Distributed systems

Prof. Dr. Ralf Lämmel
University of Koblenz-Landau
Faculty of Computer Science
Software Languages Team
Related concepts

- Parallel computing
- Distributed computing
- Concurrent computing
Concept summary

- **Parallel** programming:
  - There are several (possibly virtual) processors.
  - Forms of parallelism
    - Task parallelism -- distribute execution processes
    - Data parallelism -- distribute data across parallel computing nodes
- **Concurrent** programming: parallelism + interaction
- Multithreading: Java’s approach to task parallelism
- Multitasking: OS-level parallelism
- **Distributed** programming: client, server, ...
Distributed system approaches

- (RPC (RMI))
- Messaging
- REST
- (Web Services)

Too low-level and interoperable!

Too complicated!
Elevator speech

What if the objects need to be distributed in a network? How to communicate between different computers in an OOP-friendly manner? Basically, we want to continue to hold references to objects (perhaps remote objects), and send them messages (i.e., perform method calls) as before.

The **Java Remote Method Invocation** (Java RMI) is a Java API that performs the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage collection. [Source: http://en.wikipedia.org/wiki/Java_remote_method_invocation]

An RPC is „an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. [Source: http://en.wikipedia.org/wiki/Remote_procedure_call]

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Non-distributed programming
Caller and callee are on the same machine.

Method Invocation
- Arguments are evaluated.
- Caller location is pushed onto stack.
- Callee method is executed.
- Result is returned.
- Caller location resumes.

Object1
(Client)

MyMachine

Object2
(Server)

MyMachine

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Distributed programming

Caller and callee are on different machines.

In demo code, client and server may be on the same machine, perhaps even on the same JVM, but the invocation is handled (conceptually) over the network.

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
A first look at the “hello world” of RMI

A SAMPLE

https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javaRmiSamples

• See package “helloworld” of javaRmiSamples.
• Start the server (as described in “Makefile”).
• Start the client (by running the “Client” class).
• Observe how 41 is incremented.
A more interesting scenario: a banking application to access account data remotely at a teller (ATM).
Communication between different Java runtimes on the same machine or different machines, eventually involves the network layer of the OS.
Stubs and skeletons

- Client invokes a remote method on “stub” object.
- The stub sends the call to the server-side “skeleton”.
- That is, the stub:
  - opens a socket to the remote server,
  - marshals the method parameters,
  - forwards the data stream to the skeleton,
  - awaits and unmarshals result data stream from skeleton.
- The skeleton contains a method that:
  - receives the remote calls,
  - unmarshals the method parameters,
  - invokes the actual remote (by now, local) implementation, and
  - marshals the result and delivers it back to the stub.

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.

© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Remote-method invocation

Parameters and return values
- Remote objects – by reference
- Serializable objects – by copy
- Others – cannot be passed (exception)

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Naming service

A server-side directory that associates remote objects with names (URLs)
Components of an RMI-based application

- Services of remote objects (interfaces)
- Implementation of services (classes)
- Proxies for client-side (classes, generated)
- A server that binds services (class)
- A client that looks up services (class)
- The RMI registry (component of RMI layer)
- Security policies for client and server (optional)
Designing a client/server app (Detailed steps)

1. Design an interface for service (remote object).
2. Implement the service.
3. (Generate stub and skeleton classes.)
4. Implement a server to contact for binding.
5. Implement a client to invoke service.
6. Start the server.
7. Run the client.

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Step 1:
Design an interface for service (remote object).

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

public interface Service extends Remote {
    public long inc(long x)
        throws RemoteException;
}
```

This is a regular interface except for the special base interface and the special throws declaration.
Step 2: Implement the service.

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

public classServiceImpl
    extends UnicastRemoteObject
    implements Service {

    // Needed for serialization
    private static final long serialVersionUID = 6102178242852627613L;

    // Needed because of exception
    public ServiceImpl() throws RemoteException {
        super();
    }

    public long inc(long x) throws RemoteException {
        return ++x;
    }
}
```

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Step 3:
Generate stub and skeleton classes.

