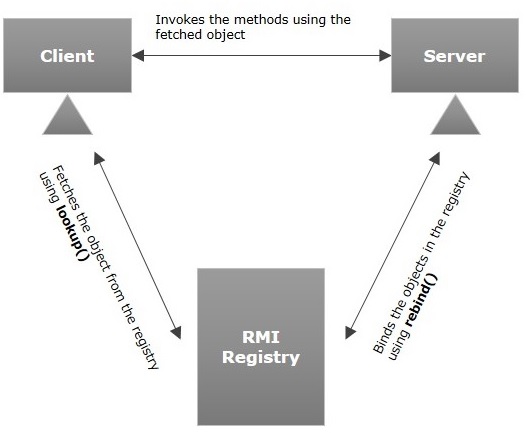
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using **bind()** or **reBind()** methods). These are registered using a unique name known as **bind name**.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).

The following illustration explains the entire process −



Goals of RMI

Following are the goals of RMI −

* To minimize the complexity of the application.
* To preserve type safety.
* Distributed garbage collection.
* Minimize the difference between working with local and remote objects.

Developing the Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the **RMIregistry**.

To develop a server program −

* Create a client class from where you want invoke the remote object.
* **Create a remote object** by instantiating the implementation class as shown below.
* Export the remote object using the method **exportObject()** of the class named **UnicastRemoteObject** which belongs to the package **java.rmi.server**.
* Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
* Bind the remote object created to the registry using the **bind()** method of the class named **Registry**. To this method, pass a string representing the bind name and the object exported, as parameters.

Following is an example of an RMI server program.

import java.rmi.registry.Registry;

import java.rmi.registry.LocateRegistry;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class Server extends ImplExample {

public Server() {}

public static void main(String args[]) {

try {

// Instantiating the implementation class

ImplExample obj = new ImplExample();

// Exporting the object of implementation class

// (here we are exporting the remote object to the stub)

Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

// Binding the remote object (stub) in the registry

Registry registry = LocateRegistry.getRegistry();

registry.bind("Hello", stub);

System.err.println("Server ready");

} catch (Exception e) {

System.err.println("Server exception: " + e.toString());

e.printStackTrace();

}

}

}

Developing the Client Program

Write a client program in it, fetch the remote object and invoke the required method using this object.

To develop a client program −

* Create a client class from where your intended to invoke the remote object.
* Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
* Fetch the object from the registry using the method **lookup()** of the class **Registry** which belongs to the package **java.rmi.registry**.

To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.

* The lookup() returns an object of type remote, down cast it to the type Hello.
* Finally invoke the required method using the obtained remote object.

Following is an example of an RMI client program.

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

public class Client {

private Client() {}

public static void main(String[] args) {

try {

// Getting the registry

Registry registry = LocateRegistry.getRegistry(null);

// Looking up the registry for the remote object

Hello stub = (Hello) registry.lookup("Hello");

// Calling the remote method using the obtained object

stub.printMsg();

// System.out.println("Remote method invoked");

} catch (Exception e) {

System.err.println("Client exception: " + e.toString());

e.printStackTrace();

}

}

}

Compiling the Application

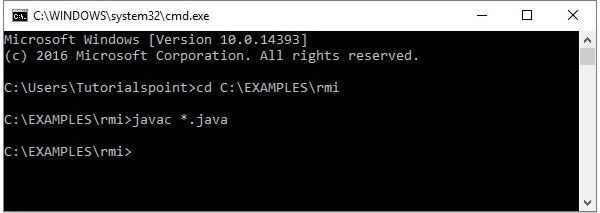
To compile the application −

* Compile the Remote interface.
* Compile the implementation class.
* Compile the server program.
* Compile the client program.

Or,

Open the folder where you have stored all the programs and compile all the Java files as shown below.

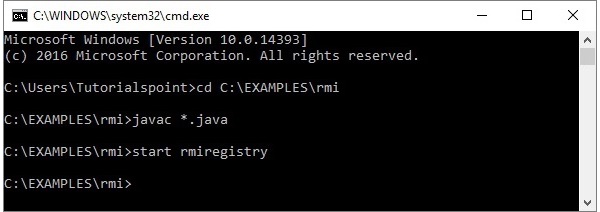
Javac \*.java



Executing the Application

**Step 1** − Start the **rmi** registry using the following command.

start rmiregistry

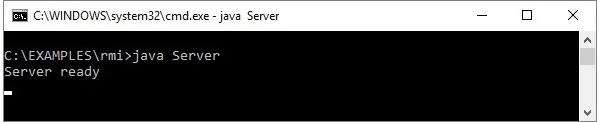


This will start an **rmi** registry on a separate window as shown below.



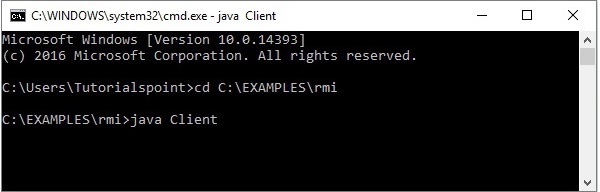
**Step 2** − Run the server class file as shown below.

Java Server

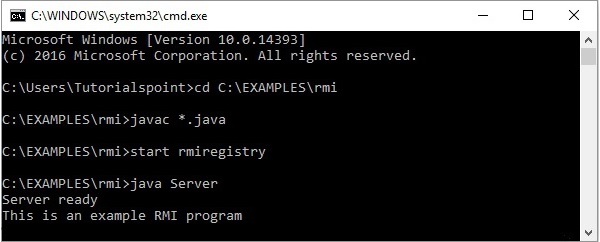


**Step 3** − Run the client class file as shown below.

java Client



**Verification** − As soon you start the client, you would see the following output in the server.



Object **Serialization**. Provides a program the ability to read or write a whole object to and from a raw byte stream. An essential feature needed by **RMI** implementation when method arguments are passed by copy.

