PLEASE NOTE
This might help you to do good or better MOHAA scripting, but it is not a scripting tutorial.

FOREWORD
With my recent endeavour into MOHAA scripting, I was quite surprised. I have seen many creative ways to do scripts in the past years, but I have never imagined such a messy code, I have witnessed in MOHAA.

The global, as well as the level scripts are really terrible examples of scripting. With this Document I will try to walk you trough my idea of a good script code in MOHAA. It will include many of the basics you probably are aware of, but non the less, you should take a moment to see if I can inspire you just a little.

THANK YOU
This tutorial got improved with the input of others, that shared their opinion and advice with me. Thanks to x-null forums users: 1337Smithy, DoubleKill, James, Purple Elephant, RyBack Others that provided input and helped: Criminal, Daggolin, NeMs, Todesengel

INDEX
VARIABLES
CVARS
ENTITIES
MAIN
BRACKETS
USING HEADER AND QUOTES
INDENTING
TYPE CASTING
NULL AND NIL
NO LOCAL STRINGS
USE THE SDK DOCUMENTATION
ASK AND LEARN FROM OTHERS
DO NOT USE GOTO
ENTITY FLAGS
EACH FRAME
USE VARIABLES OVER EVENT CALLS
NAMEING
WAIT FOR PLAYER
WAIT FOR WARMUP TIME
LOOPS
SWITCHES
SELF
PARM
EXTERNAL SCRIPT THREADS
RETURN VALUES
END
VARIABLES (local.)
There are different kinds of variables in mohaa, which are working in different Zones. I will try to give you a few examples what that means.

local.var = "some data"
Defines a local variable, which is valid only in the local Zone it was initialized in. A local zone can be a script file, a function or a Statement block.
I strongly recommend you always use local-variables unless you need global/level wide access. Using local-variables reduces the chance of naming conflicts with variables used elsewhere.

level.var = "some data"
Defines a level wide variable, which is valid across all scripts, for the duration of the current map. No matter where it is first initialized. This is very useful, if you need to access various data from different scripts, while a map or mission is active.

game.var = "some data"
Defines a global game wide variable, which is valid across all scripts, for the duration of the current running game, even across levels. So if you want to retain data across multiple levels or until the server is ended, this is the kind of variable to use. This is used to track certain advancements during single player. Be thoughtful which name you give such a game variable, to prevent naming conflicts with other game variables.

```
game.foundSecrets = 0 //global, all scripts, until server shutdown
level.treesHugged = 0 //global, all scripts, until level change
local.fishCaught = 0 //local, this script file, until level change

main:
    local.insideMain = 1 //local, valid only inside main
    if (local.insideMain == 1)
        local.firstBlock = 1
        if (local.firstBlock == 1)
            local.firstBlock = 2 //valid
            local.insideMain = 2 //valid
            game.foundSecrets = 1 //valid
            level.treesHugged = 1 //valid
            local.fishCaught = 1 //valid
        } //end second block
    } //end first block
    local.insideMain = 3 //valid
}end //end main
```

CVARS (getCvar("sv_gravity"))
Cvars are Configuration-variables, they can be permanently stored in the configuration file of a server/player. They remain in the configuration file until they are removed by a user or a specific console command. Shutting down the server will not remove a cvar from the configuration file.

In the code below you can see examples how to get and set cvar values by script.

```
local.gravity = int( getCvar("sv_gravity") ) //get cvar from server.cfg
setCvar("sv_gravity" "30") //set cvar to server.cfg
```

You should only use Cvars if you want to preserve data for the next server start. There are limits. Cvars have a length limit and the configuration file has a maximum filesize limit right below 8 Kilobyte. You are advised to store as little information as you can. Here is an example:

<table>
<thead>
<tr>
<th>Short and good</th>
<th>To long and wasteful</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCvar(&quot;coop_plH1&quot; &quot;67&quot;)</td>
<td>setCvar(&quot;cooperative_playerOne_HeroOnLastLevel&quot; &quot;67.000&quot;)</td>
</tr>
</tbody>
</table>
ENTITIES ($world)
Entities are level Objects such as players, actors, items, script_objects, health-packs, projectiles, decals, grenades...

