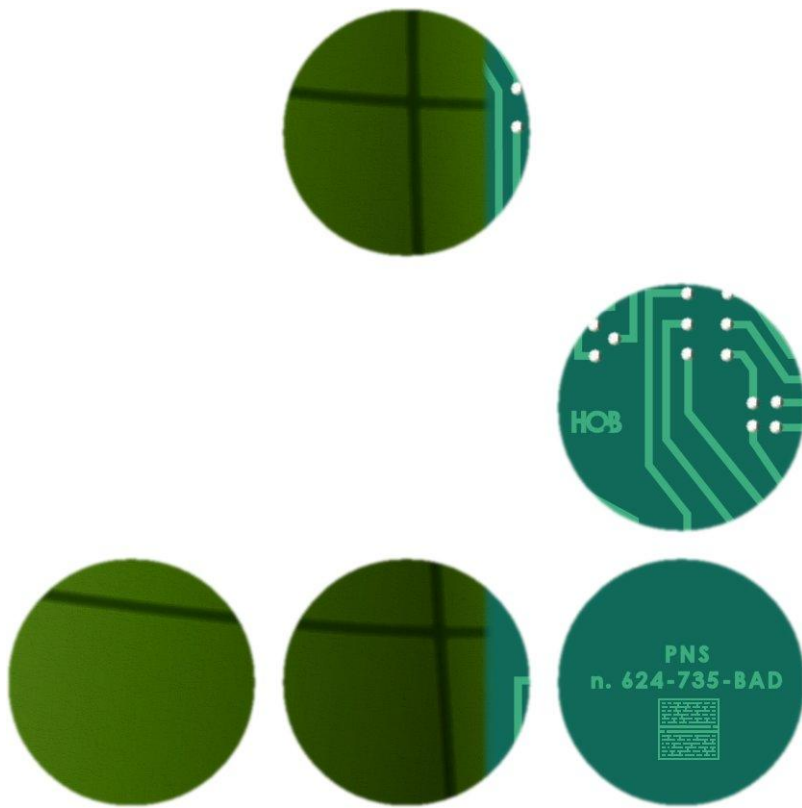


Do you want to play with Apoptosis?  
Here it is a background written for you.

# The pseudo-hacker

Renato Mite



**When the going gets geek,  
the geek gets going.**

**Renato Mite**

# **THE PSEUDO-HACKER**

**English game Kit**

**[The background](#)**

**[How to play](#)**

**[Manual of pseudocode](#)**

**[Example of algorithm](#)**

**[Matt's notes](#)**

**[Matt's solution](#)**

**© Renato Mastrulli  
All rights reserved**

## The background

In the first section of **Apoptosis**, the hacker Matthew Jaws has been kidnapped because he can manipulate P.A. Net<sup>1</sup> and hit anybody uses a **PNS**<sup>2</sup>. The kidnappers want Matt to kill a fellow of theirs, Edward Gortins, that is currently locked up in prison. Even prisoners are subject to health check with PNSes connected to HOB's net. Matt has breached the prison computer system and has sent some shocks to the hearth of the man in order to not make kidnappers suspicious. Now the health of Gortins is precarious and Matt buys time.

The kidnappers fear that Gortins reveals their dirty business any minute now and they do not want to wait anymore, they decide to kill the man in the prison infirmary the old way. Matthew has eavesdropped the intentions of the kidnappers when one of them, the stocky one, made a phone call while he was in the bathroom. Before being locked up, Matt sends you his **notes** and a message:

**"I have to save the life of a man, I have to write an algorithm to signal a false health alarm using his PNS. Look at my notes. I'll be back online in less than twenty minutes, I'll have little time and I need a working algorithm."**

Matt is thinking about a solution, but he is also preparing his escape, he has little time. Before escaping he wants to access again the server he uses to manipulate P.A. Net and wants to save Edward Gortins drawing doctors' attention to him so the man cannot be approached by the killer.

### Do you want to help him?

---

<sup>1</sup> P.A. Net stands for Public Anamnesis Net, the net of HOB company with which they gather and handle health information for the PNS users.

<sup>2</sup> Here PNS stands for Pathoneuroscope, that is a device that performs the analysis of human health status through body neural system.

## How to play

The solution of this game is an algorithm, that is a sequence of instructions to perform an action or to resolve a problem, in this case to **mislead a PNS**.

