Lua documentation for Illarion scripting v5.22

Martin [*], pharse [†], vilarion [‡]

2004-2006/2010-2015/2020

[*] martin@illarion.org, http://www.illarion.org
[†] pharse@illarion.org, http://www.illarion.org
[‡] vilarion@illarion.org, http://www.illarion.org

Contents

1. General

1.1. Formalism

System variables and variables of structures are accessed by ".".
Functions are called by ":".
If a function has no parameters, one still has to write ().
Lines that start with "?" refer to unclear commands.
Lines that start with "!" refer to suggested commands.
For variables, "*r*:" in front of them means reading access, "*rw*:" means reading and writing access.
Names in `this font` refer to illarion-specific key words.
Names in *this font* refer to lua-specific key words.
Names in ⟨*in this format*⟩ are placeholder and can be seen as variables.
Names in normal fixed font refer to a special choice of variables.
Names of functions are designed to be self explaining, therefore there are a lot of undocumented functions around.

Examples:

```
XKoordinate=TargetItem.pos.x;
User:talk(Character.say, "Hallo Welt!");
```

Important note: Lua is case sensitive.

1.2. General introduction

Everytime certain events happen (someone shift-clicks an object, a monster dies, someone looks at an object, ... see the section about "entry points"), a script is started. The name of that script is usually defined in the SQL-database in a separate row. For example, the table common, which holds information about all items in illarion (weight, ...), has a row called com_script, which holds the name of the script that is linked to each item. If someone shift-clicks an item, the lua-script that is linked to this item in common is executed. This script then consists of several functions, defining what happens in certain cases: the item can be used with another item (shift-clicks), with a character and so on. This means, a general item has the following lua-file

```
      -- item.lua
function UseItem(User, SourceItem)
    ...
end

function UseItemWithCharacter(User, SourceItem, Character)
    ...
end
```

```
function LookAtItem(User, Item)
    ...
end
...
```

Such a lua-file does not need all possible functions; if an item has no `LookAtItem(-)`function (LookAt=left-click), it simply does nothing (special) when looked at. There are also entry points for magic and NPCs, which can be found in the entry points section again.

1.3. Variable types

- ⟨*User*⟩, ⟨*Originator*⟩, ⟨*Character*⟩: Character-type variables, see chapter "Characters".
- ⟨*SourceItem*⟩, ⟨*TargetItem*⟩: Item-type variables, see chapter "Items".
- ⟨*Pos*⟩, ⟨*ItemPos*⟩, ⟨*TargetItemPos*⟩, ⟨*TargetPos*⟩: Position-type variables, see chapter "Positions"
- *dataTable* : Type that represents a Lua table, mapping data keys (strings) to data values (strings or integers).
- *Skill* : Type that represents a skill, embedded in Character like this: Character.⟨*name*⟩. Valid values for ⟨*name*⟩ are defined in the database in skills.skl_name.

2. Quickstart: Tutorials

To provide you with a way to start very quickly with scripting simple things, here's a tutorial section.

2.1. Level 0: Before we start

Before you start, you need

- Access to the script-SVN-repository; that includes free the tortoise SVN client or similar.
- A text editor (for starters, the Windows-Editor or Wordpad will do).
- Access to Illarion's testserver and the testclient and a character on the testserver with GM-rights.
- Creativity!
- Optional: DB access.

2.2. Level 1: Your first script

As your first script, we recommend to use the item with the ID 2. It is bound to the script named "`I_2_mehl.lua`". Open it in your text editor and delete the whole file except for the following lines:

```
function UseItem(User, SourceItem, ltstate)

end
```

What does this mean?

Every time someone "uses" (=shift clicks) an item with ID 2 (flour), the function `UseItem(...)` inside the script `I_2_mehl.lua` is called. The server provides this script with the following information:

- `User` contains all information about the character "using" the flour, like his position on the map, his hitpoints, skills, attributes and so on.
- `SourceItem` contains all information about the "used" item—the flour in our case—like where it is on the map, what `data`-value it has and so on.
- `TargetItem` is only used in case you "used" the flour with some other item. In that case, it contains all information about the second item.
- `ltstate` can be ignored for now as it is not important for our scripts.

