A gentle introduction to

Saverio Giallorenzo | sgiallor@cs.unibo.it

$$\frac{j \in I \quad t_c = \mathsf{eval}(e,t) \quad M(t_c) = (o_j, t') :: \tilde{m}}{\sum_{i \in I} [o_i(x_i)\ \mathtt{from}\ e]\{B_i\} \cdot t \cdot M \to B_j \cdot t \lhd (x_j, t') \cdot M[t_c \mapsto \tilde{m}]} \; \lfloor \mathsf{DCC}|_{\mathsf{Choice}} \rceil$$

$$\frac{t' = \mathsf{eval}(x,t)}{x = e; B \cdot t \cdot M \to B \cdot t \lhd (x, t') \ldots}$$

$$\frac{P \to P'}{P \mid P \ldots}$$

$$\frac{P = \mathsf{cq}(x); B \cdot t \cdot M \ldots}{\langle B_s,\ P \mid \prod \ldots} \; \lfloor \mathsf{DCC}|_{\mathsf{Cq}} \rceil$$

$$\frac{P = \ldots \quad \ldots (o, t_m)]}{\langle B_s,\ P \ldots t \cdot M'' \mid P_1 \rangle_l} \; \lfloor \mathsf{DCC}|_{\mathsf{InSend}} \rceil$$

$$\frac{P = o \ldots = l' \quad \mathsf{eval}(e_3, t) = t_c \qquad \mathsf{eval}(e_2, \ldots [t_c \mapsto M'(t_c) :: (o, t_m)]}{\langle B_s, P \mid P_1 \rangle_l \mid \langle B'_s, B \ldots \rangle_{l'} \to \langle B_s, B \cdot t \cdot M \mid P_1 \rangle_l \mid \langle B'_s, B' \cdot t' \cdot M'' \mid P_2 \rangle_{l'}} \; \lfloor \mathsf{DCC}|_{\mathsf{Send}} \rceil$$

$$\frac{P_1 = ?@e_1(e_2); B_1 \cdot \ldots M_1 \quad \mathsf{eval}(e_1, t_1) = l \quad Q = B \cdot t_\perp \lhd (x, \mathsf{eval}(e_2, t_1)) \cdot \emptyset}{\langle !(x); B,\ P \rangle_l \mid \langle B'_s,\ P_1 \mid P_2 \rangle_{l'} \to \langle !(x); B,\ Q \mid P \rangle_l \mid \langle B'_s,\ B_1 \cdot t_1 \cdot M_1 \mid P_2 \rangle_{l'}} \; \lfloor \mathsf{DCC}|_{\mathsf{Start}} \rceil$$

FORMAL CALCULUS

(like POCs and the pi-calculus)

What is Jolie?

A **Service-Oriented** Programming Language

Service-Oriented	**Object-Oriented**
Service Instances	**Objects**
Operations	**Methods**

Why SOC and Jolie?

Jolie is perfect for fast prototyping. In little time a small team of developers can build up a full-fledged distributed system.

But I already know Java!
Why shall I use Jolie?

Why SOC and Jolie?

```
SocketChannel socketChannel = SocketChannel.open();
  socketChannel.connect(
new InetSocketAddress("http://someurl.com", 80));
  Buffer buffer = . . .; // byte buffer
  while( buffer.hasRemaining() ) {
     channel.write( buffer );
}
```

Happy?

Ok, but you did not even close the channel or handled exceptions

Why SOC and Jolie?

```
SocketChannel socketChannel = SocketChannel.open();
try {
  socketChannel.connect(new InetSocketAddress("http://someurl.com",
80));
  Buffer buffer = . . .; // byte buffer
  while( buffer.hasRemaining() ) {
    channel.write( buffer );
} }
catch( UnresolvedAddressException e ) { . . . }
catch( SecurityException e ) { . . . }
/* . . . many catches later . . . */
catch( IOException e ) { . . . }
finally { channel.close(); }
```

