Jolie

A service-oriented programming language
Introduction and Basic Ideas

Paolo Baldan

Linguaggi e Modelli per il Global Computing
AA 2015/2016

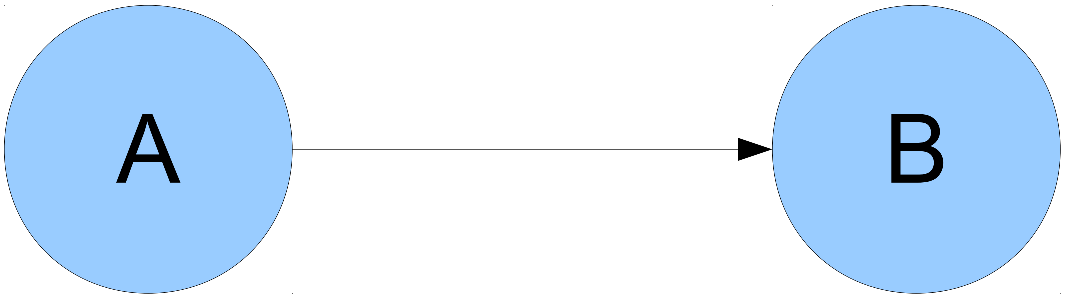
Many thanks to Fabrizio Montesi
(IT University of Copenhagen)
for the material

Programming distributed systems is hard

- Programming concurrent distributed systems is usually harder than programming non-concurrent and non-distributed ones.

- Some problems are:
  - handling concurrency
  - handling communications;
  - handling faults;
  - handling heterogeneity;
  - handling the evolution of systems.

Programming distributed systems is hard - Communications



- The basic feature for any distributed system.
- A language can access the lower level IPC facilities. E.g. in Java we can open a TCP/IP socket and send some data:

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("http://someurl.com", 80));

Buffer buffer = . . .; // Create a byte buffer with data to be sent.

while( buffer.hasRemaining() ) {
    channel.write( buffer );
}
```

Programming distributed systems is hard - Communications

- That is NOT good Java code.
- We need to remember to:
  - handle eventual exceptions;
  - remember to close the channel.
- New version (and this is actually still not perfect, but better):

```
SocketChannel socketChannel = SocketChannel.open();
try {
  socketChannel.connect(new InetSocketAddress("http://someurl.com", 80));
  Buffer buffer = . . .; // Create a byte buffer with data to be sent.

