ASSIGNMENT 8

PROBLEM STATEMENT: -

Design and develop a distributed Hotel booking application using Java RMI. A distributed hotel booking system consists of the hotel server and the client machines. The server manages hotel rooms booking information. A customer can invoke the following operations at his machine i) Book the room for the specific guest ii) Cancel the booking of a guest.

OBJECTIVE:

1. Students should be able to Implement the server-side logic that manages hotel room booking information. This includes storing information about booked rooms and handling booking and cancellation requests from clients.
2. Students should be able to Implement remote objects that implement the remote interfaces defined earlier. These objects will expose the booking and cancellation functionality to clients via RMI.
3. Students should be able to Start an RMI registry on the server machine. This registry will allow clients to look up remote objects by their names and obtain references to them.

Software Required: BlueJ IDE

THEORY:

Designing and developing a distributed Hotel Booking application using Java RMI involves creating a system where the server manages hotel room booking information, and clients interact with the server to book or cancel rooms for guests.

Components of the System:

1. Server: The server is responsible for managing hotel room booking information. It exposes remote methods that clients can invoke to book or cancel rooms.
2. Client: The client is the interface through which users interact with the system. It communicates with the server to perform booking and cancellation operations.

Remote Method Invocation (RMI):

Java RMI (Remote Method Invocation) allows Java objects to invoke methods on remote Java objects running on different JVMs (Java Virtual Machines). RMI provides a mechanism for distributed communication, enabling remote method calls between different Java applications.

Key Concepts in Java RMI:

1. Remote Interface: Define an interface that extends java.rmi. Remote. This interface declares the methods that can be invoked remotely by clients.
2. Remote Object: Implement the remote interface to create a remote object. This object is instantiated on the server and provides the implementation of the methods declared in the remote interface.
3. Server Implementation: Create a server class that instantiates the remote object, binds it to a registry, and listens for incoming client requests.
4. Client Implementation: Create a client class that looks up the remote object from the registry and invokes its methods.

Operations in the Distributed Hotel Booking System:

1. Book Room Operation:
    - The client invokes a remote method on the server to book a room for a specific guest.
    - The server updates its booking information and returns a confirmation to the client.
2. Cancel Booking Operation:
    - The client invokes a remote method on the server to cancel the booking of a guest.
    - The server updates its booking information and returns a confirmation to the client.

Workflow:

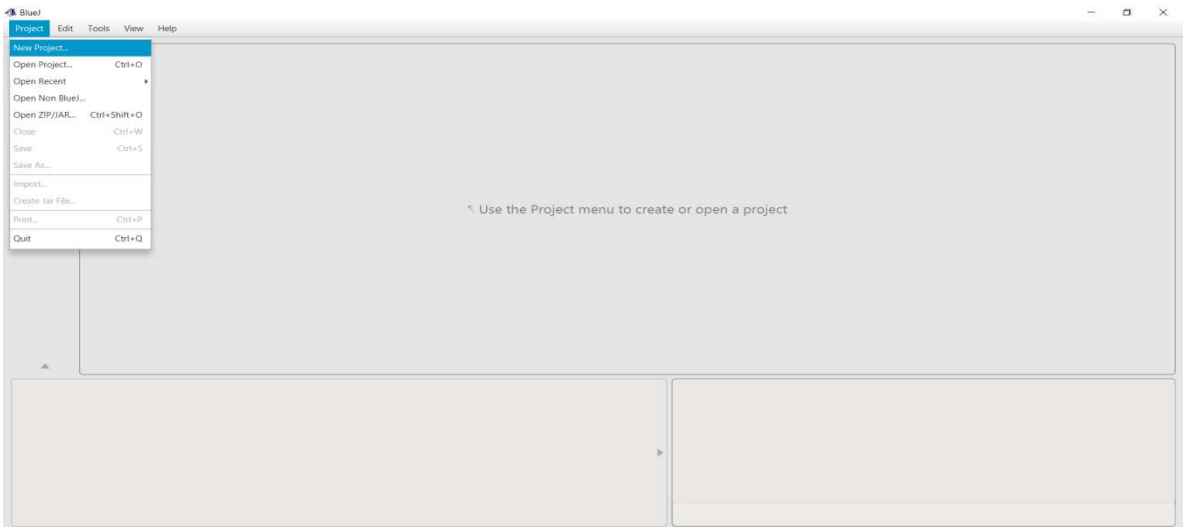
1. Server Setup:
    - The server creates an instance of the remote object implementing the booking system functionality.
    - The server binds this object to a specific name in the RMI registry, making it accessible to clients.
2. Client Interaction:
    - Clients retrieve the remote object reference from the RMI registry using its registered name.
    - Clients invoke methods on the remote object to perform booking or cancellation operations.
3. Server Response:

