

Unit 8 :
RMI and CORBA

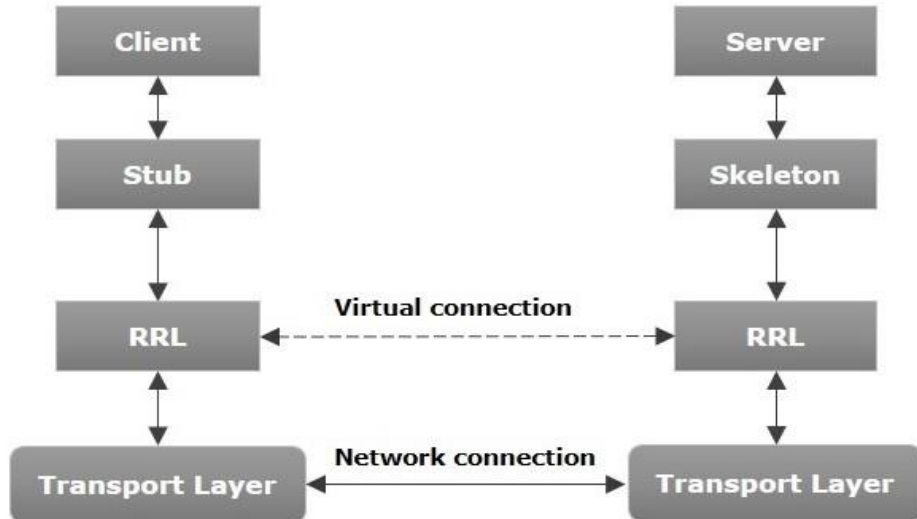
Er. Shankar pd. Dahal
pdsdahal@gmail.com

- ❖ 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**.

RMI Architecture:

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.



Transport Layer :

This layer connects the client and the server. It manages the existing connection and also sets up new connections.

Stub :

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.

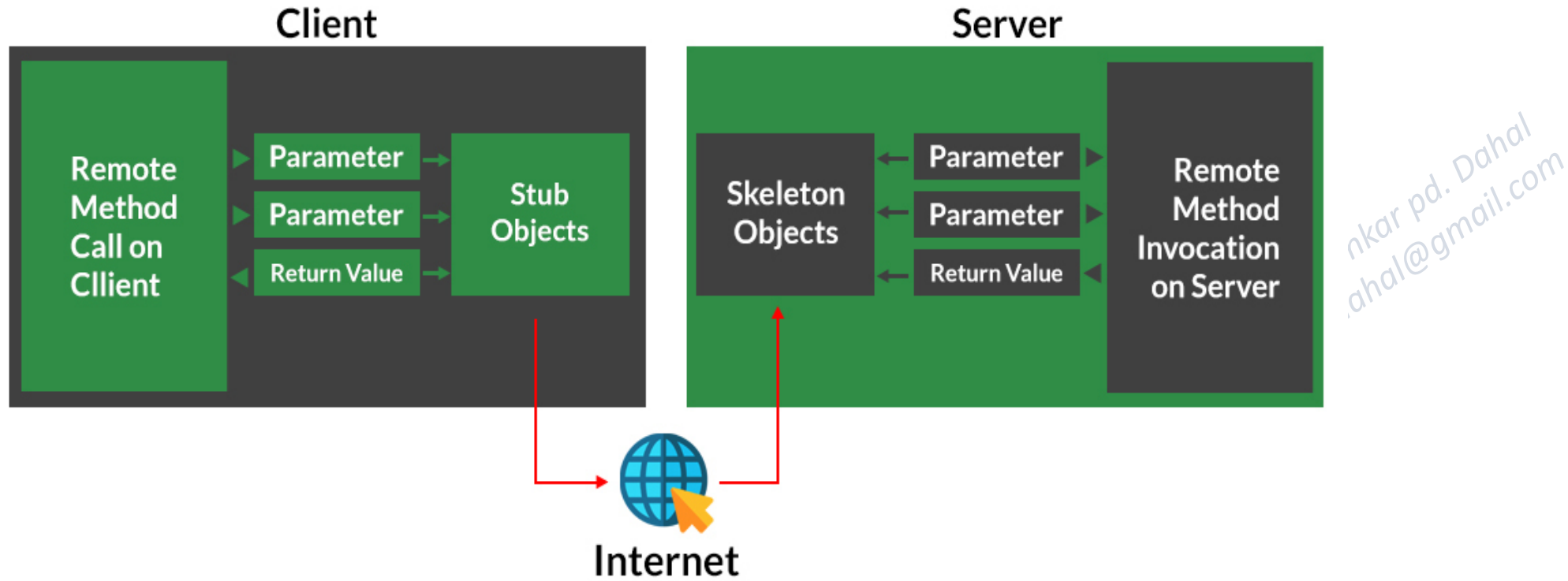
RRL(Remote Reference Layer) :