Writing instructions in **pseudocode** means writing **phrases that describe the operations to do step by step** like "shift into first gear, turn the key, [etc.]". Writing pseudocode **as programmers** means writing instructions such those of a programming language and you will see how it is easy in the short [manual of pseudocode](#).

The algorithm can contain comments and be accompanied by a **short description** that explains its functioning.

[Matt's notes](#) will give you all the necessary **information** and his **tips** to write the operations **that will set off the alarm**.

Now **choose how to play**.

**Challenge Matt** and write an algorithm no longer than 50 lines of code, excluding comments, and, if you want, a description no longer than 2000 characters that explains it.

Then you can confront yourself with his solution at the end of the kit.

**Challenge your friends**, you all choose how to write the algorithm, how many lines of code to write and even how much time you got, choose if the winner is the one who comes up with the most original solution, the solution most similar Matt's or, comparing them, the solution that would work better.

**Challenge everybody else**, share your solution with the **hashtag #aphgame** and confront yourself with other players.

**Which is your challenge?**

# Manual of pseudocode

When you read the rules of a game, you are reading pseudocode, that is a series of **phrases describing what to do**. In the same way, you can write instructions to tell Matt how set off the false alarm.

As an example, here it is an algorithm that tells what to do driving a car when approaching a traffic light.

```
function atTrafficLight()  
  if light is red  
    slow down  
    stop  
  if otherwise light is yellow and not flashing  
    slow down  
    stop  
end-if  
wait for green light or right of way  
# at this point the car can pass  
pass  
end function
```

Believe it or not, that's pseudocode.

Writing pseudocode is funnier than resolving problems for Math lessons and simpler too. The manual establishes the conventions to follow Matt's notes and write instructions that interact with the PNS. So the manual allows to compare your solution with the solution by Matt and by other players.

If you studied a little of Math and you know what a car is, it will be easy to understand the examples and the function `atTrafficLight()` written at end of the manual.

## Comments

Each row of an algorithm is an instruction to be executed, whereas phrases preceded by # (hash mark) are not instructions but indications that the programmer uses to make the algorithm clearer.

Example:    # This is a comment, not an instruction of the algorithm

## Values, Constants, Variables

For simplicity, the **values** to be used are only:

- numeric (`number`)
- boolean (`bool`) namely a value that is `true` or `false`
- null (`null`), object, array: these ones will be explained later.

The **constants** are names assigned to values that do not change in order to identify them in a more comprehensible way.

The **variables** are containers where you keep the values with which you do actions or calculations, like for example the tank of a car that keeps the fuel with which you make the engine run.

The name of variables and constants consists only of letters, numbers and symbols hyphen (-) and underscore (\_); the name of variables starts with a letter; the name of constants starts with the symbol underscore (\_).

To create a constant, you could write a phrase like this one:

```
create constant number _tankCapacity with value 50
```

that means assigning the name `_tankCapacity` to numeric value 50 (litres), but programmers use more concise instructions, like this one:

```
const number _tankCapacity = 50
```

The convention for the constants is: "const", the type of value, the name, the assignment operator (=) and the value in this order.

Variables can contain a value from the beginning or not. The convention for the variables is: "var", the type of value, the name, then the assignment operator (=) and the value follow if they are present, in this order.

```
var number liters-to-buy
```

```
var number km-to-go = 35
```

```
liters-to-buy = 10
```

## Objects, Arrays, Properties, Functions

Variables can contain also an object or an array of objects. As in the reality, an **object** is an entity that has constants, characteristic values (called properties by convention) and actions to do or undergo (called functions).

For example the car is an object, if we establish by convention that `car` is the type that defines a car, we can create a variable for this object.

```
var car kitt
```

Which `car` does `kitt` represent? The supercar? As it is written it does not represent any car, therefore it has a null value (`null`). Instead in this way:

```
kitt = MyCar
```

the variable `kitt` represents (contains) for example your car.

The **properties** of an object are variables that belong to it with which you can **get** (**get**) its values, for example the year of registration, or even **set** (**set**), for example the number of litres in the tank.

The **functions** are algorithms that can act with or without other values, can return a result or do an action without returning a result (`null` functions).

In the notes, **Matt** describes the objects and arrays you can use with their respective functions, specifying the required values and type of result, and their respective properties, specifying the type of value, if you can get (**get**) and set (**set**) it.

