Architecture of Enterprise Applications 2 Server Component – Distributed Object

Haopeng Chen

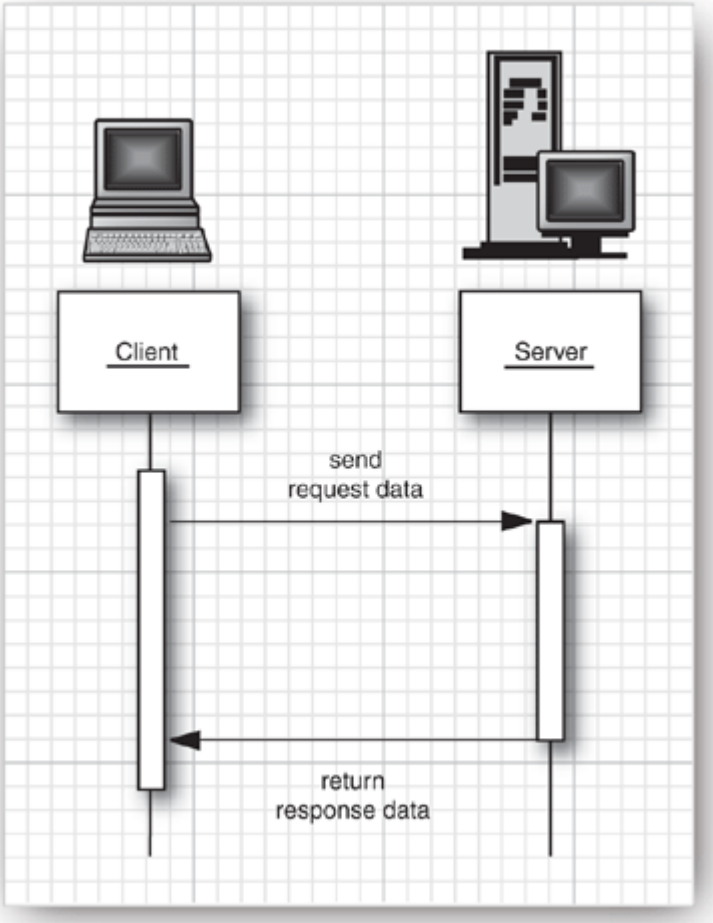
***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents

- The roles of client and server
- Remote method calls
- The RMI programming model
- Parameters and return values in remote methods
- Remote object activation
- Web services

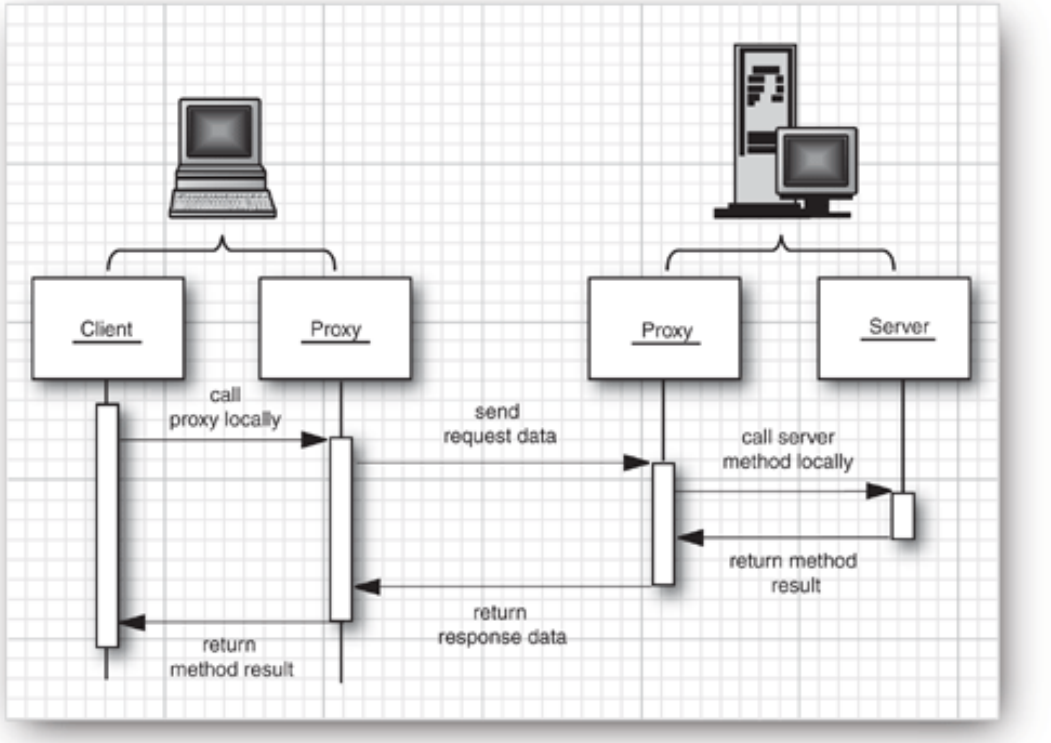
The Roles of Client and Server

- The basic idea behind all distributed programming is simple.
  - A client computer makes a request and sends the request data across a network to a server.
  - The server processes the request and sends back a response for the client to analyze.



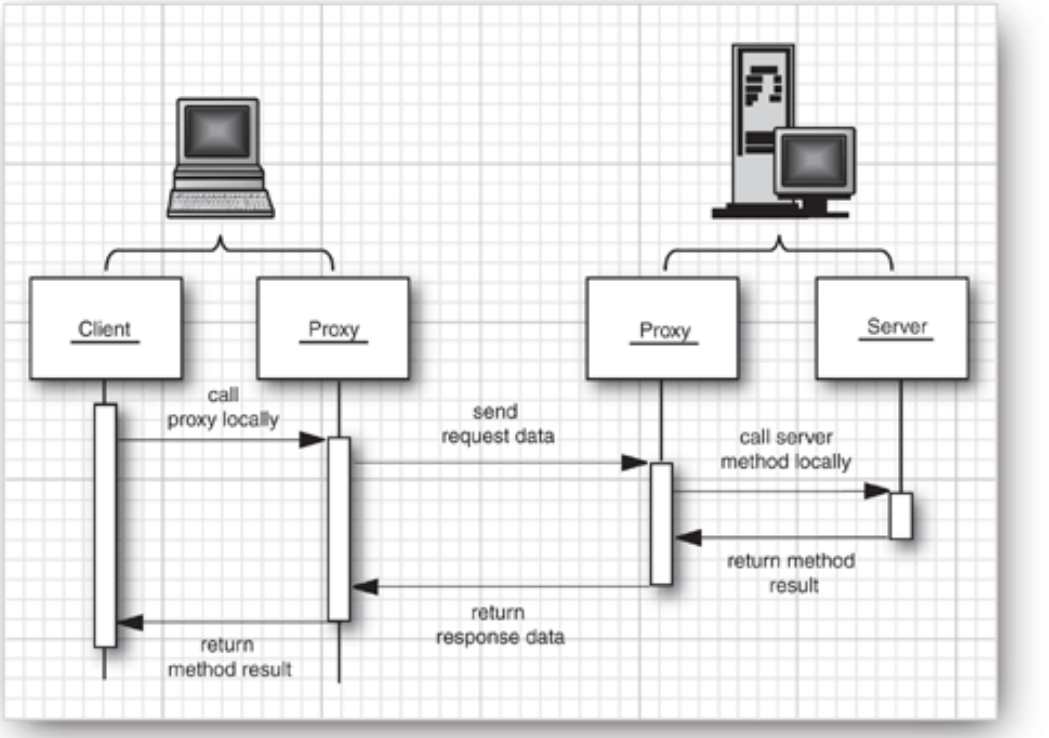
The Roles of Client and Server

- What we want is a mechanism
  - by which the client programmer makes a regular method call, without worrying about sending data across the network or parsing the response.
  - The soluti



