# Jolie

A service-oriented programming language

Introduction and Project Ideas

Fabrizio Montesi, IT University of Copenhagen

# Jolie: a **service-oriented** programming language

- Nice logo:

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE INRIA
centre de recherche SOPHIA ANTIPOLIS - MÉDITERRANÉE

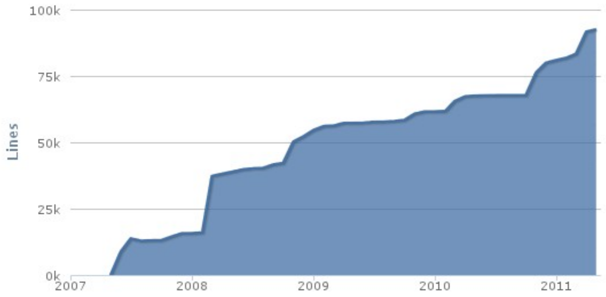
FOCUS Research Team

IT University of Copenhagen

- *Formal foundations* from the Academia.

- Tested and used in the *real world*: italianaSoftware Horsa™ We think. machines do.

- *Open source* (`http://www.jolie-lang.org/`), with a well-maintained code base:



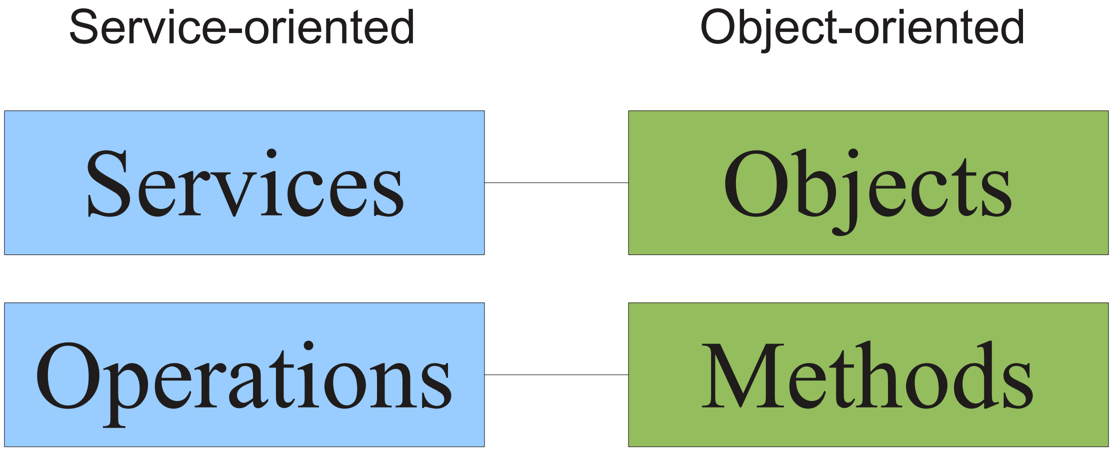
# Hello, Jolie!

- Yes, Jolie can print “Hello, world!”

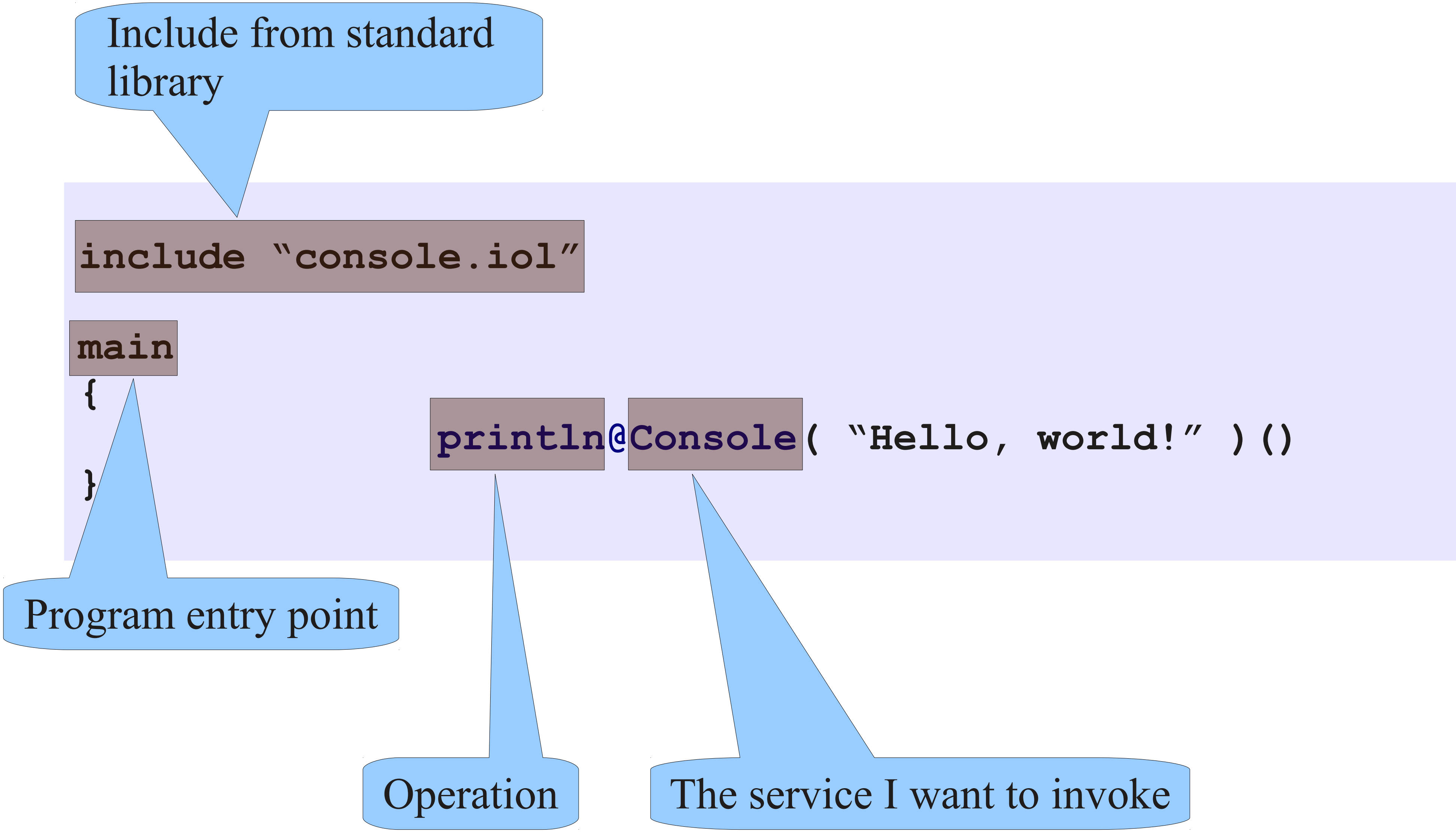
```
include “console.iol”

main
{
   println@Console( “Hello, world!” )()
}
```