The belonging of constants, properties and functions to an object is indicated with . (dot) after object's name; after the name of a function there are round brackets that eventually encase the values required. The constants can be used with the name of the type too.

Examples:    `object.property`                      `object.function()`  
                 `object._constant`                `type._constant`

Imagine a `car` object that has a property `numberOfGears` with number of available gears and a function `shiftGear` that shifts into the gear specified, here they are the instructions to get the number of gears and shift into first gear

```
var car kitt = MyCar
var number gears
gears = kitt.numberOfGears
kitt.shiftGear(1)
```

How do you know which type of property `numberOfGears` is?

You need to read the definition of the property.

Here it is an example of what you would find in the notes:

```
number get prop numberOfGears
```

The property gets the number of gears of the car.

How do you know which values the function `shiftGear` requires?

You need to read the definition of the function in the notes.

As an example, for the `shiftGear` you would find:

```
bool func shiftGear (number gearNumber)
```

The function shifts into the gear indicated by `gearNumber` and returns true if the gear is engaged.

An **array of objects** is like a filing cabinet that gathers objects of the same type and keeps them in order.

For example you create an array of objects `wheel` that can contains 5 objects in this way:

```
var wheel wheels[5]
```

The objects contained in an array are retrieved with the **index inside square brackets**, that is the number of position in the order, or, when it will be specified, with an **identification number inside round brackets**.



Going back to the example of the car, the objects `wheel` are gathered in the array `wheels` that belongs to the car. Every object `wheel` has its properties, for example `pressure` that gets or sets the value of wheel's pressure.

Do you want to know the pressure of the first wheel? You can write:

```
var car kitt = MyCar
var number first-wheel-pres
first-wheel-pres = kitt.wheels[0].pressure
```

Why 0? Because the index starts from 0 by convention.

If you find an indication like this one in the notes:

```
wheels          array of wheel [] (number serial)
```

it means that you can interact with a wheel also with its serial number:

```
var car kitt = MyCar
var number p
p = kitt.wheels(76450927).pressure
```

This is useful if the tire dealer tells you to check the pressure of the wheel with serial number 76450927 because it could belong to a defective lot and you do not know the index.

Even the arrays can have properties and functions, among them there is the property **length** that gets the number of objects it contains.

As an example, with `car.wheels.length` you get the number of wheels that a car has, in this case you know that it is five (the spare one is in the trunk), but when you do not know how many objects an array contains and you want to know the index of the last item, you can calculate it easily.

Example:    `var number last-wheel-index`  
              `last-wheel-index = (kitt.wheels.length - 1)`

## Operations

To solve a problem, often you need to do operations and therefore you need some operators. The mathematical operators such as addition (+), subtraction (-), etc. are well known, in programming languages there are other operators and there are only round brackets to group and sort operations.

Here the operators available in the game are described.

### Assignment operator

This is the operator seen until now for constants, variables and properties.

$x = a$       Assigns to  $x$  (constant, variable, property) the value of  $a$  (value, constant, variable, property, result of a function, result of another operation).

### Mathematical operators

$a + b$       Calculates the sum of the values  $a$  and  $b$

$a - b$       Calculates the difference between the value  $a$  and the value  $b$

$a * b$       Calculates the multiplication of the values  $a$  and  $b$

$a / b$       Calculates the integer quotient of the division between the values  $a$  and  $b$

$a \% b$       Calculates the remainder of the division with integer quotient between the values  $a$  and  $b$

$a // b$       Calculates the decimal quotient of the division between the values  $a$  and  $b$

$a ^ b$       Calculates the power of  $a$  raised to  $b$

### Examples:

```
var number liters-to-buy
```

```
var number cost
```

```
liters-to-buy = kitt._tankCapacity - kitt.liters-in-tank
```

```
cost = liters-to-buy * 1.80
```

There is a variant of mathematical operators that allows you to assign the result of the operation to the variable that appears before the operator.