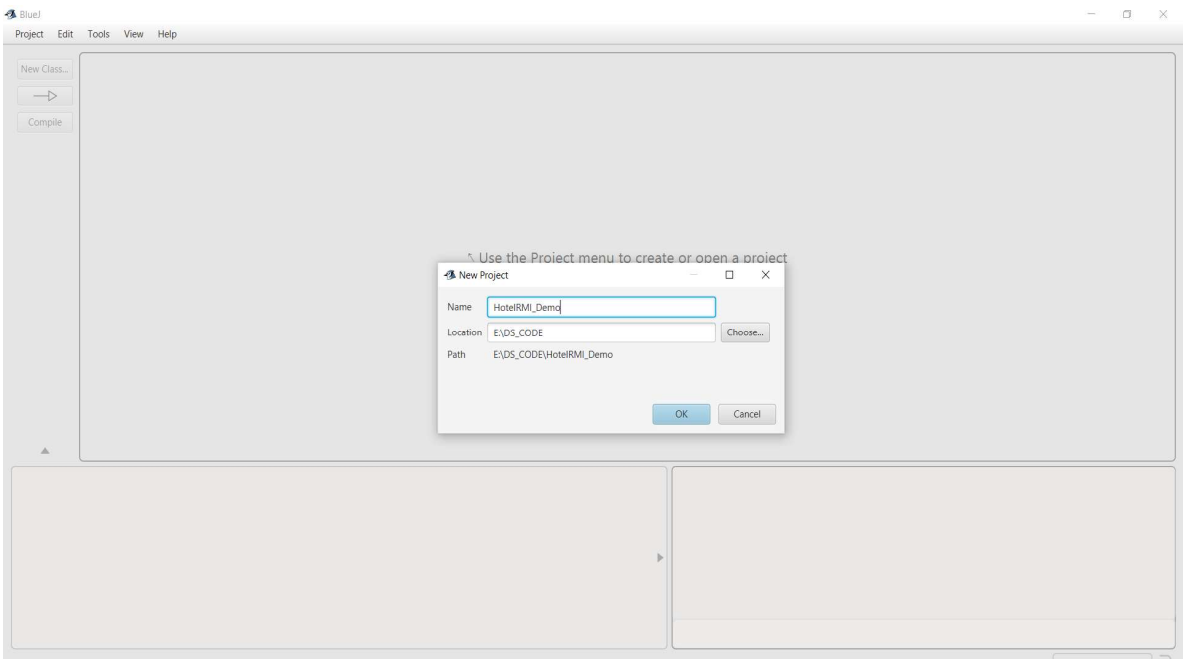
- The server processes client requests, updates booking information, and returns appropriate responses to clients.

Advantages of Java RMI in this Context:

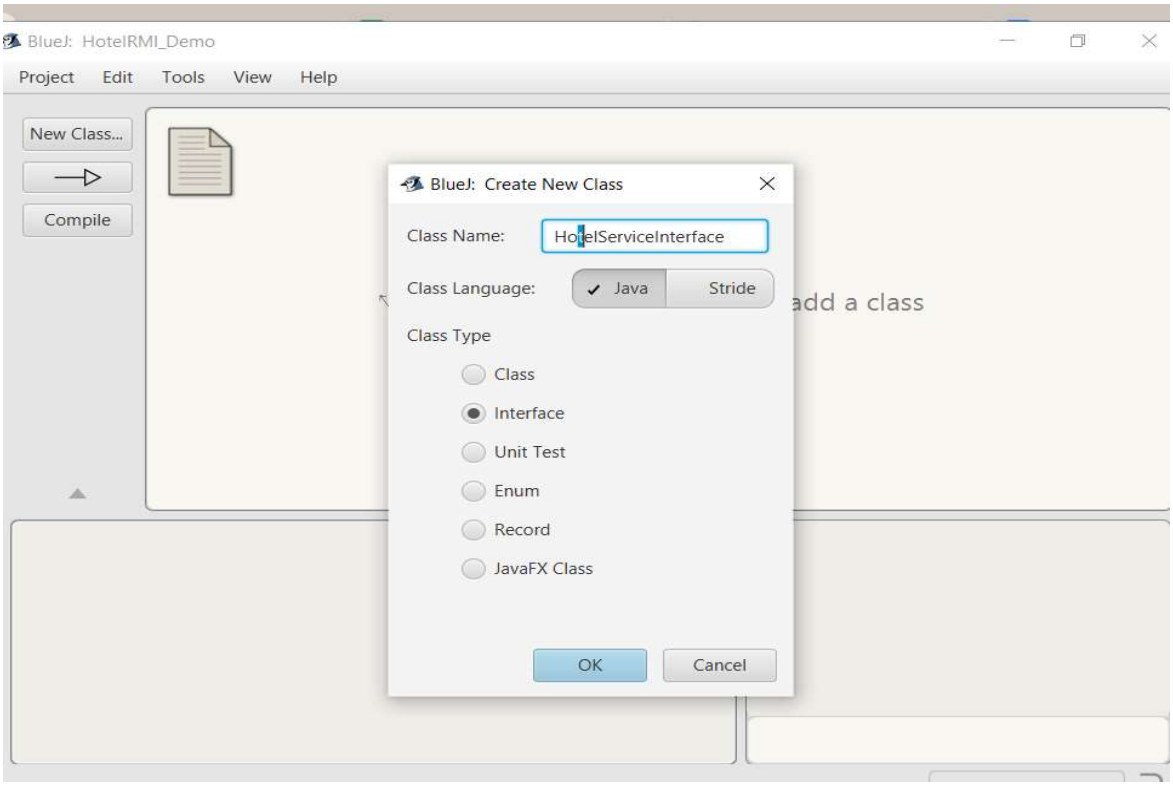
1. Ease of Development: Java RMI simplifies the development of distributed systems by abstracting network communication details.
2. Language Compatibility: Clients and servers can be written in Java, allowing seamless integration with existing Java applications.
3. Security: Java RMI provides built-in support for security features such as authentication and encryption, ensuring secure communication between clients and servers.
4. Performance: RMI is optimized for performance, making it suitable for real-time applications like hotel booking systems.