# Serialization and Deserialization in Java with Example

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

public final void writeObject(Object obj)

throws IOException

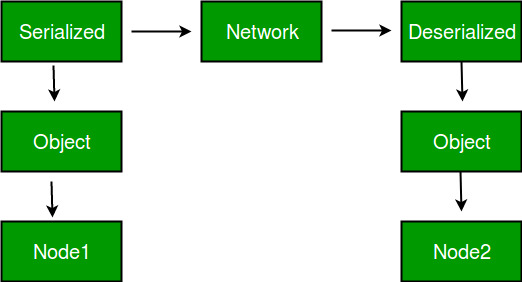
The ObjectInputStream class contains **readObject()** method for deserializing an object.

public final Object readObject()

throws IOException,

ClassNotFoundException

**Advantages of Serialization**  
1. To save/persist state of an object.  
2. To travel an object across a network.



Only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface.  
Serializable is a **marker interface** (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability.

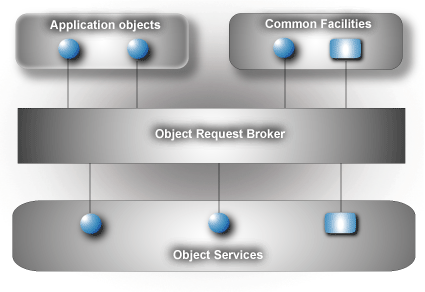
***CORBA ARCHITECTURE***

The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) to provide interoperability among distributed objects.

CORBA is essentially a design specification for an Object Request Broker (ORB), where an ORB provides the mechanism required for distributed objects to communicate with one another, whether locally or on remote devices, written in different languages, or at different locations on a network.

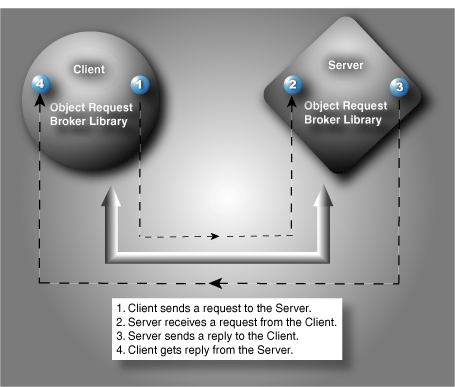
The CORBA Interface Definition Language, or IDL, allows the development of language and location-independent interfaces to distributed objects. Using CORBA, application components can communicate with one another no matter where they are located, or who has designed them. CORBA provides the location transparency to be able to execute these applications.

CORBA is often described as a "software bus" because it is a software-based communications interface through which objects are located and accessed.



Data communication from client to server is accomplished through a well-defined object-oriented interface. The Object Request Broker (ORB) determines the location of the target object, sends a request to that object, and returns any response back to the caller. Through this object-oriented technology, developers can take advantage of features such as inheritance, encapsulation, polymorphism, and runtime dynamic binding. These features allow applications to be changed, modified and re-used with minimal changes to the parent interface.

how a client sends a request to a server through the ORB:



The basic steps for CORBA development include:

http://www.ois.com/images/stories/ois/266_red_bullet1.gif**Create the IDL to Define the Application Interfaces**  
The IDL provides the operating system and programming language independent interfaces to all services and components that are linked to the ORB. The IDL specifies a description of any services a server component exposes to the client. The term "IDL Compiler" is often used, but the IDL is actually translated into a programming language.

*267_red_bullet2.gif***Translate the IDL**  
An IDL translator typically generates two cooperative parts for the client and server implementation, stub code and skeleton code. The stub code generated for the interface classes is associated with a client application and provides the user with a well-defined Application Programming Interface (API). In this example, the IDL is translated into C++.

268_red_bullet3.gif**Compile the Interface Files**Once the IDL is translated into the appropriate language, C++ in this example, these interface files are compiled and prepared for the object implementation.

269_red_bullet4.gif**Complete the Implementation**If the implementation classes are incomplete, the spec and header files and complete bodies and definitions need to be modified before passing through to be compiled. The output is a complete client/server implementation.

*270_red_bullet5.gif***Compile the Implementation**Once the implementation class is complete, the client interfaces are ready to be used in the client application and can be immediately incorporated into the client process. This client process is responsible for obtaining an object reference to a specific object, allowing the client to make requests to that object in the form of a method call on its generated API.

271_red_bullet6.gif**Link the Application**Once all the object code from steps three and five have been compiled, the object implementation classes need to be linked to the C++ linker. Once linked to the ORB library, in this example, ORB*express*, two executable operations are created, one for the client and one for the server.

272_red_bullet7.gif**Run the Client and Server**The development process is now complete and the client will now communicate with the server. The server uses the object implementation classes allowing it to communicate with the objects created by the client requests.

# The CORBA Architecture

At its core, the CORBA architecture for distributed objects shares many features with the architecture used by Java RMI. A description of a remote object is used to generate a client stub interface and a server skeleton interface for the object. A client application invokes methods on a remote object using the client stub. The method request is transmitted through the underlying infrastructure to the remote host, where the server skeleton for the object is asked to invoke the method on the object itself. Any data resulting from the method call (return values, exceptions) is transmitted back to the client by the communication infrastructure.

But that’s where the similarities between CORBA and RMI end. CORBA was designed from the start to be a language-independent distributed object standard, so it is much more extensive and detailed in its specification than RMI is (or needs to be). For the most part, these extra details are required in CORBA because it needs to support languages that have different built-in features. Some languages, like C++, directly support objects, while others, like C, don’t. The CORBA standard needs to include a detailed specification of an object model so that nonobject-oriented languages can take advantage of CORBA. Java includes built-in support for communicating object interfaces and examining them abstractly (using Java bytecodes and the Java Reflection API). Many other languages don’t. So the CORBA specification includes details about a Dynamic Invocation Interface and a Dynamic Skeleton Interface, which can be implemented in languages that don’t have their own facilities for these operations. In languages that do have these capabilities, like Java, there needs to be a mapping between the built-in features and the features as defined by the CORBA specification.

The rest of this section provides an overview of the major components that make up the CORBA architecture: the Interface Definition Language, which is how CORBA interfaces are defined; the Object Request Broker (ORB) and Object Adaptor, which are responsible for handling all interactions between remote objects and the applications that use them; the Naming Service, which is a standard service in CORBA that lets remote clients find remote objects on the network; and the inter-ORB communication protocol, which handles the low-level communication between processes in a CORBA context.