For example, if you want to buy 3 liters more and put them in a tin, you can calculate in this way:

liters-to-buy = liters-to-buy + 3

or in this way:

liters-to-buy += 3

### Mathematical operators with assignment

- $a += b$  Assigns to  $a$  (variable or property) the sum of the values  $a$  and  $b$
- $a -= b$  Assigns to  $a$  (variable or property) the difference between the value  $a$  and the value  $b$
- $a *= b$  Assigns to  $a$  (variable or property) the multiplication of the values  $a$  and  $b$
- $a /= b$  Assigns to  $a$  (variable or property) the integer quotient of the division between the values  $a$  and  $b$
- $a \% = b$  Assigns to  $a$  (variable or property) the remainder of the division with integer quotient between the values  $a$  and  $b$
- $a // = b$  Assigns to  $a$  (variable or property) the decimal quotient of the division between the values  $a$  and  $b$
- $a ^ = b$  Assigns to  $a$  (variable or property) the power of  $a$  raised to  $b$

### Comparison operators

These operators compare two values and return a **bool** value `true` if the comparison is correct, otherwise they return `false`.

- $a == b$  Returns `true` if  $a$  is equal to  $b$
- $a < b$  Returns `true` if  $a$  is less than  $b$
- $a <= b$  Returns `true` if  $a$  is less than or equal to  $b$
- $a > b$  Returns `true` if  $a$  is greater than  $b$
- $a >= b$  Returns `true` if  $a$  is greater than or equal to  $b$
- $a <> b$  Returns `true` if  $a$  is different from  $b$

## Logical operators

The logical operators act on **bool** values and return a **bool** value, they can be written with a word or a symbol.

*a* and *b*    *a* & *b*    Returns `true` if both *a* and *b* are `true`

*a* or *b*    *a* | *b*    Returns `true` if at least one among *a* and *b* is `true`

*a* xor *b*    *a* ' *b*    Returns `true` if only one among *a* and *b* is `true`

not *a*    ! *a*    Returns `true` if *a* is `false`, `false` if *a* is `true`

You will need comparison and logical operators to make choices that determine the path of an algorithm.

### Instruction IF

The instruction IF verifies a condition with **bool** value and executes one or more instructions if the value of condition is `true`.

```
if condition
    instruction(s)
end if
```

The instructions to be executed belong to block **if** and are written between **if** and **end if** with an indentation.

If you want to execute instructions also when the verification fails, you can add an **else** before **end if**.

```
if condition
    instruction(s) block if
else
    instruction(s) block else
end if
```

Between block **if** and block **else**, it is possible to add one or more blocks **else if** that verify other conditions and execute the instructions inside them if the verification succeed. If a condition returns `true`, the instructions of that block are executed and the next conditions are not verified. The instructions of block **else**, if it is present, will be executed only if all verifications fail.

```
if condition1
    instruction(s) block 1
else if condition2
    instruction(s) block 2
else if condition3
    instruction(s) block 3
...
else if conditionN
    instruction(s) block N
else
    instruction(s) block else
end if
```

If a block contains only one instruction to execute, this instruction can be written after the verification placing the keyword **do** before.

```
if condition do instruction
else if condition1 do instruction
else do instruction
```

If the last block, whatever it is, is written with the keyword **do**, **end if** is omitted.

Example: Did the tire dealer say you to check the pressure of a wheel and if it is low, to change the wheel because it belongs to a defective lot? Here they are the instructions:

```
var car kitt = MyCar
var number p
p = kitt.wheels(76450927).pressure
if (p < wheel._normalPressure) do kitt.changeWheel(76450927)
```

### Instruction CHECK

The instruction CHECK is similar to instruction IF but compares one value with others and if a comparison is `true`, executes the instructions of its block and stops comparing.

```
check a
is j
    instruction(s) block 1
is k
    instruction(s) block 2
...
is z
    instruction(s) block N
else
    instruction(s) block else
end check
```

After the keyword **is** there is a value *j*, *k*... *z* (value, constant, variable, property, result of a function, result of another operation) to be compared with the value of *a* (value, constant, variable, property, result of a function, result of another operation). If *a* is equal to one of the values, the relative block of instructions is executed and no other next comparison is verified.

An instruction **is** can verify the equality of *a* with more values separated by a comma:

```
check a  
is w, x, y  
    instruction(s) block 1  
is z  
    instruction(s) block 2  
end check
```

In this case the block of instructions is executed if *a* is equal to one of the values listed.

