A microservice-oriented programming language

Ivan Lanese

(Original slides from Fabrizio Montesi)

Jolie: a **microservice-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

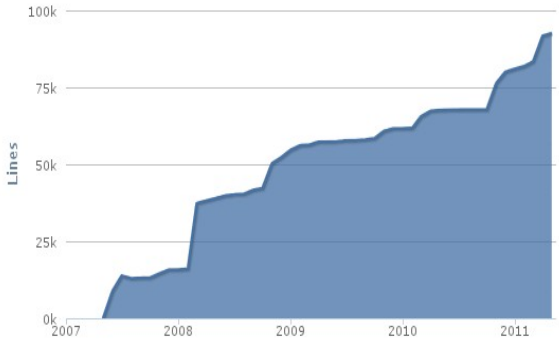
GPA Group

- Have a look at the website: http://www.jolie-lang.org/

- Based on the abstraction of (micro)service

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

- Live *open source* project, with a well-maintained code base
  - https://github.com/jolie/jolie
  - Written in Java



- Comprehensive documentation and standard library
  - https://jolielang.gitbook.io/docs/

- Installing it would be useful for the rest of the course
  - http://www.jolie-lang.org/downloads.html

- An active community on Discord
  - https://discord.gg/yQRTMNX

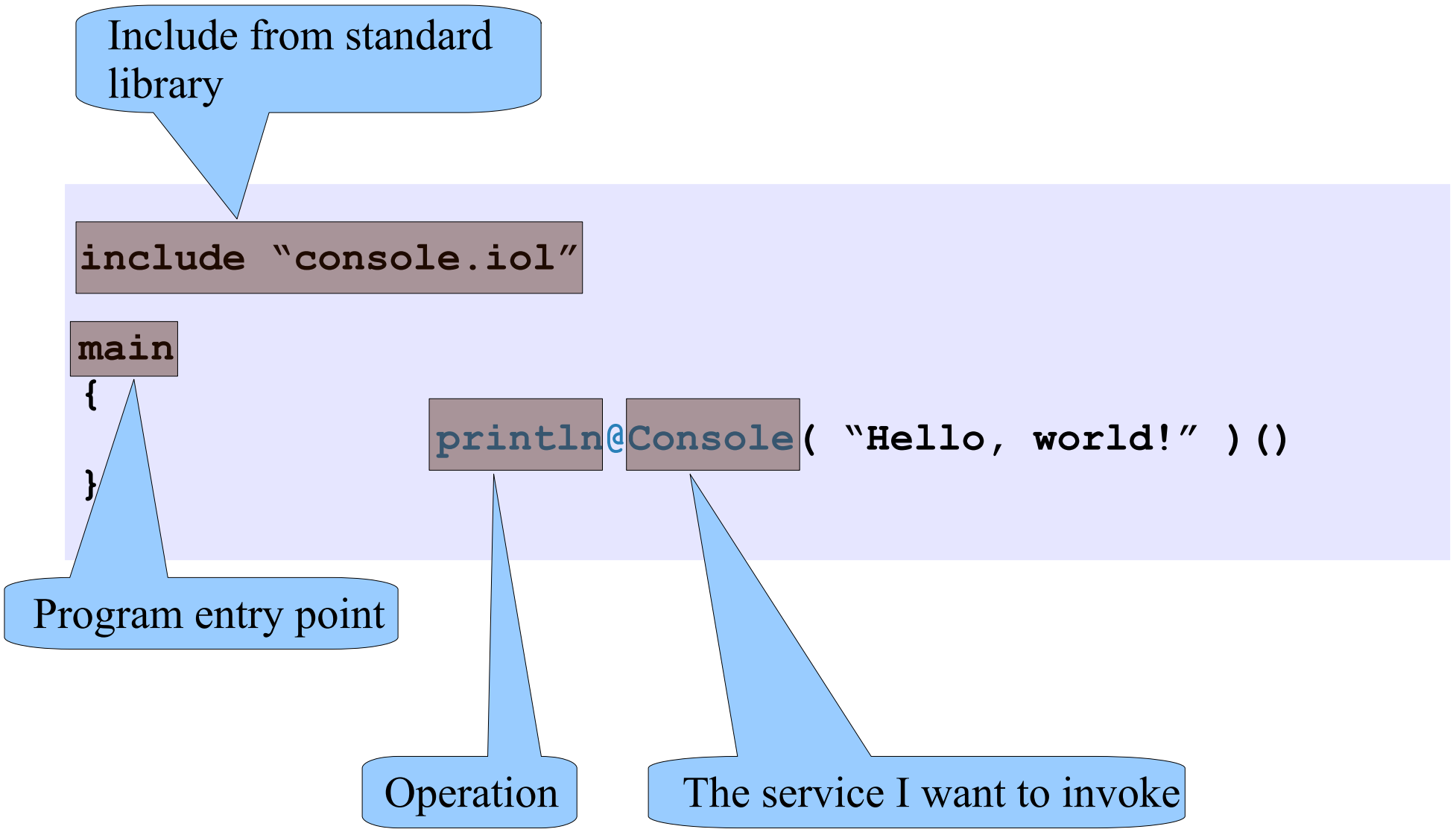
Hello, Jolie!