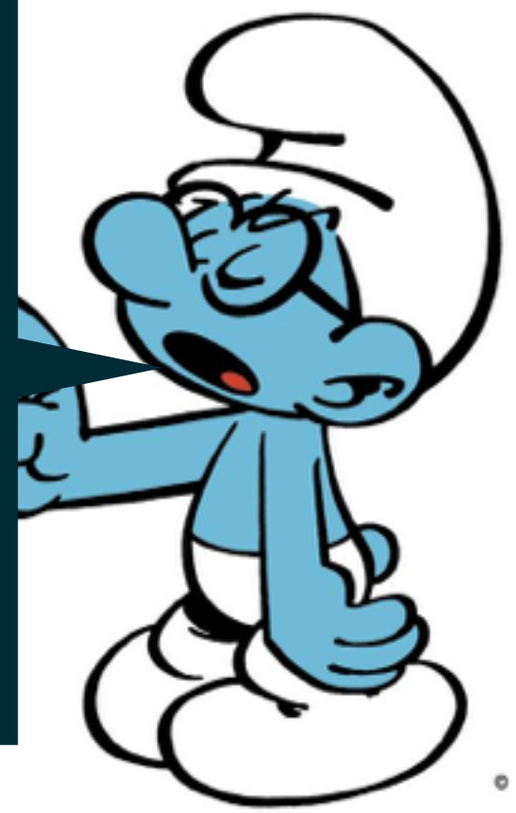
Happier now?

Yes, but what about the **server**?

Why SOC and Jolie?

```
Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
while(true) {
  int readyChannels = selector.select();
  if(readyChannels == 0) continue;
  Set<SelectionKey> selectedKeys = selector.selectedKeys();
  Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
  while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.
    } else if (key.isConnectable()) {
        // a connection was established with a remote server.
    } else if (key.isReadable()) {
        // a channel is ready for reading
    } else if (key.isWritable()) {
        // a channel is ready for writing
    }
    keyIterator.remove();
  }
}
```

Here you are



Why SOC and Jolie?

Well, ok, but again, you are not **handling exceptions**.
And what about if **different operations** use the **same channel**?
And if we wanted to use **RMIs** instead of **Sockets**?
In what **format** are you transmitting data? And if we need to **change** the **format** after we wrote the application? Do you **check** the **type of data** you receive/send?

Why SOC and Jolie?

Programming distributed systems is usually harder than programming non distributed ones.

Concerns of **concurrent** programming.

Plus (not exhaustive):

- handling **communications**;
- handling **heterogeneity**;
- handling **faults**;
- handling the **evolution** of systems.

Hello World! in Jolie

Let us get our hands dirty.