Everything that can not be changed in game, like the actual geometry of the level and all static models, are grouped together to one big $world entity. The world entity gives control over weather, fog, physics and a few more other neat things on the level.

Entities are accessible via script by their targetname. You can easily spot them by the $ in front of their targetname.

MAIN (main:)
Every level script and some of the global scripts should contain a function with the name main. This main thread is started by the game as soon as the level is loaded and all entities are spawned (with the exception of clients / players).

The main thread is mostly used to initialize setup threads. These setup threads are used to prepare the scripted sequences, Actors, exploding Objects and other entities on the level. Additional script threads can be started from Script, Triggers or Actors.

The main thread should be kept short and light weighted, so if you have a intense script to run you should start a separate thread.

The example code below shows how a main thread is presented in the level scripts. The dots ("...") are symbolic for any script code placed there.

```plaintext
1 main:
2 ...
3 end
```

NOTE: The main thread in the primary level script is started by the game without any parameters. More about parameters will come later...

BRACKETS ({} )
The biggest issue that MOHAA script code has, is the poor use of brackets. Brackets make it visually clear where a code block starts/ends. I recommend using brackets where ever you can. I use Notepad++ that can collapse and expand code if brackets are used.

Using brackets will make your work much easier, especially on large portions of code. I will give you a few examples, how to make good use of the brackets.

On the left is a example how the code looks when it is expanded, on the right how it looks like when it is collapsed in Notepad++.

If you collapse a few threads you get a pretty nice overview of your script, allowing you to navigate easier in the code.

But this is not all brackets can do, they are not just there to make it collapsable. They also make it clear how the code is suppose to work. Most errors I have seen in MOHAA code are caused, because no brackets are used. Let me give you a example.

1. without brackets

```plaintext
1 doSomething:
2    if( $someAi cansee $player )
3        $someAi attackplayer
4            printlnbold_noloc ( "can see" )
5 }end
```
The code on the first table above seems to be different from the second, but I guarantee you they are exactly the same! Without brackets only the first code line directly below the if statement is effected by the statement. The next line of code with `iprintlnbold_noloc` will always be executed regardless of the result from the if statement.

I will give you another example what consequences this can have.

1. without brackets

```java
local.player = NULL
for( local.i = 1; local.i <= $player.size; local.i++)
    local.player = $player.[local.i]
if( local.player != NULL )
    iprintlnbold_noloc ("player exists")
```

2. with brackets

```java
local.player = NULL
for( local.i = 1; local.i <= $player.size; local.i++ ){
    local.player = $player.[local.i]
}
if( local.player != NULL ){
    iprintlnbold_noloc ("player exists")
}
```

The code without brackets, suggests that it checks all players, but it checks only the last player, upon exiting the for-loop, exactly like the code with the brackets does.

So here is the catch: If this script is tested by the developer it works just fine, the error in this script will only show, if there is more than one player on the server. The Code below shows how it should have been done.

```java
local.player = NULL
for( local.i = 1; local.i <= $player.size; local.i++)
    local.player = $player.[local.i]
if( local.player != NULL ){
    iprintlnbold_noloc ("player exists")
}
```

Brackets allow you also to apply special formatting, which can be better in some cases. If you need a `codeblock` in one Line, like in the example below.

```java
if( local.water = NIL ) { local.water = "wet" }
if( local.rock = NIL ) { local.rock = "hard" }
if( local.fire = NIL ) { local.fire = "hot" }
```

**USING HEADER AND QUOTES ( ) " "**

Round brackets are used to group statements and parameters together. In most cases it does not matter in MOHAA, which is why this can be neglected, however, keep in mind that in some cases this can make a difference.

```java
iprintlnbold_noloc ( "healthonly: " + $player.healthonly ) //right
iprintlnbold_noloc  "healthonly: "+$player.healthonly  //right
iprintlnbold_noloc  "healthonly: " + $player.healthonly //wrong
```
Quotes make it clear where a string starts and where it ends. If a string is made of separate words it could be mistaken for several strings, which could cause different kinds of errors.