  while( buffer.hasRemaining() ) {
    channel.write( buffer );
  }
}
catch( UnresolvedAddressException e ) { . . . }
catch( SecurityException e ) { . . . }
/* . . . many catches later . . . */
catch( IOException e ) { . . . }
finally { channel.close(); }
```

Programming distributed systems is hard - Faults

- Applications in a distributed system can perform a *distributed transaction*.

- Example:
  - a client asks a store to buy some music;
  - the store opens a request for handling a payment on a bank;
  - the client sends his credentials to the bank for closing the payment;
  - the store sends the goods to the client.

- Looks good, but a lot of things may go **wrong**, for instance:
  - the store (or the bank) could be offline;
  - the client may not have enough money in his bank account;
  - the store may encounter a problem in sending the goods.

Programming distributed systems is hard - Heterogeneity

- In the real world, distributed systems can be heterogeneous.

- Different applications that are part of the same system could...
  - use different **communication mediums** (Bluetooth? TCP/IP?, …);
  - use different **data protocols** (HTTP? SOAP? X11?);
  - use different **versions** of the same data protocol (SOAP 1.1? 1.2?);
  - and so on...

Programming distributed systems is hard - Evolution

- Distributed systems usually *evolve over time*.

- Each application could be made by a different company.

- A company may update its application.

- Again, many possible pitfalls:
  - the updated version may use a new data protocol, unsupported by the clients;
  - the updated version may have a different interface, e.g. first it took an integer as a parameter for a functionality, now a string;
  - the updated version may have a different behaviour, e.g. first it did not require clients to log in, now it does.

Simplifying distributed systems

- Things can be made easier by hiding the low-level details.
- Two main approaches:
  - make a library/tool/framework for an existing programming language (es. Java RMI);
  - make a new programming language.

(Micro) Service-Oriented Computing (SOC)

- A design paradigm for **distributed systems**.
- A **service-oriented** system is a network of **services**.
- Services communicate through **message passing**.
- Messages are tagged with **operations** (similar to method names in OO).
- Services are typed with **interfaces**, which define **operations** and **message data types** for operations.
- Reference technology: Web Services.
  - Based on XML;
  - WS-BPEL (BPEL for short) for programming composition.

Why SOC? A few other reasons...

- Everybody was using custom solutions for distributed computing.

- We need more **integration** between existing software.
  - Programs using different data protocols cannot interact.

- We need support for more **dynamicity**.
  - **Service Discovery:** we can discover where services are located at runtime.

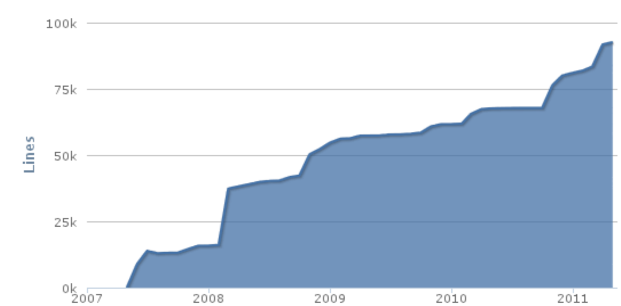
- We need support for **structured interactions (protocols** exposed**)**.
  - Many web applications implement logical orderings between actions.
  - Example: in a newspaper web portal, a user may need to log in *before* reading the news.

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

- Nice logo:

FOCUS Research Team

- *Formal foundations* from the Academia.
- Tested and used in the *real world*:
- *Open source* (`http://www.jolie-lang.org/`), with a well-maintained code base:



Jolie: comes with formal syntax and semantics

(In) $s \xrightarrow{s} \mathbf{0}$ (Out) $\bar{s} \xrightarrow{\bar{s}} \mathbf{0}$ (One-WayOut) $\bar{\omega}@z(\boldsymbol{x}) \xrightarrow{\bar{\omega}@l/z(\boldsymbol{v}/\boldsymbol{x})} \mathbf{0}$ (One-WayOutLoc) $\bar{\omega}@l(\boldsymbol{x}) \xrightarrow{\bar{\omega}@l(\boldsymbol{v}/\boldsymbol{x})} \mathbf{0}$ (One-WayIn) $\omega(\boldsymbol{x}) \xrightarrow{\omega(\boldsymbol{v}/\boldsymbol{x})} \mathbf{0}$

(Assign) $x := e \xrightarrow{x:=v/e} \mathbf{0}$ (Req-Out) $\overline{o_r}@z(\boldsymbol{x},\boldsymbol{y}) \xrightarrow{\overline{o_r}@l/z(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y})} o_r(\boldsymbol{y})$ (Req-OutLoc) $\overline{o_r}@l(\boldsymbol{x},\boldsymbol{y}) \xrightarrow{\overline{o_r}@l(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y})} o_r(\boldsymbol{y})$

(Req-In) $o_r(\boldsymbol{x},\boldsymbol{y},P) \xrightarrow{o_r(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y},P)@l} P; \bar{o}_r@l(\boldsymbol{y})$ (If then) $\chi?P : Q \xrightarrow{\chi?} P$ (Else) $\chi?P : Q \xrightarrow{\neg\chi?} Q$

(Iteration) $\chi \rightleftharpoons P \xrightarrow{\chi?} P; \chi \rightleftharpoons P$ (Not Iteration) $\chi \rightleftharpoons P \xrightarrow{\neg\chi?} \mathbf{0}$ (Synchro) $\dfrac{P \xrightarrow{s} P', Q \xrightarrow{\bar{s}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$

(Sequence) $\dfrac{P \xrightarrow{\gamma} P'}{P; Q \xrightarrow{\gamma} P'; Q}$ (Parallel) $\dfrac{P \xrightarrow{\gamma} P'}{P \mid Q \xrightarrow{\gamma} P' \mid Q}$ (Choice) $\dfrac{\epsilon_i \xrightarrow{\gamma} \mathbf{0} \quad i \in I}{\sum^{+}_{i \in I} \epsilon_i; P_i \xrightarrow{\gamma} P_i}$

Structural Congruence

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv \mathbf{0} \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \mathbf{0}; P \equiv P$$

Table 1. Rules for service behaviour lts layer

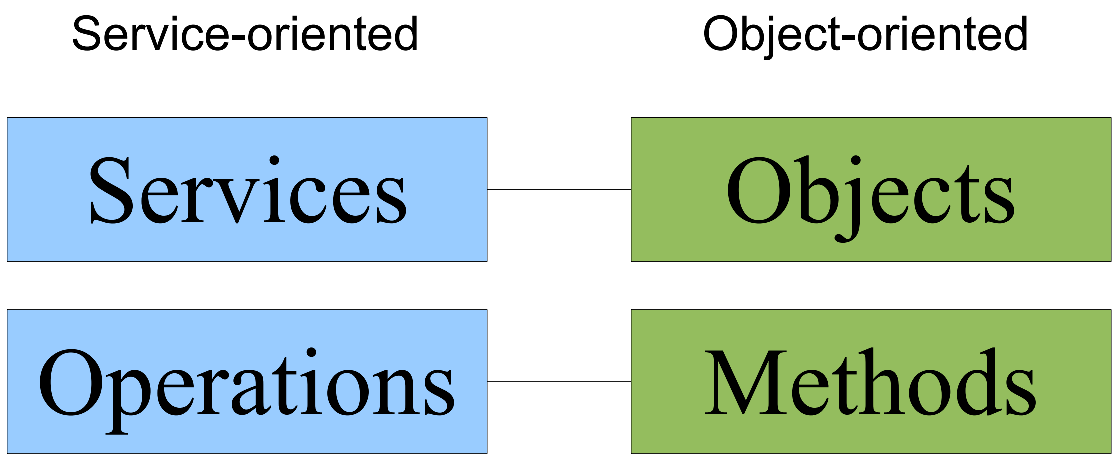
Hello, Jolie!

- 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**.
- Reminds of Object-oriented programming:



Understanding Hello World: concepts

Include from standard library

```
include "console.iol"

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

Program entry point

Operation

The service I want to invoke

Our first service-oriented application

- A program (service) defines the input/output communications it will make (operations and invocations of operations)

A

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

B

```
main
{
   number( x );
   operate on x …

}
```

- **A** sends 5 to **B** through the n**`umber`** operation (invokes the **`number`** operation of service **B)**.

