Basic Behaviour

Jolie

Composition and Workflow

Previously on **Jolie**

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

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

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

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

main
{
  sendNumber( x )
}
```

Using ports

Once defined, a **port** can be used for input (output) communications. Ports can provide **one-ways** and **request-responses.**

Input Operations	Output Operations
`ow-op( req )`	`ow-op@Port( req )`

`rr-op( req )( res ){`
`  // code block`
`}`
	`rr-op@Port( req )( res )`

Sequential 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
```

Sequential Composition

;

Ceci n'est pas une fin d'instruction

Sequential Composition

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

If **this** is the last statement **that** is definitely wrong!

Parallel 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
```

Parallel Composition

The parallel operator has **always priority** on the sequence

```
print@Console( "A" )()|
print@Console( "B" )();
print@Console( "C" )()
```

can print ABC but also BAC

Parallel Composition

```
print@Console( "A" )()|
print@Console( "B" )();
print@Console( "C" )()
```

This means:

print "A" and "B" **in parallel** and **then** print "C"

The first two statements create a **race** to access the stdout. **After their execution** the last statement can execute.

Parallel Composition

Good practice: use **scopes {}** to explicitly group parallel statements when mixed with sequences

```
print@Console( "A" )()|
print@Console( "B" )();
print@Console( "C" )()
```

is equal to

```
{ print@Console( "A" )()|
  print@Console( "B" )()
};
print@Console( "C" )()
```

But this is easier to understand

Parallel Composition

Scopes are very useful to clearly specify complex mixes of parallels and sequences of statements

```
{ print@Console( "A" )() |
  print@Console( "B" )()
};{
  print@Console( "C" )() |
  print@Console( "D" )()
}
```

Print “AB” or “BA” and then “CD” or “DC”

Input-Choice

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

```
[ input_operation_1 ]{ branch_code_1 }
[ ... ]{ ... }
[ input_operation_n ]{ branch_code_n }
```

Input-Choice

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

```
[ oneWayOperation() ] { branch_code }

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

Input-Choice

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" )() }
}
```

Service execution modalities

A service participates in a session by executing an **instance of its behaviour**.

Jolie allows to reuse a behavioural definition multiple times with the `execution` primitive.

```
execution{
 single
 | concurrent
 | sequential }
```

Default if execution is not defined

Conditionals

Conditions are used in control flow to check a boolean expression

==	equals to
!=	not equals to
<	lower than
<=	lower than or equal to
>	greater than
>=	greater than or equal to
!	negation

Conditionals

The statement `if ... else` is used to write **deterministic choices**

```
if ( cond ) {
  …
} else {
  …
}
```

`if`s can be nested

```
if( cond1 ){
 …
} else if ( cond2 ) {
 …
} else if ( cond3 ){
 …
}
```

Loops

```
while( condition ) {
  ...
}

for ( ini_code, cond, aftermath-code ) {
  ...
}
```

"main" and "init" procedures

The `main` procedure may be preceded or succeeded by the definition of auxiliary procedures that can be invoked from any other code block and can access any data associated with the specific instance they belong to.

Unlike in other major languages, procedures in Jolie do not posses a local variable scope.

“main” and “init” procedures

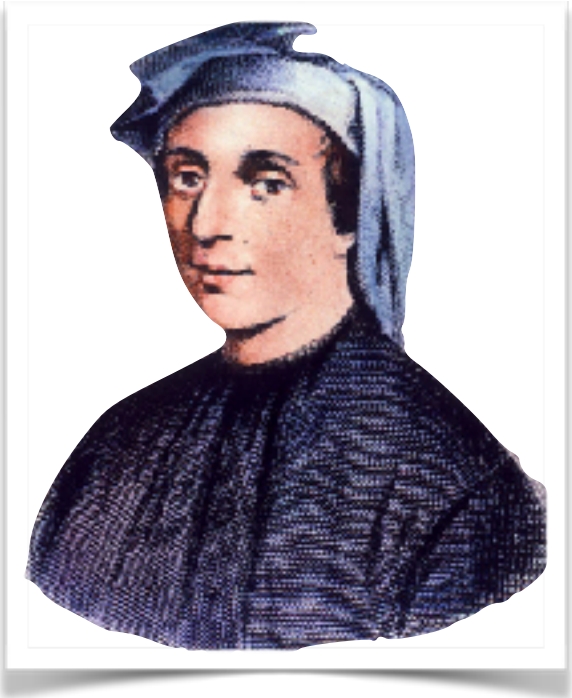
The `init` procedure, if present, is executed before the `main`. The body of the `init` procedure is executed only once, when the service is started.

```
init
{
  getCurrentDateTime@Time()( date )
}

main
{
  start();
  println@Console( "start date: " + date  )();
  getCurrentDateTime@Time()( date );
  println@Console( "current date: " + date  )()
}
```

Procedures: definition and recall

```
define procedureName
{
   ...
   code
   ...
}
```



```
include "console.iol"

define fibonacci
{
   if( f1 < end ){
      println@Console( f1 )();
      _f2 = f1+f2;
      f1 = f2;
      f2 = _f2;
      fibonacci
   }
}

main
{
   f1 = 0; f2 = 1; end = 200;
   fibonacci
}
```

Constants

It is possible to define `constants` by means of the construct constants. The declarations of the constants are divided by **commas**

```
constants {
  server_location = "socket://localhost:8080",
  ALARM_TIMEOUT = 2000,
  standard_gravity = 9.8
}
```

Constants might also be assigned on the command line.

```
jolie -C ALARM_TIMEOUT=2000 program.ol
```

which overrides `ALARM_TIMEOUT`

`global` variables

Jolie provides global variables to support sharing of data among different instances. Global variables belong to the `global` prefix

```
[ count() ]{ global.i++ }
[ print( run ) ]{
  println@Console(global.i)();
  undef( global.i )
}
```

synchronized scopes

Concurrent access to global variables can be restricted through synchronized blocks

```
synchronized( id ){ … }
```

```
[ count() ]{
  synchronized( syncToken ){
   global.i++
  }
}
[ print( run ) ]{
  println@Console(global.i)();
  undef( global.i )
}
```