- ❖ It will identify the type of server that is whether it's unicast or multicast.
- ❖ It is the layer which manages the references made by the client to the remote object.

Er. Shambur pd. Dahal
pdsdahal@gmail.com

Working of RMI : (Roles of Client and Server)

The communication between client and server is handled by using two intermediate objects: Stub object (on client side) and Skeleton object (on server-side) as also can be depicted from below media as follows:



Server :

- This is a program that typically creates a remote object to be used for method invocation.
- This object is an ordinary object except that its class implements a Java RMI interface.
- Upon creation, the object is exported and registered with a separate application called object registry.

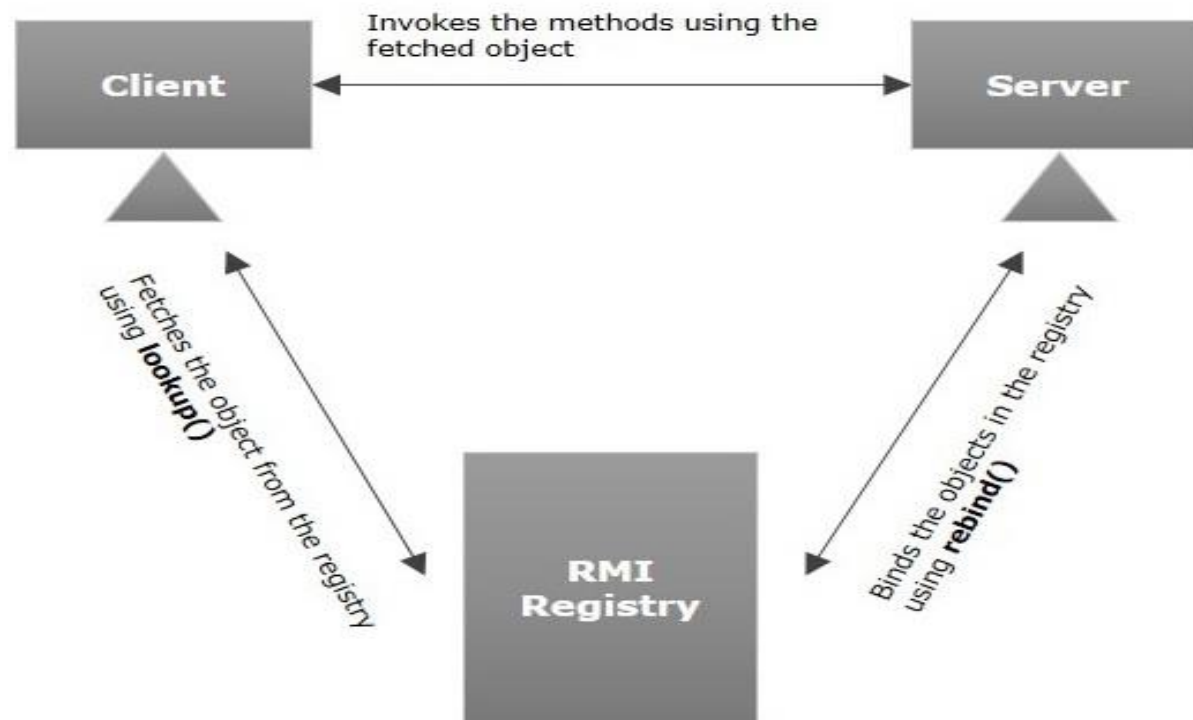
Client :

- A client program typically consults the object registry to get a reference (handle) to a remote object with a specific name.