I recommend you put your strings always into quotes.

1. local.val = getCvar whatever //wrong
2. local.val = getCvar ("whatever") //right

**INDENTING (->|)**

In this regard the MOHAA script code is exemplary, showing how **NOT** to do it! Indenting code correctly can improve the readability of your script, but it can also do the opposite if it is done incorrectly.

The example code on the left and right is identical in its function, but you will need much more time for reading and understanding the left code.

1. if( local.var == 1 )
   2. iprintlnbold_noloc "You die."
   3. else
   4. if( local.var == 2 ){
      5. iprintlnbold_noloc "You live."
   6. else
   7. iprintlnbold_noloc "Try again."
2. if( local.var == 1 ){
   3. iprintlnbold_noloc "You die."
   4. }
   5. else if( local.var == 2 ){
      6. iprintlnbold_noloc "You live."
   7. }
   8. else{
      9. iprintlnbold_noloc "Try again."

I have seen lots of code in the global script files like this, and I really can not express how bad this is, without swearing for at least 5 minutes.

My advice to you is not to write code like you are stuck between the 80's and early 90's with a Monochrome 480 x 320 Pixel monitor.

**NOTE:** Clean, easy to read code can reduce or even prevent errors.

**TYPE CASTING( int(local.var) )**

In MOHAA you can convert data to a different type. Like string, int or float, which is very useful if you want to compare values stored in Cvars or variables of a different type…

...getcvar will return the data of the Cvar as a string. You need to convert the data if you want it as a integer.

**Examples:** int(local.var), float(local.var), string(local.var)

1. local.thisIsAInteger = int( getCvar ("g_gametype") )

**NULL AND NIL ( NULL NIL)**

NULL and NIL are sort of special values, like ZERO (NULL) and NOTHING (NIL) in mohaa.

Entities that exist are never NULL, so checking against NULL can tell us if a specific entity exists or not.

1. if( $friendly1 != NULL){ ... } //if entity exist

Checking against NIL allows you to find out if a variable is empty or set.

1. if( local.isRaining != NIL){ ... } //if variable has a value

Sometimes you have to check for both, NIL and NULL, if you store entities inside variables. Check them in the right order, as shown in the code below, to get the desired result.

1. if( local.player != NIL & local.player != NULL){ ... } //has value & entity exists
NO LOCAL STRINGS (iprintlnbold_noloc(""))
If you show text to the players, that is not part of the game its default local strings, you should make use of
the command iprintlnbold_noloc. This will prevent the game from printing a error message to the console
and creating a error log of missing local strings.

USE THE SDK DOCUMENTATION
I recommend, that you download and install the MOHAA SDK and use its included documentation, to look
up the script relevant commands.

Make sure you take notes of commands that might be useful to you and test how they work. This will make
the scripting much easier for you in the long run.

ASK AND LEARN FROM OTHERS
Utilize the expiriance and wisdom of others, if you are stuck or unsure how to do it. No need to reinvent the
wheel. Even if you have to wait a day or two for a reply you might save a lot of time in the long run.
Experience and advice from others can be invaluable, especially if they have been where you are now, trying
to solve the exact same problem in their script.

DO NOT USE GOTO (goto)
MOHAA uses labels to which you can jump by using the goto command. The problem with using goto is
that it can make your code very hard to understand.