# Basics

- A Service-Oriented Architecture (SOA) is composed by **services**.
- A **service** is an application that offers **operations**.
- A service can invoke another service by calling one of its **operations**.
- Recalling Object-oriented programming:



# Understanding Hello World: concepts



```
include “console.iol”

main
{
                    println@Console( “Hello, world!” )()

}
```

Include from standard library

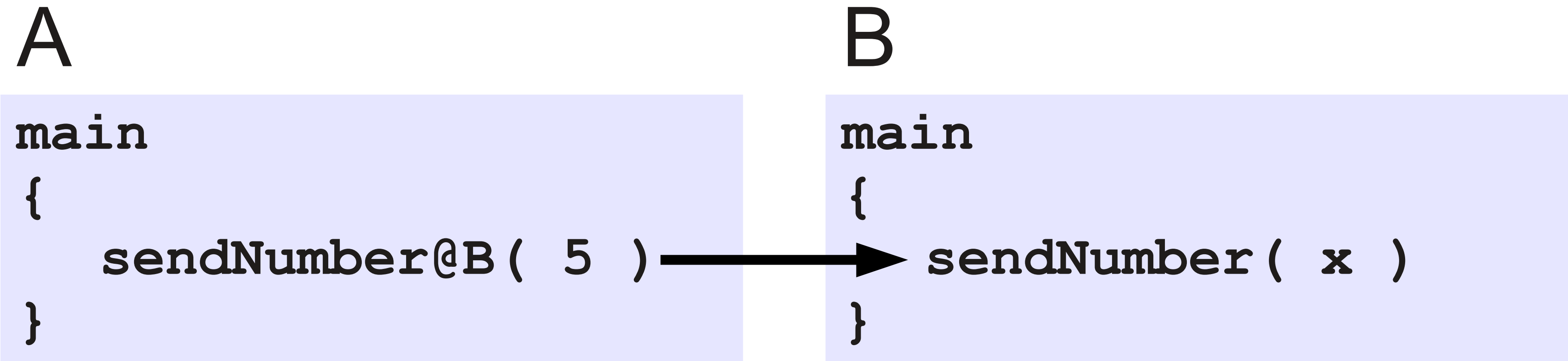
Program entry point

Operation

The service I want to invoke

# Our first service-oriented application

- A program defines the input/output communications it will make.



A

```
main
{
   sendNumber@B( 5 )
}
```

B

```
main
{
 sendNumber( x )
}
```

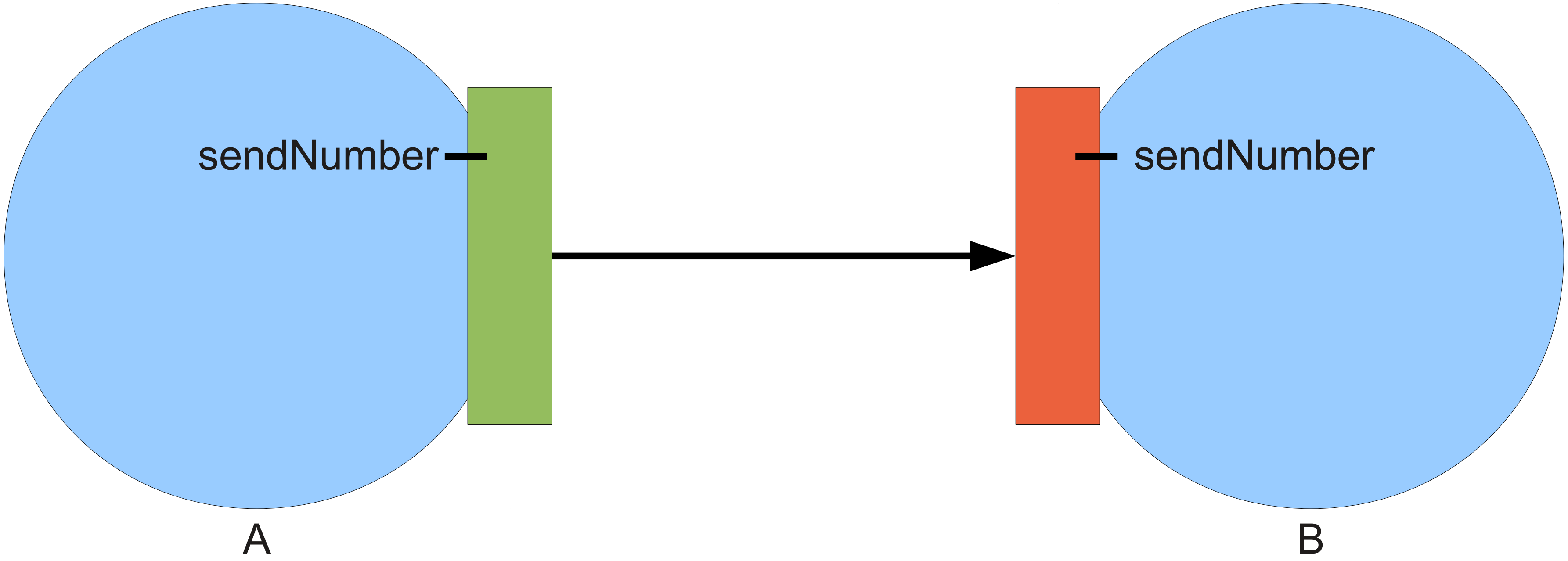
- **A** sends 5 to **B** through the sendNumber operation.

- We need to tell **A** how to reach **B**.
- We need to tell **B** how to expose sendNumber.
- In other words, how they can **communicate**!

# Ports and interfaces: overview

- Services communicate through **ports**.
- **Ports** give access to an **interface** (similar to Java interfaces?).
- An **interface** is a set of **operations** (similar to Java methods?).
- An **output port** is used to invoke **interfaces** exposed by other services.
- An **input port** is used to expose an **interface**.

- Example: a client has an **output port** connected to an **input port** of a calculator.