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 RMI registry (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:



Developing and running RMI applications involves the following steps: **(RMI Programming Model)**

1. Define the remote interface
2. Implement the remote interface
3. Develop the Server program
4. Develop the Client program
5. Compile the Java source files
6. Execute the application

1. Define the remote interface :

- ❖ A remote interface provides the description of all the methods of a particular remote object.
- ❖ The client communicates with this remote interface.

To create a remote interface :

- ❖ Create an interface that extends the predefined interface **Remote** which belongs to the package.
- ❖ Declare all the business methods that can be invoked by the client in this interface.
- ❖ Since there is a chance of network issues during remote calls, an exception named **RemoteException** may occur; throw it.

E.g.

```
import java.rmi.Remote;  
import java.rmi.RemoteException;
```

```
public interface CalculatorInterface extends Remote {  
    public int addition(int num1, int num2) throws RemoteException;  
    public int multiplictaion(int num1, int num2) throws RemoteException;  
    public int division(int num1, int num2) throws RemoteException;  
    public int subtrction(int num1, int num2) throws RemoteException;  
}
```

Er. Shankar P. Dahal
pdsdahal@gmail.com

2. Implement the remote interface :

The Remote interface, we need to :

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

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

E.g.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class SimpleCalculator extends UnicastRemoteObject implements CalculatorInterface {

    SimpleCalculator() throws RemoteException {
        super();
    }

    public int addition(int num1, int num2) throws RemoteException {
        return num1+num2;
    }

    public int multiplctaion(int num1, int num2) throws RemoteException {
        return num1*num2;
    }

    public int division(int num1, int num2) throws RemoteException {
        return num1/num2;
    }

    public int subtrction(int num1, int num2) throws RemoteException {
        return num1-num2;
    }
}
```

Er. Shankar pd. Dahal
pdsdahal@gmail.com

3. Develop the Server Program :

- ❖ Create a remote object by instantiating the implementation class.
- ❖ The server program uses createRegistry method of LocateRegistry class to create rmiregistry within the server JVM with the port number passed as an argument.
- ❖ The rebind method of Naming class is used to bind the remote object to the new name.

E.g.

```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class ServerProgram {

    public static void main(String[] args) {
        try {
            SimpleCalculator calculator = new SimpleCalculator();
            LocateRegistry.createRegistry(1900);
            Naming.rebind("rmi://localhost:1900/demo", calculator);
            System.out.println("Server Started");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Er. Shankar pd. Dahal
pdsdahal@gmail.com

4. Develop the Client Program :

- ❖ The lookup method of the Naming class is used to get the reference of the Stub object.
- ❖ The lookup method returns an object of type remote and cast it.
- ❖ Finally invoke the required method using the obtained remote object.

```
import java.rmi.Naming;  
import java.util.Scanner;
```

```
public class ClientProgram {
```

```
    public static void main(String[] args) {  
        try {  
            CalculatorInterface calculatorInterface = (CalculatorInterface) Naming.lookup("rmi://localhost:1900/demo");  
            Scanner scanner = new Scanner(System.in);  
            System.out.println("Enter a first number : ");  
            int num1 = scanner.nextInt();  
  
            System.out.println("Enter a second number : ");  
            int num2 = scanner.nextInt();  
            int result1 = calculatorInterface.addition(num1, num2);  
            System.out.println("the sum of "+num1+" , "+num2+" = "+result1);  
  
            int result2 = calculatorInterface.multiplictaion(num1, num2);  
            System.out.println("the multiplictaion of "+num1+" , "+num2+" = "+result2);  
  
            int result3 = calculatorInterface.division(num1, num2);  
            System.out.println("the division of "+num1+" , "+num2+" = "+result3);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }
```

```
}
```

Er. Shankar pd. Dahal
pd.dahal@gmail.com

5. Compile Application:

To compile application :

- ❖ Compile the remote interface.
- ❖ Compile the implementation class.
- ❖ Compile the server program.
- ❖ Compile the client program.

