

# Basic Behaviour



# Types and Data Manipulation

Saverio Giallorenzo | [sgiallor@cs.unibo.it](mailto:sgiallor@cs.unibo.it)

# Previously on Jolie

# Basic Datatypes

Jolie supports seven basic data types:

- **bool**: booleans;
- **int**: integers;
- **long**: long integers (with “L” or “l” suffix);
- **double**: double-precision float (decimal literals);
- **string**: strings;
- **raw**: byte arrays;
- **void**: the empty type.

Jolie also supports the **any** basic type, a value that can be any basic type.

# Data & Types - Part I

# Defining variables

Jolie is a **dynamically typed** language

```
a = 5; // int  
a = "Hello" // string
```

Jolie applies **file-level scoping** on variables, i.e., their scope extends for the entire file – and *includes*, if present.

# Defining variables

Jolie supports basic arithmetic operators:

add	+
subtract	-
multiply	*
divide	/
modulo	%
pre-/post-increment	++
pre-/post-decrement	--

```

a = 1;
b = 4;
n = a + b/2; // n = 3
n++; // n = 4
n = ++a + (b++)/2 // n = 4

```

# Casting variables

Variables can be cast to other types by using the corresponding casting functions

`bool()` `int()` `long()`  
`double()` `string()`

```
s = "10";  
n = 5 + int( s ); // n = 15  
  
d = "1.3";  
n = double( d ); // n = 1.3  
n = int ( n ) // n = 1
```

# Checking variable types

A variable type can be checked at runtime by means of the `instanceof` operator

```
s = "10";  
n = s instanceof string; // n = true  
n = s instanceof int; // n = false  
n = ( s = 10 ) instanceof int; // n = true
```



# Strings

Strings can be inserted enclosing them between double quotes. Character escaping works, like in C and Java, using the `\` escape character

```
s = "This is a string\n"  
  
s = "This is " + "a string\n"  
  
s = "  
JOLIE preserves formatting.  
  This line will be indented.  
    This line too.  
"
```

# Checking if variables are defined and undefining them

Once a variable is assigned, it is *defined*.

The operator `is_defined( var )` checks if a variable is defined

```
a = 1;  
is_defined( a ) // returns true  
is_defined( b ) // returns false
```

# Undefining variables

The operator `undef()` makes a variable undefined again (it removes its assigned value)

```
a = 1;  
is_defined( a ); // returns true  
undef( a );  
is_defined( a ) // returns false
```

# Dynamic Arrays

Arrays in Jolie are dynamic and can be accessed by using the `[]` operator

```
a[ 0 ] = 0;  
a[ 1 ] = 5;  
a[ 2 ] = "Hello";  
a[ 3 ] = 2.5
```

# Dynamic Arrays

in Jolie

**every variable is a dynamic array**

```
a = 1
```

is interpreted as

```
a[0] = 1
```



# Dynamic Arrays

in Jolie

**every variable is a dynamic array**

```
a.b.c = 1
```

==

```
a[ 0 ].b[ 0 ].c[ 0 ] = 1
```



# Dynamic Arrays

```
a.b.c[0] = 1;  
a.b.c[1] = 2
```

jolie tree

**VS**

```
<a>  
  <b>  
    <c>1</c>  
  </b>  
  <b>  
    <c>2</c>  
  </b>  
</a>
```

xml

```
{  
  "a": {  
    "b": [  
      { "c": "1" },  
      { "c": "2" }  
    ]  
  }  
}
```

json

# The array size operator #

```
a[ 0 ] = 0;  
a[ 1 ] = 1;  
a[ 2 ] = 2;  
a[ 3 ] = 3;  
#a // returns 4
```



# The array size operator #

```
a.b = 0;  
a.b[ 1 ] = 1;  
a.b.c = 2;  
a = 3;
```

Dare to guess?

#a

#a.b

#a.b.c

# The array size operator #

```
a[ 0 ] = 3  
|_ b [ 0 ] = 0  
| [ 1 ] = 1  
|_ c [ 0 ] = 2
```

# The array size operator #

```
a[ 0 ] = 3
|_ b [ 0 ] = 0
| [ 1 ] = 1
|_ c [ 0 ] = 2
```

Did you guess right?

#a=?    #a.b=?    #a.b.c=?

# The array size operator #

```
a[ 0 ] = 3
|_ b [ 0 ] = 0
| [ 1 ] = 1
|_ c [ 0 ] = 2
```

Did you guess right?

#a=1    #a.b=?    #a.b.c=?

# The array size operator #

```
a[ 0 ] = 3  
|_ b [ 0 ] = 0  
| [ 1 ] = 1  
|_ c [ 0 ] = 2
```

Did you guess right?

#a=1    #a.b=2    #a.b.c=?

# The array size operator #

```
a[ 0 ] = 3  
|_ b [ 0 ] = 0  
| [ 1 ] = 1  
|_ c [ 0 ] = 2
```

Did you guess right?