Step 1: Create Project and provide suitable name

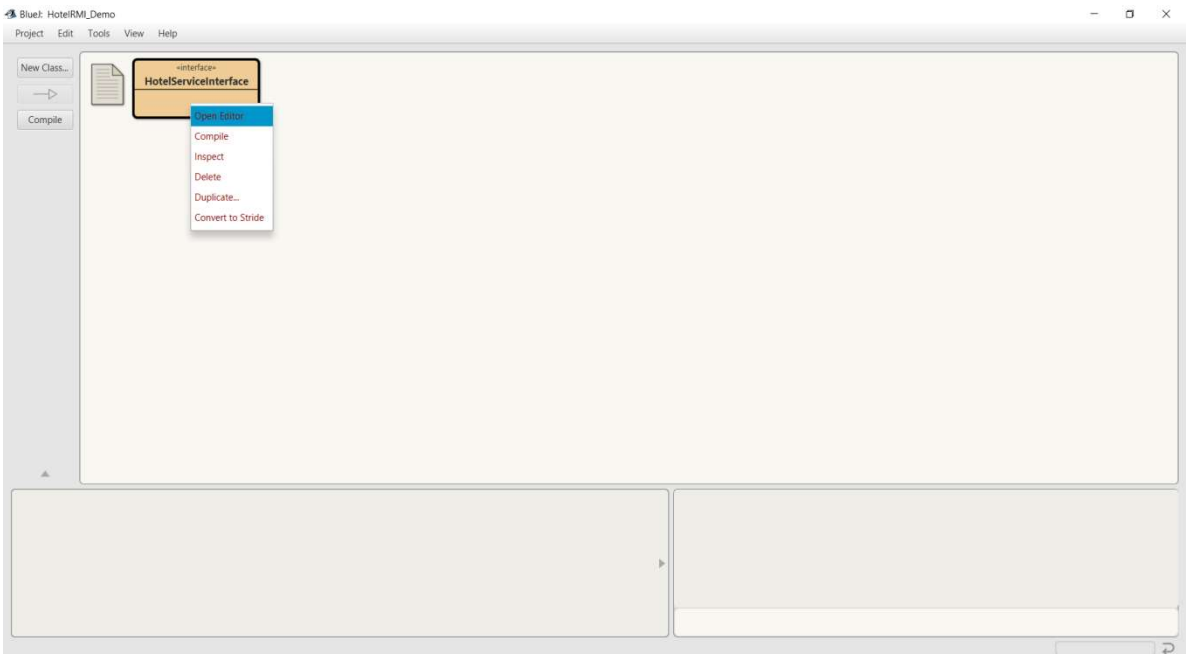




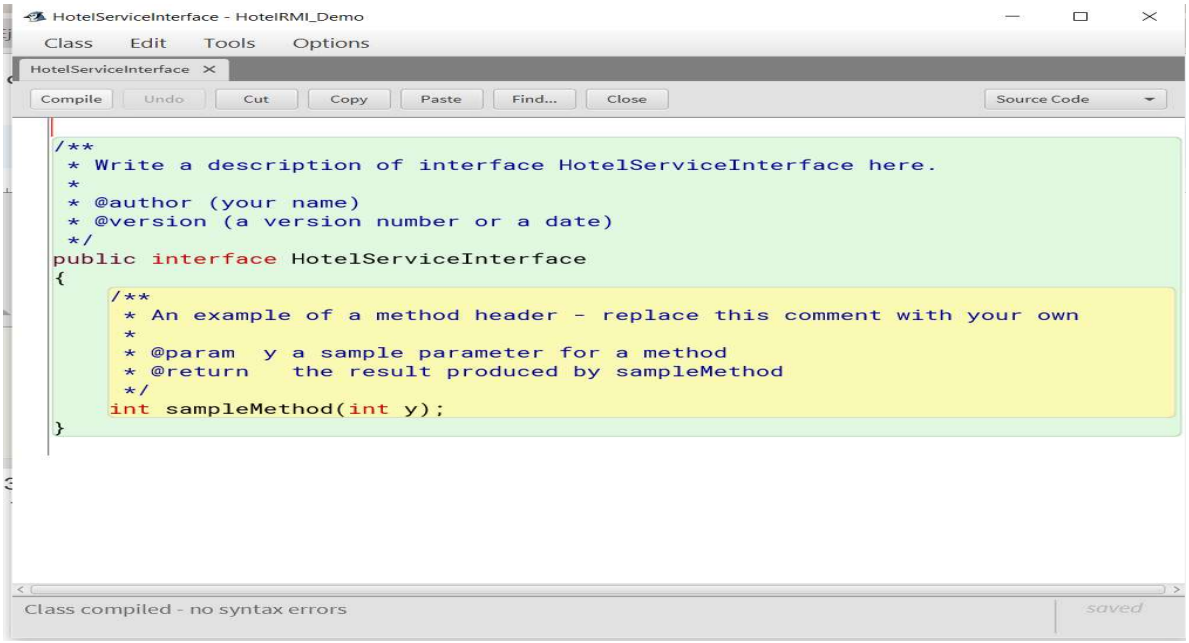
Step 2: Create Interface



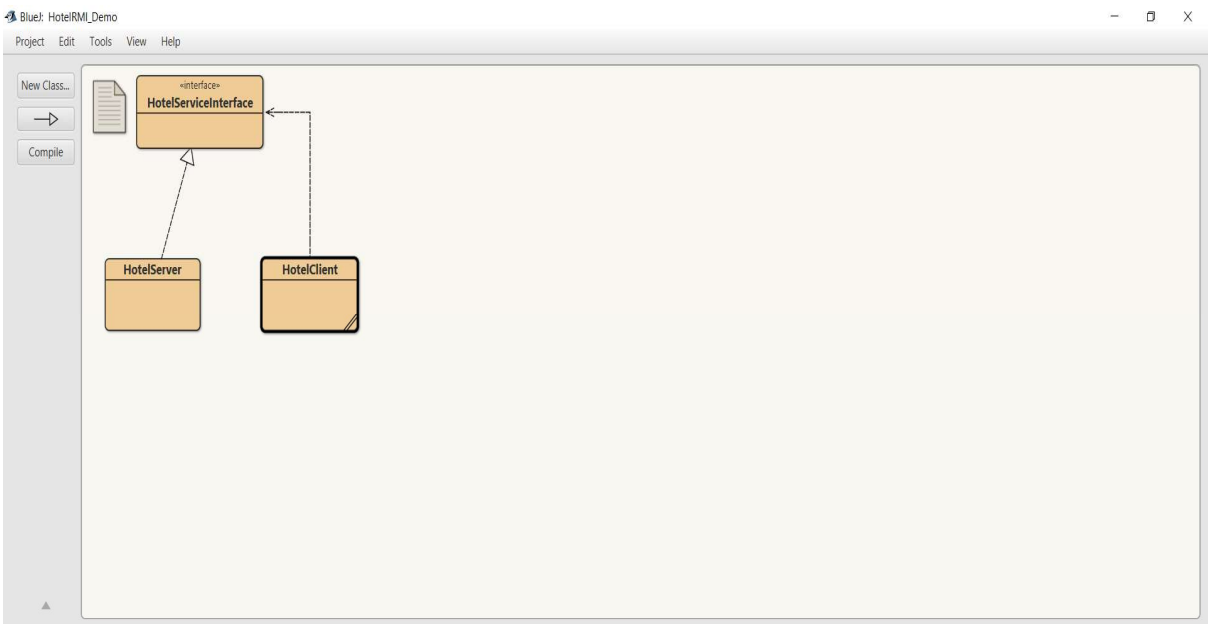