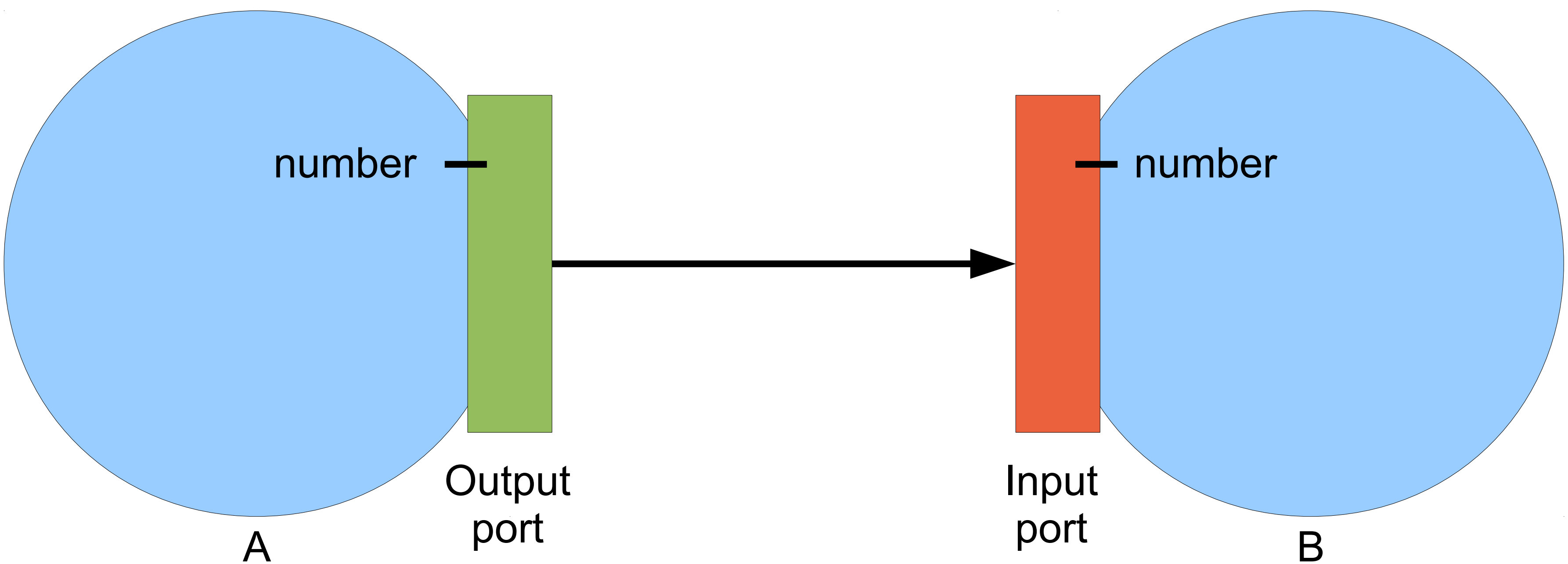
- We need to tell **B** how to expose operation **`number`**
- We need to tell **A** how to reach **B**
- In other words, how they can **communicate**!

Ports and interfaces: overview

- Services communicate through **ports**.
- **Ports** give access to **interfaces**
- An **interface** is a set of **operations** (~ methods)

- An **input port** is used to expose an **interface**.
- An **output port** is used to invoke **interfaces** exposed by other services.

- Example: A has an **output port** connected to the **input port** of B.



Our first service-oriented application

AB

interface.iol

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

A.ol

```
include "interface.iol"

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

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

B.ol

```
include "interface.iol"

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

main
{
    number( 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.
- A service can use several ports

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 }

Protocol:  https

Protocol:  xmlrpc
```

Deployment and Behaviour

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

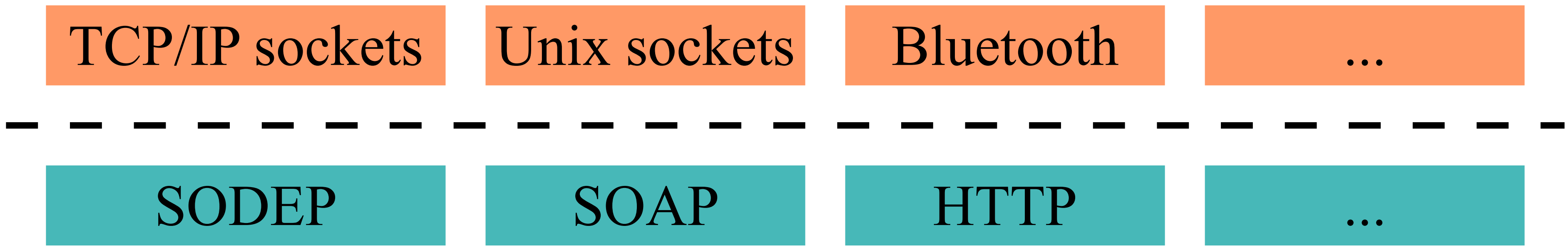
```
// B.ol

include "interface.iol"

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

main
{
    number( x ) { … }
}
```

Deployment

Behaviour

Communication abstraction

- Jolie supports many different communication mediums and data protocols.



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

- **Adapters** as services which does nothing else that receiving and forwarding from/to ports with different protocols (no behaviour)

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, **`number`** was a One-Way operation.

- Syntax for Request-Response:

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

helloWho/base

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