Use thread calls, for- and while-loops instead of goto.
The bad code on the left is using goto, while the code on the right shows how it should be done.

| 1 | $friendly turnto NULL |
| 2 | giveHealth: |
| 3 | $friendly health (1000) |
| 4 | wait 3 |
| 5 | if($friendly != NULL) |
| 6 | goto giveHealth |

| 1 | $friendly turnto NULL |
| 2 | while($friendly != NULL){ |
| 3 | $friendly health (1000) |
| 4 | wait 3 |
| 5 | } |

ENTITY FLAGS ($whatever.flags["whatever"])
Entity variables are variables that are attached to a entity, like a player, ai or vehicle. Once that entity is
removed (player disconnects), the data stored on that entity is also removed. This is actually a good thing.

The problem is how entity variables are visually displayed in the code. First the example how it looks
without flags, then how it looks with flags. Using flags will make it obvious that it is a entity variable, not a
regular script command applied to a entity.

| 1 | $captainWonders.health = 10 //set entity variable |
| 2 | $captainWonders.health 10 //set health and max health |
| 3 | local.health = $captainWonders.health //get health value |

| 1 | $captainWonders.flags["health"] = 10 //set entity variable |
| 2 | $captainWonders.health 10 //set health and max health |
| 3 | local.health = $captainWonders.flags["health"] //get health value from var |

More experienced scripters might not have as much trouble with it as beginners will, but if you can avoid
any possible source of errors, you really should.
EACH FRAME (waitframe)
Servers do not just run your script once per second, they do it a couple of times per second. How often depends on the cvar sv_fps. On default it is set to 20, that makes 20 frames per second. Or in other words, one server frame time is 0.05 seconds.

Not every server has the same settings, but for some commands to work right you need to wait one frame in-between. Luckily MOHAA has a own script command to help you there, it is called waitframe. It pauses the script (just the current thread/code, not all script) at the position it is placed, and continues as soon as the current frame time has ended and a new frame time has begun.

You will see waitframe often used inside of while and for loops, making sure the code is run only once per frame time. That makes perfect sense, because all entities on the server are also updated once per frame time, so checking for changes, needs to be done only once per frame.

USE VARIABLES OVER EVENT CALLS (getcvar)
Using the getCvar command to get values from the server creates a event in which the script code is requesting the cvar values from the game. If you use that often it can have a negative impact on the server performance. A optimized script should always minimize event calls and execute code only as it is needed.

You can store Cvar values that do not change, in script variables. These are controlled by the script, so they do compute much faster than using the getcvar command. But be warned, this only makes sense if you know that this cvar does not change while you are using it.

Grab Cvar values that don't change in your map script right above the main thread, like shown in the example. But be careful if you specify variables outside of a function they are set each time you exec the script file (exec maps/myscript.scr), unless you specify a thread (exec maps/myscript.scr::main).

```plaintext
local.g_gametype = int( getCvar( "g_gametype" ) )
main:{
  if( local.g_gametype > 0){
    wait 10
    iprintlnbold_noloc( "this is multiplayer" )
  }
}end
```

NAMEING
Naming of variables and functions is very important! Especially if you plan to work on bigger projects. Good naming has a positive impact on the code quality. I have seen alot of fancy naming, but good naming is not about making it look fancy, it is about easy to understand and fast to read code.

Here are a few established rules I recommend:

1. Function names start in lowercase and words are seperated either by a uppercase letter or an underscore.  
   *Examples: actorAnimateNow, actor_animate_now or actor_animateNow*

2. Variable Names should also follow the naming of functions, however, you may want to add the type of the variable to the variable name.  
   *Examples: local.string_missionText, level.integer_missionStatus*

3. The purpose of a variable or function should be derivable from its name.

4. Make sure you have clearifying commentaries where they are needed.  
   If it is not totally obvious what a function or variable does, make sure your comments make it clear. Not everyone would chose the same name, so not everyone will know what you intend to express.
WAIT FOR PLAYER (level wait till spawn)

Once the map is fully loaded on the server, the scripts are executed. At this moment Players just start to load the map on their computer, and it will take them a moment before they can enter the game. During this time the scripts are running without a player present. (This can also happen in singleplayer)