Done automagically.
Step 4: Implement a server to contact for binding.

```java
import java.rmi.Naming;

public class Server {
    public Server() {
        try {
            Service s = new ServiceImpl();
            Naming.rebind("rmi://localhost/Service", s);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }

    public static void main(String args[]) {
        new Server();
    }
}
```

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Name format for binding

\[\text{rmi://}\text{<host\_name>}\]
\[\text{[::<name\_service\_port>]}\]
\[\text{/}<\text{service\_name}>\]

Thus, URLs of a specific form are used for addressing remote objects.
Availability of registry

- Binding requires that the registry is locally running.
- Registry can be started programmatically as follows:

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
```

- Default TCP/IP port: 1099
Usage of “localhost”

- localhost may fail to work in binding.
- This depends on the network configuration.
- Alternatively, one can use the following idiom:

  ```java
  InetAddress addr = InetAddress.getLocalHost();
  String hostname = addr.getHostName();
  Naming.rebind("rmi://"+hostname+"/Service", s);
  ```

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Step 5:
Implement a client to invoke service.

import java.rmi.Naming;

public class Client {

    public static void main(String[] args) {
        try {
            Service s = (Service) Naming.lookup("rmi://localhost/Service");
            System.out.println(s.inc(41));
        } catch (Exception e) {
            ...
        }
    }
}

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Step 6:
Start the server.

JavaRmi> java helloworld.Server

One may want to run the server (and the client) with a security policy.

Server must be killed, e.g., with CTRL-C

If you are using Eclipse, you cannot (easily) run both server and client simultaneously from within Eclipse. One of the two would be started from the command line.
Step 7: Run the client.

JavaRmi> java helloworld.Client

42
An RMI-based banking app

A SAMPLE

https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javaRmiSamples

- A non-C/S-based version as a reference: package “banking.local”.
- The C/S-based version: package “banking.remote”.
- Think of a refactoring to derive the latter from the former.

Acknowledgement: Content for the RMI topic had been originally authored by Jürgen Starek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Distributed system approaches

- (RPC (RMI))
- Messaging
- REST
- (Web Services)

Too low-level and interoperable!

Too complicated!
Messaging

- Message delivery instead of data/reference sharing.
- Messaging based on Hoare’s formal language for describing patterns of interaction in concurrent systems called ‘Communicating Sequential Processes (CSP)’; languages such as Occam or Go are influenced by CSP.
- Message service:
  - Message-oriented middleware
  - Create, edit, read and send messages
  - Send messages to destinations
  - Publish messages to all subscribers
  - Clients potentially oblivious to each other's existence

Acknowledgement: Content for the messaging topic had been originally authored by Marcel Heinz.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Publish-subscribe messaging

Publisher1 → Server → Subscriber1
Publisher2 → Server → Subscriber2
Publisher3 → Server

Point-to-point messaging

Source1 → Queue → Target
Source2 → Queue
Source3 → Queue

Acknowledgement: Content for the messaging topic had been originally authored by Marcel Heinz.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Java Message Service (JMS) API

- Supports Publish-Subscribe, Point-To-Point messaging, asynchronous- and synchronous communication.

- **Message Types:** Empty message, JavaPrimitive Stream message, MapMessage, TextMessage, ObjectMessage, BytesMessage.

- Messages contain additional information such as a timestamp or user assigned properties.

- A JMS implementation is ActiveMQ from Apache.

- The approach offers high robustness and guaranteed delivery.

- JMS is used frequently in JavaEE Applications.

Acknowledgement: Content for the messaging topic had been originally authored by Marcel Heinz.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
The JMS API Programming Model

Acknowledgement: Content for the messaging topic had been originally authored by Marcel Heinz.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
JMS on 101

- Point-to-point
  - https://101wiki.softlang.org/Contribution:jmsPointToPoint

- Publish-subscribe
  - https://101wiki.softlang.org/Contribution:jmsPublishSubscribe

- Other technologies on 101 (related to distribution):
  - Akka
  - Hadoop

Acknowledgement: Content for the messaging topic had been originally authored by Marcel Heinz.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Distributed system approaches

- (RPC (RMI))
- Messaging
- REST
- (Web Services)

Too low-level and interoperable!

Too complicated!
Representational State Transfer (REST)

REST is „a software architecture style consisting of guidelines and best practices for creating scalable web services“.
[Source: http://en.wikipedia.org/wiki/Representational_state_transfer]
The API notion

- API — Application Programming Interface
  - An interface for functionality that is reusable by programs
- Web API
  - An interface provided over the Web (using HTTP)
Intermezzo: read data on the network instead of the file system
package org.softlang.company.features;

import java.io.File;
import java.io.IOException;
import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class Parsing {

    public static Document parseCompany(String file) throws IOException, JDOMException {
        return new SAXBuilder().build(new File(file));
    }

}
package org.softlang.company.features;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import java.io.IOException;
import java.net.URL;

public class Parsing {

    public static Document parseCompany(String url) throws IOException, JDOMException {
        return new SAXBuilder().build(new URL(url).openStream());
    }

}
More pointers for comparison

- http://101companies.org/wiki/Contribution:jdom

The difference between file versus network stream is minimal both in the case of XML and JSON.
Aspects of a RESTful web service API

- A base URI such as http://example.com/resources/
- An internet media type such as JSON
- Standard HTTP methods: GET, PUT, DELETE, POST
- Links to reference state such as http://.../resources/1
Constraints of
REST — Representational state transfer

- "Client-server": data on server, UI on client, …
- "Stateless": no client context being stored on server
- "Cacheable": responses define themselves as cacheable

REST is an architectural style — not a protocol
Flask RESTful

https://flask-restful.readthedocs.io/en/
A minimal API