"Hello World!" is enough to let you see some of the main features of Jolie and Service-Oriented Programming.

```
include "console.iol"

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

Include a service

program entry point

operation

service

Hello World! in Jolie

Let us get our hands dirty.

"Hello World!" is enough to let you see some of the main features of Jolie and Service-Oriented Programming.

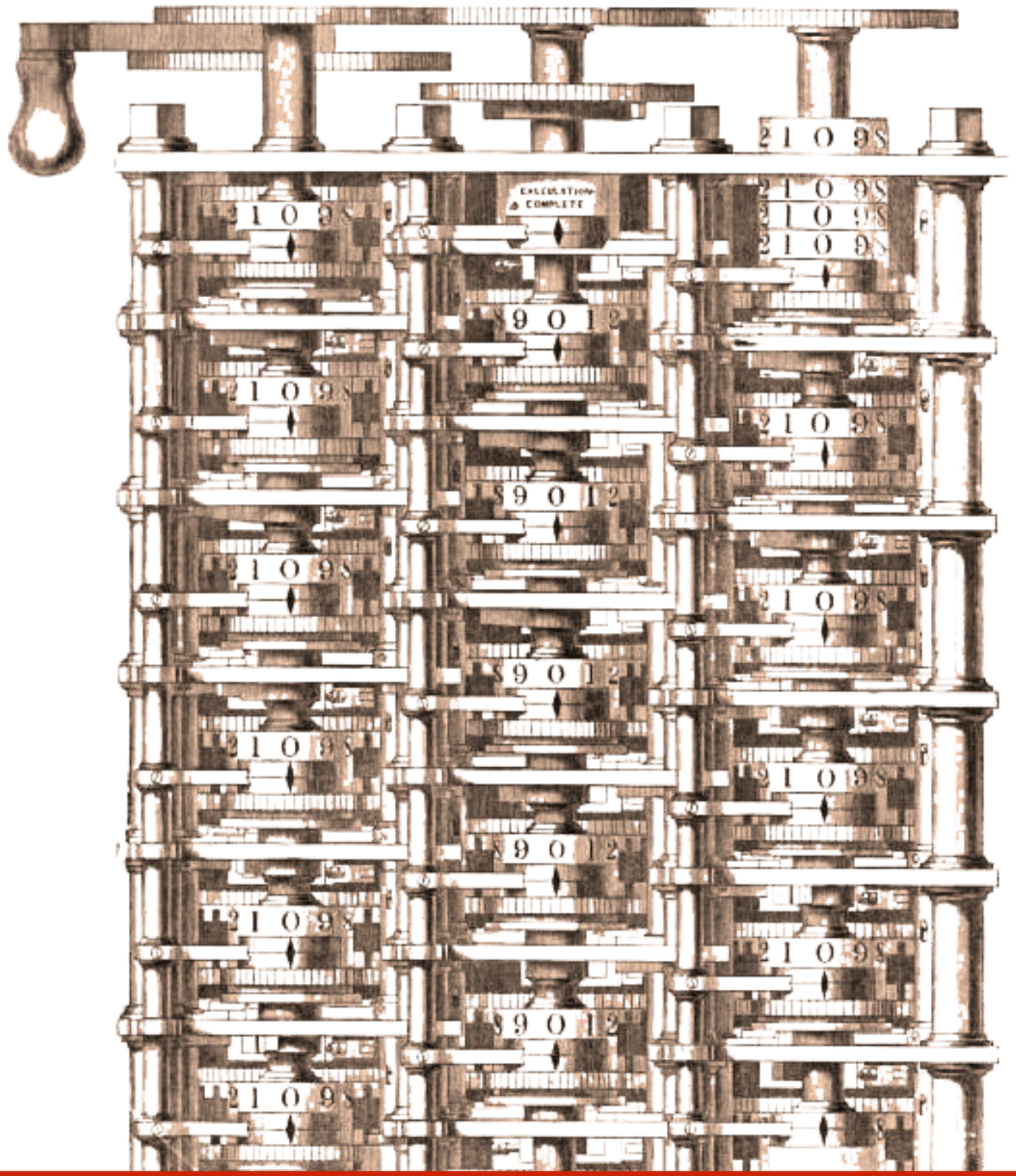
```
include "console.iol"

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

```
$ jolie hello_world.ol
```

hello_world.ol

Let us see some **Jolie in Action**

Everything starts with a **calculator**...