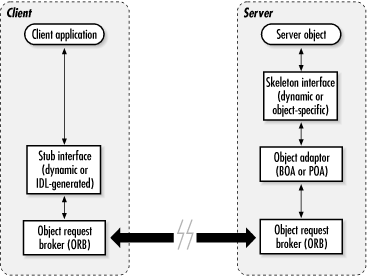
## Interface Definition Language

The Interface Definition Language provides the primary way of describing data types in CORBA. IDL is independent of any particular programming language. Mappings, or bindings, from IDL to specific programming languages are defined and standardized as part of the CORBA specification. At the time of this writing, standard bindings for C, C++, Smalltalk, Ada, COBOL, Lisp, Python and Java have been approved by the OMG. [Chapter 14](https://www.oreilly.com/library/view/java-enterprise-in/0596001525/ch14.html) contains a complete description of IDL syntax.

The central CORBA functions, services, and facilities, such as the ORB and the Naming Service, are also specified in IDL. This means that a particular language binding also specifies the bindings for the core CORBA functions to that language. Sun’s Java IDL API follows the Java IDL mapping defined by the OMG standards. This allows you to run your CORBA-based Java code in any compliant Java implementation of the CORBA standard, provided you stick to standard elements of the Java binding. Note, however, that Sun’s implementation includes some nonstandard elements; they are highlighted in this chapter where appropriate.

## The Object Request Broker and the Object Adaptor

The core of the CORBA architecture is the Object Request Broker, as shown in [Figure 4-1](https://www.oreilly.com/library/view/java-enterprise-in/0596001525/ch04s02.html#jentnut2-CHP-4-FIG-1). Each machine involved in a CORBA application must have an ORB running in order for processes on that machine to interact with CORBA objects running in remote processes. Object clients and servers make requests through their ORBs; the ORB is responsible for making the requests happen or indicating why they can’t. The client ORB provides a stub for a remote object. Requests made on the stub are transferred from the client’s ORB to the ORB servicing the implementation of the target object. The request is passed on to the implementation through an object adaptor and the object’s skeleton interface.



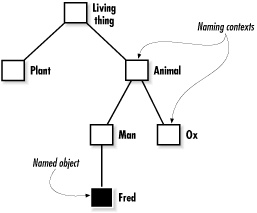
*Figure 4-1. Basic CORBA architecture*

The skeleton interface is specific to the type of object that is exported remotely through CORBA. Among other things, it provides a wrapper interface that the ORB and object adaptor can use to invoke methods on behalf of the client or as part of the lifecycle management of the object. The object adaptor provides a general facility that “plugs” a server object into a particular CORBA runtime environment. Older versions of the CORBA specification and Java IDL supported a Basic Object Adaptor (BOA) interface, while newer versions (CORBA 2.3 and later, JDK 1.4 and later) support a Portable Object Adaptor interface (we’ll discuss the difference later in the chapter). All server objects can use the object adaptor to interact with the core functionality of the ORB, and the ORB in turn can use the object adaptor to pass along client requests and lifecycle notifications to the server object. Typically, an IDL compiler is used to generate the skeleton interface for a particular IDL interface; this generated skeleton interface will include calls to the object adaptor that are supported by the CORBA environment in use.

## The Naming Service

The CORBA Naming Service (sometimes abbreviated to COSNaming, from “CORBA Object Services, Naming”) provides a directory naming structure for remote objects. The CORBA Naming Service is one of the naming and directory services supported by JNDI, so the concepts used in its API are similar to the general model of Contexts and DirContexts used in JNDI.

The naming tree always starts with a root node, and subnodes of the object tree can be created by an application. Actual objects are stored by name at the leaves of the tree. [Figure 4-2](https://www.oreilly.com/library/view/java-enterprise-in/0596001525/ch04s02.html#jentnut2-CHP-4-FIG-2) depicts an example set of objects[[12](https://www.oreilly.com/library/view/java-enterprise-in/0596001525/ch04s02.html" \l "ftn.ch04-FTNOTE-1)] registered within a Naming Service directory. The fully qualified name of an object in the directory is the ordered list of all of its parent nodes, starting from the root node and including the leaf name of the object itself. So, the full name of the object labeled “Fred” is “Living thing,” “Animal,” “Man,” “Fred,” in that order.



*Figure 4-2. A naming directory*

Each branch in the directory tree is called a naming context, and leaf objects have bindings to specific names. Each node in the naming directory is represented by an org.omg.CosNaming.NamingContext object. Each NamingContext can be asked to find an object within its branch of the tree by asking for the object by name, relative to that particular naming context. You can get a reference to the root context of the naming directory from an ORB using the resolve\_initial\_references( ) method. Once you have a reference to the root of the naming directory, you can perform lookups of CORBA objects, as well as register your own CORBA objects with the Naming Service. We’ll see more concrete details of utilizing the CORBA Naming Service later in this chapter in [Putting It in the Public Eye](https://www.oreilly.com/library/view/java-enterprise-in/0596001525/ch04s04.html).

## Inter-ORB Communication

Version 2.0 (and later) of the CORBA standard includes specifications for inter-ORB communication protocols that can transmit object requests between various ORBs running on the network. The protocols are independent of the particular ORB implementations running at either end of the communication link. An ORB implemented in Java can talk to another ORB implemented in C, as long as they’re both compliant with the CORBA standard and use the same CORBA communication protocol. The inter-ORB protocol is responsible for delivering messages between two cooperating ORBs. These messages might be method requests, return types, error messages, etc. The inter-ORB protocol also deals with differences between the two ORB implementations, like machine-level byte ordering and alignment. As a CORBA application developer, you shouldn’t have to deal directly with the low-level communication protocol between ORBs. If you want two ORBs to talk to each other, you need to ensure that they are compatible in terms of CORBA compliance levels (do they support similar levels of the CORBA specification?) and that they both speak a common, standard inter-ORB protocol.