```
sayHello@B( "John" )( result )
```

```
inputPort B {
Location:"socket://localhost:8000"
Protocol: sodep
Interfaces: MyInterface
}
```

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

Behaviour basics

- Interpreted, dynamically typed (run-time checking)
- 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 }`
- **Statements** can be composed in sequences with the **“;”** operator.

Stateful server

- the server can be persistent and stateful

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

HelloWho/base

```
main
{
    count = 0
    while ( true ) {
      sayHello( who  )( greet ) {
        greet = "Hello " + who + "! ";
        if ( count != 0 ) {
          greet = greet + "You came "
                  + count + " times"
        };
      count++
    }
}
```

- … we will see some better solution, later

Parallel Composition

- Two blocks B1 and B2 can be executed in parallel with the syntax

```
B1 | B2
```

- Example

```
{ sendNumber@B( 5 ) |
  sendNumber@C( 7 )
};
println@Console( "both operations finished" )();
```

Input choice

```
[ Input_Statement1 ]
  { P1 }

[ Input_Statement2 ]
  { P2 }
    ...
    ...
[ Input_Statementn ]
  { Pn }
```

HelloWho/shutdown

```
run = true;
count = 0;
while( run ) {
     [ sayHello( who )( greet )
         {
           greet = "Hello " + who + "! ";
           if ( count != 0 ) {
             greet = greet + "You came "
                     + count + " times"
            };
            count++
     }]
     {
         println@Console( "Answered  " + who )()
     }

     [ shutdown() ] {
         run = false
         println@Console( "Shutdown by  " + who )()
     }
}
```

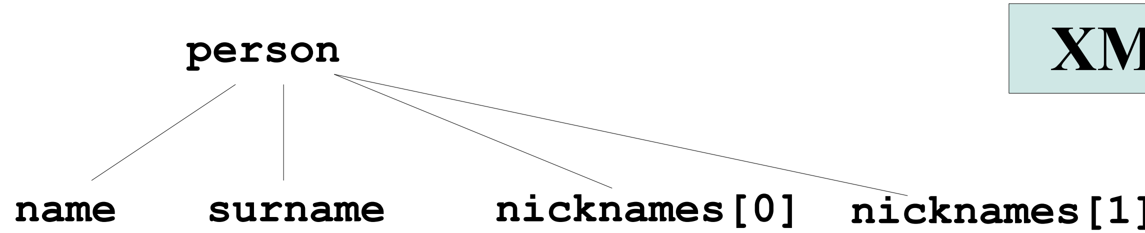
Data manipulation (1)

- 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"
```



XML-like

Data manipulation (2)

```
01person02name114Johnsurname11Smith
```

SODEP

```
person.name = "John";
person.surname = "Smith";
```

XML-like

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 (3)

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

prettyString

```
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 )()
}
```

- `team.person` same as `team.person[0]`

Data manipulation: some operators

- Deep copy: copies an entire tree onto a node.
    - `team.person[2] << john`

- Cardinality: returns the length of the array associated to a node
    - `size = #team.person`

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

- Undefining: a variable or a subtree
    - `undef(team)`

Data manipulation: some operators

- Aliasing: creates an alias towards a tree path.
  - `myPlayer -> team.person[my_player_index]`

Aliases are evaluated every time they are used (~ macro)

- With: avoding repetitive paths

DataStructures/with.ol

```
with ( a.b.c ){
    .d[ 0 ] = "zero";
    .d[ 1 ] = "one";
    .d[ 2 ] = "two";
    .d[ 3 ] = "three"
};
currentElement -> a.b.c.d[ i ];

for ( i = 0, i < #a.b.c.d, i++ ){
    println@Console( currentElement )()
}
```

Dynamic path evaluation (associative arrays, maps)

- **Static path**: `person.name`

- **Dynamic path**
  Expression in round parenthesis in a path of a data tree.
- 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 ) {
    // for all children of cities, bound to city
    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] )();
        i++;
        cities[i] = "Copenhagen"
    }
}
```

DataStructures/while.ol

Data types

- In an **interface**, each **operation** must be coupled to its **message types**.

- Syntax:
    - **`type`** *`name`*`:`**`basic_type`** `{ subnode types }`
- Where **basic_type** can be:
    - **`int`**, **`long`**, **`double`** for numbers
    - **`string`** for strings;
    - **`raw`** for byte arrays (internal use, data passing purposes);
    - **`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 istanceof int`

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
}
```

- With no indication, cardinality is defaulted to [1,1]

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 )
}
```

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
    }
}
```

A variadic calculator

```
type SumRequest:void {
    .val [1,*]:int
}

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

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

main
{
    sum( request )( response ) {
        response = 0;
        for ( i = 0, i < #request.val, i++ ) {
            response += request.val[i]
        }
    }
}
```

Summary

- Everything is a **service**
  - Exposing **interfaces** {operations with type} on **input ports**
  - Invoking operations of other **services**, **through output ports** → **network of services**

- When defining the behaviour of a **service** we can use

  - Standard control flow primitives (**if**, **while**, **for**)

  - **Sequential composition**
    "**;**"
  - **Parallel**
    { P } | { Q }
  - **Input choice**
    [op1(x)] { … }
    [op2(x)(y) { … }] { … }
    (~ select, wait for multiple operations,
    note the combination of parallel and sequential composition)

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 in **`Binding.iol`**

Dynamic binding

- Various calculators offering each an operation between **sum**, **product** and **exp** as an operation **`op`** at different addresses

- A service provider, that accept requests of sum/prod/exp and returns the binding for the corresponding calculator