The Roles of Client and Server

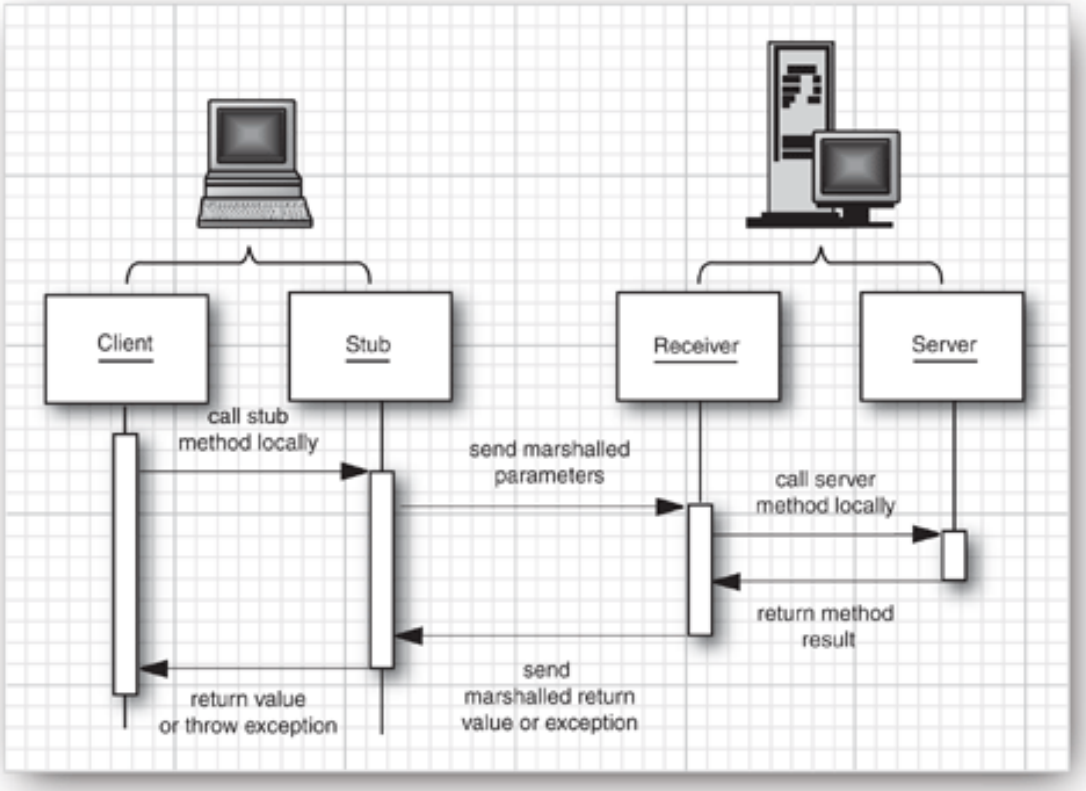
- How do the proxies communicate with each other?
  - The Java RMI technology
  - The Common Object Request Broker Architecture (CORBA)
  - The web services architecture



Remote Method Calls

- Stubs and Parameter Marshalling

```
Warehouse centralWarehouse = get stub object;
double price = centralWarehouse.getPrice("Blackwell Toaster");
```



Remote Method Calls

- Stubs and Parameter Marshalling

```
Warehouse centralWarehouse = get stub object;
double price = centralWarehouse.getPrice("Blackwell Toaster");
```

- The stub method on the client builds an information block that consists of
  - An identifier of the remote object to be used.
  - A description of the method to be called.
  - The parameters.
- The stub then sends this information to the server.

- On the server side, a receiver object performs the following actions:
  - It locates the remote object to be called.
  - It calls the desired method, passing the supplied parameters.
  - It captures the return value or exception of the call.
  - It sends a package consisting of the marshalled return data back to the stub on the client.

The RMI Programming Model

- Interfaces and Implementations

```
import java.rmi.*;
public interface Warehouse extends Remote
{
   double getPrice(String description) throws RemoteException;
}
```

  - Interfaces for remote objects must always extend the `Remote` interface defined in the `java.rmi` package.
  - All the methods in those interfaces must also declare that they will throw a `RemoteException`.

The RMI Programming Model

- On the server side, you must provide the implementation of the remote interface

```
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
   private Map<String, Double> prices;

   public WarehouseImpl() throws RemoteException
   {
      prices = new HashMap<>();
      prices.put("Blackwell Toaster", 24.95);
      prices.put("ZapXpress Microwave Oven", 49.95);
   }
   public double getPrice(String description) throws RemoteException
   {
      Double price = prices.get(description);
      return price == null ? 0 : price;
   }
}
```

The RMI Programming Model

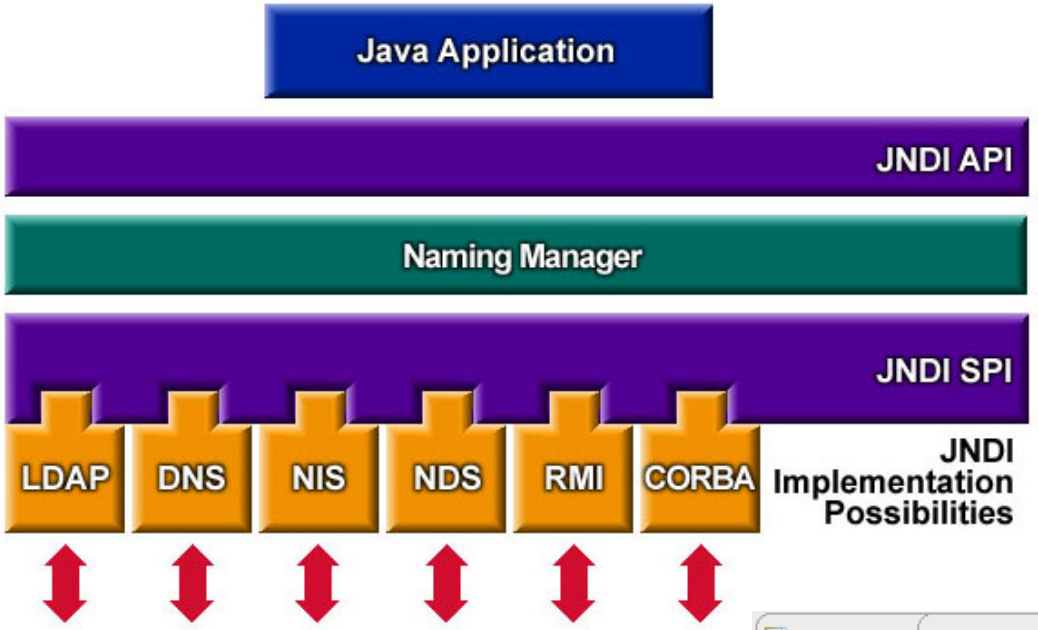
- The RMI Registry
  - The first remote object is a bootstrap registry service.
  - RMI URLs :

