

# tut\_base

## Table of contents

1 Tutorial : first steps with Choco.....	2
--	---

## 1. Tutorial : first steps with Choco

This tutorial is aimed at presenting the basic information needed to be able to write one's first application with Choco :

- how to create a problem,
- how to solve it,
- how to explore the set of solution(s).

Our running example will be the n-queens problem.

### Problem modeling

In order to solve the n-queens problem with Choco, the first thing to do is to create a problem. Here is how it is done in Choco:

```
Problem nQueens = new Problem();
```

This object will be the basis for all the subsequent information that will be added. A classical model for the n-queens problem consists in defining n variables taking their value between 1 and n : those variables represent the position of each queen on each line. In Choco, we can therefore declare an array of variables and ask to the Problem to initialise them:

```
IntVar[] queens = new IntVar[n]; // declaring an array of n variables
for (int i = 0; i < queens.length; i++) {
    queens[i] = nQueens.makeEnumIntVar("Queen" + i, 1, n);
}
```

#### Note:

We used here `makeEnumIntVar` to declare an enumerated variable because we want to explicitly handle holes in the domain of the variable. Other possibilities exist: `makeBoundIntVar`, etc.

The next step consists in adding constraints to our problem:

- any two queens should be positionned in different lines (the model itself enforces this constraint)
- any two queens should be in different columns
- any two queens should not be in the same diagonal

We will use difference and linear combination constraints:

```
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
```

## tut\_base

```
nQueens.post(nQueens.neq(queens[i], queens[j])); // different columns
nQueens.post(nQueens.neq(queens[i],
nQueens.plus(queens[j], j - i))); // diagonal
nQueens.post(nQueens.neq(queens[i],
nQueens.minus(queens[j], j - i))); // diagonal
}
}
```

Our problem is now completed: the problem has been created, variables have been created, constraints have been created and posted (they are now active in the problem). Now, it is time to actually solve the problem.

### Solving the problem

The Problem object provides three methods that can be used to solve a problem: `propagate`, `solve`, and `solveAll`.

The first one can be used to propagate the constraints of the problem i.e. as much filtering as possible impossible values from the domain of the variables (no enumeration is performed here).

```
try {
nQueens.propagate();
}
catch (ContradictionException e) {
System.err.println("This problem is over-constrained");
System.exit(-1);
}
```

#### **Warning:**

As the `propagate` method is only in charge of propagating the constraints of the problem, it may happen that a contradiction is raised. This is why any call to this method should be embedded within a `try / catch` environment.

The second one is used to really solve the problem i.e. looking for one solution. It returns a Boolean:

- Boolean.TRUE if the problem does have a solution
- Boolean.FALSE if it does not

```

if (nQueens.solve() == Boolean.TRUE) {
System.out.println("the problem has at least one solution");
}
else {
System.out.println("the problem has no solution");
}

```

**Warning:**

The `solve` method may return `null` if the search has been interrupted because of a time or backtrack limit set up for the considered problem.

**FIXME ():**

Write a tutorial about search to be referred to in the previous warning.

The third one is used to compute all the solutions of the problem.

```

nQueens.solveAll();
System.out.println("the problem has " + nQueens.getSolver().getNbSolutions()
+ " solutions");

```

**Exploring the set of solutions**

Choco allows the user to print or explore the set of solutions. Using the `solve` method leaves the variables with the values of the computed solution if it exists.

```

nQueens.solve();
System.out.println("One solution is:");
for (int i = 0; i < queens.length; i++) {
System.out.println(queens[i] + " = " + queens[i].getValue());
}

```

**Note:**

Another way of accessing the variables of the problem is to use the following code:

```

for (int i = 0; i < nQueens.getNbIntVars(); i++) {

```

## *tut\_base*

```
System.out.println(nQueens.getIntVar(i) + " = "
+ nQueens.getIntVar(i).getValue());
}
```

Choco can also be used to enumerate the set of solutions for a problem. The first step is to solve the problem, and then to enumerate the solutions using the `nextSolution()` method.

```
if (nQueens.solve() == Boolean.TRUE) {
do {
for (int i = 0; i < nQueens.getNbIntVars(); i++) {
Systemm.out.println(nQueens.getIntVar(i) + " = "
+ pb.getIntVar(i).getValue());
}
} while (pb.nextSolution() == Boolean.TRUE)
}
```

Finally, at any time the `nQueens.getSolver().getSearchSolver().maxNbSolutionStored` (5 by default) solutions are accessible in the `nQueens.getSolver().getSearchSolver().solution` array (the cell number 0 containing the last computed one).

```
System.out.println("the current solution is: ");
for (int i = 0; i < nQueens.getNbIntVars(); i++) {
System.out.println(nQueens.getIntVar(i) + " = "
+ nQueens.getIntVar(i).getValue());
}

Solution previousSolution =
nQueens.pb.getSolver().getSearchSolver().solutions;

System.out.println("the preVIOUS solution was: ");
```

```
for (int i = 0; i < nQueens.getNbIntVars(); i++) {  
    System.out.println(nQueens.getIntVar(i) + " = "  
+ previousSolution.getValue(i));  
}
```

At the end of this tutorial, we are now able to model, solve and explore the set of solutions on the n-queens problem.