Behaviours and Deployments

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

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

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

Client

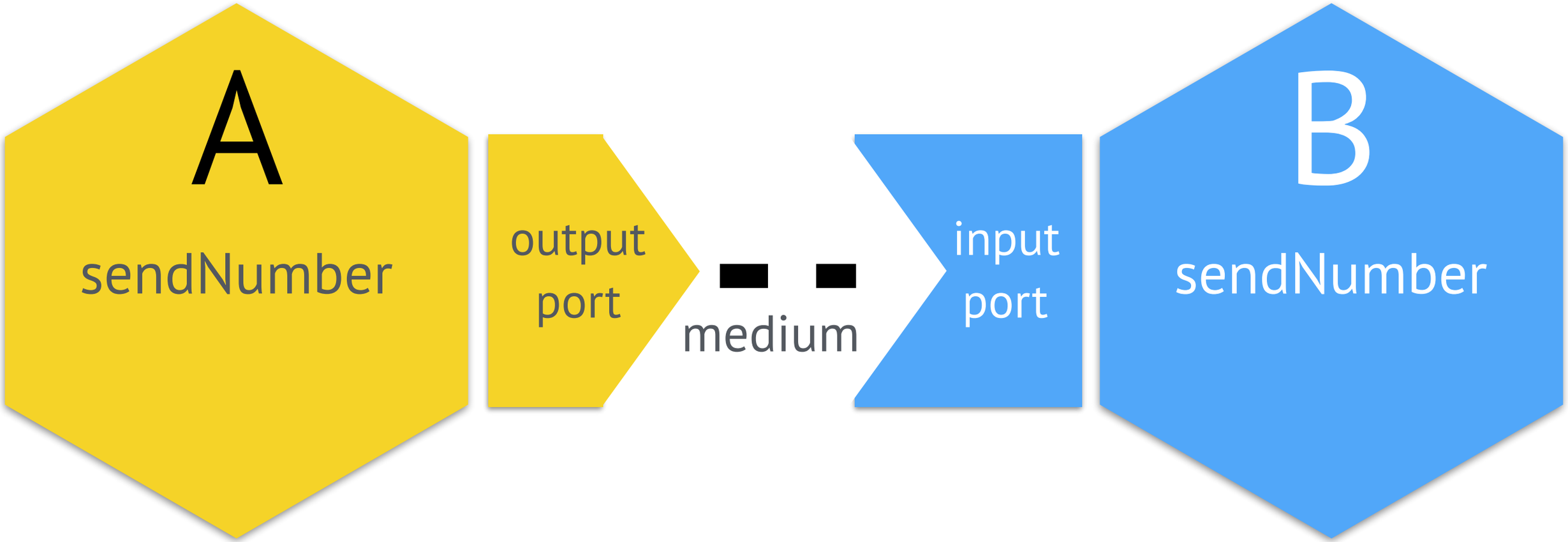
```
include "MyInterface.iol"
inputPort B {
Location:
   "socket://localhost:8000"
Protocol: sodep
Interfaces: MyInterface
}

main
{
   sendNumber( x )
}
```

Server

Deployments

Enabling Communication

Behaviours and Deployments



- 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.

A closer look on ports - Locations

A location describes:

- the **communication medium**;
- the **parameters** to set the communication up.

In Jolie a **location** is a **Uniform Resource Identifier** (URI)
with form: **medium[:parameters]**

	Medium	Parameters
TCP/IP	**socket://**	**www.google.it:80**
Bluetooth	**btl2cap://**	`localhost: 3B9FA89520078C303355AAA694238F07;name=Vision;encrypt= false;authenticate=false`
Local	**localsocket:**	**/tmp/mysocket.socket**
Java RMI	**rmi://**	**myRmiUrl.com/MyService**
In-Memory	**local**	

A closer look on ports - Protocols

A protocol defines the format the data is sent (**encoded**) and received (**encoded**)

In Jolie protocols are names and possibly additional parameters:

```
json/rpc
```

```
sodep
```

```
https
```

```
soap
```

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

Behaviours

Composing Interactions

Interactions via Operations

Input Operations	
	`oneWay( req )`

	`reqRes( req )( res ){`
	`  // code block`
	`}`

Output Operations	
	`oneWay@Port( req )`
	`reqRes@Port( req )( res )`

Behaviour Composition

The sequence operator **;** denotes that the **left operand** of the statement is executed **before** the one on the right.

```
println@Console( "A" )();
println@Console( "B" )()
```

Prints

```
A
B
```

Behaviour Composition

The parallel operator | states that both left and right operands execute concurrently

```
println@Console( "A" )() |
println@Console( "B" )()
```

can print

```
A
B
```

but also

```
B
A
```

Behaviour Composition

The input choice implements **input-guarded non-deterministic choice**.

```
[ oneWayOperation() ] { branch_code }
[ oneWayOperation2() ] {branch_code2}


[ requestResponseOperation()(){
  rr_code }
 ] { branch_code }
```

Behaviour Composition

The input choice implements **input-guarded non-deterministic choice**.

```
main {
 [ buy( stock )( response ) {
  buy@Exchange( stock )( response )
 } ] { println@Console( "Buy order forwarded" )() }

 [ sell( stock )( response ) {
  sell@Exchange( stock )( response )
 }] { println@Console( "Sell order forwarded" )() }
}
```

