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**

```java
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
}
```

**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**

```java
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();
    }
}
4 Remote object registration

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

```
rmic HelloServer
```

```
1
define server object interface

2
develop client program

3
define server implementation class

4
create and register server object

5
client stub (Bytecode)

Server skeleton (Bytecode)
```

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)
    }
}
```

5 Client implementation

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

This 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);
    }
}
```

Parameters with primitive data types are passed with their values between JVMs; for object parameters, a
 distinction is made between local and remote:

1. local object parameter

   RMI places the object itself, rather than the object reference.

   The transmitted object must implement the interface java.io.Serializable or
   java.io.Externalizable.

   Classes requiring special handling must implement

```
private void writeObject(java.io.ObjectOutputStream) throws IOException
    private void readObject(java.io.ObjectInputStream) throws IOException. ClassNotFoundException;
```

2. remote object parameter

   RMI transmits the stub of the remote object; the stub is a reference to the remote object.
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.

**Definitions**

- RMI characteristics
- RMI architecture
- Locating remote objects
- Developing RMI applications
- Parameter Passing in RMI
- Distributed garbage collection

Servlet Properties

Servlet 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.
  - `service` when a client request is forwarded to the servlet.
  - Servlets are invoked via HTTP requests (get or post method), e.g.
    ```html
    <form method="post" action="http://myhost:8080/servlet/formServlet">
      .... arguments of the form ..... 
    </form>
    ```
import javax.servlet.http.*;
import java.io.*;

public class MyServlet extends HttpServlet {
    /* called by the servlet engine to initialize servlet */
    public void init() throws ServletException {
        // setup the servlet context
    }
    /* process the HTTP Get request */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // process the HTTP Get request
    }
    /* process the HTTP Post request */
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // process the HTTP Post request
    }
    /* called by the servlet engine to release the resource */
    public void destroy() {
        // close resources
    }
    // other methods
}

Example - CurrentTime
Example - Registration of Students
This HTML form and the associated servlet process student registrations.

HTML Form

<html>
<head>
<title>Student Registration Form</title>
</head>
<body>
<p>Student Registration Form</p>
<form action="http://localhost:8080/example/servlet/GetParameters" method="GET">
  <p>Last Name <input type="text" name="lastName" size="20"></p>
  <p>First Name <input type="text" name="firstName" size="20"></p>
  <p>Gender:
    <input type="radio" name="gender" value="M"> Male
    <input type="radio" name="gender" value="F"> Female</p>
  <p>Major <select name="major" size="1">
    <option value="CS">Computer Science</option>
    <option value="MA">Mathematics</option>
  </select></p>
  <p>Minor <select name="minor" size="2" multiple>
    <option>Computer Science</option>
    <option>Mathematics</option>
    <option>Economics</option>
  </select></p>
</form>
</body>
</html>

Java Code:

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetParameters extends HttpServlet {

    /** process the HTTP GET request */
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // obtain parameters from the client
        String lastName = request.getParameter("lastName");
        String firstName = request.getParameter("firstName");
        String gender = request.getParameter("gender");
        String major = request.getParameter("major");
        String[] minors = request.getParameterValues("minor");
        String tennis = request.getParameter("tennis");
        String soccer = request.getParameter("soccer");
        String golf = request.getParameter("golf");
        String remarks = request.getParameter("remarks");

        out.println("Last Name: 
        + lastName + 
        First Name: 
        + firstName + 
        Gender: 
        + gender + 
        Minor: 
        + minors + 
        Tennis: 
        + tennis + 
        Soccer: 
        + soccer + 
        Golf: 
        + golf + 
        Remarks: 
        + remarks + 
        close() // close stream
    }
```
Servlets (Java Servlets) are programs invoked by a client and executed on the server host, used to extend the functionality of the server.

**Servlet Properties**
- Servlet Lifecycle
- HttpServlet Interface
- Structure of a Servlet

**Issues**
The following section discusses several important basic issues of distributed applications:
- Data representation in heterogeneous environments.
- Discussion of an execution model for distributed applications.
- What is the appropriate error handling?
- What are the characteristics of distributed transactions?
- What are the basic aspects of group communication (e.g., algorithms used by ISIS)?
- How are messages propagated and delivered within a process group in order to maintain a consistent state?

