Exercise: Model the "Dining Philosophers Problem" in Alloy
Author: Martin Monperrus
Version: 24. Sep. 2012
URL: http://www.monperrus.net/martin/exercise-alloy-dining-philosophers.pdf

You have to model in Alloy the "Dining philosophers problem"[1] so as to prove that deadlocks can not appear under certain conditions.

The Dining philosophers problem is a synchronization problem. It consists of n philosophers around a dinner table, with one fork between each pair of adjacent philosophers. All philosophers want to eat, but require 2 forks for doing so. The synchronization risk is a deadlock where each philosopher holds one fork and waits for taking the other.

The basic Alloy model of the problem can be expressed as follows:

```
sig Philosopher {
  right: Fork,
  left: Fork
}

sig Fork {}
```

The goal of this assignment is to express in Alloy all constraints associated with the model, to express the deadlock predicate, to express an allocation strategy and to verify that this strategy prevents the apparition of deadlocks.

Question 1: Add facts or predicates to specify that there is only one table of philosophers, with one fork between each pair of adjacent philosophers. Generate a couple of instances with different numbers of instances to check the validity of your code.

Let us now model the state of the system as follows:

```
sig State {
  allocation: Philosopher -> set Fork
}
```

Question 2: Express the following predicates:

```
// is true if philosopher p can eat in this state
// I.e. if he has two forks in hand
pred canEat[p:Philosopher, s:State] {
  ....
}
```

```
// is true if philosopher p can take a fork nearby him,
// I.e. if one of the forks is available
pred canTakeAFork[p:Philosopher, s:State] {
   ...
}
```

1see http://en.wikipedia.org/wiki/Dining_philosophers_problem

Question 3: Express what is valid state: forks are not shared and philosophers hold only nearby forks.

```
pred isValid[s:State] {
  ....
}
```

Question 4: Express in Alloy what is a deadlock.

```
pred deadlock[s:State] {
  ....
}
```

Question 5: How to find an instance of a deadlock?

Question 6: We now want to find a criterion to avoid deadlocks. A simple one is the "waiter" solution: the waiter monitors the whole table and check that not all forks are taken at the same time. Write the corresponding predicate:

```
pred waiterAgrees[s:State] {
  ....
}
```

Question 7: Write the assertion to verify the "waiter" solution.

```
assert checkIfWaiterRuleIsOK  {
  ....
}

check checkIfWaiterRuleIsOK for 10 but 1 State
```