Let's try the following script:

```
function UseItem(User, SourceItem, ltstate)
    User:inform(User.name.." has used me!");
end
```

Commit this script to the svn-repository, log into the testserver (if you haven't already), reload the item-scripts by saying "`!rd`" with your character and wait until it finishes with "***Definitions reloaded***". Then produce flour (by saying "`!create 2`", you get one) and shift-click it. In my case, what appears is: `Ciryon: Ciryon has used me!`

3. Positions

3.1. Functions

posStruct ⟨*position*⟩=`position(`*int* ⟨*x*⟩, *int* ⟨*y*⟩, *int* ⟨*z*⟩`)`

Creates position-structure for the point (x, y, z).

boolean posStruct ⟨*posA*⟩ == *posStruct* ⟨*posB*⟩

Compares two position structs. Returns ⟨*true*⟩ if they are equal, ⟨*false*⟩ otherwise.

text `tostring(`*posStruct* ⟨*pos*⟩`)`

Returns "(" .. pos.x .. ", " .. pos.y .. ", " .. pos.z .. ")".

3.2. Variables

rw: *int* ⟨*position*⟩.x
rw: *int* ⟨*position*⟩.y
rw: *int* ⟨*position*⟩.z
Usage: XCoordinate=User.pos.x

3.3. Additional information

Note that a position from a character struct is only a pointer. Thus it will change if the character changes its position. Example:

```
User:forceWarp( position(0,0,0) );
testPos = User.pos; -- testPos is (0,0,0)
User:forceWarp( position(1,1,1) ); -- now User.pos AND testPos is (1,1,1)
```

Avoid this by copying the single x,y and z coordinates in a new position struct.

4. Colours

4.1. Functions

`colour(`*int* ⟨*red*⟩, *int* ⟨*green*⟩, *int* ⟨*blue*⟩ [, *int* ⟨*alpha*⟩ = 255]`)`

> Creates colour structure for given RGBA values. All parameters range from 0 to 255. For the optional ⟨*alpha*⟩ 0 means transparent, 255 means opaque.

4.2. Variables

rw: *int* ⟨*colour*⟩.`red`
rw: *int* ⟨*colour*⟩.`green`
rw: *int* ⟨*colour*⟩.`blue`
rw: *int* ⟨*colour*⟩.`alpha`
Usage: alpha = c.alpha

5. Characters

5.1. Functions

5.1.1. Text/Speech

void ⟨*character*⟩:`talk(`*int* ⟨*texttype*⟩,*text* ⟨*text*⟩`)`

⟨*texttype*⟩ can be `Character.say`, `Character.whisper` or `Character.yell`.
Lets a character say/whisper/yell some ⟨*text*⟩.
Example: `User:talk(Character.say`, "Hello world!"`)`

void ⟨*character*⟩:`talk(`*int* ⟨*texttype*⟩,*text* ⟨*germanText*⟩, *text* ⟨*englishText*⟩`)`

Same as `talk(...)` except that players will only hear speech in their own language.

void ⟨*character*⟩:`inform(`*text* ⟨*Text*⟩, *int* ⟨*informtype*⟩ = `Character.mediumPriority)`
void ⟨*character*⟩:`inform(`*text* ⟨*germanText*⟩, *text* ⟨*englishText*⟩, *int* ⟨*informtype*⟩ = `Character.mediumPriority)`

Informs a player with a short ⟨*Text*⟩ and has no effect when used with other character types. Except for debugging the second syntax should be used to add native language support. Different priorities can be selected. These determine how prominent the ⟨*Text*⟩ is shown on the screen. Valid priorities are: `Character.lowPriority`, `Character.mediumPriority` as default if this parameter is omitted and `Character.highPriority`.
Examples: `User:inform(`"Du bist betrunken.", "You are drunk."`)`
`User:inform(`"A raindrop falls on your head.", `Character.lowPriority)`

void ⟨*character*⟩:`introduce(`*Player* ⟨*player*⟩`)`

Introduces ⟨*player*⟩ to ⟨*character*⟩ if character is a player as well. Otherwise it has no effect.

void ⟨*character*⟩:`move(`*int* ⟨*direction*⟩,*boolean* ⟨*active move*⟩`)`

⟨*character*⟩ makes a step into ⟨*direction*⟩. ⟨*active move*⟩ is true if the move was done actively (normal case) and false otherwise.

