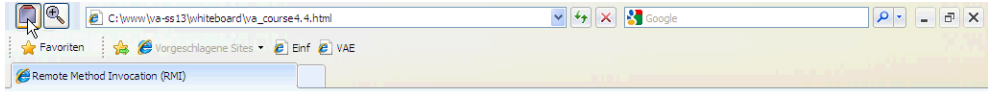
Script **generated by TTT**

Title: Distributed_Applications (27.05.2013)

Date: Mon May 27 09:11:29 CEST 2013

Duration: 41:20 min

Pages: 16



RMI supports communication among objects residing on different Java virtual machines (JVM). RMI is an RPC of the object-oriented Java environment.

Generated by Targeteam

1 Defining a remote interface

A remote interface is the set of methods that can be invoked remotely by a client.

- The remote interface must be declared public.
- The remote interface must extend the `java.rmi.Remote` interface.
- Each method must throw the `java.rmi.RemoteException` exception.
- If the remote methods have any remote objects as parameters or return types, they must be interfaces rather than implementation classes.
- **Example: remote interface definition**

```
public interface HelloInterface extends java.rmi.Remote {
    /* this method is called by remote clients and it is implemented by
    the remote object */
    public String sayHello ( ) throws java.rmi.RemoteException
}
```

Generated by Targeteam

2 Implementing the remote interface

Definition of an implementation class that defines the methods of the remote interface;

the abstract class `java.rmi.server.RemoteServer` provides the basic semantics to support remote references.

`java.rmi.server.RemoteServer` has subclasses

`java.rmi.server.UnicastRemoteObject` : defines a non-replicated remote object whose references are valid only while the server process is alive.

`java.rmi.activation.Activatable` : defines a remote object which can be instantiated on demand (if it has not been started already).

- **Example: Remote interface implementation**

```
import java.io.*;
import java.rmi.* ;
import java.rmi.server.* ;
import java.util.Date.* ;
public class HelloServer extends UnicastRemoteObject implements
HelloInterface{
    public HelloServer( ) throws RemoteException {
       super( );
       /* call superclass constructor to export this object */
    }
    public String sayHello( ) throws RemoteException {
       return "Hello World, the current system time is " + new Date( );
```

Definition of an implementation class that defines the methods of the remote interface;

the abstract class `java.rmi.server.RemoteServer` provides the basic semantics to support remote references.

`java.rmi.server.RemoteServer` has subclasses

`java.rmi.server.UnicastRemoteObject` : defines a non-replicated remote object whose references are valid only while the server process is alive.

`java.rmi.activation.Activatable` : defines a remote object which can be instantiated on demand (if it has not been started already).

- **Example: Remote interface implementation**

```
import java.io.*;
import java.rmi.* ;
import java.rmi.server.* ;
import java.util.Date.* ;
public class HelloServer extends UnicastRemoteObject implements
HelloInterface{
    public HelloServer( ) throws RemoteException {
       super( );
       /* call superclass constructor to export this object */
    }
    public String sayHello( ) throws RemoteException {
```

2 Implementing the remote interface

Definition of an implementation class that defines the methods of the remote interface;

the abstract class `java.rmi.server.RemoteServer` provides the basic semantics to support remote references.

`java.rmi.server.RemoteServer` has subclasses

`java.rmi.server.UnicastRemoteObject` : defines a non-replicated remote object whose references are valid only while the server process is alive.

`java.rmi.activation.Activatable` : defines a remote object which can be instantiated on demand (if it has not been started already).

- **Example: Remote interface implementation**

```
import java.io.*;
import java.rmi.* ;
import java.rmi.server.* ;
import java.util.Date.* ;
public class HelloServer extends UnicastRemoteObject implements
HelloInterface{
    public HelloServer( ) throws RemoteException {
       super( );
       /* call superclass constructor to export this object */
    }
    public String sayHello( ) throws RemoteException {
```

2 Implementing the remote interface

Definition of an implementation class that defines the methods of the remote interface;

the abstract class `java.rmi.server.RemoteServer` provides the basic semantics to support remote references.

`java.rmi.server.RemoteServer` has subclasses

`java.rmi.server.UnicastRemoteObject` : defines a non-replicated remote object whose references are valid only while the server process is alive.

`java.rmi.activation.Activatable` : defines a remote object which can be instantiated on demand (if it has not been started already).

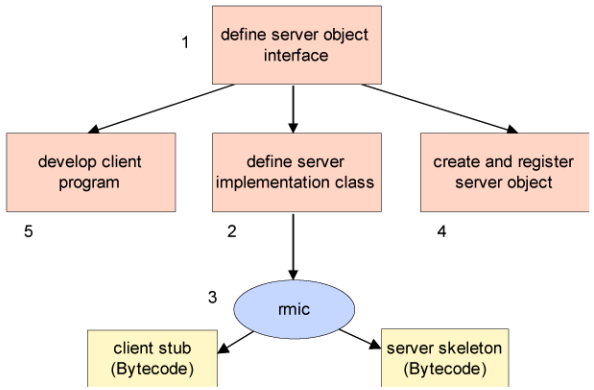
- **Example: Remote interface implementation**

```
import java.io.*;
import java.rmi.* ;
import java.rmi.server.* ;
import java.util.Date.* ;
public class HelloServer extends UnicastRemoteObject implements
HelloInterface{
    public HelloServer( ) throws RemoteException {
       super( );
       /* call superclass constructor to export this object */
    }
    public String sayHello( ) throws RemoteException {
       return "Hello World, the current system time is " + new Date( );
```

3 Generating stubs and skeletons

The tool rmic generates stub and skeleton from the implemented class (up to Java version 5).

```
rmic HelloServer
```



ception

4 Remote object registration

Every remotely accessible object must be registered in a registry in order to make it available;

stubs are needed for registration.

the registry is started at the host of the remote object.