#a=1    #a.b=2    #a.b.c=1

# Data & Types - Part II

# Managing complex data structures - **Deep Copy** Operator

## Deep Copy Operator

dst << src

```
birds.dove      = 1;  
birds.swan     = 2;  
  
mammals.lion   = 2;  
mammals.puma  = 3;  
  
fish.tuna      = 1;  
  
zoo.fly        << birds;  
zoo.walk       << mammals;  
zoo.swim       << fish
```



# Managing complex data structures - **Deep Copy Operator**

```

zoo
|_ fly
|   |_ dove
|   |_ swan
|
|_ walk
|   |_ lion
|   |_ puma
|
|_ swim
    |_ tuna

```

```

birds.dove      = 1;
birds.swan      = 2;

mammals.lion    = 2;
mammals.puma    = 3;

fish.tuna       = 1;

zoo.fly        << birds;
zoo.walk       << mammals;
zoo.swim       << fish

```

# Managing complex data structures - **Deep Copy Operator**

**Attention:** `d << s` overwrites all the correspondent sub-nodes of `s` rooted in `d`, leaving the other sub-nodes unaffected

```
d.greeting      = "hello";
d.first         = "to the";
d.first.second  = "world";
d.first.third   = "!";

s.first.first   = "to a";
s.first.second  = "brave";
s.first.third   = "new";
s.first.fourth  = "world";

d << s
```

## Before

```
d
|_ greeting = "hello"
|_ first = "to the"
   |_ first.second = "world"
   |_ first.third = "!"
```

## After

```
d
|_ greeting = "hello"
|_ first
   |_ first = "to a"
   |_ second = "brave"
   |_ third = "new"
   |_ fourth = "world"
```

# Managing complex data structures - **Inline Trees**

It is possible to  
compose trees inline  
with syntax

```
{  
  .node1 = 1,  
  .node2 = "2",  
  .node3 = var3  
}
```

```
zoo.fly << {  
  .dove = 1,  
  .swan = 2  
};  
zoo.walk << {  
  .lion = 2,  
  .puma = 3  
};  
zoo.swim << {  
  .tuna = 1  
};
```

# Navigating complex data structures - **Dynamic Lookup**

Nested variables can be identified by means of a string expression evaluated at runtime.

Dynamic look-up is obtained as a subpath with a **string within round parenthesis**

```

zoo
|_ fly
|   |_ dove
|   |_ swan
|
|_ walk
|   |_ lion
|   |_ puma
|
|_ swim
|   |_ tuna

```

```
zoo.( "fly" ).dove
```

```
zoo.( "f" + "l" + "y" ).dove
```

```
zoo.( "f" + "l" + "y" ).( "dove" )
```

```
fly = "fly"
```

```
zoo.( fly ).dove
```

# Navigating complex data structures - 'with' Operator

**with** operator provides a shortcut for repetitive variable paths.

```
with ( zoo ){  
    .fly.dove = 1;  
    .fly.swan = 2  
    .mammals.lion = 2;  
    .mammals.puma = 3  
    .fish.tuna = 1  
}
```

# Navigating complex data structures - 'with' Operator

**with** operator  
provides a shortcut  
for repetitive  
variable paths.

**withs can  
be nested!**

```
with ( zoo ) {  
  with ( .fly ) {  
    .dove = 1;  
    .swan = 2  
  };  
  with ( .mammals ) {  
    .lion = 2;  
    .puma = 3  
  };  
  with ( .fish ) {  
    .tuna = 1  
  }  
}
```

# Navigating complex data structures - 'with' Operator

**with** operator provides a shortcut for repetitive variable paths.

it means it is evaluated for each **.subpath** inside the **with**



```
with ( arr[ #arr ] ) {  
  .a = "1";  
  .b = "2";  
  .c = "3"  
}
```

evaluates to

```
arr[ #arr ].a = "1";  
arr[ #arr ].b = "2";  
arr[ #arr ].c = "3"
```

# Navigating complex data structures - 'foreach' Operator

```
foreach ( kind : zoo ){  
    foreach( species : zoo.( kind ) ){  
        println@Console( "zoo." +  
            kind + "." + species )()  
    }  
}
```

## Returns



```
zoo.fly.dove  
zoo.fly.swan  
zoo.swim.tuna  
zoo.walk.lion  
zoo.walk.puma
```

The **foreach** operator looks for any child-node inside the given **root**. For every child assigns its **name** to the given variable and executes the internal code block.



# Navigating complex data structures - **Aliases**

An **alias** is a pointer to to another variable path. Aliases are created with the **->** operator

```
birds    -> zoo.fly;  
mammals  -> zoo.walk;  
fishes   -> zoo.swim
```

# Navigating complex data structures - Aliases

```
currentKind -> zoo.( kind );  
foreach ( kind : zoo ) {  
    foreach ( species : currentKind ) {  
        println@Console( species )()  
    }  
}
```

prints



dove  
swan  
tuna  
lion  
puma

# Navigating complex data structures - Aliases

```
with ( a.b.c ) {  
  .d[ 0 ] = "zero";  
  .d[ 1 ] = "one";  
  .d[ 2 ] = "two";  
  .d[ 3 ] = "three";  
};  
  
currElem[ 0 ] -> a.b.c.d[ i ];  
  
for ( i = 0, i < #a.b.c.d, i++ ) {  
  println@Console( currElem )()  
}
```

Prints



```
zero  
one  
two  
three
```