You can insert comparison operators different than equality between **is** and the value, and more comparisons separated by a comma.

```
check a  
is < j  
    instruction(s) block 1  
is k, <= s  
    instruction(s) block 2  
...  
is >= z  
    instruction(s) block N  
else  
    instruction(s) block else  
end check
```

In this case the block of instructions is executed if one of the comparisons after **is** returns `true` and no other next comparison is verified.

The instructions of block **else**, if it is present, will be executed if no comparison is `true`.

## Instruction FOR

The instruction FOR creates a loop that executes a block of instructions more times changing a numeric variable in a range of values.

```
for  $a = b$  to  $c$   
    instruction(s)  
end for
```

The variable  $a$  gets the value of  $b$  and the block of instructions is executed each time, adding one unit (1), until the variable  $a$  contains the value  $c$  in the last execution.

You can specify the value that changes the variable  $a$  with keyword **go**:

<pre><b>for</b> <math>a = b</math> <b>to</b> <math>c</math> <b>go</b> <math>x</math>     <i>instruction(s)</i> <b>end for</b></pre>	<pre><b>for</b> <math>a = b</math> <b>to</b> <math>c</math> <b>go</b> <math>-x</math>     <i>instruction(s)</i> <b>end for</b></pre>
---	--

If the value of **go** is positive or omitted (that is equal to 1), the value  $b$  must be less than value  $c$ , if  $b$  is equal to  $c$  the instructions are executed only one time, if  $b$  is greater than  $c$  the instructions are never executed.

If the value of **go** is negative, the value  $b$  must be greater than value  $c$ , if  $b$  is equal to  $c$  the instructions are executed only one time, if  $b$  is less than  $c$  the instructions are never executed.

You can choose to interrupt a loop **for** with keyword **break** put inside the block of instructions:

```
for  $a = b$  to  $c$   
    instruction(s)  
    break  
    instruction(s)  
end for
```



If the variable  $a$  has already been created, after the loop it will contain the last value it got, or the value that it had if the instructions have not been executed. If the variable  $a$  has not been created, it will be created by the instruction **for** and it exists only within the loop.

As an example, the instruction **for** can be used to check all the wheels of the car and inflate them if necessary.

```
var number p
for i = 0 to (kitt.wheels.length - 1)
  if (kitt.wheels[i].pressure < wheel._normalPressure)
    p = wheel._normalPressure - kitt.wheels[i].pressure
    kitt.wheels[i].inflate(p)
  end if
end for
```

### Instruction FOR EACH

The instruction FOR EACH is a variant of the instruction FOR that you can use to execute a block of instructions for each object of an array.

```
for each  $a$  in  $b$ 
  instruction(s)
end for
```

The instruction **for each** executes the instructions in it for each item of array  $b$  according the index order, assigning each time the item to variable  $a$ . If the array is empty, the instructions are not executed.

If the variable  $a$  has already been created, after the loop **for each** it will contain the last value it got, or the value that it had if the instructions have not been executed.

If the variable *a* has not been created, it will be created by the instruction **for each** and it exists only within the loop.

Therefore the previous example can be rewritten in this way:

```
var number p
for each r in kitt.wheels
  if (r.pressure < wheel._normalPressure)
    p = wheel._normalPressure - r.pressure
    r.inflate(p)
  end if
end for
```

### Instruction LOOP

The instruction LOOP creates a loop that executes a block of instructions zero, one or more times according to a **bool** value specified.

<b>while</b> <i>a</i>	<b>loop</b>
<i>instruction(s)</i>	<i>instruction(s)</i>
<b>break</b>	<b>break</b>
<b>loop</b>	<b>while</b> <i>a</i>

The keyword **while** analyzes a **bool** value *a* (value, constant, variable, property, result of a function, result of a logic operation or comparison), the loop continues if the instruction **while** finds *a* equal to `true` and stops with keyword **break** or when **while** finds *a* equal to `false`. If *a* is already `false`, the instructions of the block are executed one time if **while** is written at end of block, otherwise the loop is not executed.

**until** *a**instruction(s)***break****loop****loop***instruction(s)***break****until** *a*

The keyword **until** analyzes a **bool** value *a* (value, constant, variable, property, result of a function, result of a logic operation or comparison), the loop continues if the instruction **until** finds *a* equal to `false` and stops with keyword **break** or when **until** finds *a* equal to `true`. If *a* is already `true`, the instructions of the block are executed one time if **until** is written at end of block, otherwise the loop is not executed.