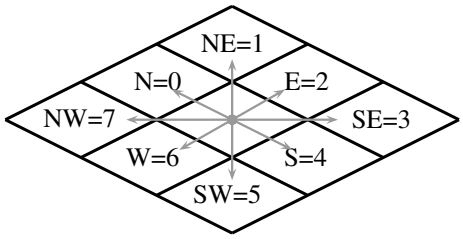

Figure 5.1.: The 8 possible directions

void ⟨*character*⟩:`turn(`*int* ⟨*direction*⟩`)`
void ⟨*character*⟩:`turn(`*posStruct* ⟨*position*⟩`)`

Turns ⟨*character*⟩ into the given ⟨*direction*⟩ or towards the given ⟨*position*⟩.

text ⟨*character*⟩:`alterMessage(`*text* ⟨*Text*⟩,*int* ⟨*LanguageSkill*⟩`)`

Returns the altered ⟨*Text*⟩ with respect to the ⟨*LanguageSkill*⟩ given.

void ⟨*character*⟩:`sendCharDescription(`*int* ⟨*id*⟩, *text* ⟨*text*⟩`)`

Shows the ⟨*text*⟩ as character description of the character with ID ⟨*id*⟩ (just next to the avatar) only to this ⟨*character*⟩.

void ⟨*character*⟩:`sendBook(`*int* ⟨*id*⟩`)`

Tells the client to display book ⟨*id*⟩ for ⟨*character*⟩. Books are stored as client resources.

5.1.2. Skills and Attributes

NOTE: There are two kinds of attributes: fixed and variable ones. All attributes that are not meant to develop during the game are fixed, e.g. strength, perception, age, sex etc.

On the other hand there are the variable attributes, e.g. any skill (of course), hitpoints, mana etc. Any change to those will be permanently written to the database.

Any change to fixed attributes is only temporary and will be null upon the next login. Example: You want to create an amulet that gives a +2 STR bonus when worn. It does not suffice to change the strength when the amulet is put on, you have to use a Long Time Effect that does the change again after a login (and checks if the amulet is still in its place).

text ⟨*character*⟩:`getSkillName(`*Skill* ⟨*skill*⟩`)`

Returns the name of the given ⟨*skill*⟩ in the player's language.

int ⟨*character*⟩:`getSkill(`*Skill* ⟨*skill*⟩`)`
Character:skillvalue ⟨*character*⟩:`getSkillValue(`*Skill* ⟨*skill*⟩`)`

skillvalue is a table with two fields, major and minor, representing the skill and the minor skill.

int ⟨*character*⟩:`setSkill(`*Skill* ⟨*skill*⟩, *int* ⟨*major*⟩, *int* ⟨*minor*⟩`)`

Directly sets major and minor skill. Returns the new major skill. If the skill does not exist in the database, nothing is set and 0 is returned.

int ⟨*character*⟩:`increaseSkill(`*Skill* ⟨*skill*⟩, *int* ⟨*value*⟩`)`
int ⟨*character*⟩:`increaseMinorSkill(`*Skill* ⟨*skill*⟩, *int* ⟨*value*⟩`)`

Increase major and minor skill, respectively. Return the new major skill. If the skill does not exist in the database, nothing is increased and 0 is returned.

void ⟨*character*⟩:`learn(`*Skill* ⟨*skill*⟩, *int* ⟨*actionPoints*⟩,*int* ⟨*learnLimit*⟩`)`

⟨*skill*⟩: Constant of the skill.
⟨*actionPoints*⟩: Number of actionPoints used up for the action which resulted in learning.
⟨*learnLimit*⟩: The skill will not be advanced beyond this limit and never beyond 100.

int ⟨*character*⟩:`increaseAttrib(`*text* ⟨*AttribName*⟩, *int* ⟨*value*⟩`)`

Increases the attribute given (see below) and returns the new attribute value. Use ⟨*value*⟩=0 to read the attribute's value. Note that this command also sends a player update to all characters in range if necessary.

void ⟨*character*⟩:`setAttrib(`*text* ⟨*AttribName*⟩, *int* ⟨*value*⟩`)`

⟨*AttribName*⟩ can be: "sex", "age", "body_height", "attitude", "luck", "strength", "dexterity", "constitution", "agility", "intelligence", "perception", "willpower", "essence", "foodlevel", "hitpoints", "mana", "poisonvalue". And "sex" can be: Character.male, Character.female