The Internet Inter-ORB Protocol (IIOP) is an inter-ORB protocol based on TCP/IP. TCP/IP is by far the most commonly used network protocol on the Internet, so IIOP is the most commonly used CORBA communication protocol. There are other standard CORBA protocols defined for other network environments, however. The DCE Common Inter-ORB Protocol (DCE-CIOP), for example, allows ORBs to communicate on top of DCE-RPC.

## BENEFITS

CORBA's benefits include language- and OS-independence, freedom from technology-linked implementations, strong data-typing, high level of tunability, and freedom from the details of distributed data transfers.

**Language independence**

CORBA was designed to free engineers from limitations of coupling their designs to a particular software language. Currently there are many languages supported by various CORBA providers, the most popular being Java and C++. There are also C++11, C-only, Smalltalk, Perl, Ada, Ruby, and Python implementations, just to mention a few.

**OS-independence**

CORBA's design is meant to be OS-independent. CORBA is available in Java (OS-independent), as well as natively for Linux/Unix, Windows, Solaris, OS X, OpenVMS, HPUX, Android, LynxOS, VxWorks, ThreadX, INTEGRITY, and others.

**Freedom from technologies**

One of the main implicit benefits is that CORBA provides a neutral playing field for engineers to be able to normalize the interfaces between various new and legacy systems. When integrating C, C++, Object Pascal, Java, Fortran, Python, and any other language or OS into a single cohesive system design model, CORBA provides the means to level the field and allow disparate teams to develop systems and unit tests that can later be joined together into a whole system. This does not rule out the need for basic system engineering decisions, such as threading, timing, object lifetime, etc. These issues are part of any system regardless of technology. CORBA allows system elements to be normalized into a single cohesive system model.  
For example, the design of a [multitier architecture](https://en.wikipedia.org/wiki/Multitier_architecture) is made simple using [Java Servlets](https://en.wikipedia.org/wiki/Java_Servlet) in the web server and various CORBA servers containing the business logic and wrapping the database accesses. This allows the implementations of the business logic to change, while the interface changes would need to be handled as in any other technology. For example, a database wrapped by a server can have its database schema change for the sake of improved disk usage or performance (or even whole-scale database vendor change), without affecting the external interfaces. At the same time, C++ legacy code can talk to C/Fortran legacy code and Java database code, and can provide data to a web interface.

**Data-typing**

CORBA provides flexible data typing, for example an "ANY" datatype. CORBA also enforces tightly coupled datatyping, reducing human errors. In a situation where Name-Value pairs are passed around, it is conceivable that a server provides a number where a string was expected. CORBA Interface Definition Language provides the mechanism to ensure that user-code conforms to method-names, return-, parameter-types, and exceptions.

**High tunability**

Many implementations (e.g. ORBexpress (Ada, C++, and Java implementation)[[4]](https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture" \l "cite_note-4) and OmniORB (open source C++ and Python implementation))[[5]](https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture#cite_note-5) have options for tuning the threading and connection management features. Not all ORB implementations provide the same features.

**Freedom from data-transfer details**

When handling low-level connection and threading, CORBA provides a high level of detail in error conditions. This is defined in the CORBA-defined standard exception set and the implementation-specific extended exception set. Through the exceptions, the application can determine if a call failed for reasons such as "Small problem, so try again", "The server is dead" or "The reference does not make sense." The general rule is: Not receiving an exception means that the method call completed successfully. This is a very powerful design feature.

**Compression**

CORBA marshals its data in a binary form and supports compression. IONA, Remedy IT, and [Telefónica](https://en.wikipedia.org/wiki/Telefonica" \o "Telefonica) have worked on an extension to the CORBA standard that delivers compression. This extension is called ZIOP and this is now a formal OMG standard.

***JAVA BEANS***

JavaBeans are [classes](https://www.geeksforgeeks.org/classes-objects-java/)that [encapsulate](https://www.geeksforgeeks.org/encapsulation-in-java/) many objects into a single object (the bean). It is a java class that should follow following conventions:

1. Must implement [Serializable](https://www.geeksforgeeks.org/serialization-in-java/).
2. It should have a public no-arg constructor.
3. All properties in java bean must be private with public getters and setter methods.

|  |
| --- |
| // Java program to illustrate the  // structure of JavaBean class  public class TestBean {  private String name;  public void setName(String name)      {          this.name = name;      }  public String getName()      {          return name;      }  } |

**Syntax for setter methods:**

1. It should be public in nature.
2. The return-type should be void.
3. The setter method should be prefixed with set.
4. It should take some argument i.e. it should not be no-arg method.

**Syntax for getter methods**

1. It should be public in nature.
2. The return-type should not be void i.e. according to our requirement we have to give return-type.
3. The getter method should be prefixed with get.
4. It should not take any argument.

For Boolean properties getter method name can be prefixed with either “get” or “is”. But recommended to use “is”.

|  |
| --- |
| // Java program to illustrate the  // getName() method on boolean type attribute  public class Test {  private boolean empty;  public boolean getName()      {          return empty;      }  public boolean isempty()      {          return empty;      }  } |

**Implementation**

|  |
| --- |
| // Java Program of JavaBean class  package geeks;  public class Student implements java.io.Serializable  {  private int id;  private String name;  public Student()      {      }  public void setId(int id)      {          this.id = id;      }  public int getId()      {          return id;      }  public void setName(String name)      {          this.name = name;      }  public String getName()      {          return name;      }  } |

|  |
| --- |
| // Java program to access JavaBean class  package geeks;  public class Test {  public static void main(String args[])      {          Student s = new Student(); // object is created          s.setName("GFG"); // setting value to the object          System.out.println(s.getName());      }  } |

Output:

GFG

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

The name "Bean" was given to encompass this standard, which aims to create [reusable](https://en.wikipedia.org/wiki/Code_reuse) [software components](https://en.wikipedia.org/wiki/Component-based_software_engineering) for [Java](https://en.wikipedia.org/wiki/Java_(programming_language)).

It is a reusable software component written in Java that can be manipulated visually in an application builder tool.

Features

**Introspection**

Introspection is a process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API because it allows another application such as a design tool, to obtain information about a component.

**Properties**

A property is a subset of a Bean's state. The values assigned to the properties determine the behaviour and appearance of that component. They are set through a setter method and can be obtained by a getter method.