# Our first service-oriented application

interface.iol

```
interface MyInterface {
OneWay:
    sendNumber(int)
}
```

A.ol

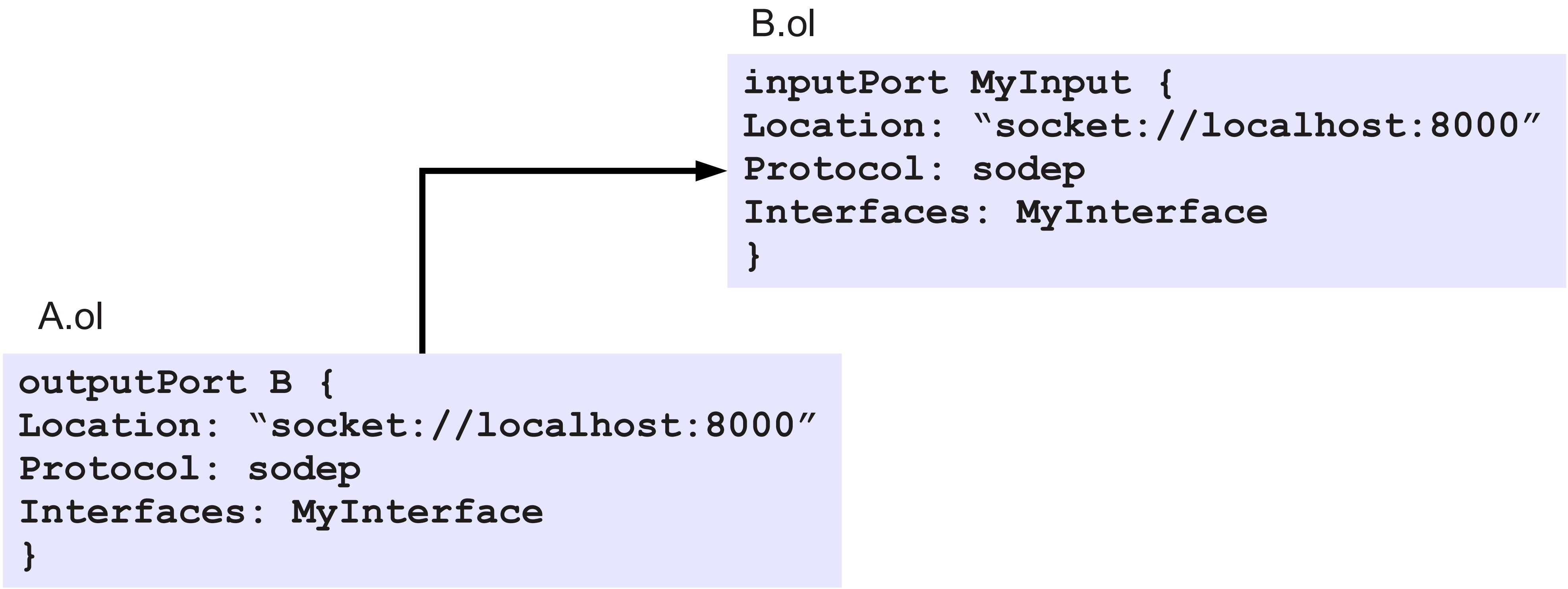
```
include “interface.iol”

outputPort B {
Location:
    “socket://localhost:8000”
Protocol: sodep
Interfaces: MyInterface
}

main
{
    sendNumber@B( 5 )
}
```

B.ol

```
include “interface.iol”

inputPort MyInput {
Location:
    “socket://localhost:8000”
Protocol: sodep
Interfaces: MyInterface
}

main
{
    sendNumber( x )
}
```

# Anatomy of a port

- A port specifies:
  - the **location** on which the communication can take place;
  - the **protocol** to use for encoding/decoding data;
  - the **interfaces** it exposes.
- There is no limit to how many ports a service can use.



B.ol

```
inputPort MyInput {
Location: “socket://localhost:8000”
Protocol: sodep
Interfaces: MyInterface
}
```

A.ol

```
outputPort B {
Location: “socket://localhost:8000”
Protocol: sodep
Interfaces: MyInterface
}
```

# Anatomy of a port: location

- A location is a URI (Uniform Resource Identifier) describing:
  - the **communication medium** to use;
  - the parameters for the communication medium to work.

- Some examples:
  - TCP/IP: `socket://www.google.com:80/`
  - Bluetooth: `btl2cap://localhost:3B9FA89520078C303355AAA694238F07;name=Vision;encrypt=false;authenticate=false`
  - Unix sockets: `localsocket:/tmp/mysocket.socket`
  - Java RMI: `rmi://myrmiurl.com/MyService`

# Anatomy of a port: protocol

- A protocol is a name, optionally equipped with configuration parameters.
- Some examples: sodep, soap, http, xmlrpc, …

```
Protocol: sodep

Protocol: soap

Protocol: http { .debug = true }
```

# Deployment and Behaviour

- A JOLIE program is composed by two definitions:
  - **deployment**: defines how to execute the behaviour and how to interact with the rest of the system;
  - **behaviour**: defines the workflow the service will execute.

```
// B.ol


include “interface.iol”

inputPort MyInput {
Location: “socket://localhost:8000”
Protocol: sodep
Interfaces: MyInterface
}
```

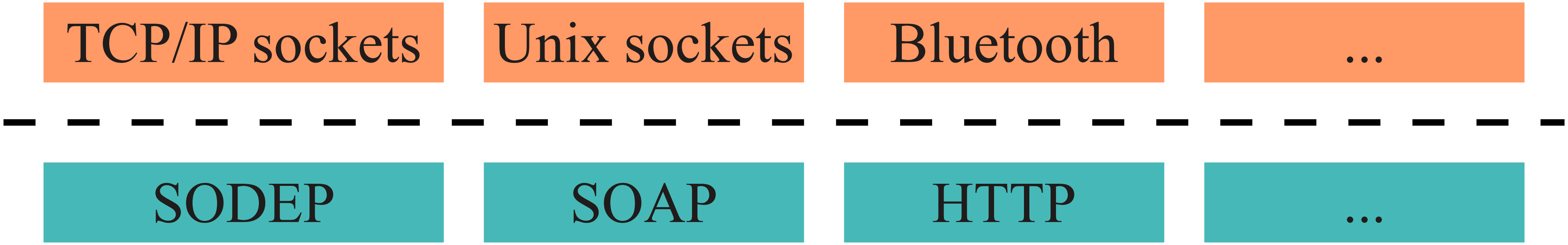
Deployment

```
main
{
    sendNumber( x )
}
```

Behaviour

# Communication abstraction

- Jolie supports many different communication mediums and data protocols.



TCP/IP sockets

Unix sockets

Bluetooth

...

SODEP

SOAP

HTTP

...

- A program just needs its port definitions to be changed in order to support different communication technologies!

# Operation types

- JOLIE supports two types of operations:
  - One-Way: receives a message;
  - Request-Response: receives a message and sends a response back.

- In our example, **`sendNumber`** was a One-Way operation.

- Syntax for Request-Response:

```
interface MyInterface {
RequestResponse:
    sayHello(string)(string)
}
```

```
sayHello@B( “John” )( result )
```

```
sayHello( name )( result ) {
    result = “Hello “ + name
}
```

# Behaviour basics

- Statements can be composed in sequences with the ; operator.
- We refer to a block of code as **`B`**

- Some basic statements:

  - assignment: **`x = x + 1`**
  - if-then-else: **`if ( x > 0 ) { B } else { B }`**
  - while: **`while ( x < 1 ) { B }`**
  - for cycle: **`for ( i = 0, i < x, i++ ) { B }`**