Be aware that any attribute change to fixed attributes like "strength" will only last for the current session and will be reset to the database value upon the next login.

boolean ⟨*character*⟩:`isBaseAttributeValid(`*text* ⟨*attribute*⟩, *int* ⟨*value*⟩`)`

Returns whether ⟨*value*⟩ is acceptable for the given ⟨*attribute*⟩ and the race of the character, respecting limits given in table raceattr.

int ⟨*character*⟩:`getBaseAttributeSum()`

Returns the current sum of the eight primary attributes: agility, constitution, dexterity, essence, intelligence, perception, strength and willpower.

int ⟨*character*⟩:`getMaxAttributePoints()`

Returns the value which `getBaseAttributeSum()` needs to result in, so that the base attributes can be saved. Can be used to make tests before actually trying to save the base attributes.

int ⟨*character*⟩:`getBaseAttribute(`*text* ⟨*attribute*⟩`)`

Returns the base value of the given ⟨*attribute*⟩, that is the value that this attribute normally has, when no special effects are active.

boolean ⟨*character*⟩:`setBaseAttribute(`*text* ⟨*attribute*⟩, *int* ⟨*value*⟩`)`

Sets the base value of the given ⟨*attribute*⟩ and returns ⟨*true*⟩ if `isBaseAttributeValid(...)` would return ⟨*true*⟩. Otherwise is a no-op and returns ⟨*false*⟩.

boolean ⟨*character*⟩:`increaseBaseAttribute(`*text* ⟨*attribute*⟩, *int* ⟨*amount*⟩`)`

If `isBaseAttributeValid(...)` would return ⟨*true*⟩, increases or decreases the base value of the given ⟨*attribute*⟩ and returns ⟨*true*⟩. Otherwise is a no-op and returns ⟨*false*⟩.

boolean ⟨*character*⟩:`saveBaseAttributes()`

Saves the eight primary base attributes to the database, iff `getBaseAttributeSum()` == `getMaxAttributePoints()`. On failure resets primary attribute values to database values. Returns whether the operation was successful or not.

void ⟨*character*⟩:`setSkinColour(`*colour* ⟨*c*⟩`)`

Sets skin colour.

void ⟨*character*⟩:`setHairColour(`*colour* ⟨*c*⟩`)`

Sets hair and beard colour.

colour ⟨*character*⟩:`getSkinColour()`

Returns skin colour.

colour ⟨*character*⟩:`getHairColour()`

Returns hair color.

void ⟨*character*⟩:`setHair(`*int* ⟨*hairID*⟩`)`

Returns the ID of the present hair, 0 for no hair.

void ⟨*character*⟩:`setBeard(`*int* ⟨*beardID*⟩`)`

Returns the ID of the present beard, 0 for no beard.

int ⟨*character*⟩:`getHair()`

Returns the ID of the present hair, 0 for no hair.

int ⟨*character*⟩:`getBeard()`

Returns the ID of the present beard, 0 for no beard.

int ⟨*character*⟩:`setRace(`*int* ⟨*race*⟩`)`

Temporarily sets the race of a character. See table 5.1 for details.

int ⟨*character*⟩:`getRace()`

Returns the race of a character. See table 5.1 for details.

int ⟨*character*⟩:`getMonsterType()`

Returns the monster-ID of a monster. For a list of the current monster-IDs, please consult the database or one of the other developers.

int ⟨*character*⟩:`getFaceTo()`

Returns an integer between 0 and 7 inclusively that indicates the direction the character is facing to. For a list of the directions, see Fig.(5.1).

int ⟨*character*⟩:`getType()`

Returns Character.player for players, Character.monster for monsters and Character.npc for NPCs.