**Customization**

A customizer can provide a step-by-step guide that the process must follow to use the component in a specific context.

**Events**

Beans ( may / can / should ) ( support / interact with ) the EventObject EventListener model. *-- please help to ( correct / improve ) this! –*

**Persistence**

Persistence is the ability to save the current state of a Bean, including the values of a Bean's properties and instance variables, to nonvolatile storage and to retrieve them at a later time.

**Methods**

A bean should use [accessor methods](https://en.wikipedia.org/wiki/Mutator_method" \l "Java_example" \o "Mutator method) to [encapsulate](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)) the properties. A bean can provide other methods for business logic not related to the access to the properties.

Advantages[[edit](https://en.wikipedia.org/w/index.php?title=JavaBeans&action=edit&section=2" \o "Edit section: Advantages)]

* The properties, events, and methods of a bean can be exposed to another application.
* A bean may register to receive events from other objects and can generate events that are sent to those other objects.
* Auxiliary software can be provided to help configure a bean.
* The configuration settings of a bean can be saved to persistent storage and restored.

Disadvantages[[edit](https://en.wikipedia.org/w/index.php?title=JavaBeans&action=edit&section=3" \o "Edit section: Disadvantages)]

* A class with a [zero-argument constructor](https://en.wikipedia.org/wiki/Nullary_constructor) is subject to being instantiated in an invalid state.[[1]](https://en.wikipedia.org/wiki/JavaBeans#cite_note-Bloch-1) If such a class is instantiated manually by a developer (rather than automatically by some kind of framework), the developer might not realize that the class has been improperly instantiated. The compiler cannot detect such a problem, and even if it is documented, there is no guarantee that the developer will see the documentation.
* JavaBeans are inherently mutable and so lack the advantages offered by [immutable objects](https://en.wikipedia.org/wiki/Immutable_objects).[[1]](https://en.wikipedia.org/wiki/JavaBeans#cite_note-Bloch-1)
* Having to create getters for every property and setters for many, most, or all of them can lead to an immense quantity of [boilerplate code](https://en.wikipedia.org/wiki/Boilerplate_code).

Following are the unique characteristics that distinguish a JavaBean from other Java classes −

* It provides a default, no-argument constructor.
* It should be serializable and that which can implement the **Serializable** interface.
* It may have a number of properties which can be read or written.
* It may have a number of "**getter**" and "**setter**" methods for the properties.

JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.

A JavaBean property may be **read, write, read only**, or **write only**. JavaBean properties are accessed through two methods in the JavaBean's implementation class –

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | get**PropertyName**()  For example, if property name is *firstName*, your method name would be **getFirstName()** to read that property. This method is called accessor. |
| 2 | set**PropertyName**()  For example, if property name is *firstName*, your method name would be **setFirstName()** to write that property. This method is called mutator. |

A read-only attribute will have only a **getPropertyName()** method, and a write-only attribute will have only a **setPropertyName()** method.

JavaBeans Example

Consider a student class with few properties −

package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable {

private String firstName = null;

private String lastName = null;

private int age = 0;

public StudentsBean() {

}

public String getFirstName(){

return firstName;

}

public String getLastName(){

return lastName;

}

public int getAge(){

return age;

}

public void setFirstName(String firstName){

this.firstName = firstName;

}

public void setLastName(String lastName){

this.lastName = lastName;

}

public void setAge(Integer age){

this.age = age;

}

}

Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>

Here values for the scope attribute can be a **page, request, session** or **application based** on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other **useBean declarations** in the same JSP.

Following example shows how to use the useBean action −

<html>

<head>

<title>useBean Example</title>

</head>

<body>

<jsp:useBean id = "date" class = "java.util.Date" />

<p>The date/time is <%= date %>

</body>

</html>

You will receive the following result − −

The date/time is Thu Sep 30 11:18:11 GST 2010

Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax −

<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">

<jsp:setProperty name = "bean's id" property = "property name"

value = "value"/>

<jsp:getProperty name = "bean's id" property = "property name"/>

...........

</jsp:useBean>

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.

Following example shows how to access the data using the above syntax −

<html>

<head>

<title>get and set properties Example</title>

</head>

<body>

<jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">

<jsp:setProperty name = "students" property = "firstName" value = "Zara"/>

<jsp:setProperty name = "students" property = "lastName" value = "Ali"/>

<jsp:setProperty name = "students" property = "age" value = "10"/>

</jsp:useBean>

<p>Student First Name:

<jsp:getProperty name = "students" property = "firstName"/>

</p>

<p>Student Last Name:

<jsp:getProperty name = "students" property = "lastName"/>

</p>

<p>Student Age:

<jsp:getProperty name = "students" property = "age"/>

</p>

</body>

</html>

Let us make the **StudentsBean.class** available in CLASSPATH. Access the above JSP. the following result will be displayed −

Student First Name: Zara

Student Last Name: Ali

Student Age: 10

## JavaBeans API[[edit](https://en.wikipedia.org/w/index.php?title=JavaBeans&action=edit&section=4" \o "Edit section: JavaBeans API)]

The JavaBeans functionality is provided by a set of classes and interfaces in the java.beans package.

| **Interface** | **Description** |
| --- | --- |
| AppletInitializer | Methods in this interface are used to initialize Beans that are also [applets](https://en.wikipedia.org/wiki/Java_applet). |
| BeanInfo | This interface allows the designer to specify information about the events, methods and properties of a Bean. |
| Customizer | This interface allows the designer to provide a graphical user interface through which a bean may be configured. |
| DesignMode | Methods in this interface determine if a bean is executing in design mode. |
| ExceptionListener | A method in this interface is invoked when an exception has occurred. |
| PropertyChangeListener | A method in this interface is invoked when a bound property is changed. |
| PropertyEditor | Objects that implement this interface allow the designer to change and display property values. |
| VetoableChangeListener | A method in this interface is invoked when a Constrained property is changed. |
| Visibility | Methods in this interface allow a bean to execute in environments where the GUI is not available. |

## JavaBean conventions

In order to function as a JavaBean [class](https://en.wikipedia.org/wiki/Class_(computer_science)), an object class must obey certain conventions about method naming, construction, and behaviour. These conventions make it possible to have tools that can use, reuse, replace, and connect Java Beans.