Last stand - that ORC example

```
include "net.inc"
val BingSpell =
    BingSpellFactoryPropertyFile
    ("orc/orchard/orchard.properties")
Println(y)
< y <
  ( Prompt("Input a string: ") > x >
    ( BingSpell(x) | (Rwait(250) >> x) ) )
```

Last stand - that ORC example

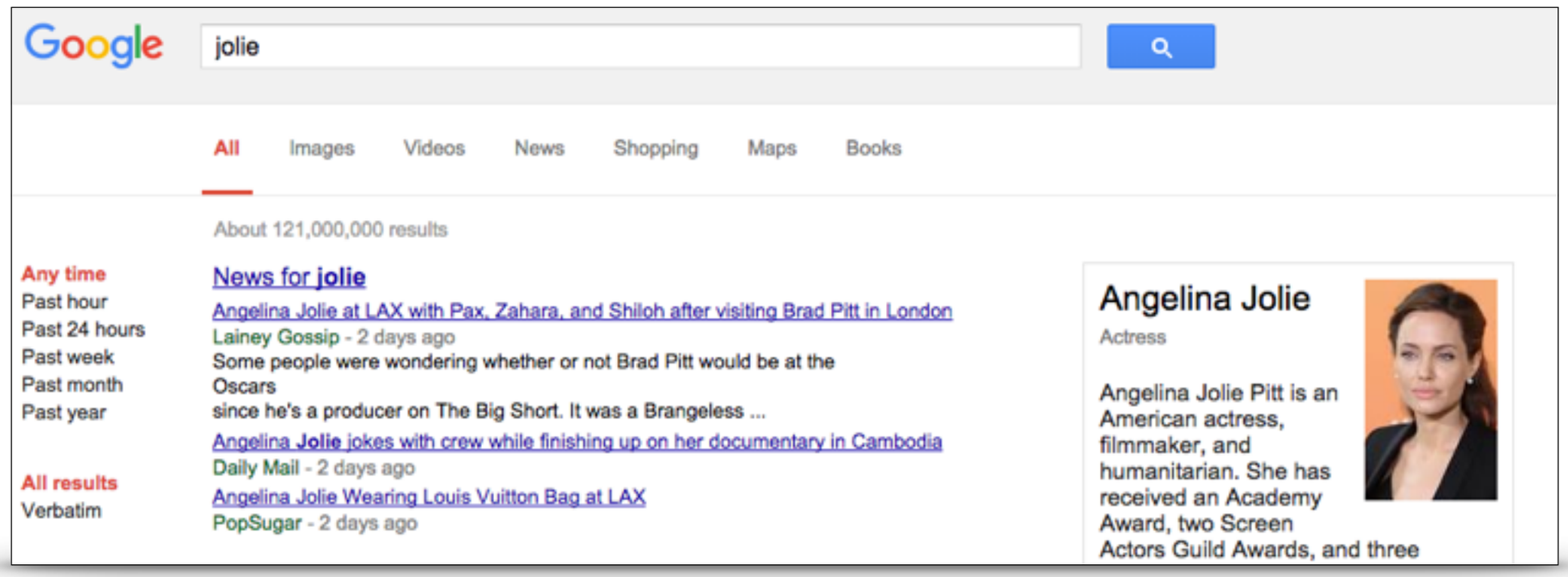
```
include "console.iol"
include "time.iol"

timeout = 250;
timeout.operation = "timeout";
txt = "Beutiful";
{
 spellCheck@BingSpell({ .text = txt, .location = myLoc })
 |
 setNextTimeout@Time( timeout )
};
[ spellCheckResponse( text )]{ println@Console( text )() }
[ timeout() ]{ throw( TimeoutException ) }
```

Before you take off

Jolie Website

http://www.jolie-lang.org

still working out the SEO…



The Jolie Interpreter

Last release

http://www.jolie-lang.org/downloads.html

- Requires JRE 1.6+
- Download jolie-installer.jar
- open a console and run

```
java -jar jolie-installer.jar
```

Sources

Jolie is an **open source** project with continuous updates and a well documented codebase

https://github.com/jolie/jolie

"This *is* the programming language you are looking for"