I have been told that there is a command to wait for players in multiplayer, but I have never tested it.

```plaintext
level waittill playerspawn
```

The example code below shows how it could be done in single- and multiplayer.

```plaintext
waitForPlayers: {
  if (int (getCvar("g_gameType"))) { //Singleplayer
    level waittill spawn //wait until player spawn
    local.player = $player[1]
    local.player healthonly 111
  }
  else { //Multiplayer
    while($player == NULL) { //loop until a player joins
      waitframe
      }
      for (local.i = 1; local.i <= $player.size; local.i++) { //handle all players
        local.player = $player[local.i] //get a player
        if (local.player != NULL) { //if player exists
          local.player healthonly 111
        }
      }
  }
}
```

WAIT FOR WARMUP TIME (g_warmup)

If you are making scripts for multiplayer, you might want parts of the script to start once the actual match starts. The Cvar g_warmup is used to set a timelimit that should reflect a little over the average loading time so all players can enter the game before it starts in multiplayer.

The code below is a good example how to pause your script/function until the warmup time is over.

```plaintext
while (int (getCvar("g_warmup"))) >= level.time{
  waitframe
}
```

LOOPS (while & for)

The most common loops are for and while. If not used right they can create a infinity loop error. Such a error will be caught by the server and the current game will be terminated. On a multiplayer server that will seem like the server has crashed, but it actually just shut down with a error.

A infinity loop error is assumed if a loop has been iterated (cycled) for a coupe of thousand times, within a single server frame (0.05 sec at sv_fps 20).

If you need your loop to run for a longer duration of time, you should use a command that will pause your loop for a short time.

You can use wait or waitframe.

waitframe will pause your loop temporarily and wait until the beginning of the next server frame time, before resuming. This will run your loop once per frame and prevent your loop from being errored out.

wait followed by a float or integer (wait 1.5), will pause your loop temporarily and wait until the time in seconds has passed, before resuming.
Loops have two special commands, that can be very useful! **Continue** will make a loop skip forward to the next Iteration/Cycle. **Break** will break out of the loop, this will end the current loop.

The **while** loop is iterated once per server frame.

The **while** loop will break (end instantly) if the entity $enemy does no longer exist.

The **for** loop will interate as often as there are players on the server.

The **for** loop will skip forward to the next iteration and ignore the rest of the code in the current iteration if the current player does not exist.

```cpp
//loop as long as the entity does exist
while( $enemy != NULL){
  //end the for loop if enemy is dead
  if( isAlive $enemy != 1){ break }

  //handle each player in mp
  for( local.i=1;local.i<=$player.size;local.i++){//grab player from array
    local.player = $player[local.i]
    //if player does not exists, go to next cycle
    if( local.player == NULL ){ continue }
  }
  //will only execute if player != null
  local.player heal 1
}

//end while loop if enemy dead enemy
if( isAlive $enemy != 1 ){ break }

//heal enemy
$enemy heal 1

//wait for this frametime to end
waitframe
}
```

**Example of break and alternative means.** The two examples are equal in their function, but sometimes it is better to use a break and sometimes it is not to. This depends on the structure of your code and if there are multiple conditions you want to check in a specific order or just right at the start or end of the loop.

```cpp
while( local.playerInTank && $player.health > 0 ){ //do something here
  ... 
}
```

```cpp
while( local.playerInTank ){//do something here
  if( $player.health > 0 ){ break }
  ... 
}
```