The required conventions are as follows:

* The class must have a public [default constructor](https://en.wikipedia.org/wiki/Default_constructor) (with no arguments). This allows easy instantiation within editing and activation frameworks.
* The class [properties](https://en.wikipedia.org/wiki/Property_(programming)) must be accessible using *get*, *set*, *is* (can be used for boolean properties instead of get), *to* and other methods (so-called [accessor methods](https://en.wikipedia.org/wiki/Accessor" \o "Accessor) and [mutator methods](https://en.wikipedia.org/wiki/Mutator_method" \o "Mutator method)) according to a standard [naming convention](https://en.wikipedia.org/wiki/Naming_conventions_(programming)). This allows easy automated inspection and updating of bean state within frameworks, many of which include custom editors for various types of properties. Setters can have one or more than one argument.
* The class should be [serializable](https://en.wikipedia.org/wiki/Serialization" \l "Java" \o "Serialization). (This allows applications and frameworks to reliably save, store, and restore the bean's state in a manner independent of the [VM](https://en.wikipedia.org/wiki/Virtual_machine) and of the platform.)

### Code example[[edit](https://en.wikipedia.org/w/index.php?title=JavaBeans&action=edit&section=6" \o "Edit section: Code example)]

**package** **player**;

**public** **class** **PersonBean** **implements** java.io.Serializable {

*/\*\* Properties \*\*/*

**private** boolean deceased = **false**;

**private** List list;

*/\*\* Property "name", readable/writable. \*/*

**private** String name = **null**;

*/\*\* No-arg constructor (takes no arguments). \*/*

**public** PersonBean() {

}

**public** List getList() {

**return** list;

}

**public** void setList(**final** List list) {

**this**.list = list;

}

*/\*\**

*\* Getter for property "name".*

*\*/*

**public** String getName() {

**return** name;

}

*/\*\**

*\* Setter for property "name".*

*\**

*\* @param value*

*\*/*

**public** void setName(**final** String value) {

**this**.name = value;

}

*/\*\**

*\* Getter for property "deceased"*

*\* Different syntax for a boolean field (is v.s. get)*

*\*/*

**public** boolean isDeceased() {

**return** deceased;

}

*/\*\**

*\* Setter for property "deceased".*

*\* @param value*

*\*/*

**public** void setDeceased(boolean value) {

deceased = value;

}

}

**TestPersonBean.java**:

**import** **player.PersonBean**;

*/\*\**

*\* Class "TestPersonBean".*

*\*/*

**public** **class** **TestPersonBean** {

*/\*\**

*\* Tester method "main" for class "PersonBean".*

*\**

*\* @param arguments*

*\*/*

**public** **static** void main(**final** String[] arguments) {

**final** PersonBean person = **new** PersonBean();

person.setName("Bob");

person.setDeceased(**false**);

person.setList(**new** ArrayList());

*// Output: "Bob [alive]"*

System.out.print(person.getName());

System.out.println(person.isDeceased() ? " [deceased]" : " [alive]");

}

}

**<jsp:useBean** id="person" class="player.PersonBean" scope="page"**/>**

**<jsp:setProperty** name="person" property="\*"**/>**

**<html>**

**<body>**

Name: **<jsp:getProperty** name="person" property="name"**/><br/>**

Deceased? **<jsp:getProperty** name="person" property="deceased"**/><br/>**

**<br/>**

**<form** name="beanTest" method="POST" action="testPersonBean.jsp"**>**

Enter a name: **<input** type="text" name="name" size="50"**><br/>**

Choose an option:

**<select** name="deceased"**>**

**<option** value="false"**>**Alive**</option>**

**<option** value="true"**>**Dead**</option>**

**</select>**

**<input** type="submit" value="Test the Bean"**>**

**</form>**

**</body>**

**</html>**

## ****What are JavaBean Properties?****

A JavaBean property can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define. It may be of the following mode: read, write, read-only, or write-only. JavaBean features are accessed through two [methods](https://www.edureka.co/blog/java-methods/):

**1.*getEmployeeName ()***

For example, if the employee name is firstName, the method name would be getFirstName() to read that employee name. This method is known as an ***accessor.***Properties of getter methods are as follows:

1. Must be public in nature
2. Return-type should not be void
3. The getter method should be prefixed with the word *get*
4. It should not take any argument

**2.*setEmployeeName ()***

For example, if the employee name is firstName, the method name would be setFirstName() to write that employee name. This method is known as a ***mutator.***Properties of setter methods:

1. Must be public in nature
2. Return-type should be void
3. The setter method has to be prefixed with the word *set*
4. It should take some argument

Now that you have attained some theoretical knowledge about JavaBeans, let us move on and understand the implementation process.

**Example Program: Implementation of JavaBeans**

The example program shown below demonstrates how to implement JavaBeans.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class Employee implements java.io.Serializable  {  private int id;  private String name;  public Employee()      {      }  public void setId(int id)      {          this.id = id;      }  public int getId()      {          return id;      }  public void setName(String name)      {          this.name = name;      }  public String getName()      {          return name;      }  } |

Next program is written in order to access the JavaBean class that we created above:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Employee1 {  public static void main(String args[])  {      Employee s = new Employee();      s.setName("Chandler");      System.out.println(s.getName());  }  } |

**Output:**

Chandler

Bean DEVOLOPER KIT

The **Bean** Developer **Kit** (BDK), available from the **Java** Soft site, is a simple example of a **tool** that enables you to create, configure, and connect a set of Beans.