Step 3: Interface has been Created . Right click on Block of Interface.



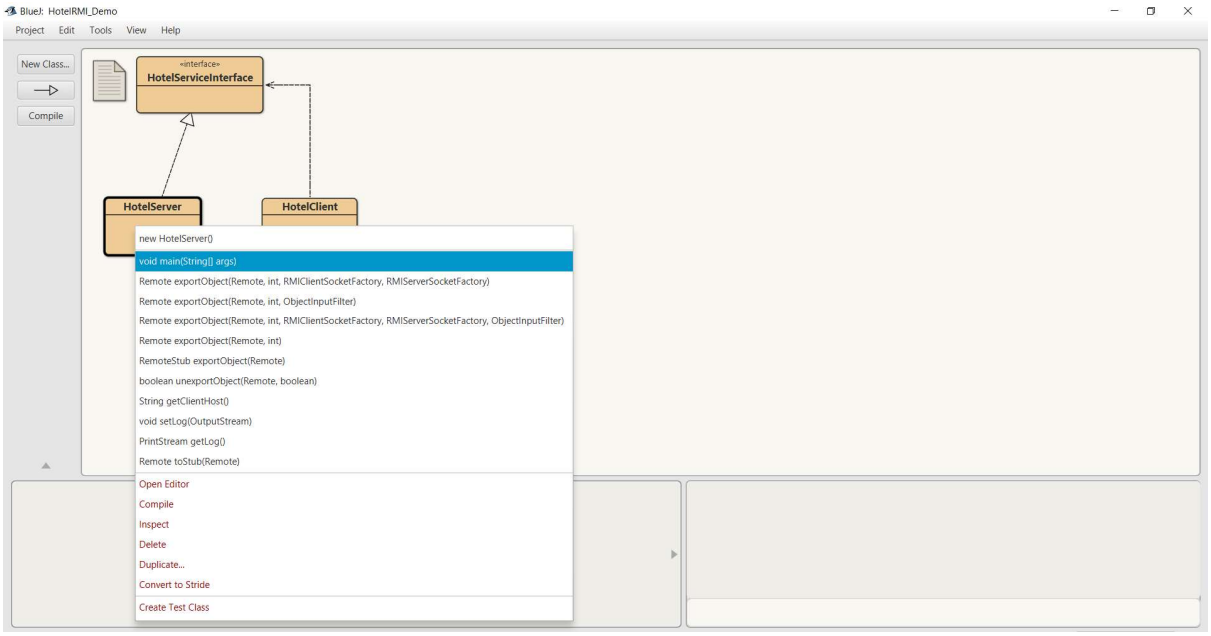
Step 4 : Click on Open Editor. Write the code of Interface here and Compile it.