- A client asking to a service provider the binding to the appropriate service and then using the operation

[see `serviceDiscovery`]

Fault Handling

Scope, Fault, Throw

- A **scope** is a behavioural container denoted by a name
- **Fault handlers** can be associated to a scope
- Faults thrown within a scope are
  - **Handled** by corresponding handler, if any
  - Passed to the parent scope, otherwise

```
…
scope ( scope_name ) {
  install (
    fault_name1 => handler code
     fault_name2 => handler code
  );
  …
  throw ( fault_name )
  }
…
```

- Uncaught faults in a **request-response** operation passed to the invoker

Scope, Fault, Throw

- Example: A calculator that only accepts positive numbers

```
fault_calculator
```

Termination and recovery

- A scope can be **terminated** by a **faulty parallel scope**
- A **termination handler** can be installed for managing the situation and bringing the activity to a safe state

```
…
scope ( scope_name ) {
  install (
    this => termination handler code
  );
  …

}

|    // parallel scope

scope ( sibling ) {
    …
    …
    throw ( fault_xy )
}
```

- If **`scope_name`** has children scopes with termination handlers, these are triggered before

Compensation

- When a scope sucessfully terminates, its termination recovery code is made accessible to the enclosing scope (as a **compensation**)
- It can be called (in a handler) with **`comp(sub_scope)`**

```
main
{
    install( a_fault =>
      println@Console( "Fault handler for a_fault" )();
      comp( example_scope ) // Access to recovery of subscope
    );

    scope( example_scope )
    {
      install( this =>
        println@Console( "recovering step" )()
    );
    …
    };
    throw( FaultName )
}
```

Compensation

- The current recovery handler can be referred to with **`cH,`** thus allowing an incremental construction of recovery code

terminationCompensation

```
main
{
    install( a_fault =>
      println@Console( "Fault handler for a_fault" )();
      comp( example_scope ) // Access to recovery of subscope
    );

    scope( example_scope )
    {
        …
        … some work …
      install( this =>
        println@Console( "recovering step 1" )());
        …
        … some work …
      install( this =>
        cH;
        println@Console( "recovering step 2" )());

    };
    throw( FaultName )
}
```

Architectural Composition

Embedding

- A mechanism for integrating multiple services in a single one

- Possibly written in different languages (not only Jolie, but also Java and Javascript currently supported)

```
embedded {
    Language : path [ in OutputPort ]
}
```

- The output port (if specified) is bound to the local input port of the embedded service

Embedding Javascript

- The JS service (`Calculator.js`)

```
importClass( java.lang.System );
importClass( java.lang.Integer );

function sum( request )
{
    var x = request.getFirstChild("x").intValue();
    var y = request.getFirstChild("y").intValue();
    System.out.println( "Got request for " + x + " and " + y );
    return Integer.parseInt(x + y); // JS represents any number as
                                    // ac64-bit floating point number.
}
```

Embedding Javascript

```
// port for connecting to the local port of the Javascript service
outputPort CalculatorJS {
Interfaces: CalculatorInterface
}

// embeds the JS service
embedded {
JavaScript:
    "Calculator.js" in CalculatorJS
}

// port for exposing the embedded service (verbatim of the JS service)
inputPort Service {
Location: "socket://localhost:8000"
Protocol: sodep
Interfaces: CalculatorInterface
}


main
{
    sum (request)(response) {
        sum@CalculatorJS(request)(response)
    }
}
```

Aggregation

- A service can expose operations which are delegated to other services

```
// external service
outputPort A {
    Location: "socket://urlA.com:80/"
    Protocol: soap
    Interfaces: InterfaceA
}

// external service
outputPort B {
    Location: "socket://urlB.com:80/"
    Protocol: xmlrpc
    Interfaces: InterfaceB
}

// Expose the services from A, B and a locally implemented one
inputPort Input {
    Location: "socket://url.com:8000/"
    Protocol: sodep
    Interfaces: MyInterface
    Aggregates: A, B
}

main {
    // implement the operations specified in MyInterface
}
```

Aggregation

- Could be just a forwarder (or protocol connector)

```
// external service
outputPort A {
    Location: "socket://urlA.com:80/"
    Protocol: soap
    Interfaces: InterfaceA
}

// external service
outputPort B {
    Location: "socket://urlB.com:80/"
    Protocol: xmlrpc
    Interfaces: InterfaceB
}

// Expose the services from A, B and a locally implemented one
inputPort Input {
    Location: "socket://url.com:8000/"
    Protocol: sodep
    Aggregates: A, B
}
```

And More ...

- Redirection
- Couriers (aggregation with code addition)
- Web Services …
  (use web-services and export Jolie service as a web-service)

Session Management

Multiple executions: sessions

- The calculator works, but it terminates after executing once ...
- We would like it to keep going and accept further requests
- We introduce **sessions**.
- A session is an **execution instance** of a service **behaviour**.