**The Beans Development Kit**

The BDK is delivered separately from the JDK. You can [download the BDK](http://java.sun.com/beans/software/bdk_download.html)[ (outside of the tutorial)](http://java.sun.com/beans/software/bdk_download.html) freely from the JavaBeans web site.This site contains instructions for installing the BDK on your system. Here is a general description of the BDK files and directories:

* README.html contains an entry point to the BDK documentation
* LICENSE.html contains the BDK license agreement
* GNUmakefile and Makefile are Unix and Windows makefiles (.gmk and .mk suffixes) for building the demos and the BeanBox, and for running the BeanBox
* beans/apis contains
  + a java directory containing JavaBeans source files
  + a sun directory containing property editor source files
* beans/beanbox contains
  + makefiles for building the BeanBox
  + scripts for running the BeanBox
  + a classes directory containing the BeanBox class files
  + a lib directory containing a BeanBox support jar file used by MakeApplet's produced code
  + sun and sunw directories containing BeanBox source (.java) files
  + a tmp directory containing automatically generated event adapter source and class files, .ser files, and applet files automatically generated by MakeApplet
* beans/demos contains
  + makefiles for building the demo Beans
  + an HTML directory containing an applet wrapper demonstration that must be run in appletviewer, HotJava, or JDK1.1-compliant browsers
  + a sunw directory containing
    - a wrapper directory containing a Bean applet wrapper
    - a demos directory containing demo source file
* beans/doc contains
  + demos documentation
  + a javadoc directory containing JavaBeans and JavaBeans-related class and interface documentation
  + miscellaneous documentation
* beans/jars contains jar files for demo Beans

**Introspection** is the automatic process of analyzing a bean's design patterns to reveal the bean's properties, events, and methods. ... **Introspection** uses reflection, the relationship between **Introspection** and Reflection can be seen as similar to JavaBeans and other **Java** classes

# Introspection

Introspection is the automatic process of analyzing a bean's design patterns to reveal the bean's properties, events, and methods. This process controls the publishing and discovery of bean operations and properties. This lesson explains the purpose of introspection, introduces the Introspection API, and gives an example of introspection code.

## Purpose of Introspection

A growing number of Java object repository sites exist on the Internet in answer to the demand for centralized deployment of applets, classes, and source code in general. Any developer who has spent time hunting through these sites for licensable Java code to incorporate into a program has undoubtedly struggled with issues of how to quickly and cleanly integrate code from one particular source into an application.

The way in which introspection is implemented provides great advantages, including:

1. Portability - Everything is done in the Java platform, so you can write components once, reuse them everywhere. There are no extra specification files that need to be maintained independently from your component code. There are no platform-specific issues to contend with. Your component is not tied to one component model or one proprietary platform. You get all the advantages of the evolving Java APIs, while maintaining the portability of your components.
2. Reuse - By following the JavaBeans design conventions, implementing the appropriate interfaces, and extending the appropriate classes, you provide your component with reuse potential that possibly exceeds your expectations.

## Introspection API

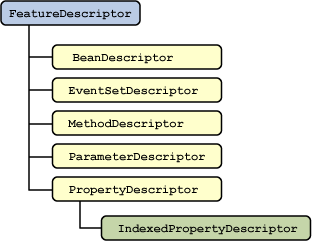
The JavaBeans API architecture supplies a set of classes and interfaces to provide introspection.

The [BeanInfo](http://download.oracle.com/javase/7/docs/api/java/beans/BeanInfo.html" \t "_blank) (in the API reference documentation) interface of the java.beans package defines a set of methods that allow bean implementors to provide explicit information about their beans. By specifying BeanInfo for a bean component, a developer can hide methods, specify an icon for the toolbox, provide descriptive names for properties, define which properties are bound properties, and much more.

The [getBeanInfo(beanName)](http://download.oracle.com/javase/7/docs/api/java/beans/Introspector.html" \l "getBeanInfo(java.lang.Class)" \t "_blank) (in the API reference documentation) of the [Introspector](http://download.oracle.com/javase/7/docs/api/java/beans/Introspector.html" \t "_blank) (in the API reference documentation) class can be used by builder tools and other automated environments to provide detailed information about a bean. The getBeanInfo method relies on the naming conventions for the bean's properties, events, and methods. A call to getBeanInfo results in the introspection process analyzing the beanÂ’s classes and superclasses.

The Introspector class provides descriptor classes with information about properties, events, and methods of a bean. Methods of this class locate any descriptor information that has been explicitly supplied by the developer through BeanInfo classes. Then the Introspector class applies the naming conventions to determine what properties the bean has, the events to which it can listen, and those which it can send.

The following figure represents a hierarchy of the FeatureDescriptor classes:



Each class represented in this group describes a particular attribute of the bean. For example, the isBound method of the [PropertyDescriptor](http://download.oracle.com/javase/7/docs/api/java/beans/PropertyDescriptor.html" \t "_blank) class indicates whether a PropertyChangeEvent event is fired when the value of this property changes.

# Properties

To define a property in a bean class, supply public getter and setter methods. For example, the following methods define an int property called mouthWidth:

public class FaceBean {

private int mMouthWidth = 90;

public int getMouthWidth() {

return mMouthWidth;

}

public void setMouthWidth(int mw) {

mMouthWidth = mw;

}

}

A builder tool like NetBeans recognizes the method names and shows the mouthWidth property in its list of properties. It also recognizes the type, int, and provides an appropriate editor so the property can be manipulated at design time.

This example shows a property than can be read and written. Other combinations are also possible. A read-only property, for example, has a getter method but no setter. A write-only property has a setter method only.

A special case for boolean properties allows the accessor method to be defined using is instead of get. For example, the accessor for a boolean property running could be as follows:

public boolean isRunning() {

// ...

}

Various specializations of basic properties are available and described in the following sections.

## Indexed Properties

An *indexed* property is an array instead of a single value. In this case, the bean class provides a method for getting and setting the entire array. Here is an example for an int[] property called testGrades:

public int[] getTestGrades() {

return mTestGrades;

}

public void setTestGrades(int[] tg) {

mTestGrades = tg;

}

For indexed properties, the bean class also provides methods for getting and setting a specific element of the array.

public int getTestGrades(int index) {

return mTestGrades[index];

}

public void setTestGrades(int index, int grade) {

mTestGrades[index] = grade;

}

## Bound Properties

A *bound* property notifies listeners when its value changes. This has two implications:

1. The bean class includes addPropertyChangeListener() and removePropertyChangeListener() methods for managing the bean's listeners.
2. When a bound property is changed, the bean sends a PropertyChangeEvent to its registered listeners.

PropertyChangeEvent and PropertyChangeListener live in the java.beans package.