**Example of continue and alternative means.** Using continue can prevent your code inside a loop from becoming too much interleaved. As you can see on the left example code with all these if statements. If you notice that your code is getting to interleaved and to hard to read, you should consider making use of break and continue instead, like on the example code on the right.

```cpp
for( local.i=1; local.i<=$player.size; local.i++){
  local.player = $player[local.i]
  //if player exists and is alive
  if( local.player != NULL && local.player.health > 0){
    //if player is alive
    if( isAlive $enemy{)
      if( isAlive $enemy2 ){ //continue with next cycle, if player health > 10
        if( local.player.health < 11 ){ if( local.player.health >= 10 ){ local.player heal 0.5
          continue
        }
        continue
      }
      //abort if any enemy is dead
      if( !isAlive $enemy1 ){ break }
    }
    //continue with next cycle if player missing or dead
    if( local.player == NULL || local.player.health <= 0){
      continue
    }
    //continue with next cycle if player health > 10
    if( local.player.health > 10 ){ continue
    }
    //heal current player upto 50%
    local.player heal 0.5
}
```

```cpp
for( local.i=1; local.i<=$player.size; local.i++){
  local.player = $player[local.i]
  //continue with next cycle if player missing or dead
  if( local.player == NULL || local.player.health <= 0){
    continue
  }
  //continue with next cycle if player health > 10
  if( local.player.health > 10 ){ continue
  }
  //heal current player upto 50%
  local.player heal 0.5
}
SWITCH

Switch as alternative to a series of if/else statements has three distinct advantages:

1. **Speed** - A Switch construct is generally faster than an if/else construct.

2. **Readability** - In many cases a switch construct is easier to read than an if/else construct.

3. **Default** - In a switch construct the fallback to default can provide some safety.

```plaintext
1 switch by contents of var
2 switch ( local.var ){
3     case 1://if local.var has value 1
4         ...
5         break
6     case 2://if local.var has value 2
7         ...
8         break
9     case 5://if local.var has value 5
10        ...
11        break
12 default://otherwise, default here
13        ...
14        break
15 }
```

On the example code above you can see a classical example of a switch construct. The switch converts the value of the expression (in this case the variable local.var) into a string for comparison.

**SELF**

In MOHAA scripting you will often see the use of the object `self`.

Self is the object that started the current function.

This means `self` can be a different object each time it is being used. To figure out what `self` is you need to know where or what started this function.

```plaintext
1 main:
2     //example function using self
3     if( self == NIL || self == NULL ){ end }
4     self health 1000
5     self scale 2
6     }end
```

Triggers, Entities, and scripts (AI/Global) can start functions in the script, and whatever started the function will be accessible as `self` in the function.

Actors can run threads and also become `self`, they can then perform advanced script actions.

If a function is started by a trigger this trigger will be `self` in that function it did start.

**MOHAA allows you also to get the entity that activated (entered/used) the trigger, with another object reference (parm.other), more about this in the next chapter.**

The example code shows how a function needs to be started for the `entity1` to become `self` inside the main function of `somescr.scr`:

```plaintext
1 $entity1 thread somescr.scr::main
2 $entity2 exec somescr.scr::main
```

**PARM.** (parm.other)

There are several parm Reference Objects, each has its own purpose and can only be used under certain circumstances. Each parm Reference Objects stores only one Object at a time, which is globally accessible.

This means that the Object can be overwritten by the game at any given moment.

To keep the object, you need to put it immediately into a local variable!