E.g. `javac *.java`

6. Execute the application

Step-1 : Start rmi rmiregistry using the following command:

e.g. `start rmiregistry`

Step-2 : Run the server class

e.g. `java serverclassName`

Step-3 : Run the client class

e.g. `java clientClassName`

Er. Shankar pd. Dahal
pdsdahal@gmail.com

Stubs and Parameter Marshalling :

- ❖ Client makes procedure call (just like a local procedure call) to the client stub.
- ❖ Server is written as a standard procedure.
- ❖ Stubs take care of packaging arguments and sending messages.
- ❖ Packaging parameters is called **marshalling**

The stub's job is to:

- ❖ Marshall the parameters passed.
- ❖ Create an identifier of the remote object and method to be used.

Er. Shankar pd. Dahal
pdsdahal@gmail.com

CORBA :

- The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together.
- CORBA is a standard for distributing objects across networks so that operations on those objects can be called remotely.
- CORBA is not associated with a particular programming language, and any language with a CORBA binding can be used to call and implement CORBA objects. Objects are described in a syntax called Interface Definition Language (IDL).

CORBA includes four components:

1. Object Request Broker (ORB) :

The Object Request Broker (ORB) handles the communication, marshaling, and unmarshaling of parameters so that the parameter handling is transparent for a CORBA server and client applications.

2. CORBA server:

The CORBA server creates CORBA objects and initializes them with an ORB. The server places references to the CORBA objects inside a naming service so that clients can access them.

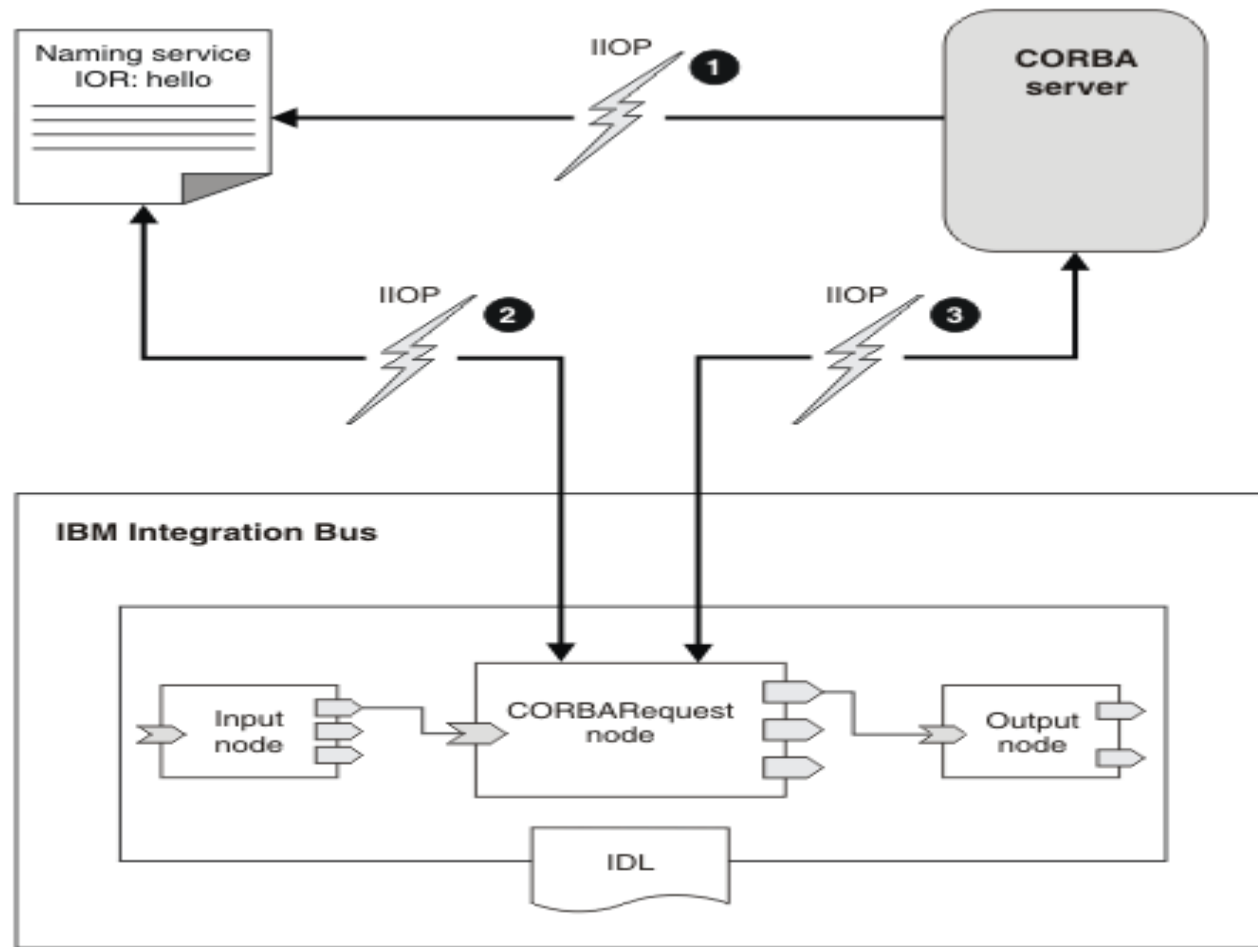
3. Naming service:

The naming service holds references to CORBA objects.

4. CORBARequest node:

The CORBARequest node acts as a CORBA client.

The following diagram shows the layers of communication between IBM® Integration Bus and CORBA.



- ❖ Use CORBA nodes to connect IBM® Integration Bus with CORBA Internet Inter-Orb Protocol (IIOP) applications.
- ❖ Interface Definition Language (IDL)

The diagram illustrates the following steps.

1. CORBA server applications create CORBA objects and put object references in a naming service so that clients can call them.
2. At deployment time, the node contacts a naming service to get an object reference.
3. When a message arrives, the node uses the object reference to call an operation on an object in the CORBA server.

Difference between RMI and CORBA :

| RMI | CORBA |
|---|---|
| RMI is a Java-specific technology. | CORBA has implementation for many languages. |
| It uses Java interface for implementation. | It uses Interface Definition Language (IDL) to separate interface from implementation. |
| RMI objects are garbage collected automatically. | CORBA objects are not garbage collected because it is language independent and some languages like C++ does not support garbage collection. |
| RMI programs can download new classes from remote JVM's. | CORBA does not support this code sharing mechanism. |
| RMI passes objects by remote reference or by value. | CORBA passes objects by reference. |
| Java RMI is a server-centric model. | CORBA is a peer-to-peer system. |
| RMI uses the Java Remote Method Protocol as its underlying remoting protocol. | CORBA use Internet Inter- ORB Protocol as its underlying remoting protocol. |
| The responsibility of locating an object implementation falls on JVM. | The responsibility of locating an object implementation falls on Object Adapter either Basic Object Adapter or Portable Object Adapter. |

Java Beans

- ❖ JavaBeans is a portable, platform-independent model written in Java Programming Language. Its components are referred to as beans.
- ❖ JavaBeans are classes which encapsulate several objects into a single object. It helps in accessing these object from multiple places.
- ❖ JavaBeans contains several elements like Constructors, Getter/Setter Methods and much more.
- ❖ It is a java class that should follow **following conventions**:
 1. Must implement Serializable.
 2. It should have a public no-arg constructor.
 3. All properties in java bean must be private with public getters and setter methods.

Why use JavaBean?

- ❖ According to Java white paper, it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

Advantages of JavaBean:

- ❖ The JavaBean properties and methods can be exposed to another application.
- ❖ It provides an easiness to reuse the software components.
- ❖ A Bean may register to receive events from other objects and can generate events that are sent to other objects.
- ❖ The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- ❖ A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.

Disadvantages of JavaBean

- ❖ JavaBeans are mutable. So, it can't take advantages of immutable objects.
- ❖ Creating the setter and getter method for each property separately may lead to the boilerplate code.