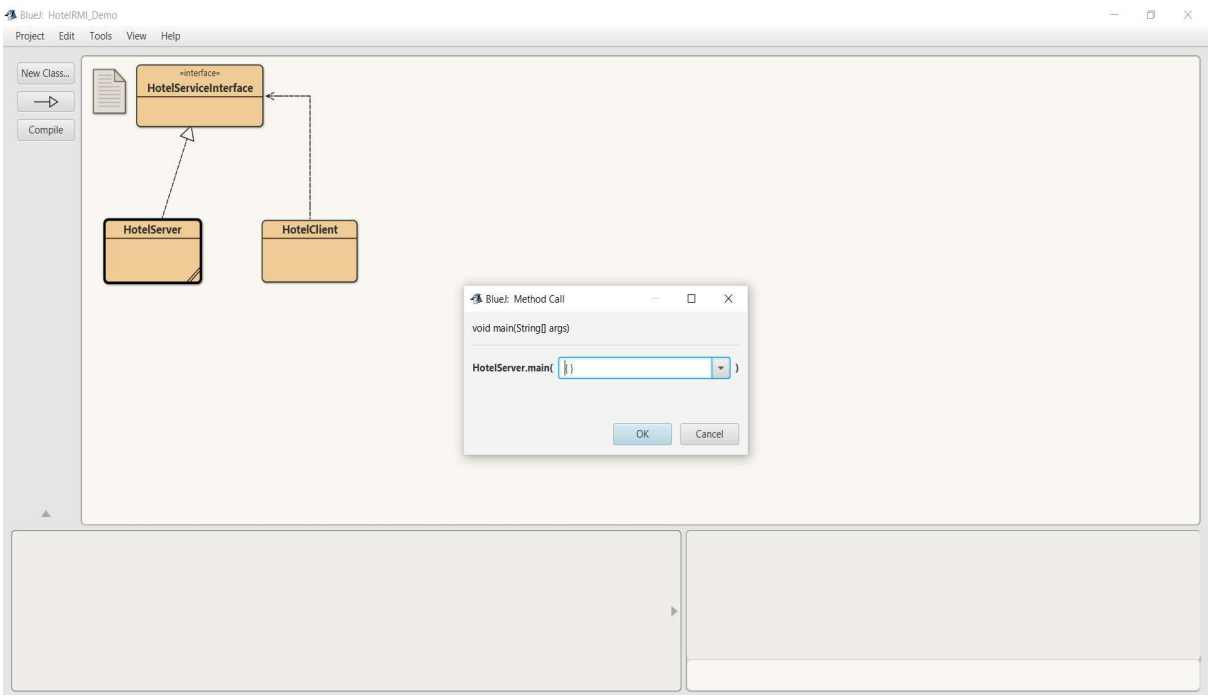


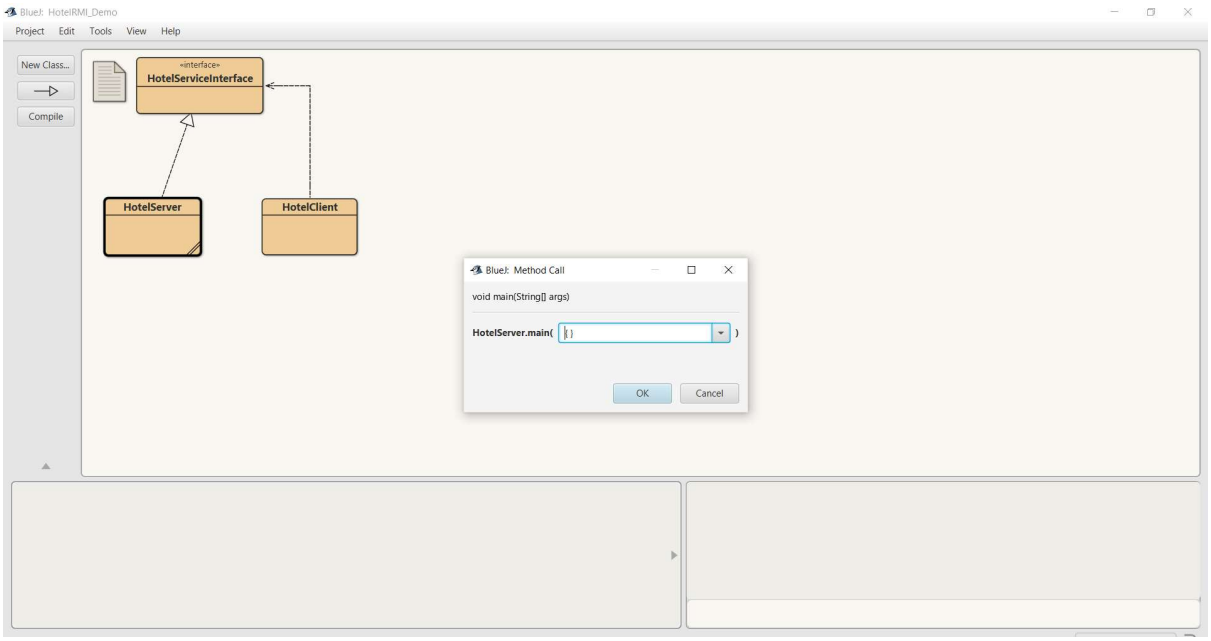
Step 5: In the similar way create two more classes HotelServer, and HotelClient. Open the editor and write the code and compile the both classes. On successfully completed the above procedure the diagram will be.