```plaintext
1 local.other = parm.other
```
parm.other
Can be used if a function is started by a trigger. Whatever activated the trigger which started the function shown in the code below will be parm.other.
1 //example function using parm.other, needs to be started by a trigger
2 showTargetnameOther:
3  local.other = parm.other
4  if ( local.other == NIL || local.other == NULL ){ end } //exit if invalid/missing
5  iprintlnbold_noloc( "parm.other has the targetname:"+local.other.targetname )
6 }

parm.owner
Is similar to parm.other, but it is used to get the owner of the projectile that has activated the trigger which started the function.
1 showTargetnameOwner:
2  local.other = parm.owner //get activating entity
3  local.owner = parm.owner //get owner of activating entity
4  if ( local.owner == NIL || local.owner == NULL ){ //fallback to other if no owner
5    if ( local.other == NIL || local.other == NULL ){ end } //exit if missing
6    local.owner = local.other //fallback
7  }
8  iprintlnbold_noloc( "parm.owner targetname:"+local.owner.targetname )
9 }

parm.previousthread
Returns the thread as a object that was previously started, it is used to have control of a thread outside of the actual thread. The example code below is from one of the global game scripts.
1 //example how to use parm.previousethread
2 friendlythink:
3  //start a thread
4  thread friendlythinkstart
5  //retrive the started thread
6  local.thread = parm.previousthread
7  //wait until actor dies
8  self waittill death
9  //check if thread is still running, delete if it is
10  if (local.thread){
11    local.thread delete
12  }
13 }end

parm.movefail and parm.movedone
This is used to check if the actor move has failed or succeeded. The example function below would be started in the script like this: $actor thread actor_runTo $doorWay
1 actor runTo local.runTo:
2  //disable ai thinking for actor
3  self exec global/disable_ai.scr
4  //make actor go to that entity its location
5  self runto local.runTo
6  //pause this function here until actor has finished moving
7  self waittill movedone
8  //show actor moving status
9  if( parm.movedone == 1 ){ iprintln_noloc("Success: "+self.targetname) }
10  if( parm.movefail == 1 ){ iprintln_noloc("Failure: "+self.targetname) }
11  //re-enable ai thinking for actor
12  self exec global/enable_ai.scr
13 }end
EXTERNAL SCRIPT THREADS ( exec/thread )

Starting functions in others script files is a common thing to do in mohaa scripting. This is different than starting a function in the same script file, as demonstrated below.

<table>
<thead>
<tr>
<th>Same file</th>
<th>Different File</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread myFunction</td>
<td>thread myFolder/myFile.scr::myFunction</td>
</tr>
</tbody>
</table>

You will often see exec being used in mohaa to call functions in external script files. There are two ways you can execute a file with exec. With a function name attached, like shown on the left in the table below, or without a function name.

**NOTE:** There is a huge difference between the two ways of doing this!

<table>
<thead>
<tr>
<th>Executing only a function</th>
<th>Executing the complete file</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec myFolder/myFile.scr::myFunction</td>
<td>thread myFolder/myFile.scr</td>
</tr>
</tbody>
</table>

This starts only one function, in this case the function:
- myFunction.

This executes the script from top to bottom.
- local.var1
- level.var1
- main:

In other words, it executes the script file the same way, the game does execute the main level script file.

```plaintext
1 //file wide variable
2 local.var1 = NIL
3 //level wide variable
4 level.var1 = NIL
5
6 //main function
7 main:{
8 ...
9 }end
10
11 //myFunction
12 myFunction:{
13 ...
14 }end
```

RETURN VALUES

Functions can return values, this is especially interesting if they are used for checking. In the code below the entity in the local.entity variable is checked against all valid players on the server. If any valid player is touching the entity the function will return 1, otherwise it will return 0.

```plaintext
1 if(exec coop_mod/replace.scr::istouching local.entity){
2  iprintInbold_noloc("A player is touching: "+local.entity)
3  wait 5
4 }}
```

Using an external function for this, rather than typing all the code in this file, reduces greatly the amount of code and it keeps the code comprehensible. This is why HaZardModding Coop Mod uses this technique.

You can use `exec`, `waitexec` and `waitthread` for this, if you are using `exec` be aware that you can not use any kind of wait command (waitframe/wait) in the external function, use `waitexec` or `waitthread` then.

The example code below shows how a function returning a value could look like.

```plaintext
1 returnValue local.var1 local.var2:{
2  local.result = 0
3  if(local.var1 == local.var2){ local.result = 1 }
4 }end local.result
```
Your code should be beautiful, practical but above all functional.
If you have to choose between beauty and practical, go with practical.
But make sure it is functional at all times, by conducting thoroughl tests.

Thank you for reading, I hope I could give you some useful Advice.
If you have any feedback, feel free to contact me.

Contact HaZardModding Group:
On Discord: https://discord.gg/vW7yskc (recommended)
On ModDB: https://www.moddb.com/messages/compose?to=Chrissstrahl