### Functions and instruction RETURN

As already seen, the result of a function is assigned to a variable with the assignment operator (=):

```
x = function()
```

Functions that are not `null` return a value with the instruction RETURN.

#### Example:

```
number func calcFillUpCost(car a, number price-per-liter)
  var number liters-to-buy
  liters-to-buy = (a._tankCapacity - a.liters-in-tank)
  return (liters-to-buy * price-per-liter)
end func

var car kitt = MyCar
var number cost
cost = calcFillUpCost(kitt, 1.80)
```

### Example of algorithm

Here the function `atTrafficLight()` written using the objects `car` and `trafficLight` with constants, properties and functions described in the notes below.

#### Notes

`car`

`bool get prop rightOfWay`

Gets a value that indicates if the car has the right of way.

`bool get prop running`

Gets a value that indicates if the car is running.

`null func go()`

It makes the car go.

`null func slowdown()`

It slows down the car.

`null func stay()`

The car stays still.

`null func stop()`

It stops the car.

`trafficLight`

`_none = 0`      # Constant that represents none of the lights

`_green = 1`      # Constant that represents the green light

`_yellow = 2`      # Constant that represents the yellow light

`_red = 3`      # Constant that represents the red light

`number get prop light`

Gets the value of the light currently on (see constants).

`bool get prop flashing`

Gets a value that indicates if the light is flashing.

### Algorithm

```
null func atTrafficLight (car a, trafficLight s)
    until ((s.light == s._green) or (s.light == s._none))
        if a.running          # Car arrives at the traffic light
            check s.light
                is s._yellow
                    if s.flashing do break
                    a slowdown()
                    a.stop()
                is s._red
                    a slowdown()
                    a.stop()
            end check
        else do a.stay()      # Stays still at traffic light
    loop
    # Check right of way
    until a.rightOfWay
        a.stay()
    loop
    a.go() # Passes
end func
```

### Description

The function enters in a loop if the light of traffic light x is yellow or red, in the loop it stops the car y or keeps the car still until the light change. It interrupts the loop if the light is yellow and flashes.

When the traffic light is off or is flashing, when the light is or becomes green, the execution proceeds up to the loop that verifies the right of way and the car passes as soon as it has the right of way.

## Matt's notes

I need a function `falseAlarm` with this definition:

```
null func falseAlarm (PNSDevice pns, UserBody body)
```

where I pass you the objects *pns* and *body*, the first is an object `PNSDevice` that represents the PNS worn by the man I want to save, the second is an object `UserBody` that represents the man and his body values.

Here it is what I know about the HOB system:

### **BodyParam**

It represents a vital parameter of human body and it is used to retrieve data from PNS.

There are many of them, but so far I know only these constants that identify (see property `id`) the corresponding vital parameters:

<code>_HeartRate = 9</code>	<code># Heartbeat</code>
<code>_BloodPressure = 13</code>	<code># Blood pressure</code>
<code>_LungsRate = 15</code>	<code># Respiratory rate</code>
<code>_LiverStatus = 206</code>	<code># Liver general status</code>
<code>_Temperature = 332</code>	<code># Body temperature</code>
<code>_BladderStatus = 674</code>	<code># Bladder general status</code>

Properties:

number get prop id

Gets the number that identifies the parameter.

number get prop criticalTolerance

Gets the current tolerance of the body for the vital parameter. If the value of vital parameter is near to the critical value inside this tolerance (see `criticalValue` and `insideTolerance`), the value can become critical.

number get prop criticalValue

Gets the critical value for the vital parameter. If the vital parameter is at critical value, the PNS deems the parameter at risk.

bool get prop isDangerous

Gets a value that indicates if a critical value can be dangerous.

For all the parameters I know, except `_BladderStatus`, the property `isDangerous` returns `true`.

Functions:

bool func insideTolerance(number value)

Returns `true` if the specified value is inside the range of critical tolerance.

## UserBody

params array of BodyParam [] (number id)

Array of vital parameters, you can retrieve them by index or identifier (see `BodyParam`). `params` has only the property `length`.

bool get prop danger

Gets a value that indicates if the body is in danger. This property returns `true` and the **PNS sets off an alarm** with dangerous vital parameters (`isDangerous == true`) in one of these combinations:

- at least three parameters reach the critical value

(see `PNSDevice.dangerParamCount`)

- two parameters reach the critical value and at least six parameters remain inside critical tolerance for at least 20 readings of data (see `BodyParam.insideTolerance` and `PNSLog.average`).