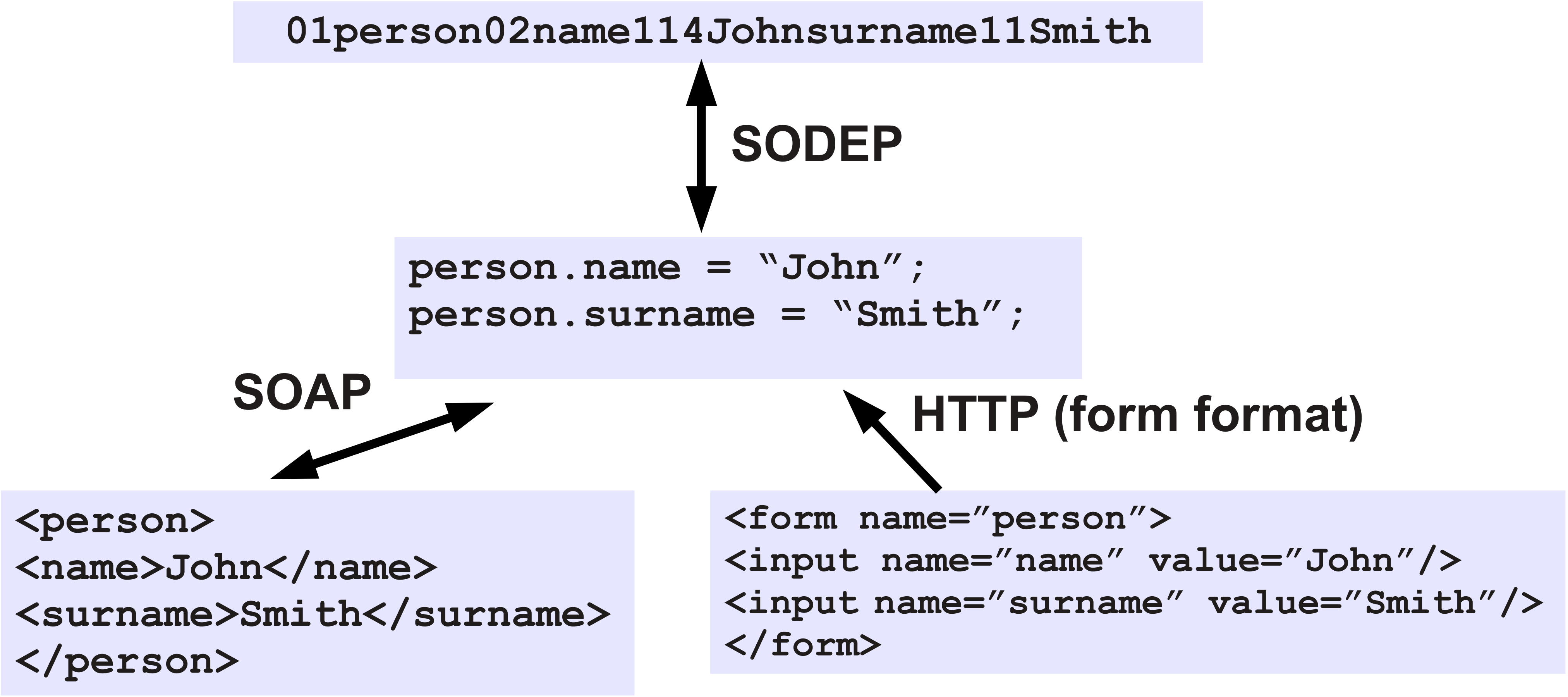
# Data manipulation (1)

- In JOLIE, every variable is a tree:

```
person.name = “John”;
person.surname = “Smith”
```

- Every tree node can be an array:

```
person.nicknames[0] = “Johnz”;
person.nicknames[1] = “Jo”
```



```
01person02name114Johnsurname11Smith
```

SODEP

```
person.name = “John”;
person.surname = “Smith”;
```

SOAP

```
<person>
<name>John</name>
<surname>Smith</surname>
</person>
```

HTTP (form format)

```
<form name=”person”>
<input name=”name” value=”John”/>
<input name=”surname” value=”Smith”/>
</form>
```

# Data manipulation (2)

- You can dump the structure of a node using the standard library.

```
include “console.iol”
include “string_utils.iol”

main
{
    team.person[0].name = “John”;
    team.person[0].age = 30;
    team.person[1].name = “Jimmy”;
    team.person[1].age = 24;

    team.sponsor = “Nike”;
    team.ranking = 3;

    valueToPrettyString@StringUtils( team )( result );
    println@Console( result )()
}
```

# Data manipulation: some operators

- Deep copy: copies an entire tree onto a node.
    - `team.person[2] << john`
- Cardinality: returns the length of an array.
    - `size = #team.person`
- Aliasing: creates an alias towards a tree.
    - `myPlayer -> team.person[my_player_index]`

```
for( i = 0, i < #team.person, i++ ) {
    println@Console( team.person[i].name )()
}
```

# Dynamic path evaluation

- Also known as associative arrays.
- Static variable path: `person.name`
- One can use an expression in round parenthesis when writing a path in a data tree. **Dynamic path evaluation.**
- Example:

  - We make a map of cities indexed by their names:
    - `cityName = “Copenhagen”;`
    - `cities.(cityName).state = “Denmark”`
  - Note that:

    `cities.(“Copenhagen”)`
  - is the same as:

    `cities.Copenhagen`
  - can be browsed with the foreach statement:

```
foreach( city : cities ) {
    println@Console( cities.(city).state )()
}
```

# Data manipulation: question

- What will be printed to screen?

```
include “console.iol”
include “string_utils.iol”

main
{
    cities[0] = “Copenhagen”;
    i = 0;
    while( i < #cities ) {
        println@Console( cities[i] )();
        cities[i] = “Copenhagen”;
        i++
    }
}
```

# Data types

- In an **interface**, each **operation** must be coupled to its **message types**.
- Types are defined in the deployment part of the language.
- Syntax:
  - **type** *name*:**basic_type** { subtypes }
- Where **basic_type** can be:
  - **int**, **long**, **double** for numbers
  - **string** for strings;
  - **raw** for byte arrays;
  - **void** for empty nodes;
  - **any** for any possible basic value;
  - **undefined**: makes the type accepting any value and any subtree.

```
type Team:void {
    .person:void {
        .name:string
        .age:int
    }
    .sponsor:string
    .ranking:int
}
```

# Casting and runtime basic type checking

- For each basic data type, there is a corresponding primitive for:
  - casting, e.g. **`x = int( s )`**
  - runtime checking, e.g. **`x = is_int( y )`**

# Data types: cardinalities

- Each node in a type can be coupled with a **range** of possible occurences.
- Syntax:
    - **type** *name*[min,max]**:basic_type** { subtypes }
- One can also have:
    - `*` for any number of occurences (>= 0);
    - `?` for [0,1].

```
type Team:void {
    .person[1,5]:void {
        .name:string
        .age:int
    }
    .sponsor:string
    .ranking:int
}
```

# Data types and operations

- Data types are to be associated to operations.

```
type SumRequest:void {
    .x:int
    .y:int
}

interface CalculatorInterface {
RequestResponse:
    sum( SumRequest )( int )
}
```

# Parallel and input choice

- Parallel composition: **`B | B`**

```
sendNumber@B( 5 ) | sendNumber@C( 7 )
```

- Input choice:

```
[ ok( message ) ] { P1 }

[ shutdown() ] { P2 }

[ printAndShutdown( text )() {
    println@Console( text )()
} ] { P3 }
```

```
run = 1;
while( run ) {
    [ print( message ) ] {
        println@Console( text )()
    }
    [ shutdown() ] { run = 0 }
}
```

# A calculator service

```
type SumRequest:void {
    .x:int
    .y:int
}

interface CalculatorInterface {
RequestResponse:
    sum(SumRequest)(int)
}

inputPort MyInput {
Location: “socket://localhost:8000/”
Protocol: sodep
Interfaces: CalculatorInterface
}

main
{
    sum( request )( response ) {
        response = request.x + request.y
    }
}
```

# Dynamic binding

- In an SOA, a fundamental mechanism is that of *service discovery*.
- A service dynamically (at runtime) discovers the location and a protocol for communicating with another service.
- In JOLIE we obtain this by manipulating an output port as a variable.