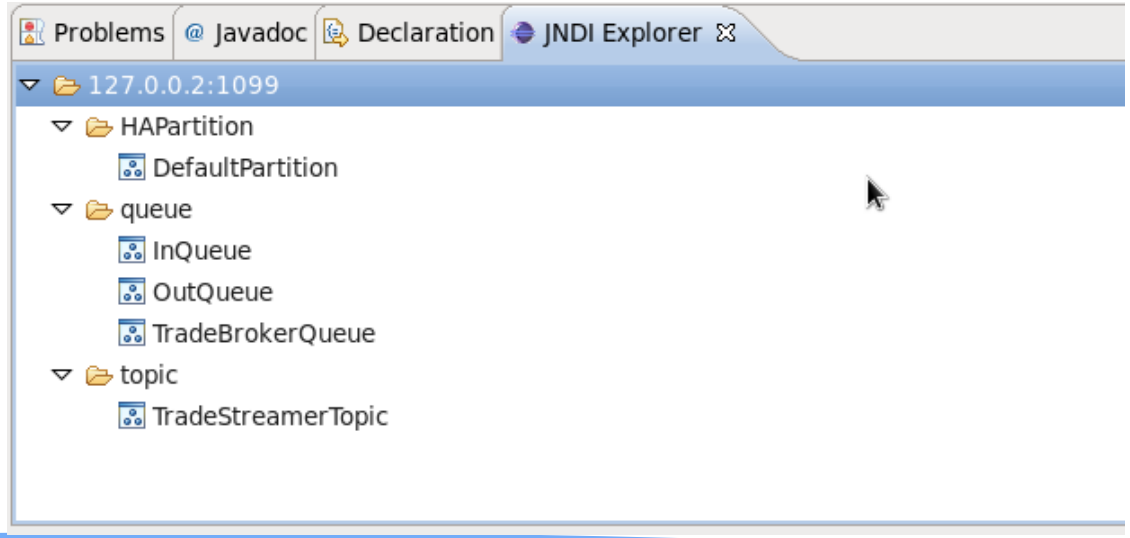
```
rmi://regserver.mycompany.com:1099/central_warehouse
```

- Here is the code for
  - registering a `WarehouseImpl` object with the RMI registry on the same server:

```
WarehouseImpl centralWarehouse = new WarehouseImpl();
Context namingContext = new InitialContext();
namingContext.bind("rmi:central_warehouse",
                   centralWarehouse);
```

Java Naming and Directory Interface





JNDI API

- On Server-side

```
Context ctx = new InitialContext();
ctx.bind("jdbc/AcmeDB", vds);
ctx.rebind("jdbc/ZenithDB", vds);
```

- On Client-side

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/ZenithDB");
```

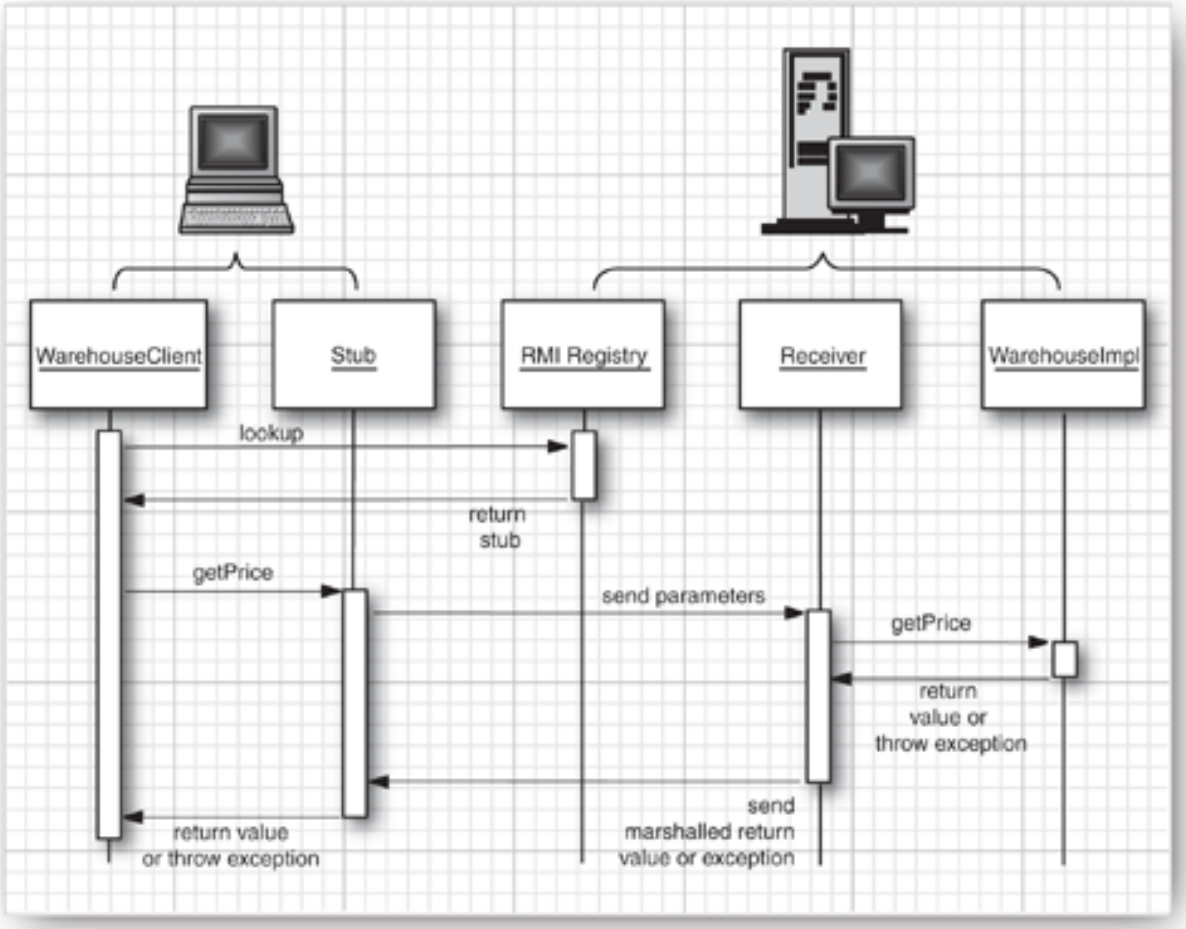
The RMI Programming Model

- The Server-side program