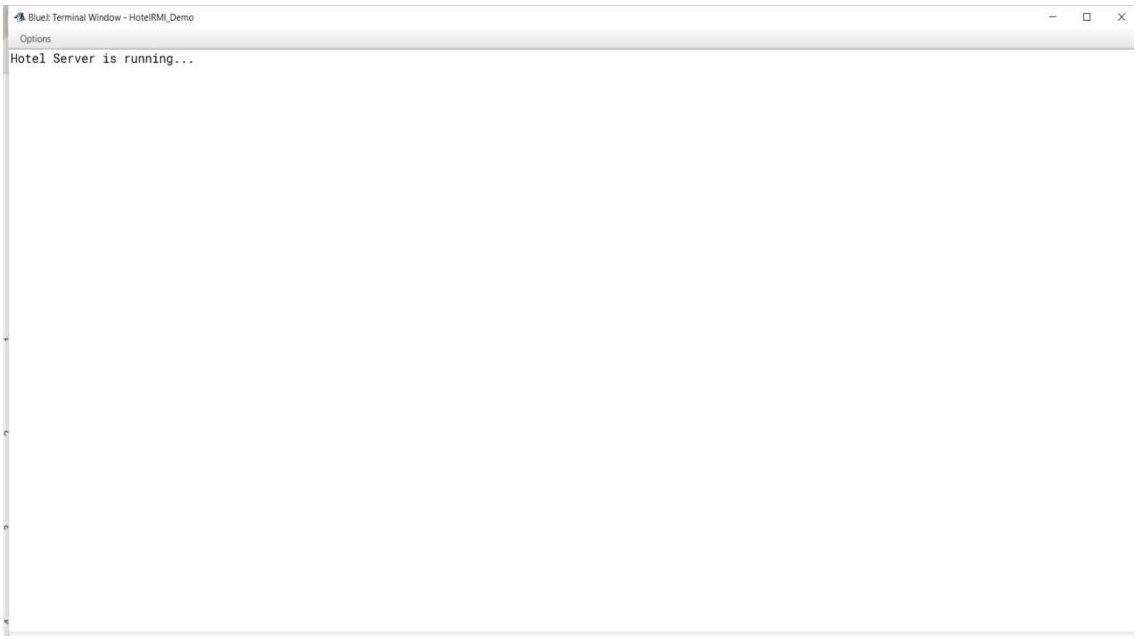
Step 6: Write Click on the block of HotelServer Class and select the main method call.



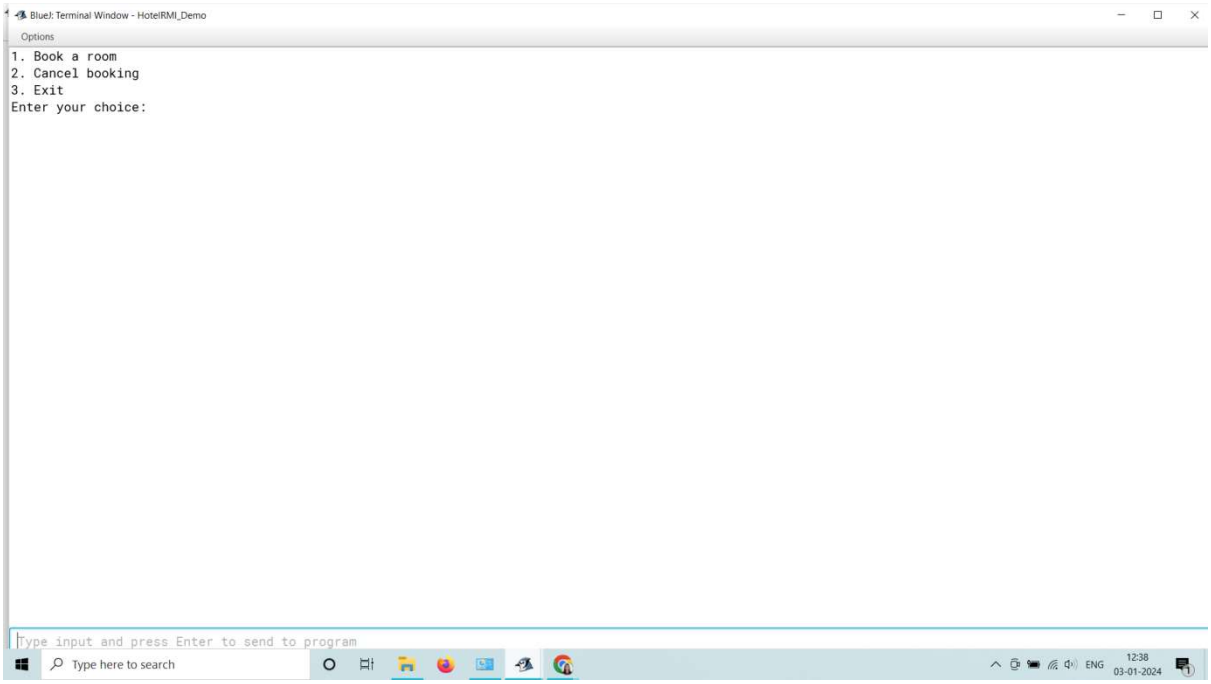




Step 7: Observe the Hotel Server is running on Terminal Window.