**External data representation**
- Independence from hardware characteristics while exchanging messages means: use of external data representation.
- Marshalling and unmarshalling
- Centralized transformation
- Decentralized transformation
- Common external data representation
- XML as common data representation

**External data representation**
- Independence from hardware characteristics while exchanging messages means: use of external data representation.
- Marshalling and unmarshalling
- Centralized transformation
- Decentralized transformation
- Common external data representation
- XML as common data representation
Two aspects of a common external data representation are of importance:
- a machine-independent format for data representation, and
- a language for description of complex data structures.

Examples: XDR ("eXternal Data Representation") by Sun and ASN.1 (Abstract Syntax Notation). Other formats are
- Corba's common data representation: structured and primitive types can be passed as arguments and results.
- Java's object serialization: flattening of single objects or tree of objects.

Examples:
- XDR ("eXternal Data Representation") by Sun and ASN.1 (Abstract Syntax Notation). Other formats are
- Corba's common data representation: structured and primitive types can be passed as arguments and results.
- Java's object serialization: flattening of single objects or tree of objects.

**Common external data representation**

For the representation of numbers in main memory, one of the following methods are generally used:
- "little endian" representation: the lower part of a number is stored in the lower memory area.
- "big endian" representation: the higher part of a number is stored in the lower memory area, e.g. the Sun-Sparc architecture.

**Example representation of the number 1347**

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Endian</td>
<td>00000011</td>
<td>00000101</td>
<td>00000101</td>
<td>01000011</td>
</tr>
<tr>
<td>Little Endian</td>
<td>01000011</td>
<td>00000101</td>
<td>00000101</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Convention: for network transfer, numbers which encompass several bytes are structured according to a well-defined representation, such as "big endian".
no shared address space for client and server; pointer transfer is problematic.
1. prohibit pointers in a remote procedure call,
2. dereference pointers in a remote procedure call,
   serialize the data structure the pointer is pointing to ("marshal"), and transfer the whole data structure.
   - no use of null pointers; instead, we use boolean variables.
   - in heterogeneous environments no use of function pointers.
   However, in a homogeneous Java environment function pointers can be dereferenced and the function transferred to the server site.
3. pointer transfer.

Heterogeneous environment means different data representations
   requirement to enable data transformation.

Independence from hardware characteristics while exchanging messages means; use of external data representation.

Marshalling and unmarshalling
Centralized transformation
Decentralized transformation
Common external data representation
XML as common data representation

Request for the invocation of the Java method

```xml
<soap:Body>
  <m:echoString>
    xmlns:n="http://tempuri.org/mapping.server.Primitive"
    xsi:type="xsd:string">cat</value>
  </m:echoString>
</soap:Body>
```

Request for Java method invocation

```xml
<soap:Body>
  <m:echoInts>
    xmlns:n="http://tempuri.org/mapping.server.Array"
    xsi:type="xsd:int[3]">
    <int:href="#id0">
    </m:echoInts>
    <id0 id="id0">
      soapenc:root="0"
      xsi:type="soapenc:Array"
      soapenc:ArrayType="xsd:int[3]">
      <i xsi:type="xsd:int">1</i>
      <i xsi:type="xsd:int">2</i>
      <i xsi:type="xsd:int">3</i>
    </id0>
</soap:Body>
```
Complex data types can be mapped to XML for transmission across the network.

**Example: primitive datatypes**
SOAP provides built-in support for encoding arrays.

**Example: array datatype**
Complex data types are mapped to XML schema types:

- SOAP platforms provide API for creating custom mapping.
- `e.g. writeSchema` to specify an XML schema definition

<table>
<thead>
<tr>
<th>Level of Abstraction</th>
<th>Application specific data encoding language</th>
<th>XML</th>
<th>General data encoding language</th>
<th>ASN 1</th>
<th>Network data encoding language</th>
<th>Sun XDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>XML</td>
<td></td>
<td>General data encoding language</td>
<td>ASN 1</td>
<td>Network data encoding language</td>
<td>Sun XDR</td>
</tr>
<tr>
<td>low</td>
<td>Application specific data encoding language</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>