```
public class WarehouseServer
{
   public static void main(String[] args) throws
                                          RemoteException, NamingException
   {
      System.out.println("Constructing server implementation...");
      WarehouseImpl centralWarehouse = new WarehouseImpl();

      System.out.println("Binding server implementation to registry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:central_warehouse", centralWarehouse);

      System.out.println("Waiting for invocations from clients...");
   }
}
```

The RMI Programming Model

- The Client-side Program



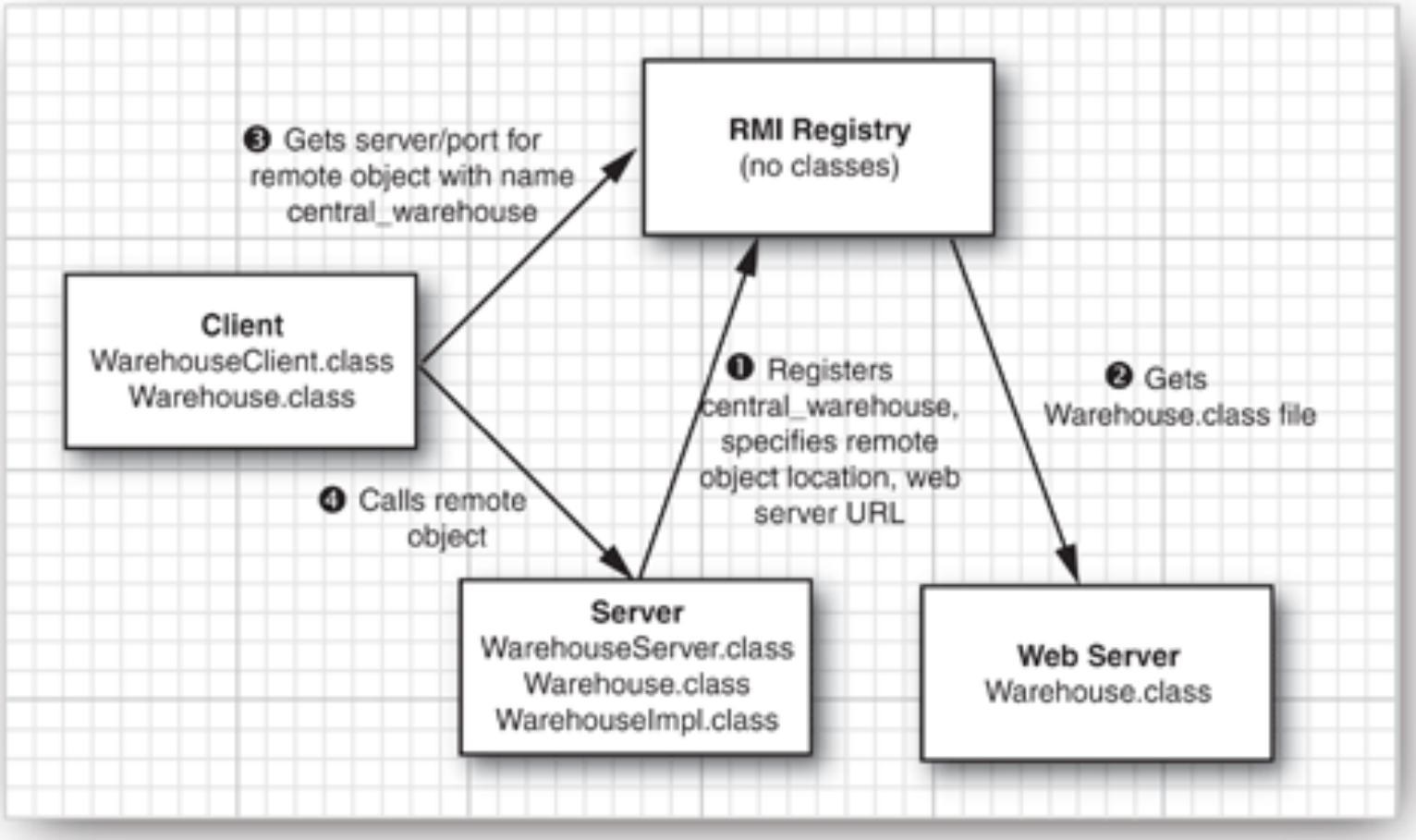
The RMI Programming Model

- The Client-side Program

```
public class WarehouseClient
{
   public static void main(String[] args) throws
                                          NamingException, RemoteException
   {
      Context namingContext = new InitialContext();

      String url = "rmi://localhost/central_warehouse";
      Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);

      String descr = "Blackwell Toaster";
      double price = centralWarehouse.getPrice(descr);
      System.out.println(descr + ": " + price);
   }
}
```

The RMI Programming Model

- Deploying the Program



The RMI Programming Model

- Running the Program
  - Add `ClassDir` to `CLASSPATH`
  - For example: `E:\Projects\JavaEE\rmiserver\bin`
  - Create a `jndi.properties` file, and copy it into `/Server` and `/Client` class dir.

```
java.naming.factory.initial=com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url=rmi://localhost:1099
```

  - Run the `rmiregistry` in a windows console
  - Run the `WarehouseServer` in another windows console, you will see:

```
Constructing server implementation...
Binding server implementation to registry...
Waiting for invocations from clients...
```

  - Run the `WarehouseClient` in the third windows console, you will see:

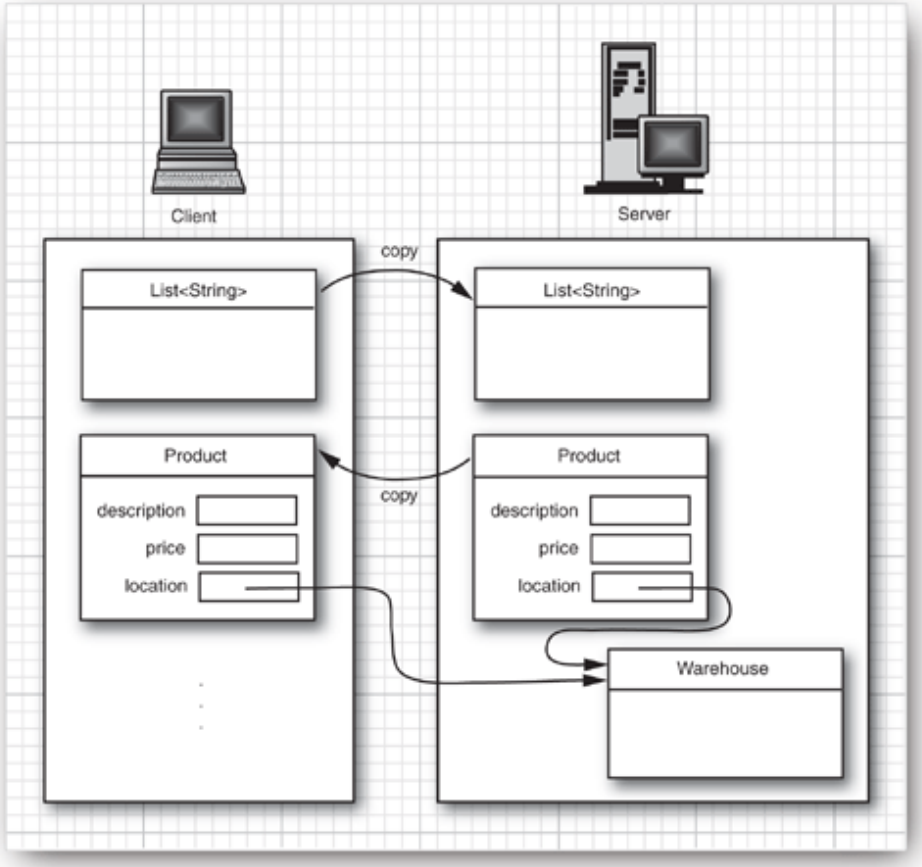
```
RMI registry bindings: central_warehouse
Blackwell Toaster: 24.95
```

Parameters and Return Values in Remote Methods

- There are two mechanisms for transferring values between virtual machines.
  - Objects of classes that implement the `Remote` interface are transferred as remote references.
  - Objects of classes that implement the `Serializable` interface but not the `Remote` interface are copied using serialization.

Transferring Nonremote Objects


```
public interface Warehouse extends Remote
{
   double getPrice(String description) throws RemoteException;
   Product getProduct(List<String> keywords) throws RemoteException;
}
```



Transferring Nonremote Objects


```
public class Product implements Serializable
{
   private String description;
   private double price;
   private Warehouse location;

   public Product(String description, double price)
   {
      this.description = description;
      this.price = price;
   }

   public String getDescription() { return description; }

   public double getPrice() { return price; }

   public Warehouse getLocation() { return location; }

   public void setLocation(Warehouse location) {
      this.location = location;
   }
}
```

Transferring Nonremote Objects


```
public class Book extends Product
{
   private String isbn;

   public Book(String title, String isbn, double price)
   {
      super(title, price);
      this.isbn = isbn;
   }

   public String getDescription()
   {
      return super.getDescription() + " " + isbn;
   }
}
```

Transferring Nonremote Objects


```
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
   private Map<String, Product> products;
   private Warehouse backup;

   public WarehouseImpl(Warehouse backup) throws RemoteException
   {
      products = new HashMap<>();
      this.backup = backup;
   }

   public void add(String keyword, Product product)
   {
      product.setLocation(this);
      products.put(keyword, product);
   }
```

Transferring Nonremote Objects


```
public double getPrice(String description) throws RemoteException {
   for (Product p : products.values())
      if (p.getDescription().equals(description)) return p.getPrice();
   if (backup == null) return 0;
   else return backup.getPrice(description);
}
public Product getProduct(List<String> keywords) throws RemoteException {
   for (String keyword : keywords) {
      Product p = products.get(keyword);
      if (p != null) return p;
   }
   if (backup != null)
      return backup.getProduct(keywords);
   else if (products.values().size() > 0)
      return products.values().iterator().next();
   else
      return null;
   }
}
```

Transferring Nonremote Objects


```
public class WarehouseServer
{
   public static void main(String[] args) throws RemoteException, NamingException
   {
      System.out.println("Constructing server implementation...");
      WarehouseImpl backupWarehouse = new WarehouseImpl(null);
      WarehouseImpl centralWarehouse = new WarehouseImpl(backupWarehouse);

      centralWarehouse.add("toaster", new Product("Blackwell Toaster", 23.95));
      backupWarehouse.add("java", new Book("Core Java vol. 2", "0132354799", 44.95));

      System.out.println("Binding server implementation to registry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:central_warehouse", centralWarehouse);

      System.out.println("Waiting for invocations from clients...");
   }
}
```

Transferring Nonremote Objects


```
public class WarehouseClient
{
   public static void main(String[] args) throws NamingException, RemoteException
   {
      Context namingContext = new InitialContext();

      System.out.print("RMI registry bindings: ");
      NamingEnumeration<NameClassPair> e = namingContext.list("rmi://localhost/");
      while (e.hasMore())
         System.out.println(e.next().getName());

      String url = "rmi://localhost:1099/central_warehouse";
      Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);

      Scanner in = new Scanner(System.in);
      System.out.print("Enter keywords: ");
      List<String> keywords = Arrays.asList(in.nextLine().split("\\s+"));
      Product prod = centralWarehouse.getProduct(keywords);

      System.out.println(prod.getDescription() + ": " + prod.getPrice());
   }
}
```

Transferring Nonremote Objects

- Running the Program
  - Add `ClassDir` to `CLASSPATH`
  - For example: `E:\Projects\JavaEE\WareHouseServer\bin`
  - Create a `jndi.properties` file, and copy it into `/Server` and `/Client` class dir.

```
java.naming.factory.initial=com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url=rmi://localhost:1099
```

  - Run the `rmiregistry` in a windows console
  - Run the `WarehouseServer` in another windows console, you will see:

```
Constructing server implementation...
Binding server implementation to registry...
Waiting for invocations from clients...
```

  - Run the `WarehouseClient` in the third windows console, you will see:

```
RMI registry bindings: central_warehouse
Enter Keywords:
```

Remote Object Activation

- The activation mechanism
  - lets you delay the object construction so that a remote object is only constructed when at least one client invokes a remote method on it.