```python
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True)
```

Source: https://flask-restful.readthedocs.io/en/
API access at the command line

$ python api.py
  * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
  * Restarting with stat
  * Debugger is active!
  * Debugger PIN: 265-716-942

127.0.0.1 -- [22/Jun/2017 10:08:39] "GET / HTTP/1.1" 200 --

$ curl http://127.0.0.1:5000/
{
  "hello": "world"
}
Management of TODO items

At the command line

$ curl http://localhost:5000/todo1 -d "data=Remember to buy Bordeaux" -X PUT
{"todo1": "Remember to buy Bordeaux"}

$ curl http://localhost:5000/todo1
{"todo1": "Remember to buy Bordeaux"}

$ curl http://localhost:5000/todo2 -d "data=Eat cheese" -X PUT
{"todo2": "Eat cheese"}

Source: https://flask-restful.readthedocs.io/en/
Management of TODO items

At the Python prompt

```python
>>> from requests import put, get

>>> put('http://localhost:5000/todo1', data={'data': 'Remember to buy Bordeaux'}).json()
{u'todo1': u'Remember to buy Bordeaux'}

>>> get('http://localhost:5000/todo1').json()
{u'todo1': u'Remember to buy Bordeaux'}

>>> put('http://localhost:5000/todo2', data={'data': 'Eat cheese'}).json()
{u'todo2': u'Eat cheese'}
```

Source: https://flask-restful.readthedocs.io/en/
The API for managing TODO items

```python
from flask import Flask, request
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

todos = {}

class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        todos[todo_id] = request.form['data']
        return {todo_id: todos[todo_id]}

api.add_resource(TodoSimple, '/<string:todo_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

Source: https://flask-restful.readthedocs.io/en/
Path matching

todos = {}

```python
class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        todos[todo_id] = request.form['data']
        return {todo_id: todos[todo_id]}

api.add_resource(TodoSimple, '/todo/<int:todo_id>')
```

```
$ curl http://localhost:5000/todo/1 -d "data=Remember the beer" -X PUT
{
   "1": "Remember the beer"
}
```

Source: https://flask-restful.readthedocs.io/en/

© 2010-17, Ralf Lämmel, rlaemmel@acm.org
The following table shows how the HTTP methods are typically used to implement a RESTful API.