Er. Shankar pd. Dahal
pdsdahal@gmail.com

Syntax for setter methods:

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

Syntax for getter methods:

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

Er. Shankar pd. Dahal
pdsdahal@gmail.com

Bean Properties

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

Types of properties :

- a. Simple Properties
- b. Boolean Properties
- c. Indexed Properties

Er. Shankar pd. Dahal
pdsdahal@gmail.com

A. Simple Properties :

- ❖ Simple property are basic, independent, individual properties like firstName, lastName, height, width etc.
- ❖ The accessor and mutator method should follow standard naming conventions.

e.g.

```
private String firstName;
```

```
public void setFirstName(String firstName) {  
this.firstName = firstName;  
}
```

```
public String getFirstName() {  
return firstName;  
}
```

Er. Shankar pd. Dahal
pdsdahal@gmail.com

B. Boolean Properties :

- ❖ They are simple properties.
- ❖ The accessor and mutator method should follow standard naming conventions.
- ❖ The getter methods follow an optional design pattern.

e.g.

```
private boolean connect;
```

```
public boolean isConnect() {  
return connect;  
}
```

```
public void setConnect(boolean connect) {  
this.connect = connect;  
}
```

Er. Shankar pd. Dahal
pdsdahal@gmail.com

C. Indexed Properties :

- ❖ It is a property that can take an array of values.
- ❖ It supports a range of values instead of single value.

e.g.

```
private int[] rollNo;
```

```
public int[] getRollNo() {  
return rollNo;  
}
```

```
public void setRollNo(int[] rollNo) {  
this.rollNo = rollNo;  
}
```

Er. Shankar pd. Dahal
pdsdahal@gmail.com

Design Pattern

- ❖ Property design patterns are used to identify the properties of a Bean. Property design patterns are closely related to accessor and mutator methods.

Types of property design patterns

- a. Simple property design patterns
- b. Boolean property design patterns
- c. Indexed property design patterns

Multicast Event Design Patterns

- ❖ Beans that support multiple event listeners are multicast event sources. The automatic JavaBeans introspection facility uses a pair of special event registration methods for Beans that broadcast to multiple event listeners: one method allows interested parties to be added as listeners, and the other allows listeners to be removed. Following are the design patterns governing these event registration methods:

```
public void addEventListenerType (EventListenerType x);  
public void removeEventListenerType (EventListenerType x);
```

Er. Shankar pd. Dahal
pdsdahal@gmail.com

Simple property design pattern

- ❖ Simple properties in JavaBeans consist of all single-valued properties, which include all built-in Java data types as well as classes and interfaces.
- ❖ For example, properties of type int, long, float, Color, Font, and boolean are all considered simple properties.
- ❖ The design patterns for simple property accessor and mutator methods as follow:

```
public PropertyType getPropertyName();  
public void setPropertyName(PropertyType x);
```

Boolean property design pattern

- ❖ Boolean properties have an optional design pattern for the getter method that can be used to make it more clear that the property is boolean.
- ❖ The design pattern for the boolean getter method follows:

```
public boolean isPropertyName();
```

Following is an example of a pair of accessor and mutator methods for a boolean property named **visible**:

```
public boolean isVisible();  
public void setVisible(boolean v);
```

Er. Shankar pd. Dahal
pdsdahal@gmail.com

Indexed property design patterns

- ❖ An indexed property is a property representing an array of values.
- ❖ Indexed properties require slightly different accessor methods, it only makes sense that their design patterns differ a little from those of other properties.
- ❖ Following are the design patterns for indexed properties:

```
public PropertyType getProperty(int index);
```

```
public void setPropertyName(int index, PropertyType x);
```

```
public PropertyType[] getPropertyName();
```

```
public void setPropertyName(PropertyType[] x);
```

The first pair of design patterns defines accessor methods for getting and setting individual elements in an indexed property.

The second pair of patterns defines accessor methods for getting and setting the entire property array as a whole.

Er. Shankar pd. Dahal
pdsdahal@gmail.com