```
class WarehouseImpl extends
    Activatable implements Warehouse { . . . }
```

- You must provide a constructor that takes two parameters:
  - An activation ID (which you simply pass to the superclass constructor).
  - A single object containing all construction information, wrapped in a `MarshalledObject`.

Remote Object Activation


```
public interface Warehouse extends Remote{
   double getPrice(String description) throws RemoteException;
}

public class WarehouseImpl extends Activatable implements Warehouse
{
   private Map<String, Double> prices;
   public WarehouseImpl(ActivationID id,
                        MarshalledObject<Map<String, Double>> param)
         throws RemoteException, ClassNotFoundException, IOException
   {
      super(id, 0);
      prices = param.get();
      System.out.println("Warehouse implementation constructed.");
   }
   public double getPrice(String description) throws RemoteException
   {
      Double price = prices.get(description);
      return price == null ? 0 : price;
   }
}
```

Remote Object Activation

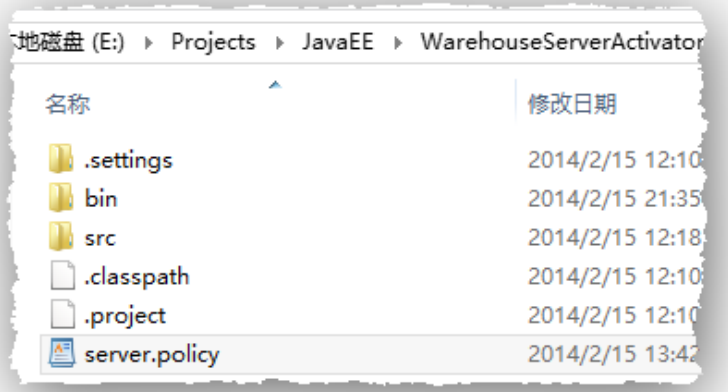

```
public class WarehouseActivator
{
   public static void main(String[] args) throws RemoteException, NamingException,
         ActivationException, IOException
   {
      System.out.println("Constructing activation descriptors...");

      Properties props = new Properties();
      props.put("java.security.policy",
                new File("server.policy").getCanonicalPath());

      ActivationGroupDesc group = new ActivationGroupDesc(props, null);
      ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);

      Map<String, Double> prices = new HashMap<>();
      prices.put("Blackwell Toaster", 24.95);
      prices.put("ZapXpress Microwave Oven", 49.95);

      MarshalledObject<Map<String, Double>> param =
                     new MarshalledObject<Map<String, Double>>(prices);

      String codebase = "http://localhost:8080/";
```

Remote Object Activation


```
        ActivationDesc desc = new ActivationDesc(id,
                                        "WarehouseImpl", codebase, param);

        Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);

        System.out.println("Binding activable implementation to registry...");
        Context namingContext = new InitialContext();
        namingContext.bind("rmi:central_warehouse", centralWarehouse);
        System.out.println("Exiting...");
    }
}
```

- Server.policy

```
grant
{
    permission java.security.AllPermission;
};
```



Remote Object Activation


```
public class WarehouseClient
{
   public static void main(String[] args) throws NamingException, RemoteException
   {
      Context namingContext = new InitialContext();

      System.out.print("RMI registry bindings: ");
      Enumeration<NameClassPair> e = namingContext.list("rmi://localhost/");
      while (e.hasMoreElements())
         System.out.println(e.nextElement().getName());

      String url = "rmi://localhost/central_warehouse";
      Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);

      String descr = "ZapXpress Microwave Oven";//"Blackwell Toaster";
      double price = centralWarehouse.getPrice(descr);
      System.out.println(descr + ": " + price);
   }
}
```

Remote Object Activation

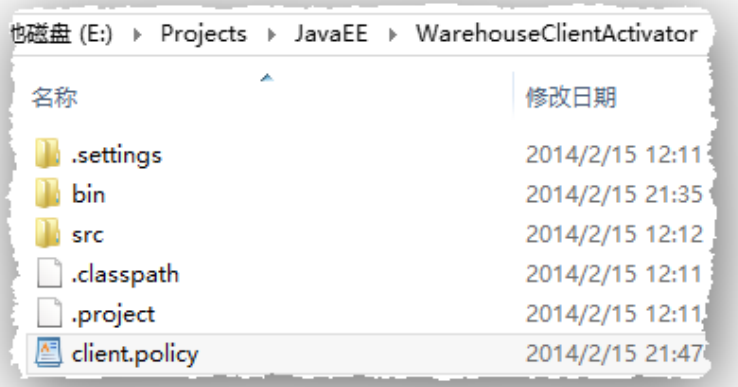
- Client.policy

```
grant
{
    permission com.sun.rmi.rmid.ExecPermission
       "${java.home}${/}bin${/}java";
    permission com.sun.rmi.rmid.ExecOptionPermission
       "-Djava.security.policy=*";
 };
```

- rmid.policy

```
grant
{
    permission com.sun.rmi.rmid.ExecPermis
       "${java.home}${/}bin${/}java";
    permission com.sun.rmi.rmid.ExecOptionP
       "-Djava.security.policy=*";
 };
```

- Copy the `rmid.policy` to `java.home/bin`



Remote Object Activation

1. Compile all source files.
2. Start the RMI registry with `rmiregistry`
3. Start the RMI activation daemon with

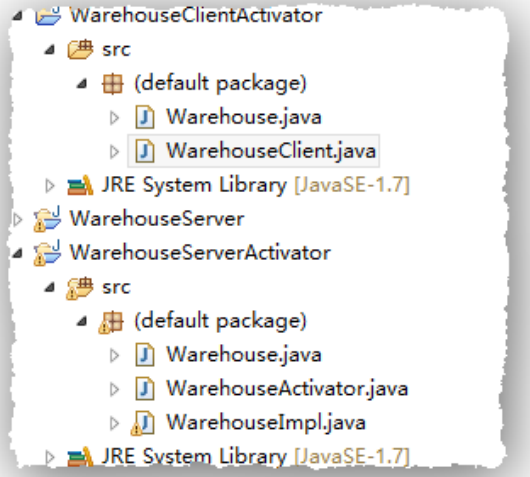
```
rmid -J-Djava.security.policy=rmid.policy
```

4. Run the activation program from the server directory.

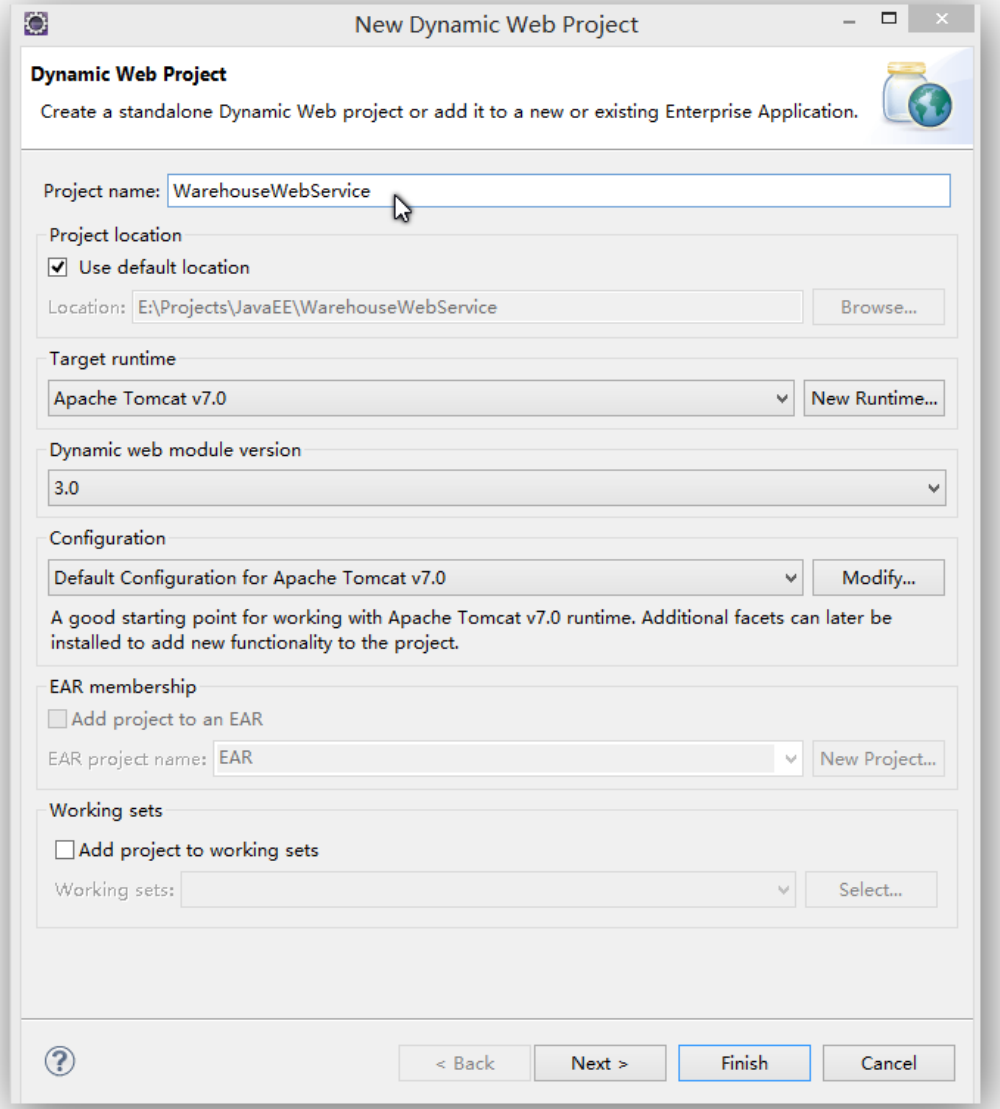
```
java -Djava.rmi.server.codebase=http://localhost:8080/
     WarehouseActivator
```

5. Run the client program from the client directory.