<table>
<thead>
<tr>
<th>Resource</th>
<th>GET</th>
<th>PUT</th>
<th>POST</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Collection URI, such as</strong></td>
<td><strong>List</strong> the URIs and perhaps other details of the collection's members.</td>
<td><strong>Replace</strong> the entire collection with another collection.</td>
<td><strong>Create</strong> a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.</td>
<td><strong>Delete</strong> the entire collection.</td>
</tr>
<tr>
<td><a href="http://example.com/resources">http://example.com/resources</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Element URI, such as</strong></td>
<td><strong>Retrieve</strong> a representation of the addressed member of the collection, expressed in an appropriate Internet media type.</td>
<td><strong>Replace</strong> the addressed member of the collection, or if it doesn't exist, <strong>create</strong> it.</td>
<td><strong>Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it.</strong></td>
<td><strong>Delete</strong> the addressed member of the collection.</td>
</tr>
<tr>
<td><a href="http://example.com/resources/item17">http://example.com/resources/item17</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed system approaches

- (RPC (RMI))
- Messaging
- REST
- (Web Services)

Too low-level and interoperable!

Too complicated!
Elevator speech

How to communicate across OS, programming language, type system, protocol? For instance, how to wire up a .NET component with a Java component? Also, how to reuse “services” across the internet/web?

See also: http://www.infoworld.com/d/architecture/infoclipz-service-oriented-architecture-soa-899

A Web Service is „a software system designed to support interoperable machine-to-machine interaction over a network“. [Source: W3C and http://en.wikipedia.org/wiki/Web_service]
Service-Oriented Architecture

- Goal: Combine and reuse
- Units: interoperable services
- Features: Loose coupling with
  - OS
  - language
  - technologies, etc.

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Illustration of services

A SAMPLE

https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javawsSamples/
package de.unikoblenz.sle;

import net.webservicex.GlobalWeather;

/**
 * Retrieve weather forecast
 *
 * @author pek
 *
 * public class WhatIsTheWeather {

 public static final void main(String[] args) {
 GlobalWeatherSoap weatherService = new GlobalWeather().getGlobalWeatherSoap();
 String weather = weatherService.getWeather("Koeln / Bonn", "Germany");
 System.out.println(weather);
 }

}
```java
package de.unikoblenz.sle;

import net.webservicex.Country;

/**
 * Retrieve information about Germany
 */
@Author Rak
public class WhatGMTAndISOare {

    public static void main(String[] args) {
        CountrySoap countryInfo = new Country().getCountrySoap();
        String countryName = "Germany";
        String GMTzone = countryInfo.getGMTbyCountry(countryName);
        String ISOcode = countryInfo.getISOCountryCodeByCountyName(countryName);
        System.out.println(GMTzone);
        System.out.println(ISOcode);
    }
}
```

---

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
WebService as an implementation of SOA

Service Broker

Publishes WSDL

Service Provider

Service contract

Interact

Service

Client

Finds WSDL

Service Consumer

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.

© 2010-17, Ralf Lämmel, rlaemmel@acm.org
WebService as an implementation of SOA

Service

Service contract

WSDL

XML message

XML message

SOAP

Client

Java

Java

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
WebService as an implementation of SOA

Java

Service

WSDL

Service contract

XML message

SOAP

Client

NET

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
WebService as an implementation of SOA

.NET

Service

Service contract

WSDL

XML message

SOAP

Java

Client

XML message

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
Running example: Counting words

- **Service:** count words of a string
- **Input:** string (space separated words)
- **Output:** int for number of words

https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javawsSamples/wcservice
Web Services

Description Language 1.1

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
SOAP messages for the running example

Request

<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:count xmlns:ns2="http://sle.unikoblenz.de/">
      Hello .NET
    </ns2:count>
  </S:Body>
</S:Envelope>

Response

<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns1:countResponse xmlns:ns1="http://sle.unikoblenz.de/">
      2
    </ns1:countResponse>
  </S:Body>
</S:Envelope>
Simple Object Access Protocol

- SOAP uses an internet application layer protocol as a transport protocol (SMTP, RPC, HTTP)
- Uses XML as message format

- Platform independent
- Language independent
Java API for XML Web Services (JAX-WS)

- Annotations for WebServices creation
- Server- and client-side support
- Part of Java SE 6
- Part of Java EE 5 platform
Java

Service

WSDL

Service contract

XML message

XML message

SOAP

Java

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.