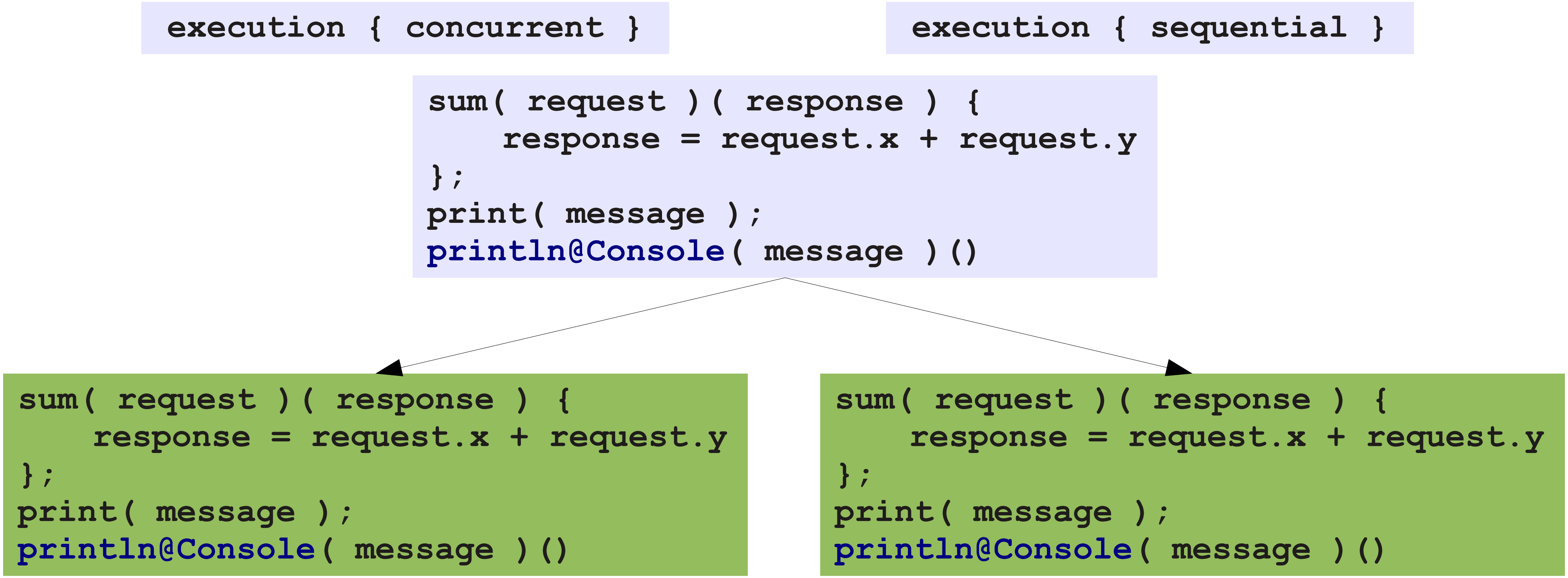
```
outputPort Calculator {
Interfaces: CalculatorInterface
}

main
{
    Calculator.location = “socket://localhost:8000/”;
    Calculator.protocol = “sodep”;
    request.x = 2;
    request.y = 3;
    sum@Calculator( request )( result )
}
```

- Type for bindings defined in

  `$JOLIE_DIR/include/types/Binding.iol`

# Multiple executions: sessions

- The calculator works, but it terminates after executing once.
- We want it to keep going and accept other requests.
- We introduce **sessions**.
- A session is an **execution instance** of a service **behaviour**.
- In JOLIE, sessions can be executed **concurrently** or **sequentially**.



```
execution { concurrent }
```

```
execution { sequential }
```

```
sum( request )( response ) {
    response = request.x + request.y
};
print( message );
println@Console( message )()
```

```
sum( request )( response ) {
    response = request.x + request.y
};
print( message );
println@Console( message )()
```

```
sum( request )( response ) {
    response = request.x + request.y
};
print( message );
println@Console( message )()
```

# More

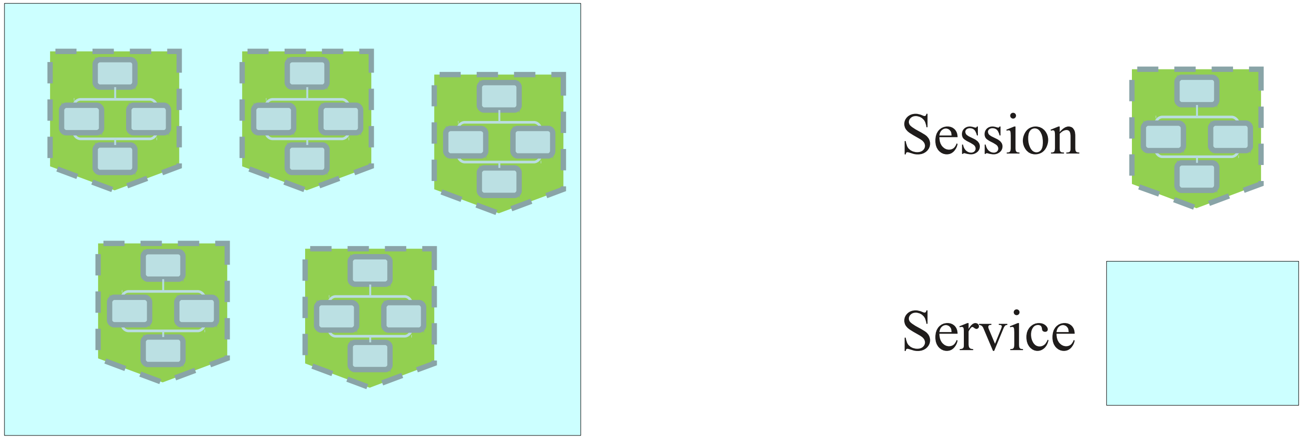
- A service may engage in different **separate conversations** with other parties.
  - Example: a chat server may manage different chat rooms.

- Each conversation needs to be supported by a private execution state.
  - Example: each chat room needs to keep track of the posted messages.

- We call this support **session**.

