

1. Getting started with ChocLet

Launching

To launch the shell of ChocLet, just run the following command .

```
$ java -jar choclet.jar  
>
```

Variable declaration

You can declare variables using the keyword **let**.

```
let x = 10;  
let y = "some_string";  
let z = 13.385;
```

The types of the variables are inferred, and you can't change the type of a variable.

```
let x = 10; // x is an int  
x = 0.1; // Error, No operator '=' for type 'int' and 'double'
```

ChocLet give some ways to declare arrays.

```
let x = [1, 2, 3]; // x is an array of int, of size 3  
let y = [100 of float]; // y is an array of float, of size 100
```

Warning

All the values inside an allocated array, are null.

```
let x = [10 of int];  
// let y = x [0] + 10; // NullPointerException  
x [0] = 3; // Ok  
let y = x [0] + 10; // Ok
```

You can concatenate arrays of the same type using the operator \sim .

```
let x = [1, 2, 3] ~ [4, 5, 6];  
let y = "same_for_" ~ "strings";  
println (x); // [1, 2, 3, 4, 5, 6];  
println (y); // same for strings
```

Source file

You can write source code in a file. The file must have the extension **.clt**. To run a file, there is two ways, the first one is to pass the file as an argument of the jar file.

```
|| $ java -jar choclet.jar myfile.clt
```

The second one is to import the function described in the file using the keyword **import**. It won't execute the code inside the file, but import all declared function.

```
|| > import myfile;
```

Functions

```
|| def foo (a, b) {  
||     println (a, b);  
|| }  
  
|| // foo is a function and we call it  
|| foo (1, "hi_!!");
```

Flow Control

Values can be controlled conditionally using the **if** and **else** statements.

```
|| let n = 5;  
|| if n < 0 {  
||     println (n, "_is_negative");  
|| } else if n > 0 {  
||     println (n, "_is_positive");  
|| } else {  
||     println (n, "_is_zero");  
|| }
```

You can also do loops with the **while** keyword.

```
|| let n = 1;  
|| // Loop while n is less than 101  
|| while n < 101 {  
||     if n % 2 == 0 {  
||         println ("even");  
||     } else {  
||         println ("odd");  
||     }  
||     n = n + 1;  
|| }
```

Or iterate over a range of value with the keyword **for**

```
for i in 0 .. 101 {  
  if i % 2 == 0 {  
    println ("even");  
  } else {  
    println ("odd");  
  }  
}
```

2. Choco interface

ChocLet is designed to use Choco solver.

```
let a = choco.int (0, 10);  
let b = choco.int (0, 10);
```

This declaration means that **a**, and **b** can have a value between 0 to 10.

```
(a != b).post ();
```

In this instruction, we inform choco, that we don't want that **a** and **b** have the same value.

```
while choco.solve ()  
  println (a, "!=" , b);
```

We request a resolution, and while there is a valid solution, we print the value of **a** and **b**.

In choco there is some special constraint on boolean, that will be posted automatically. Those constraints are the simple and double implication.

```
(a != b) -> (c != d);  
(a == b) <-> (c <= d);
```

Choco have some predefined global constraint, and some times we want to use them.

```
in choco {  
  // allDifferent is a choco function declared somewhere  
  def allDifferent -> ChocoConstraint;  
}  
  
// creating an array of 10 var, between 0 and 10  
let a = choco.intArray ("A", 10, 0, 10);  
choco.allDifferent (a).post (); // Ok  
choco.solve ();  
println (a);
```

Usefull global constraints

Choco give us some simplification to compute some information on arrays of ChocoInt. It give us access to the sum.

```
let resultOfSum = 10;
choco.sum (myarray, "=", resultOfSum).post ();
// The operator inside of the function can be either '<', '>', '=',
// '!=', '<=', '>='
// It will ensure that the sum of the array respect the operator on the
// result.

// It can be used to store the sum into a ChocoInt too.
let resultVar = choco.int (0, 1000);
choco.sum (myarray, "=", resultVar).post ();
```

Or the number of time an element appears in the array.

```
let resultOfCount = choco.int (0, myArray.len);
let itemToCount = 1;
choco.count (itemToCount, myArray, resultOfCount).post ();
```

BinPacking

The last important one for us is the binPacking constraint. This constraint will take 3 elements :

- *position* : An Array of position
- *size* : An Array of size, with the same length as position
- *capacity* : An Array of capacity

The array of position will represent a position in the capacity array, it means that all the value stored in position must have a range between $[0, capacity.len[$.

This constraint will ensure that the sum of the size of the elements with a position b

$$\forall b \in [0, |capacity|[\left(\sum_{i=0}^{|size|} S \mid S = \begin{cases} 0, & \text{if } position_i \neq b \\ size_i, & \text{otherwise} \end{cases} \right) \leq capacity_b \quad (1)$$

It can be used in ChocLet after declaring the global constraint.

```

let position = choco.intArray ("P", 3, 0, 2);
let size = [10, 5, 9];
let capacity = choco.intArray ("C", 2, 0, 20);

// The last parameter is just an offset, we don't use it
choco.binPacking (position, size, capacity, 0).post ();
// The capacity array will be informed.
// Possible solutions are capacity = [15, 9] or [19, 5] ...
//                               position = [0, 0, 1] or [0, 1, 0] ...

```