## PNSDevice

number get prop dangerParamCount

Gets the number of vital parameters that currently have reached the critical value.

## PNSLog func getLog(number paramID)

Returns an object `PNSLog` that interacts with the vital parameter specified by `paramID`.

# PNSLog

It represents a container of data that interacts with a vital parameter.

values     array of number []

Array of all values read, stored in reverse chronological order, the most recent values have lower index, so `values[0]` is always the last value read. `values` has the property `length` and the function `average()` that returns the average of all the values.

number get prop lastValue

Gets the last value read, it is equal to `values[0]`.

bool get prop steady	*1	*2 (before)
----------------------	----	-------------

Gets `true` if the value of parameter is stable, `false` if it is changing.

```
bool get prop goingCritical      *1    *2 (before)
```

Gets `true` if the value of parameter is approaching the range of critical tolerance.



`bool get prop insideCritical` \*1 \*2 (before)

Gets `true` if the value of parameter is inside the range of critical tolerance.

`bool get prop isCritical`

Reads a new value and gets `true` if the value is equal to the critical value.

`number func average(number n)`

Returns the average of the last `n` values stored in the log.

`null func readNewValue()`

Performs a new reading and stores the value of the vital parameter.

`null func storeValue(number value, number pass)` \*2 (after)

Stores a value in the log as if it had been read from the body, to call this function you need a numeric pass.

- \*1 The value of property (`true` or `false`) is determined according to the average (see `PNSLog.average`) of the last 20 values read.
- \*2 Each time these properties and functions are used, the PNS performs a reading of the parameter value automatically, before or after using them, so it inserts the value in the array `values` and updates the status of the parameter that, as already said for \*1, is determined by the average of last values read.

Bug: Readings repeated in a short time store values higher than the real values.

**Tips:** use the function `readNewValue` to push vital parameters to their critical value, repeated readings store higher values but they also affect the health of the man pushing it to a critical status for real, therefore do not go too far with the parameter `_HeartRate`, the heart of the man is beaten-up.

I have a pass `_tmp-pass` for the function `storeValue`, you can use it only one time because after that it is not valid anymore.

Matt is in one of those situations in which there is not just one solution.

Every programmer knows that an algorithm can be written in different ways or can be improved, therefore do not think of Matt's solution but write your own solution and have fun.

## You are ready to play.

Use the hashtag **#aphgame** to share your solution,  
talk about the game, challenge your friends...

## Did you have fun?

Tell me

Twitter [@renatomite](https://twitter.com/renatomite)

GooglePlus [+Renato Mite](https://plus.google.com/+RenatoMite)

**Turn the page and see Matt's solution.**

### Algorithm

```

null func falseAlarm (PNSDevice pns, UserBody body)

    var number avg
    var number gap
    var number num = 10000
    var number aid = -1
    var PNSLog logs[4]
    var number count
    var number pass = _tmp-pass
    # search for a dangerous parameter at risk
    for each param in body.params
        if param.isDangerous and (param.id <> BodyParam._HeartRate) and →
→   (param.id <> BodyParam._LungsRate) and (param.id <> BodyParam._BloodPressure)
            logs[0] = pns.getLog(param.id)
            avg = logs[0].average(20)
            gap = (param.criticalValue - avg)
            if (gap <= param.criticalTolerance) # gap is inside the range of critical tolerance
                if (gap <= 0) # average greater than the critical value
                    aid = param.id
                    break # exits the loop and uses this parameter
                else if (gap < num)
                    num = gap
                    aid = param.id # uses this parameter if it does not find a better one
                end if
            end if
        end if
    end for
    if (aid < 0) do aid = BodyParam._Temperature
    # pushes the parameters to their critical value
    num = 0

```

```

logs[0] = pns.getLog(BodyParam._HeartRate)
logs[1] = pns.getLog(BodyParam._LungsRate)
logs[2] = pns.getLog(BodyParam._BloodPressure)
logs[3] = pns.getLog(aid)
loop
  count = 0
  logs[1].readNewValue()
  logs[2].readNewValue()
  logs[3].readNewValue()
  if logs[1].isCritical do count += 1
  if logs[2].isCritical do count += 1
  if logs[3].isCritical do count += 1
  if (count > 1)
    if logs[0].insideCritical and (pass <> -1)
      logs[0].storeValue(body.params(BodyParam._HeartRate).criticalValue * 1.3, pass)
      pass = -1
    end if
  end if
  num += 1
until ((num == 1000) or body.danger)
end func