Multiple executions: sessions

- 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 )()
```

- `execution { single }` is the default and can be omitted

Multiple executions: sessions

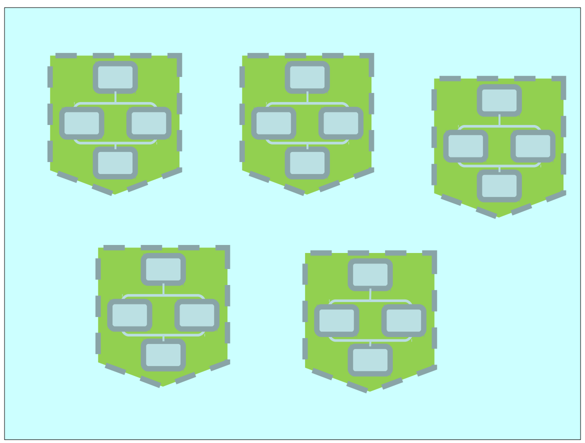
- In the `sequential` and `concurrent` cases, the behavioural definition inside **`main`** must be an input statement
  - input
  - input choice

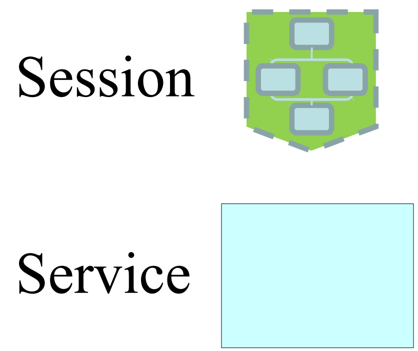
- Kind of guarded pi-calculus "!" …

- Such inputs are called **starting operations** and determine the activation of a new service instance

More on sessions

- A service may have **multiple sessions** running in parallel (conceptually several sessions running inside the service)



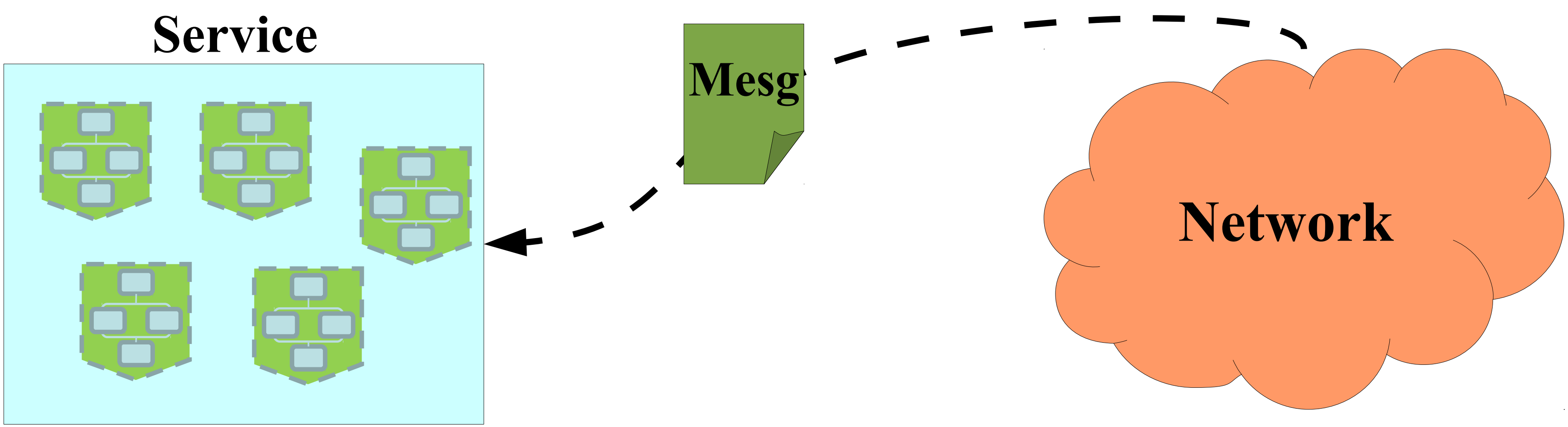


- It may engage in different **separate conversations** with other parties
  - *Example*: a chat service managing different chat rooms.

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

Message routing

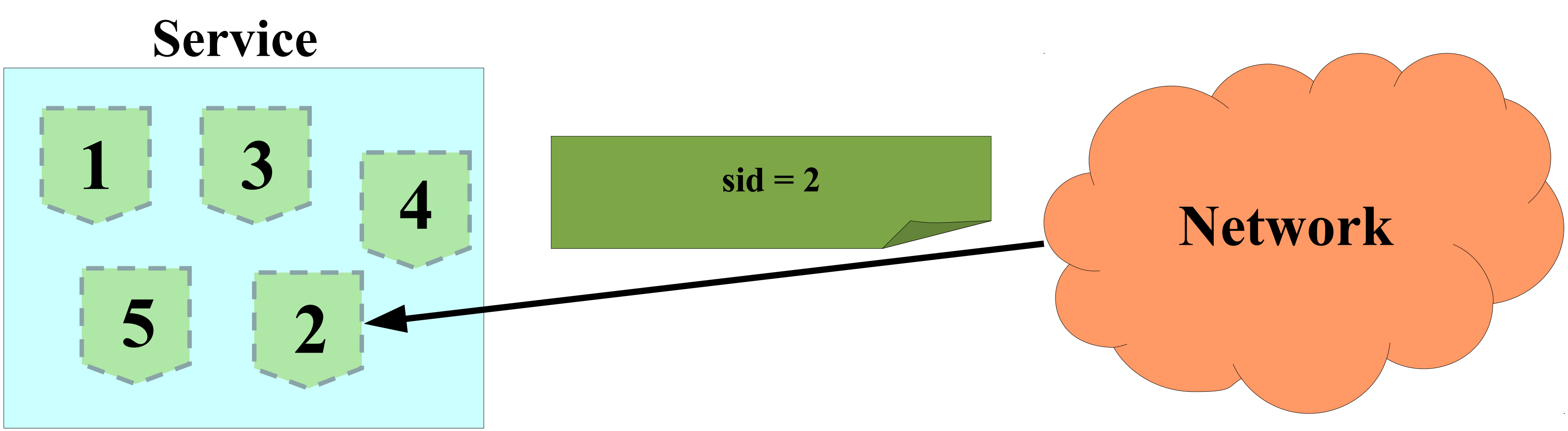
- What happens when a service receives a message from the network?
- We need to assign the message to a session!