Documentation

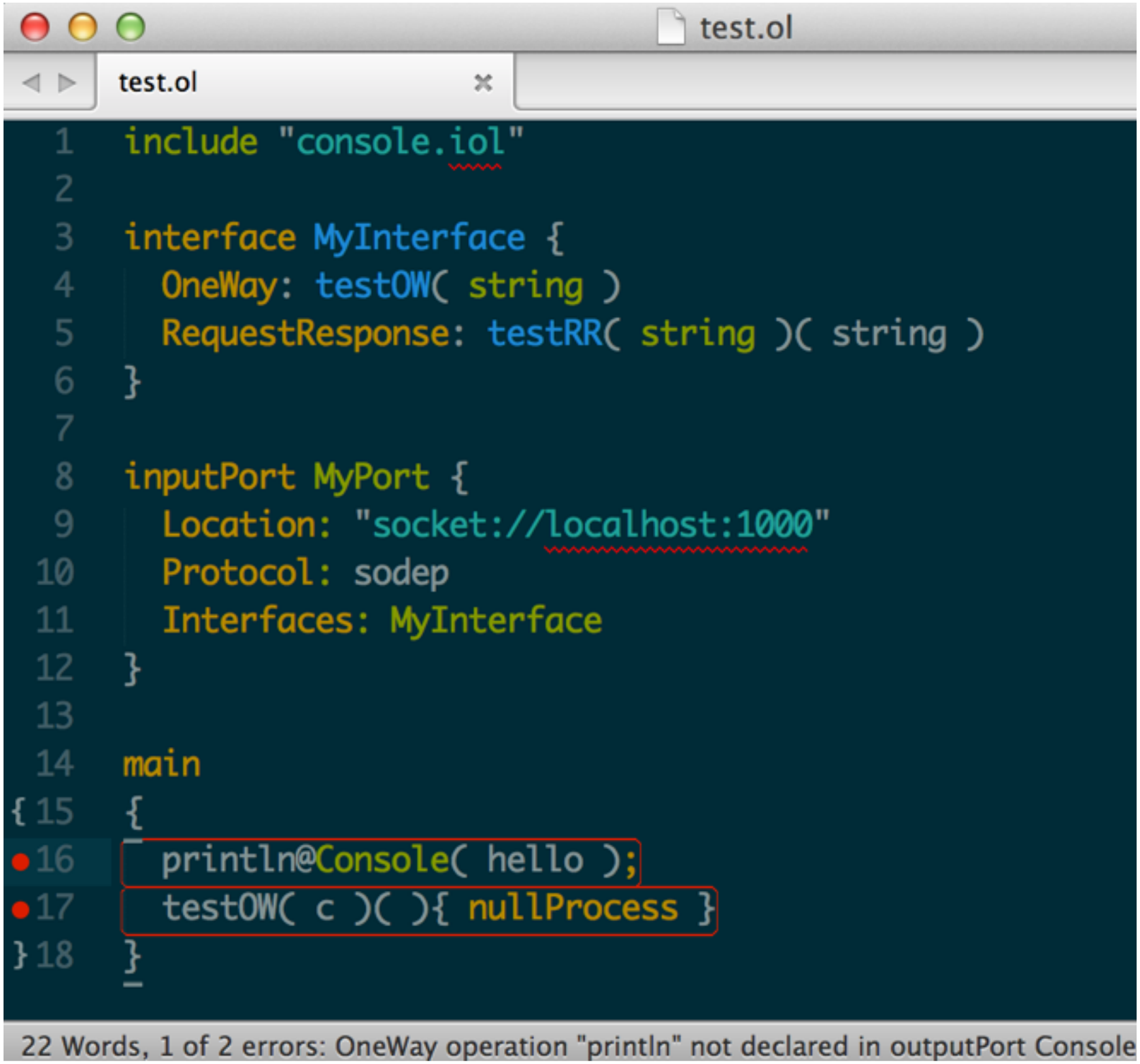
Comprehensive and ever-growing
documentation and **Standard Library**.

http://docs.jolie-lang.org

Editors

Sublime Text but also Atom

Syntax highlight, online checking, etc.



Thanks for your time!