```

### Description

First the algorithm executes a for each of the vital parameters to search for a parameter whose average is greater than its critical value or the most near possible, then executes a loop that repeatedly reads the values of the parameters `_LungsRate`, `_BloodPressure` and the parameter found (in absence of that, uses `_Temperature`) to exploit the bug that stores higher values. When in the loop at least two of the parameters read are critical, it writes in the log of `_HeartRate` a value 30% greater than the critical one if the parameter is inside the range of critical tolerance. In this way at least three parameters should have reached the critical value and the PNS should set off the alarm. The loop ends when the PNS sets off the alarm or after 1000 iterations, because if it continued indefinitely it could kill the man before his killer does.

# Get to know Apoptosis

**Apoptosis is my first book, a sci-fi novel that I usually introduce in this way:**

A medical researcher, a revolutionary diagnostic innovation, a digital network for public health and a hacker calling everything into question... before the Apoptosis.

## PLOT

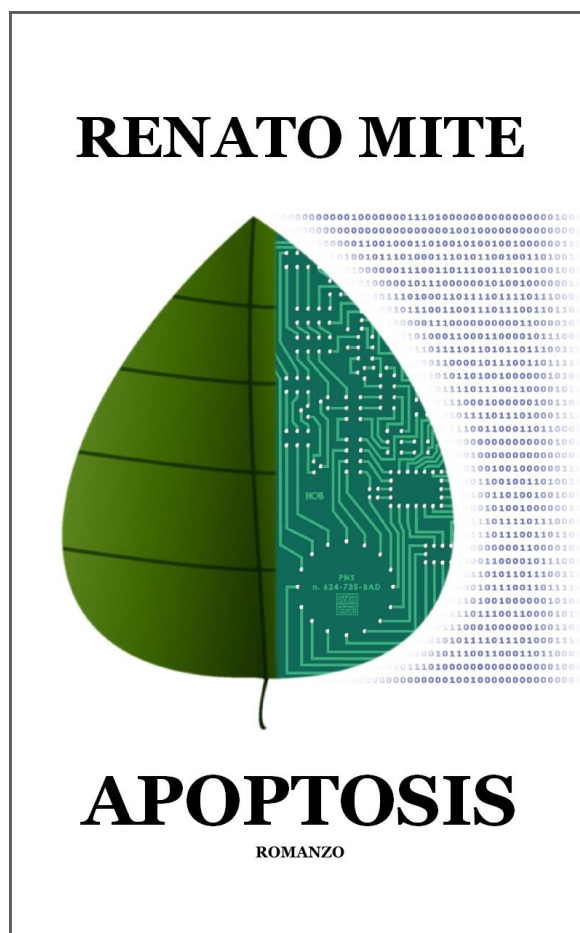
The company HOB Medicines revolutionised medicine with the Pathoneuroscopy, the diagnostic investigation through body neural system, and created the PNS, a device for personal diagnosis connected to the P.A. Net, the digital network for public health.

Everybody wears a PNS, but somebody doubts its efficiency: George Tobell, the researcher that led the way for Pathoneuroscopy, affected by a neuropathy caused by a prototype of PNS; and Matthew Jaws, an hacker obsessed by HOB which wants to get to the bottom of destiny of harbingers, legendary sick persons object of HOB experimentation.

When George dies, Matthew will take possession of his treatise on Pathoneuroscopy that sheds some lights on secrets of HOB.

Matthew gets hired in the control room of the P.A. Net in the HOB Building, there works also the analyst who has made an agreement with the tycoon of HOB to catch the hacker of their net. In the laboratories of the skyscraper works a young researcher, Jason Stemberg, which will discover that anyone using the PNS is in danger.

The paroxysm is imminent and will entangle them all.



**Now Apoptosis is available only in Italian. If you think it is worth a reading, spread the word, a publisher can notice it and have it translated in your language.**

**Thank you.  
Renato Mite**

[www.renatomite.it](http://www.renatomite.it)