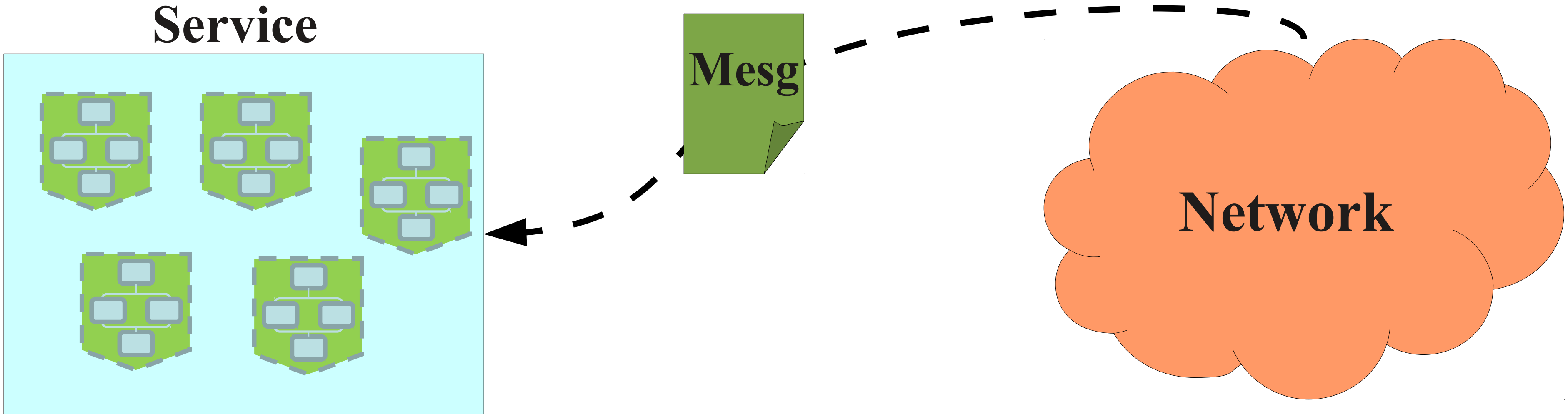
- Sessions are independent of each other: they run in parallel.
  - Some call them **threads** equipped with a **private state**.

- Therefore, a service has many parallel sessions running inside of it:



# Message routing

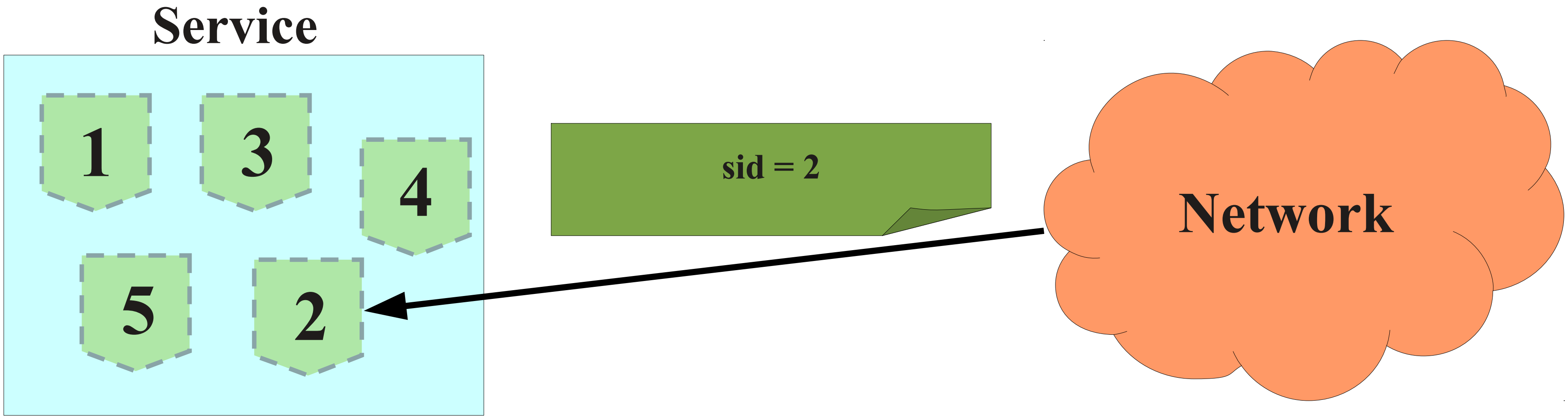
- What happens when a service receives a message from the network?
- We need to assign the message to a session!



- How can we establish which session the message is meant for?

# Session identifiers

- A widely used mechanism for routing messages to sessions.
- Each session has a **session identifier** (sid).
- All received messages contain an sid.
- The service gives the message to the session with the same sid.

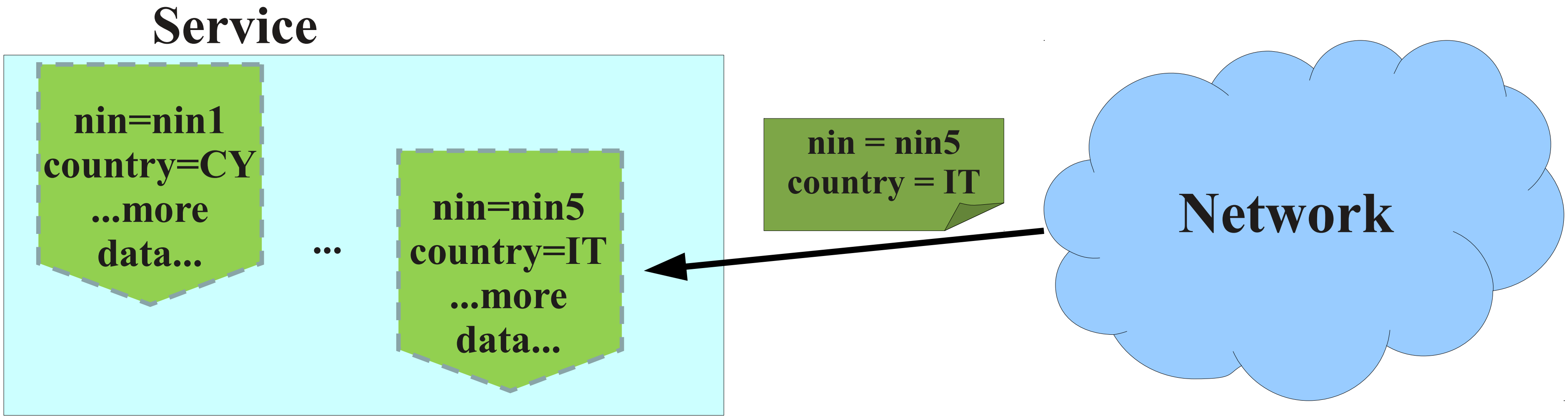


# Correlation sets

- A *generalisation* of session identifiers.

- A session is identified by the **values** of some of its variables.
  - These variables form a **correlation set** (or **cset**).
  - Similar to unique keys in relational databases.

- Example:
  - in a service where we have a session for every person in the world a correlation set could be formed by the national identification number and the country.