- **Example for object registration**

```
import java.rmi.* ;

public class RegisterIt {

    public static void main (String args [])
        try { // Instantiate the object
           HelloServer obj = new HelloServer( );
           System.out.println ("Object instantiated: " + obj);
           Naming.rebind("/HelloServer", obj);
           System.out.println("HelloServer bound in registry");
        } catch (Exception e) {
           System.out.println(e)
        }                                          cep
    }

}
```


5 Client implementation

This step encompasses the writing of the client that uses remote objects.

The client must incorporate a registry lookup in order to obtain a reference to the remote object.

The client interacts with the remote interface, ***never*** with the object implementation.

- **Example: Client implementation**

```
import java.rmi.*;
public class HelloClient {
    public static void main (String args [ ]) {
        if (System.getSecurityManager( ) == null)
        System.setSecurityManager (new RMISecurityManager( ) );
        try { String name = "//" + args [0] + "/HelloServer";
            HelloInterface obj = (HelloInterface) Naming.lookup (name);
            String message = obj.sayHello( );
            System.out.println(message);

        } catch (Exception e) {
           System.out.println("HelloClient exception: " + e);
        }
    }
}
```

Missing access rights results in the exception:

Developing RMI applications

The steps developing an RMI application differs slightly from the development steps of a traditional RPC application.

1. Defining a remote interface
2. Implementing the remote interface
3. Generating stubs and skeletons
4. Remote object registration
5. Client implementation

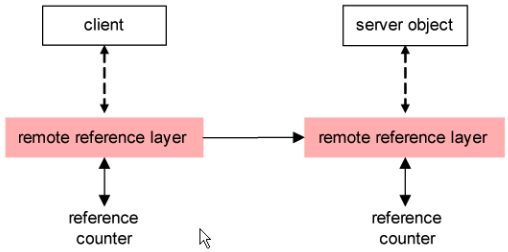
At the end the client must be started.

Remote Method Invocation (RMI)

RMI supports communication among objects residing on different Java virtual machines (JVM). RMI is an RPC of the object-oriented Java environment.

Distributed garbage collection

Utilization of life references for each JVM; reference counter represents the number of life references.



The first client access creates a referenced message sent to the server.

If there is no valid client reference, then an unreferenced message is sent to the server.

Time limit of references ("lease time", e.g. 10 minutes); the connection to the server must be renewed by the client, otherwise the reference becomes invalid.

Remote Method Invocation (RMI)

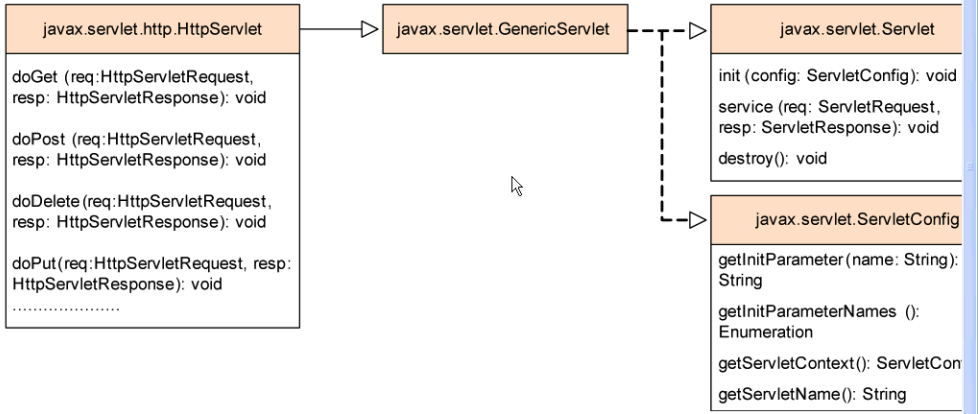
RMI supports communication among objects residing on different Java virtual machines (JVM). RMI is an RPC of the object-oriented Java environment.

HttpServlet Interface

HttpServlet inherits abstract class GenericServlet which implements interfaces Servlet and ServletConfig.

GenericServlet defines a generic protocol-independent servlet

HttpServlet defines a servlet for the HTTP protocol



doGet is invoked to respond to a GET request

doPost is invoked to respond to a POST request

doDelete is invoked to respond to a DELETE request; normally used to delete a file on the server

Servlet Properties

execution of a servlet in the context provided by the servlet engine.

Apache Tomcat : free, open-source implementation of Java servlet technology.

methods specified within each servlet object and invoked by the servlet engine

init: when a servlet is initialized.

shutdown: when a servlet is no longer needed.

service: when a client request is forwarded to the servlet.

servlets are invoked via HTTP requests (get or post method), e.g.

```
<form method="post"
action="http://myhost:8080/servlet/formServlet">
.... arguments of the form .....
```

Interface `javax.servlet.Servlet` specifies the methods to be implemented by the servlets.

```
public void init() throws ServletException;

public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException;

public void destroy();
```