- 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 a sid (e.g., cookie in http header reqs).
- The service gives the message to the session with the same sid.

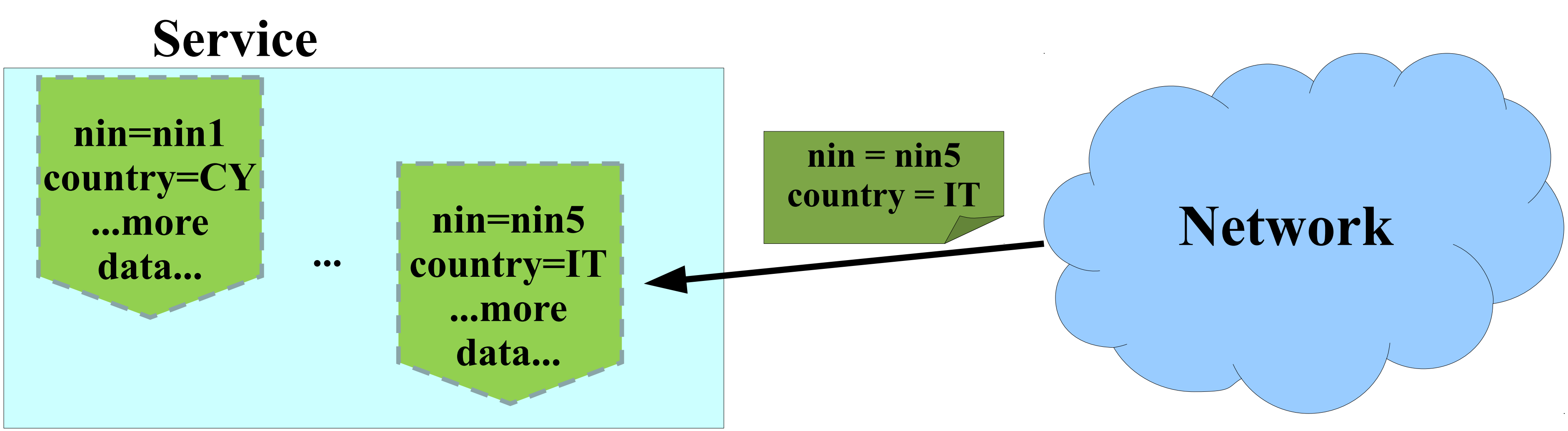


Correlation sets

- 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

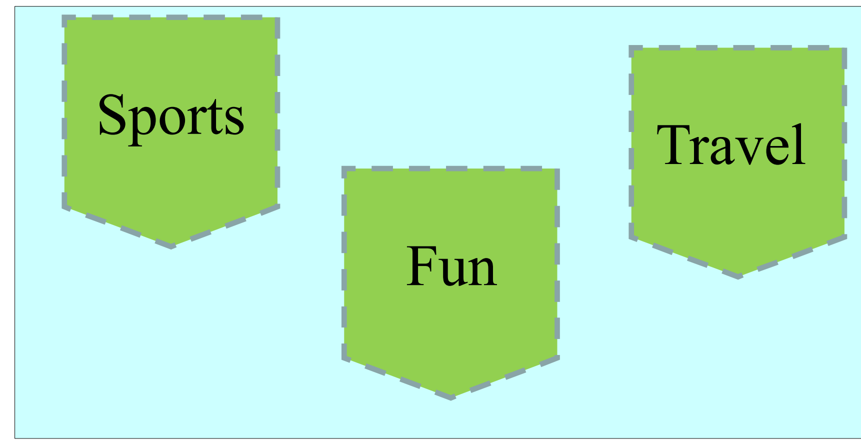
- Pros
  - Programmability of correlation can be used for **integration**.
  - Each cset is a different way of identifying a session: support for **multiparty interactions**.
- Cons
  - Almost totally controlled by the programmer: easy to make mistakes (static analysis and typing support).

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 (pointing to `new`)

Correlating chats

- Two correlation sets (both identifying the instance):
- `name` for the name of the chat,
- `adminToken` unique key for closing the chat

- `openRoom`:
starting message which creates a new instance, initialising the correlation set to name=openRequest.room, adminToken=fresh value

- `publish`
destination determined by `message.room`

- `close`
destination determined by `closeRequest.adminToken`

Correlation sets

- Syntax