# Session identifiers VS correlation sets

*Session identifiers*

- Pros
  - Usually handled by the middleware: hard to make mistakes.
- Cons
  - All clients must send the sid as expected: no support for integration.

*Correlation sets*

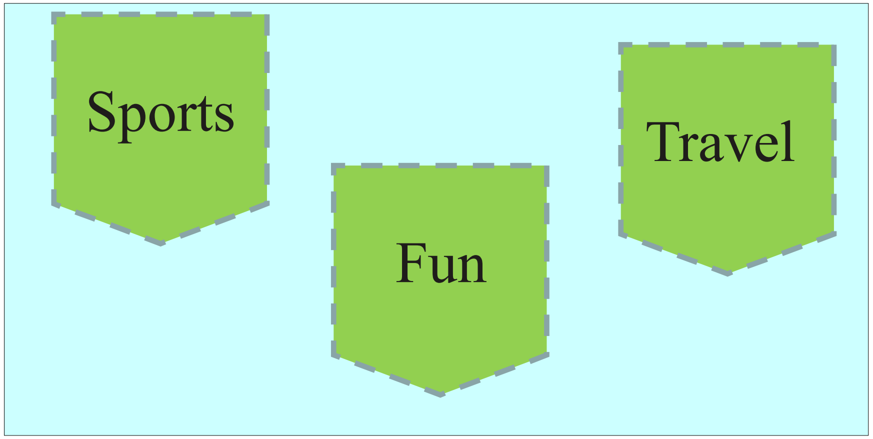
- Pros
  - Programmability of correlation can be used for **integration**.
  - Each cset is a different way of identifying a session: support for **multiparty interactions**.
- Cons
  - Almost totally controlled by the programmer: easy to make mistakes.

# Example: chat service

- We model a chat service handling separate chat rooms. Each room is a session.

```
interface ChatInterface {
RequestResponse:
    openRoom(OpenRequest)(OpenResponse)
OneWay:
    publish(PublishMesg),
    close(CloseMesg)
}
```

**Chat service**



Session starter

```
main
{
    openRoom( openRequest )( response ) {
        // Create the chat room...
    }; run = true;
    while ( run ) {
        [ publish( message ) ] { println@Console( message.content )() }
        [ close( closeRequest ) ] { run = false }
    }
}
```

# Correlating chats

- We want:
  - to publish messages in the right rooms; 1
  - to let the room creator close it, but only her! 2

- So we create two correlation sets:

```
interface ChatInterface {
RequestResponse: openRoom(OpenRequest)(OpenResponse)
OneWay: publish(PublishMesg), close(CloseMesg)
}

cset { name: OpenRequest.room PublishMesg.roomName }  1
cset { adminToken: CloseMesg.adminToken }  2
```

```
main
{
    openRoom( openRequest )( csets.adminToken ) {
        csets.adminToken = new
    }; run = true;
    while ( run ) {
        [ publish( message ) ] { println@Console( message.content )() }
        [ close( closeRequest ) ] { run = false }
    }
}
```

Fresh value generator

# Exercise (together)

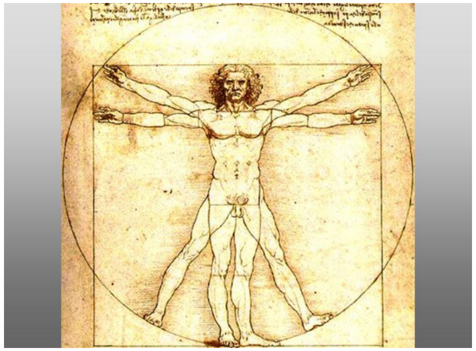
- We design an SOA for handling exams between students and professors.
- A student can start an examination session.
- A professor can ask a question in the session.
- The student answers and the professor can either accept or reject.
- The student is notified.

- **Questions**

- **Architecture: roles and services.**
  - What are the involved services? **Roles.**
  - Who controls the execution flow? **Orchestrator.**
- **Work flow: operations, data types and activity composition.**
  - Who starts the session?
  - How does the session behave?

# Some other things you can do with Jolie

# Leonardo

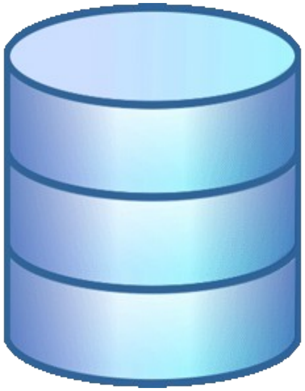


- A web server in pure Jolie.
- Can fit in a slide.

  (ok, I reduced the font size a little)
- ~50 LOCs

```
include "console.iol"
include "file.iol"
include "string_utils.iol"
include "config.iol"

execution { concurrent }

interface HTTPInterface {
RequestResponse:
        default(undefined)(undefined)
}

inputPort HTTPInput {
Protocol: http {
        .debug = DebugHttp; .debug.showContent = DebugHttpContent;
        .format -> format; .contentType -> mime;
        .default = "default"
}
Location: Location_Leonardo
Interfaces: HTTPInterface
}

init {
        documentRootDirectory = args[0]
}

main {
        default( request )( response ) {
                scope( s ) {
                        install(
                                FileNotFound =>
                                println@Console( "File not found: " + file.filename )()
                        );
                        s = request.operation;
                        s.regex = "\\?";
                        split@StringUtils( s )( s );
                        file.filename = documentRootDirectory + s.result[0];
                        getMimeType@File( file.filename )( mime );
                        mime.regex = "/";
                        split@StringUtils( mime )( s );
                        if ( s.result[0] == "text" ) {
                                file.format = "text";
                                format = "html"
                        } else {
                                file.format = format = "binary"
                        };
                        readFile@File( file )( response )
                }
        }
}
```

# Jolie and DBMS



| id | name | surname |
|---|---|---|
| 1 | John | Smith |
| 2 | Donald | Duck |

```
query@Database
   ( “select * from people” )( result );
print@Console( result.row[1].surname )() // “Duck”
```

- Equipped with protection from SQL injection.

# Jolie and Java

```
public class StringUtils
    extends JavaService
{
    public String trim( String s )
    {
        return s.trim();
    }
}
```

```
include “string_utils.iol”5

main
{
    trim@StringUtils
        ( “ Hello “ )( s )
    // now s is “Hello”
}
```

# Also...

- Jolie is based on the service-oriented programming paradigm, but it is a **general purpose programming language**.

- You can use it even for controlling a media player (ECHOES), or the brightness level of your Apple keyboard (Jabuka).

- Lots of other applications... ask about them!

# Ideas for student projects

# Support for new communication technologies

- Jolie can be easily extended to support new communication means.
  Some examples...

- For open source enthusiasts.
  - D-Bus. It is used by Skype and all major interoperable programs in many *nix operating systems. Example of application: make a simple Jolie program that reads Skype chats for you, or speaks what you write during a call! Or, make automated voice replies!
    Or, exploit existing PDF viewers for implementing a remote presentation system!
  - The KDE project uses a library inspired by Jolie for implementing communication between desktop components: we could build upon it to make KDE programmable directly from Jolie!