```
java –Djava.security.manager
     –Djava.security.policy==client.policy
     WarehouseClient
```



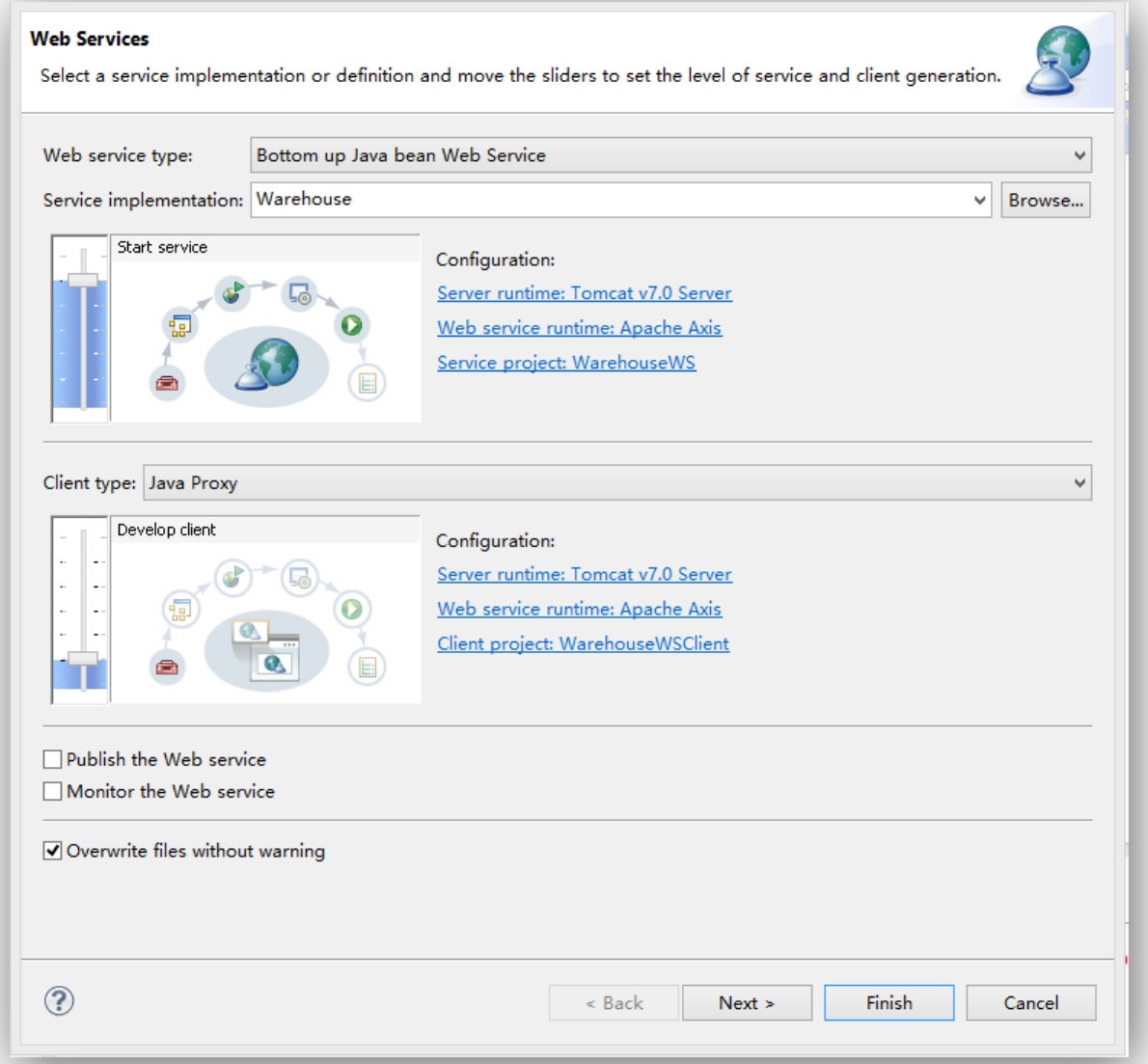
Web Services



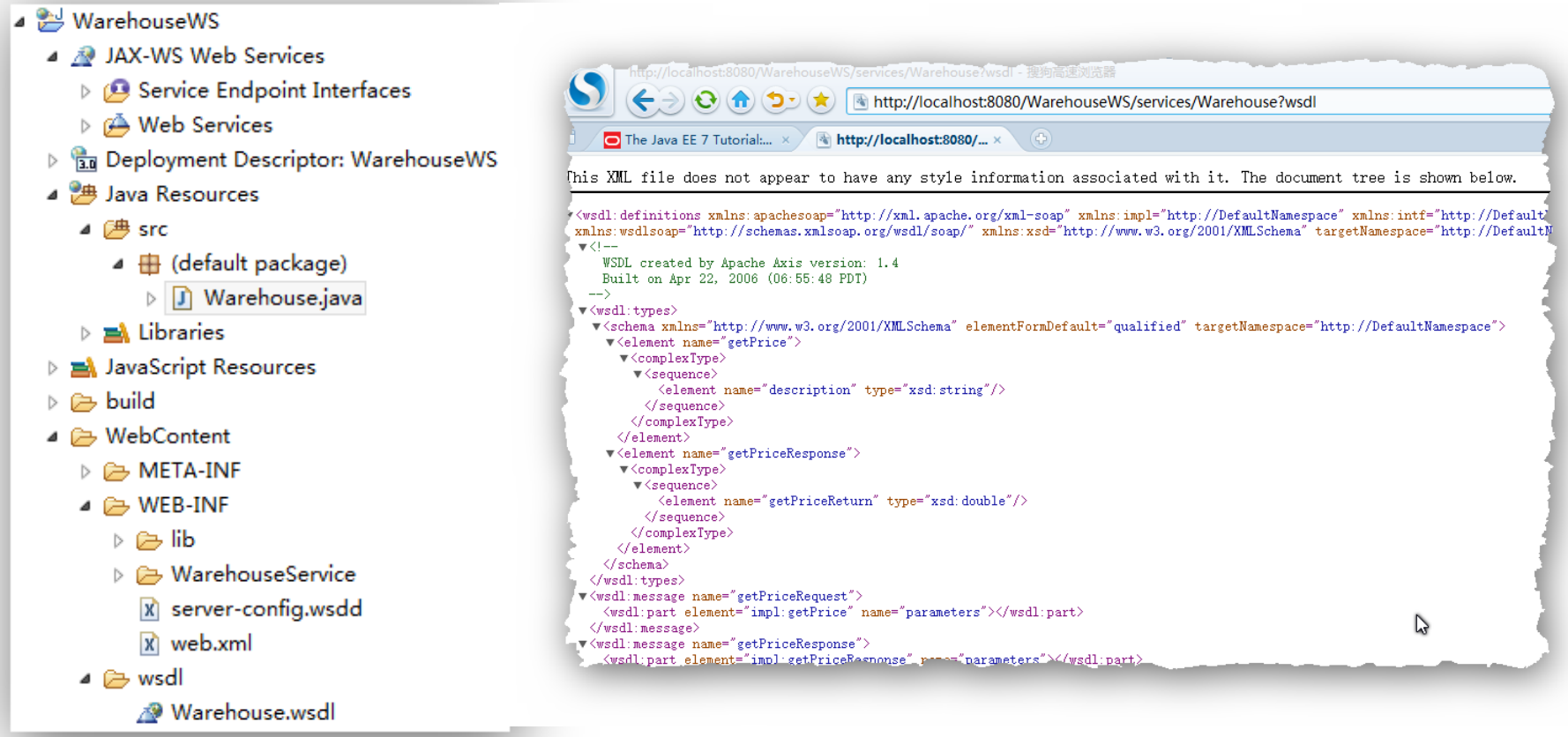
Web Services