Step 8: In the similar way Right click on the block of Hotel Client Class and select the main method call. Observe the Hotel Client is running on Terminal Window.



Step 9

Give the appropriate choice in the Terminal window and Run the code successfully.

Implementation

Hotel Client

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

public class HotelClient {
   public static void main(String[] args) {
      try {
         // Look up the RMI server object from the registry
         HotelServiceInterface hotelService = (HotelServiceInterface)
Naming.lookup("rmi://localhost/HotelService");

         Scanner scanner = new Scanner(System.in);

         while (true) {
            System.out.println("1. Book a room");
            System.out.println("2. Cancel booking");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");

            int choice = scanner.nextInt();
            scanner.nextLine(); // consume the newline character

            switch (choice) {
               case 1:
                  System.out.print("Enter guest name: ");
                  String guestName = scanner.nextLine();

                  System.out.print("Enter room number: ");
                  int roomNumber = scanner.nextInt();

                  boolean booked = hotelService.bookRoom(guestName, roomNumber);

                  if (booked) {
                     System.out.println("Room booked successfully!");
                  } else {
                     System.out.println("Room booking failed.");
                  }
                  break;

               case 2:
                  System.out.print("Enter guest name for cancellation: ");
                  String cancelGuestName = scanner.nextLine();

                  boolean canceled = hotelService.cancelBooking(cancelGuestName);

                  if (canceled) {
```

```
                    System.out.println("Booking canceled successfully!");
                } else {
                    System.out.println("Booking cancellation failed.");
                }
                break;

            case 3:
                System.out.println("Exiting the client application.");
                System.exit(0);
                break;

            default:
                System.out.println("Invalid choice. Please enter a valid option.");
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Hotel server

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;

public class HotelServer extends UnicastRemoteObject implements
HotelServiceInterface {
  private Map<Integer, String> bookedRooms;

  public HotelServer() throws RemoteException {
    bookedRooms = new HashMap<>();
  }

  @Override
  public synchronized boolean bookRoom(String guestName, int roomNumber)
throws RemoteException {
    if (!bookedRooms.containsKey(roomNumber)) {
      bookedRooms.put(roomNumber, guestName);
      System.out.println("Room " + roomNumber + " booked for guest: " +
guestName);
      return true;
```

```
        } else {
            System.out.println("Room " + roomNumber + " is already booked.");
            return false;
        }
    }

    @Override
    public synchronized boolean cancelBooking(String guestName) throws
RemoteException {
        for (Map.Entry<Integer, String> entry : bookedRooms.entrySet()) {
            if (entry.getValue().equals(guestName)) {
                bookedRooms.remove(entry.getKey());
                System.out.println("Booking for guest " + guestName + " canceled.");
                return true;
            }
        }
        System.out.println("No booking found for guest " + guestName);
        return false;
    }

    public static void main(String[] args) {
        try {
            HotelServer server = new HotelServer();

            // Create and export the RMI registry on port 1099
            java.rmi.registry.LocateRegistry.createRegistry(1099);

            // Bind the server object to the registry
            Naming.rebind("HotelService", server);

            System.out.println("Hotel Server is running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Hotel service Interface

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

public interface HotelServiceInterface extends Remote {
    boolean bookRoom(String guestName, int roomNumber) throws RemoteException;
    boolean cancelBooking(String guestName) throws RemoteException;
```

}

CONCLUSION:

In this way we have by implementing the distributed Hotel Booking application using Java RMI, we can create a robust and scalable system for managing hotel room bookings efficiently.

ORAL QUESTION

1. Can you explain the benefits of using a distributed system for a hotel booking application?

2. What are the main components of the system, and how do they interact with each other?

3. What is Java RMI (Remote Method Invocation), and how does it facilitate communication between distributed components?

4. What are the necessary methods that need to be defined in the remote interface to support these operations?

Code:

HotelClient.java

```
import java.rmi.Naming;
import java.util.Scanner;
public class HotelClient {
 public static void main(String[] args) {
 try {
 // Look up the RMI server object from the registry
 HotelServiceInterface hotelService = (HotelServiceInterface)
Naming.lookup("rmi://localhost/HotelService");
 Scanner scanner = new Scanner(System.in);
 while (true) {
 System.out.println("1. Book a room");
 System.out.println("2. Cancel booking");
 System.out.println("3. Exit");
 System.out.print("Enter your choice: ");
 int choice = scanner.nextInt();
 scanner.nextLine(); // consume the newline character
 switch (choice) {
 case 1:
 System.out.print("Enter guest name: ");
 String guestName = scanner.nextLine();
 System.out.print("Enter room number: ");
 int roomNumber = scanner.nextInt();
 boolean booked = hotelService.bookRoom(guestName, roomNumber);
 if (booked) {
 System.out.println("Room booked successfully!");
 } else {
 System.out.println("Room booking failed.");
 }
 break;
 case 2:
 System.out.print("Enter guest name for cancellation: ");
 String cancelGuestName = scanner.nextLine();
 boolean canceled = hotelService.cancelBooking(cancelGuestName);
 if (canceled) {
 System.out.println("Booking canceled successfully!");
 } else {
 System.out.println("Booking cancellation failed.");
 }
 break;
 case 3:
 System.out.println("Exiting the client application.");
```