- For other enthusiasts.
  - You can integrate Jolie with your preferred wireless technology to access your house appliances. Control your house lights!

# Language developments

- Jolie is looking for a **static type system** for variables. Right now, we only have runtime type checking when we send or receive messages.
  This can be tedious!

- Jolie is pretty fast already, but it could be faster!
  Sometimes we copy data when we would not need it.
  With a simple **static analysis for self-invocations** we could speed up some programming patterns by a factor of 10 or more!

- We need some **syntactic shortcuts** and new **language primitives** for making manipulating data structures easier.
  For instance, I would like to be able to write stuff such as:

```
sum@Calculator( { .x = 5, .y = 6 } )( result )
```

# Behavioural compatibility

- Look at these two programs:

Server

```
login( user );
readNewspaper()( newspaper );
logout( user )
```

Client

```
readNewspaper@Server()( x );
login( me )
```

- They are **incompatible!!** The order of interactions is wrong.
- This is very hard to see in other languages (even impossible in many cases).
- In Jolie, it becomes **evident**.
- So: we could build a static analysis that checks these errors!
- Technically, this would be based on **Multiparty Session Types**.

# Software Distribution Architecture

- Installing Jolie and distributing software made with it is still not ideal...

- ...but we could use Jolie to code a software distribution architecture that takes care of automatic updating, notifications, and distribution.

- This work would empower the new (coming) Jolie website.

- ...and also be the standard distribution platform for Jolie software.

# Crash handling

- What happens when a Jolie program **crashes**?
    - Messages are lost
    - Execution state must be recovered “by hand”
    - Potentially very inconvenient when many sessions are open.

- We need to develop a mechanism for **graceful crash handling**, similarly to DBMS.

# Application server and reverse proxy

- Jolie can already be used as an application server for multiple services.
- We need to enhance this to be configurable and have a GUI.
- Ultimately, we want a Jolie-based software for easily handling a collection of web applications and service-oriented software in general!
- Example: make a Jolie program, upload it, and run it on the application server.
- Support cloud computing!

# Joliepse, the Jolie IDE

- Joliepse is a prototype of a Jolie IDE based on Eclipse.
- We need to integrate it with existing tools for documentation (joliedoc), testing (joliedummy), and Java-Jolie interaction (jolie2java).
- This would make the development of complex Jolie software much faster!

# Survey - Comparison with BPEL

- WS-BPEL is the current reference language for service-oriented computing composition.

- How does it differ from Jolie?

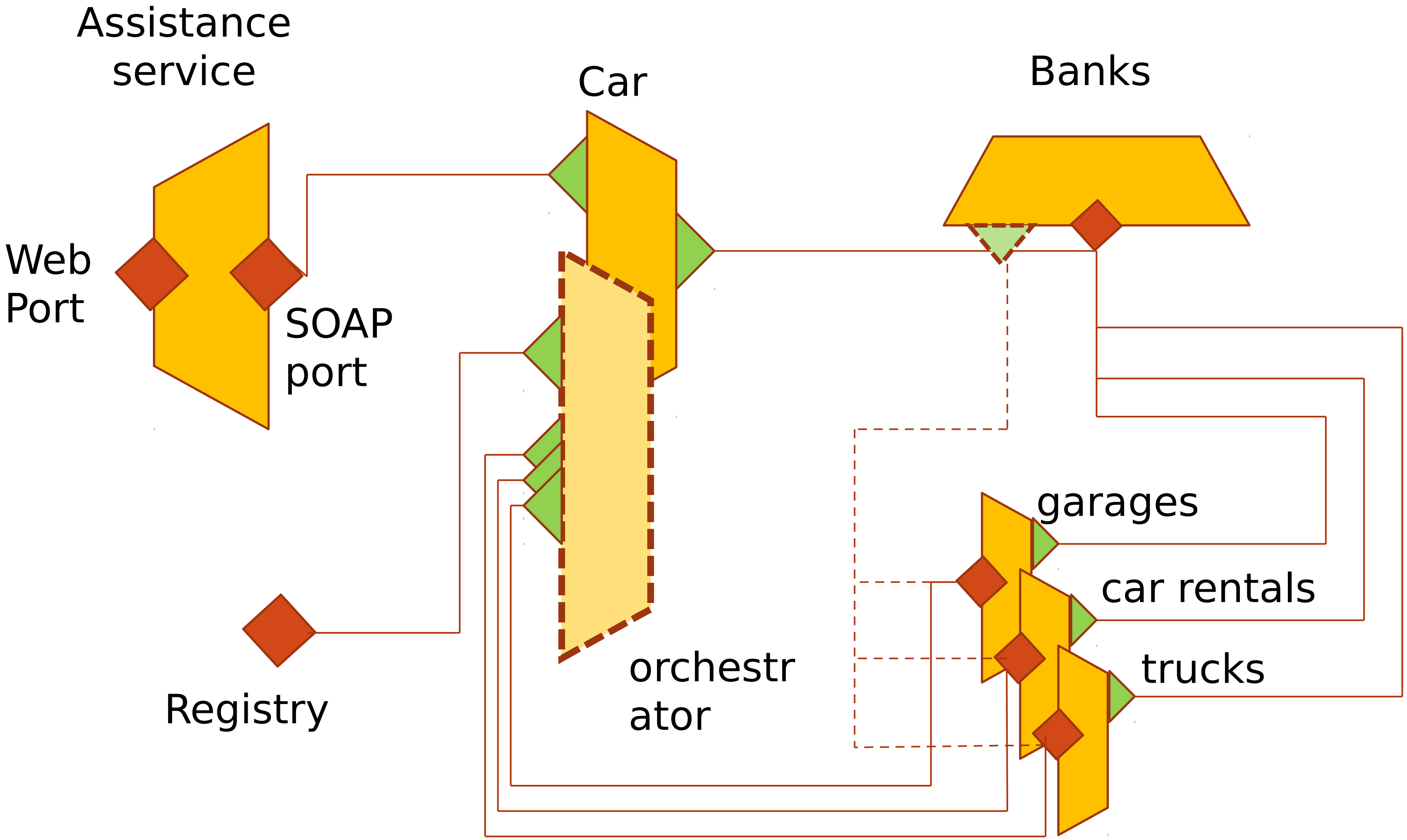
- How does programming services in Jolie differ from other languages?

- We need a survey of the main differences between different technologies.

- Preliminary (sketchy) results: a Jolie program usually reduces a BPEL program by an order of magnitude.

# Architectural graphical designer - SOA Circuits

- We need a graphical designer for Service-Oriented Architectures!
- Also called SOA circuits, they describe the connections between services.
- Example: automotive scenario for handling car engine failures:



# Testing suite

- We need a tool for automatically generating **tests** for a Jolie program.
- A tester would invoke the operations of the Jolie program and receive its outputs.
- Similar to JUnit, but for services!

# Many other possibilities...

- So come and talk to us if you are curious or have some idea of your own!