- Our first Jolie program:

```
include "console.iol"

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

Understanding Hello World: concepts



Our first real microservice-oriented application

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

A

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

B

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

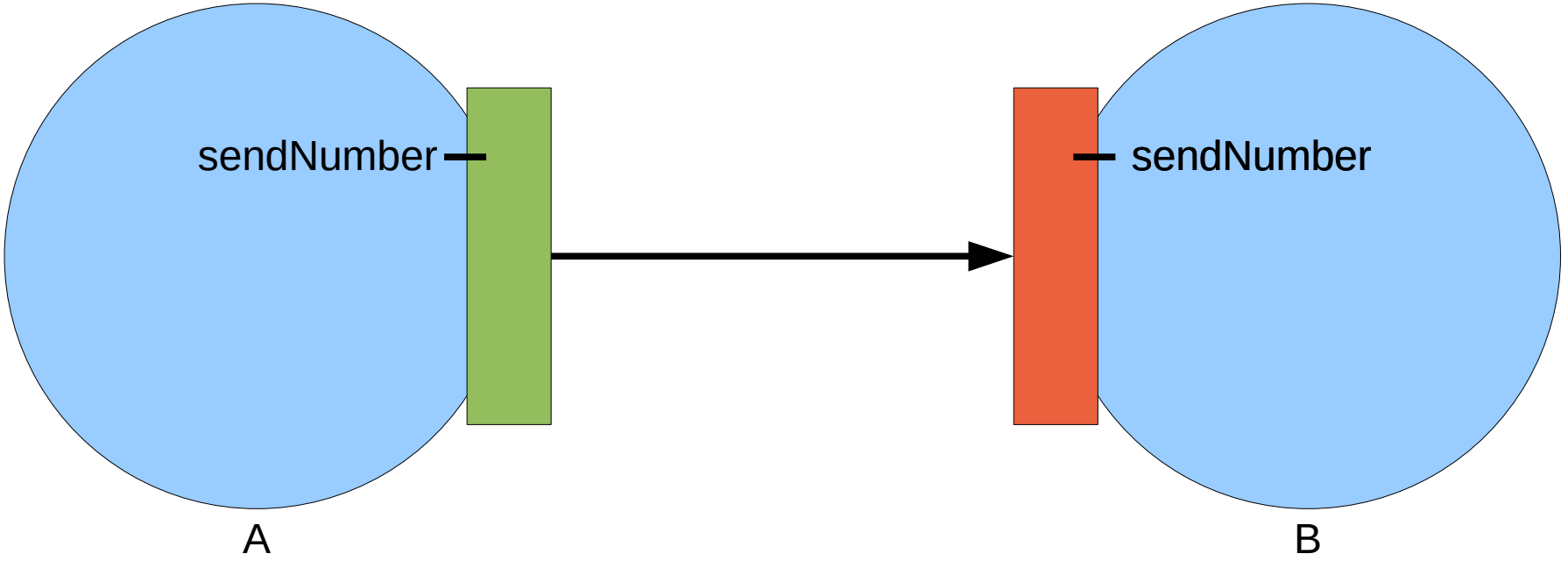
- **A** sends 5 to **B** through the sendNumber operation.

- 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**.
- An **interface** is a set of **operations**.
- 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 microservice-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:
  - Bluetooth:
  - Unix sockets:
  - Java RMI:

```
socket://www.google.com:80/

btl2cap://
localhost:3B9FA89520078C303355AAA694238F07;name=Vision;e
ncrypt=false;authenticate=false

localsocket:/tmp/mysocket.socket


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
}
```

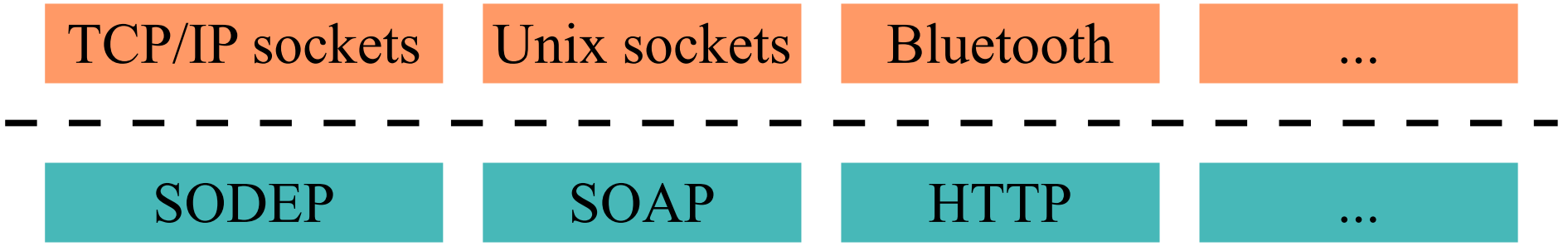
Deployment

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

Behaviour

Communication abstraction

- Jolie supports many different communication mediums and data protocols.



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

Operation types

- JOLIE supports the two classical types of operations for SOA:
  - One-Way: sends a message;
  - Request-Response: sends a message and waits for a response.

- 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 }`**

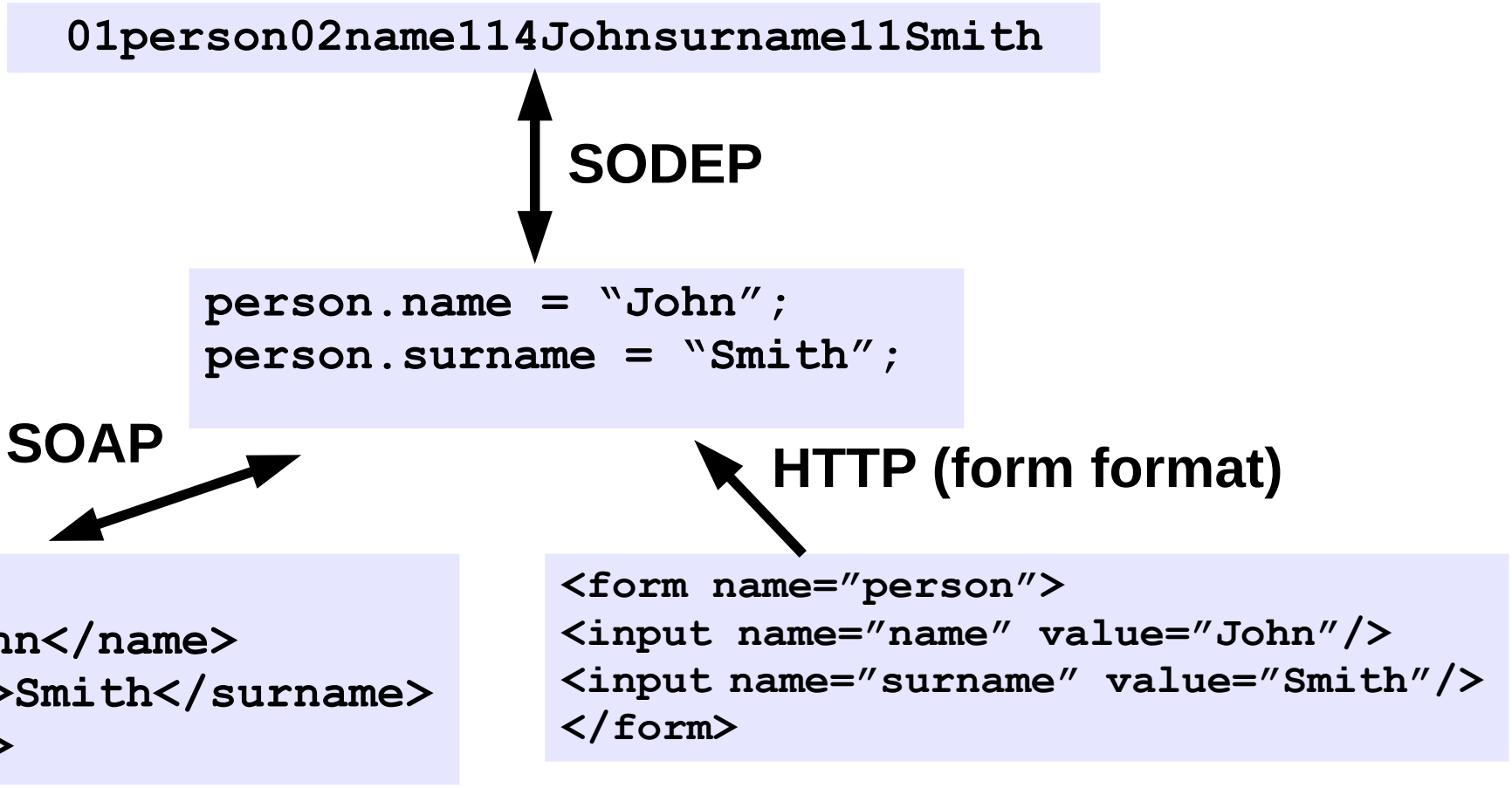
Data manipulation (1)

- 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"
```