The java.beans package also includes a class, PropertyChangeSupport, that takes care of most of the work of bound properties. This handy class keeps track of property listeners and includes a convenience method that fires property change events to all registered listeners.

The following example shows how you could make the mouthWidth property a bound property using PropertyChangeSupport. The necessary additions for the bound property are shown in bold.

**import java.beans.\*;**

public class FaceBean {

private int mMouthWidth = 90;

**private PropertyChangeSupport mPcs =**

**new PropertyChangeSupport(this);**

public int getMouthWidth() {

return mMouthWidth;

}

public void setMouthWidth(int mw) {

**int oldMouthWidth = mMouthWidth;**

mMouthWidth = mw;

**mPcs.firePropertyChange("mouthWidth",**

**oldMouthWidth, mw);**

}

**public void**

**addPropertyChangeListener(PropertyChangeListener listener) {**

**mPcs.addPropertyChangeListener(listener);**

**}**

**public void**

**removePropertyChangeListener(PropertyChangeListener listener) {**

**mPcs.removePropertyChangeListener(listener);**

**}**

}

Bound properties can be tied directly to other bean properties using a builder tool like NetBeans. You could, for example, take the value property of a slider component and bind it to the mouthWidth property shown in the example. NetBeans allows you to do this without writing any code.

## Constrained Properties

A *constrained* property is a special kind of bound property. For a constrained property, the bean keeps track of a set of *veto* listeners. When a constrained property is about to change, the listeners are consulted about the change. Any one of the listeners has a chance to veto the change, in which case the property remains unchanged.

The veto listeners are separate from the property change listeners. Fortunately, the java.beans package includes a VetoableChangeSupport class that greatly simplifies constrained properties.

Changes to the mouthWidth example are shown in bold:

import java.beans.\*;

public class FaceBean {

private int mMouthWidth = 90;

private PropertyChangeSupport mPcs =

new PropertyChangeSupport(this);

**private VetoableChangeSupport mVcs =**

**new VetoableChangeSupport(this);**

public int getMouthWidth() {

return mMouthWidth;

}

public void

setMouthWidth(int mw) **throws PropertyVetoException** {

int oldMouthWidth = mMouthWidth;

**mVcs.fireVetoableChange("mouthWidth",**

**oldMouthWidth, mw);**

mMouthWidth = mw;

mPcs.firePropertyChange("mouthWidth",

oldMouthWidth, mw);

}

public void

addPropertyChangeListener(PropertyChangeListener listener) {

mPcs.addPropertyChangeListener(listener);

}

public void

removePropertyChangeListener(PropertyChangeListener listener) {

mPcs.removePropertyChangeListener(listener);

}

**public void**

**addVetoableChangeListener(VetoableChangeListener listener) {**

**mVcs.addVetoableChangeListener(listener);**

**}**

**public void**

**removeVetoableChangeListener(VetoableChangeListener listener) {**

**mVcs.removeVetoableChangeListener(listener);**

**}**

}

# Bean Customization

Customization provides a means for modifying the appearance and behavior of a bean within an application builder so it meets your specific needs. There are several levels of customization available for a bean developer to allow other developers to get maximum benefit from a bean's potential functionality

# Customizers

A customizer is a user interface for customizing an entire Bean, as opposed to a single property. The characteristics and behaviors of the Bean that can be modified by a customizer are not limited to its exposed properties. There can be any number of other settings that are needed to configure a Bean that are not considered properties.

The complexity of the configuration options for a Bean is another possible reason to provide a customizer. This gives you an opportunity to present these choices in a way that makes more sense to a user, and might even provide a mechanism which can guide the user through the process.

If you provide a customizer, you must also provide a BeanInfo class for your Bean in order to identify its associated customizer class. There is no standard naming convention that can be followed, nor is there any kind of customizer manager class provided by the JDK. This is another area that seems inconsistent. I don’t see any reason why a run-time registration mechanism, like that provided by the java.beans.PropertyEditorManager, could not have been provided for registering Bean customizers at run-time. Likewise, I think a naming convention that appends the string Customizer to the Bean class name would have been useful, allowing for the creation of customizers without the need for creating a BeanInfo class. However, it is almost trivial to create a BeanInfo class solely for the purpose of specifying an associated customizer class.

**Customizers**

You have learned that builder tools provide support for you to create your own property editors. What other needs should visual builders meet for complex, industrial-strength beans? Often it is undesirable to have all the properties of a bean revealed on a single (sometimes huge) property sheet. What if one single root choice about the type of the bean rendered half the properties irrelevant? The JavaBeans specification provides for user-defined customizers, through which you can define a higher level of customization for bean properties than is available with property editors.

When you use a bean *Customizer*, you have complete control over how to configure or edit a bean. A Customizer is an application that specifically targets a bean's customization. Sometimes properties are insufficient for representing a bean's configurable attributes. Customizers are used where sophisticated instructions would be needed to change a bean, and where property editors are too primitive to achieve bean customization.

All customizers must:

* Extend java.awt.Component or one of its subclasses.
* Implement the java.beans.Customizer interface This means implementing methods to register PropertyChangeListener objects, and firing property change events at those listeners when a change to the target bean has occurred.
* Implement a default constructor.
* Associate the customizer with its target class via BeanInfo.getBeanDescriptor.

**Persistence**

*Persistence*is the ability to save the current state of a Bean, including the values of a Bean’sproperties and instance variables, to nonvolatile storage and to retrieve them at a later time. The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans.

The easiest way to serialize a Bean is to have it implement the **java.io.Serializable** interface, which is simply a marker interface. Implementing **java.io.Serializable** makes serialization automatic. Your Bean need take no other action. Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements **java.io.Serializable**, then automatic serialization is obtained.

When using automatic serialization, you can selectively prevent a field from being saved through the use of the **transient** keyword. Thus, data members of a Bean specified as **transient** will not be serialized.

If a Bean does not implement **java.io.Serializable**, you must provide serialization yourself, such as by implementing **java.io.Externalizable**. Otherwise, containers cannot save the configuration of your component.

**Customizers**

A Bean developer can provide a *customizer* that helps another developer configure the Bean. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