```
import java.util.*;
import javax.jws.*;

@WebService
public class Warehouse {
   public Warehouse() {
      prices = new HashMap<String, Double>();
      prices.put("Blackwell Toaster", 24.95);
      prices.put("ZapXpress Microwave Oven", 49.95);
   }

   public double getPrice(@WebParam(name="description") String description)
   {
      Double price = prices.get(description);
      return price == null ? 0 : price;
   }

   private Map<String, Double> prices;
}
```

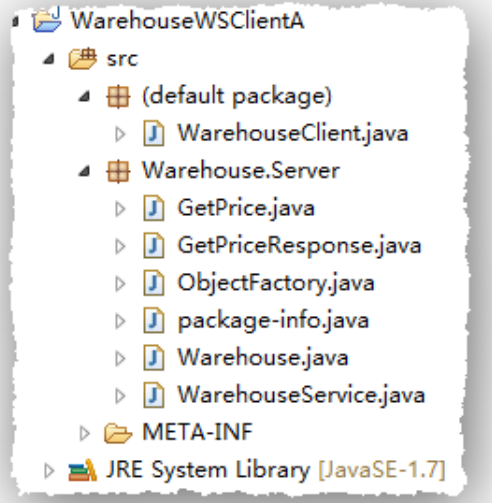
Web Services



Web Services



Web Service Client - A



- Create a plain java project

- Generate the necessary files for client and add them to the project

```
wsimport -keep -p warehouse.server
http://localhost:8080/WebServices/warehouse?wsdl
```

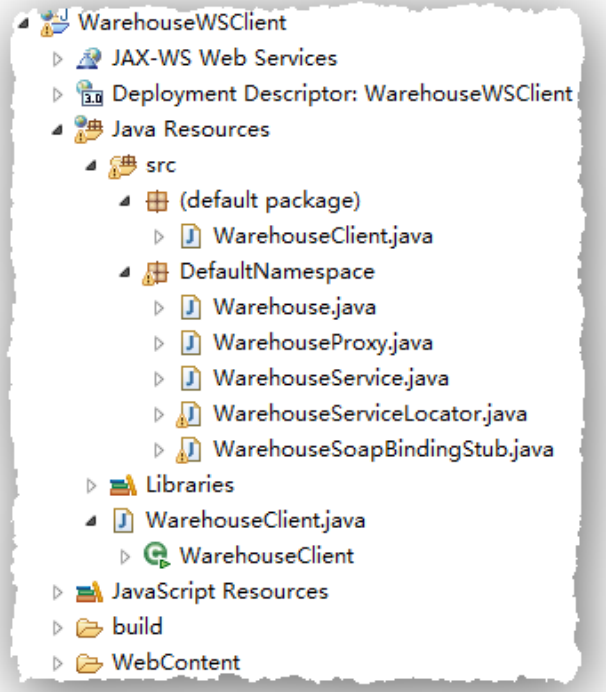
- Write a Web Service Client

```
public class WarehouseClient
{
   public static void main(String[] args) throws NamingException, RemoteException
   {
      WarehouseService service = new WarehouseService();
      Warehouse port = service.getPort(Warehouse.class);

      String descr = "ZapXpress Microwave Oven";
      double price = port.getPrice(descr);
      System.out.println(descr + ": " + price);
   }
}
```

Web Service Client - B

- Use the generated Client

```
public class WarehouseClient
{
    public static void main(String[] args) throws NamingException, RemoteException
    {
      WarehouseServiceLocator locator = new WarehouseServiceLocator();
      Warehouse warehouse = null;
      try{
          warehouse = locator.getWarehouse();
      }catch(Exception e){};

      String descr = "Blackwell Toaster";
      double price = warehouse.getPrice(descr);
      System.out.println(descr + ": " + price);
    }
}
```



References

- Core Java (volume II) 9th edition
  - http://horstmann.com/corejava.html
- The Java EE 7 Tutorial
  - http://docs.oracle.com/javaee/7/tutorial/doc/javaeetutorial7.pdf

Thank You!