```
cset { correlation_var1 : alias_11 alias_12 ...
       ...
       correlation_varn : alias_n1 alias_n2 ... }
```

- Exactly one correlation set linking all its variables to (the type of) an operation → the correlation set for the operation!

→

- Given a message the instance it refers to (if any) is determined

Correlation sets

- When a service receives a message through an input port, there are three possibilities

  - The msg correlates with an instance
    → passed over to the instance

  - The msg does not correlate with any instance and its operation is a starting operation
    → new behaviour instance is created
    correlation set of the starting operation (if any) initialized atomically

  - The msg does not correlate with any behaviour instance and its operation is not a starting operation
    → CorrelationError fault

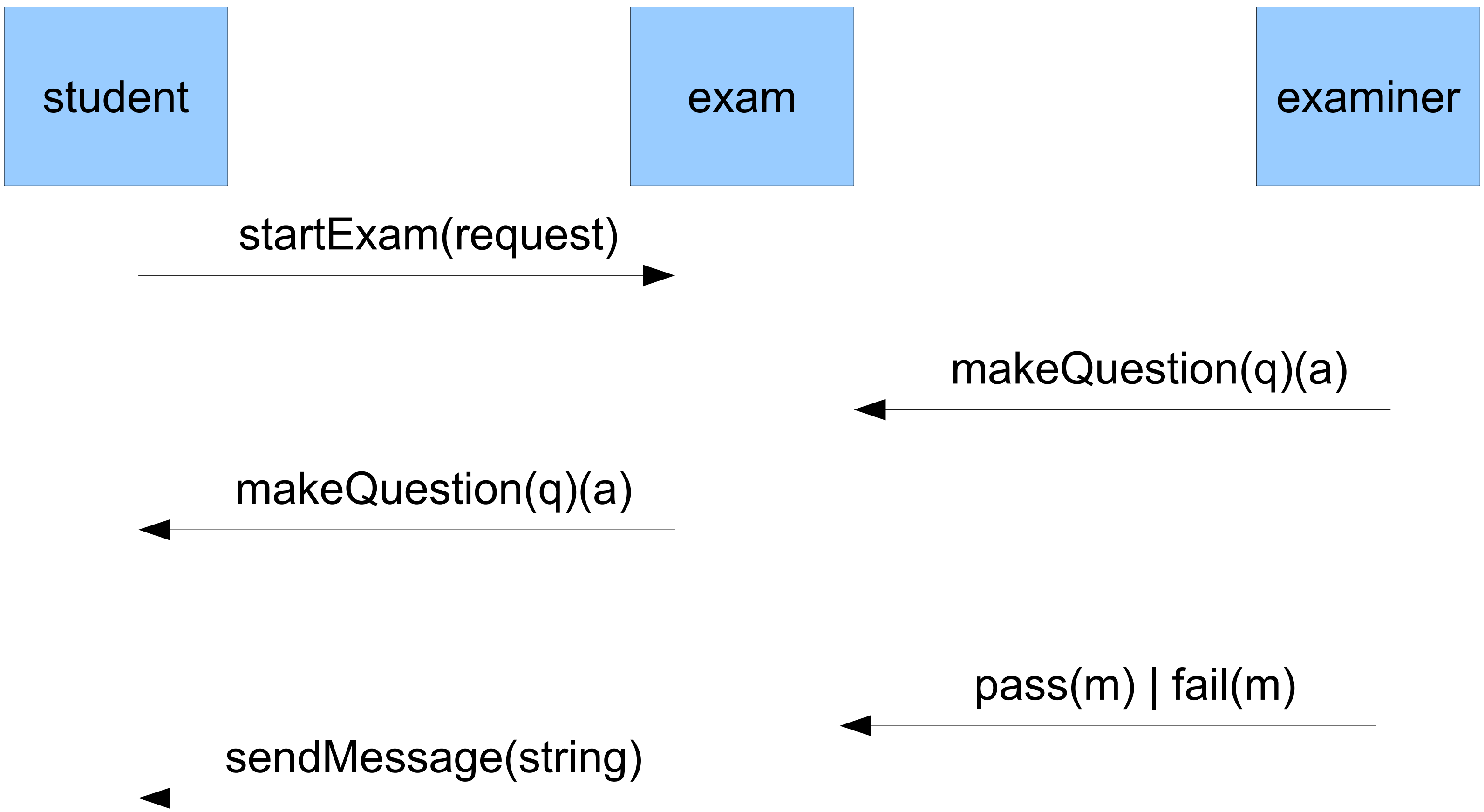
Exercise (together)

- 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**

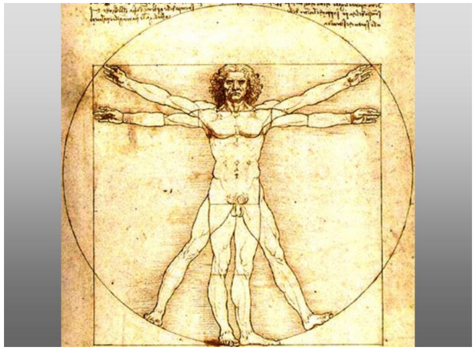
- **Architecture: roles and services.**
  - What are the involved services? **Roles.**
  - Who controls the execution flow? **Orchestrator.**
- **Work flow: operations, data types and activity composition.**
  - Who starts the session?
  - How does the session behave?

Exercise


student
exam
examiner
startExam(request)
makeQuestion(q)(a)
makeQuestion(q)(a)
pass(m) | fail(m)
sendMessage(string)

Some other things you can do with Jolie

Leonardo

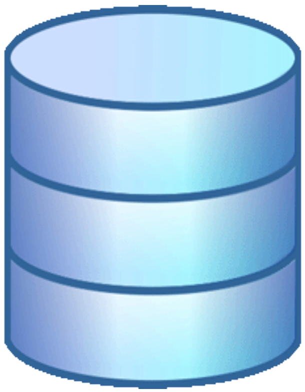


- A web server in pure Jolie.
- Can fit in a slide.

  (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"

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!