```
 System.exit(0);
 break;
 default:
 System.out.println("Invalid choice. Please enter a valid option.");
 }
 }
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}
```

HotelServer.java

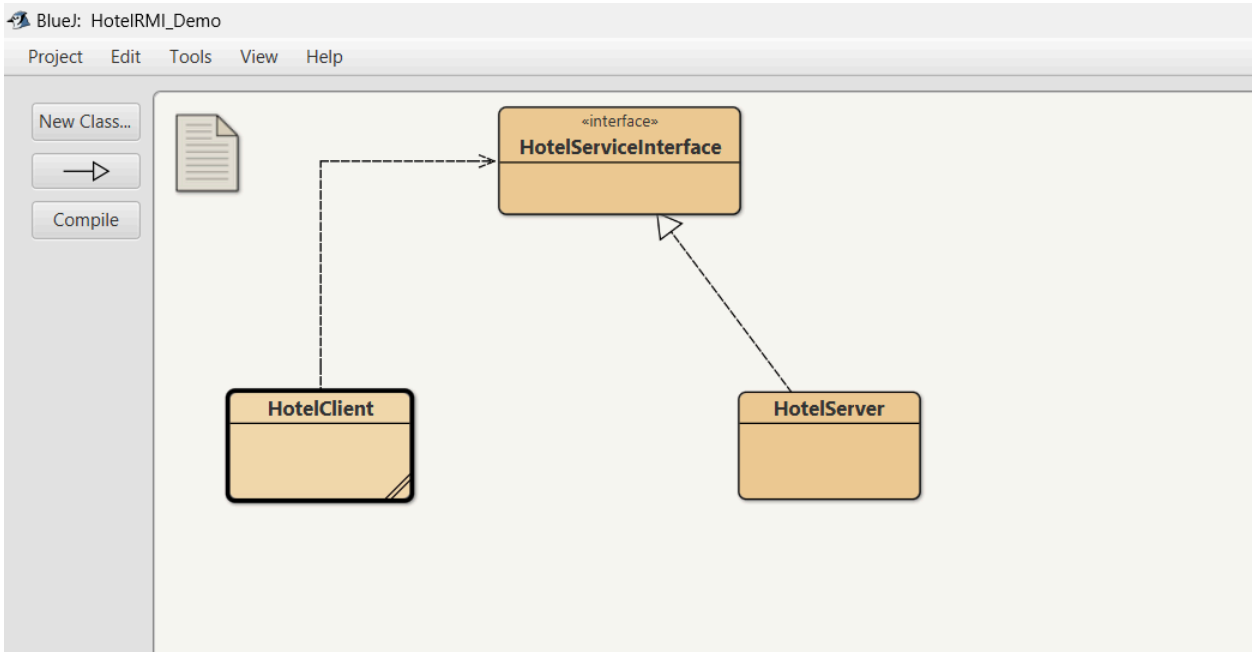
```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;
public class HotelServer extends UnicastRemoteObject implements
HotelServiceInterface {
 private Map<Integer, String> bookedRooms;
 public HotelServer() throws RemoteException {
 bookedRooms = new HashMap<>();
 }
 @Override
 public synchronized boolean bookRoom(String guestName, int
roomNumber)
throws RemoteException {
 if (!bookedRooms.containsKey(roomNumber)) {
 bookedRooms.put(roomNumber, guestName);
 System.out.println("Room " + roomNumber + " booked for guest: " +
guestName);
 return true;
 } else {
 System.out.println("Room " + roomNumber + " is already booked.");
 return false;
 }
 }
 @Override
 public synchronized boolean cancelBooking(String guestName) throws
RemoteException {
 for (Map.Entry<Integer, String> entry : bookedRooms.entrySet()) {
 if (entry.getValue().equals(guestName)) {
 bookedRooms.remove(entry.getKey());
```

```
 System.out.println("Booking for guest " + guestName + " canceled.");
 return true;
 }
 }
 System.out.println("No booking found for guest " + guestName);
 return false;
 }
 public static void main(String[] args) {
 try {
 HotelServer server = new HotelServer();
 // Create and export the RMI registry on port 1099
 java.rmi.registry.LocateRegistry.createRegistry(1099);
 // Bind the server object to the registry
 Naming.rebind("HotelService", server);
 System.out.println("Hotel Server is running...");
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}
```

HotelServiceInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HotelServiceInterface extends Remote {
 boolean bookRoom(String guestName, int roomNumber) throws
RemoteException;
 boolean cancelBooking(String guestName) throws RemoteException;
}
```

Output:



BlueJ: Terminal Window - HotelRMI_Demo

Options

```
Hotel Server is running...
1. Book a room
2. Cancel booking
3. Exit
Enter your choice: 1
Enter guest name: abc
Enter room number: 101
Room 101 booked for guest: abc
Room booked successfully!
1. Book a room
2. Cancel booking
3. Exit
Enter your choice: 2
Enter guest name for cancellation: abc
Booking for guest abc canceled.
Booking canceled successfully!
1. Book a room
2. Cancel booking
3. Exit
Enter your choice: 3
Exiting the client application.
```