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: question

- What will be printed to screen?

```
include "console.iol"
include "string_utils.iol"

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

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 )()
}
```

becomes

```
myPlayer -> team.person[i];
for( i = 0, i < #team.person, i++ ) {
    println@Console( myPlayer.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 = "Bologna";`
    - `cities.(cityName).state = "Italy"`
  - Note that:

    `cities.("Bologna")`
  - is the same as:

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

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

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 { types of subtrees }`
- Where **basic_type** can be:
  - **`int`**, **`long`**, **`double`** for numbers;
  - **`bool`** for booleans;
  - **`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[1,5]: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.
- Range of root is always [1,1] and is omitted
- Syntax (`bt` stands for basic type):
  - **`type`** *`name`*`:bt { .name1[min,max]: bt { … }`
    …
    `.name1[min,max]: bt { … }`
    `}`
- `max` can also be *, for any number of occurrences (>=0)
- One can also have:
  - **`*`** for [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 }
```

User interaction

- This is done via services provided by the standard library
- We have already seen println from console.iol for output

```
println@Console(5)()
```

- An easy (and graphical) way of performing input is showInputDialog from ui/swing_ui.iol

```
showInputDialog@SwingUI("Insert your name:")(x.name)
```

- If interested you can find further user interaction services in the Jolie documentation

Exercise: A calculator service

- Define a calculator service offering operations for sum of two numbers, product of two numbers, and average of an array of numbers
- Define suitable client(s) for the service

A (simpler) 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
    }
}
```