© 2010-17, Ralf Lämmel, rlaemmel@acm.org
JAX-WS (Server side)

```java
@WebService
public class Wordings {
  @WebMethod
  public int count(String text) {
    // TODO this is just a stub
    return 42;
  }
}
```

```
<message name="Wording_count">
  <part name="count" type="xs:string"/>
</message>

<message name="Wording_countResponse">
  <part name="countResponse" type="xs:int"/>
</message>

<portType name="Wording">
  <operation name="count">
    <input message="tns:Wording_count"/>
    <output message="tns:Wording_countResponse"/>
  </operation>
</portType>
```

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.

© 2010-17, Ralf Lämmel, rla@cs.stonybrook.edu
JAX-WS (Client side)

> `$JAVA_6_HOME/bin/wsoimport URL -keep`  

<table>
<thead>
<tr>
<th>Generated file</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count.java</td>
<td>Wrapper for request message</td>
</tr>
<tr>
<td>CountResponse.java</td>
<td>Wrapper for response message</td>
</tr>
<tr>
<td>ObjectFactory.java</td>
<td>JAXB XML Registry</td>
</tr>
<tr>
<td>package-info.java</td>
<td>JAXB package annotations</td>
</tr>
<tr>
<td>Wording.java</td>
<td>Service Endpoint Interface</td>
</tr>
<tr>
<td>WordingService.java</td>
<td>WSDL service</td>
</tr>
</tbody>
</table>

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.  
© 2010-17, Ralf Lämmel, rlaemm@acm.org
@WebServiceClient(
    name = "WordingService",
    targetNamespace = "http://sle.unikoblenz.de/",
    wsdlLocation = "http://localhost:8080/TextStat/Wording.wsdl"
)

public class WordingService extends Service {

    public WordingService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public WordingService() {
        super(WORDINGSERVICE_WSDL_LOCATION,
             new QName("http://sle.unikoblenz.de/", "WordingService"));
    }

    @WebEndpoint(name = "WordingPort")
    public Wording getWordingPort() {
        return (Wording) super.getPort(
            new QName("http://sle.unikoblenz.de/", "WordingPort"),
            Wording.class);
    }
}

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.

© 2010-17, Ralf Lämmel, raemmel@acm.org
Dynamic proxy implementing SEI

```java
@WebService(
    name = "Wording",
    targetNamespace = "http://sle.unikoblenz.de/"
)@XmlSeeAlso({ObjectFactory.class})
```

JAXB use for WSDL⟷Java mapping

```java
public interface Wording {
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "count",
                    targetNamespace = "http://sle.unikoblenz.de/",
                    className = "de.unikoblenz.sle.Count")
    @ResponseWrapper(localName = "countResponse",
                    targetNamespace = "http://sle.unikoblenz.de/",
                    className = "de.unikoblenz.sle.CountResponse")
    @WebMethod
    public int count(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0);
}
```

Service Endpoint method

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.

© 2010-17, Ralf Lämmel, rlammel@acm.org
JAX-WS (Client side)

```xml
<portType name="Wording">
  <operation name="count" parameterOrder="count"> ... 
</portType>

<service name="WordingService">
  <port binding="tns:WordingBinding" name="WordingPort">
    <soap:address location="http://127.0.0.1:8080/TextStat/Wording"/>
  </port>
</service>

```

Client code

```java
public class Client {
    public static void main(String[] args) {
        Wording wording = new WordingService().getWordingPort();
        System.out.println(wording.count("Count this!"));
    }
}
```

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlammel@acm.org
Illustration of the wordcount service/client

https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javawsSamples

A SAMPLE

Omitted topics

- WS-*, additional specifications (e.g. WS-Security)
- Orchestration (WS-BPEL)
- Choreography (WS-CDL)
- ...

Acknowledgement: Content for the web-service topic had been originally authored by Ekaterina Pek.
© 2010-17, Ralf Lämmel, rlaemmel@acm.org
End of Lecture