void ⟨*character*⟩:`increasePoisonValue(`⟨*value*⟩`)`
int ⟨*character*⟩:`getPoisonValue()`
void ⟨*character*⟩:`setPoisonValue(`*int* ⟨*value*⟩`)`
int ⟨*character*⟩:`getMentalCapacity()`
void ⟨*character*⟩:`setMentalCapacity(`*int* ⟨*value*⟩`)`
void ⟨*character*⟩:`increaseMentalCapacity(`*int* ⟨*value*⟩`)`
int ⟨*character*⟩:`getMagicType()`

returns MagicType

void ⟨*character*⟩:`setMagicType(`*int* ⟨*MagicType*⟩`)`

MagicType: "mage"=0, "priest"=1, "bard"=2, "druid"=3

int ⟨*character*⟩:`getMagicFlags(`*int* ⟨*MagicType*⟩`)`
void ⟨*character*⟩:`teachMagic(`*int* ⟨*MagicType*⟩,*int* ⟨*MagicFlag*⟩`)`
int ⟨*character*⟩:`getPlayerLanguage()`

Returns the player's language: Player.german or Player.english.

Name	⟨*rID*⟩	Name	⟨*rID*⟩	Name	⟨*rID*⟩
human	0	blackwolf	41	blacktroll	80
dwarf	1	greywolf	42	redtroll	81
halfling	2	redwolf	43	blackzombie	82
elf	3	redraptor	48	transparentzombie	83
orc	4	silverbear	49	redzombie	84
lizardman	5	blackbear	50	blackhellhound	85
gnome	6	bear	51	transparenthellhound	86
troll	9	raptor	52	greenhellhound	87
mumie	10	zombie	53	redhellhound	88
skeleton	11	hellhound	54	redimp	89
beholder	12	imp	55	blackimp	90
blackbeholder	13	irongolem	56	blueirongolem	91
transparentbeholder	14	ratman	57	redratman	92
brownmummy	15	dog	58	greenratman	93
bluemummy	17	beetle	59	blueratman	94
sheep	18	fox	60	reddog	95
spider	19	slime	61	greydog	96
demonskeleton	20	chicken	62	blackdog	97
redspider	21	bonedragon	63	greenbeetle	98
greenspider	22	blackbonedragon	64	copperbeetle	99
bluespider	23	redbonedragon	65	redbeetle	100
pig	24	transparentbonedragon	66	goldbeetle	101
boar	25	greenbonedragon	67	greyfox	102
transparentspider	26	bluebonedragon	68	redslime	103
wasp	27	goldbonedragon	69	blackslime	104
redwasp	28	redmummy	70	transparentslime	105
stonegolem	30	greymummy	71	brownchicken	106
brownstonegolem	31	blackmummy	72	redchicken	107
redstonegolem	32	goldmummy	73	blackchicken	108
silverstonegolem	33	transparentskeleton	74		
transparentstonegolem	34	blueskeleton	75		
cow	37	greenskeleton	76		
bull	38	goldgolem	77		
wolf	39	goldskeleton	78		
transparentwolf	40	bluetroll	79		

Table 5.1.: List of available races with Race-IDs (⟨*rID*⟩)

5.1.3. Quest progress

void ⟨*character*⟩:`setQuestProgress(`*int* ⟨*questID*⟩,*int* ⟨*progress*⟩`)`

A questprogress can be set for a specific quest.

int ⟨*progress*⟩[, *int* ⟨*time*⟩] ⟨*character*⟩:`getQuestProgress(`*int* ⟨*questID*⟩`)`

Returns the ⟨*progress*⟩ for a specific quest. Optionally also returns the ⟨*time*⟩ when this progress was last set as Unix timestamp.

5.1.4. Item handling

int ⟨*character*⟩:`createItem(`*int* ⟨*itemID*⟩,*int* ⟨*count*⟩,*int* ⟨*quality*⟩, *dataTable* ⟨*data*⟩`)`

Item is created in the belt or backpack of ⟨*character*⟩. If that is not possible, the items will not be created. The function returns an integer that gives the number of items that cannot be created. `world:createItemFromId` might be a good choice in addition.

void ⟨*character*⟩:`createAtPos(`*int* ⟨*Position_body*⟩,*int* ⟨*itemId*⟩,*int* ⟨*count*⟩`)`

Creates an item at a special body position (see below).

void ⟨*character*⟩:`changeQualityAt(`*int* ⟨*Position_body*⟩,*int* ⟨*qly-amount*⟩`)`

Changes the quality by amount at position_body.

int ⟨*character*⟩:`eraseItem(`*int* ⟨*itemID*⟩,*int* ⟨*count*⟩`)`
int ⟨*character*⟩:`eraseItem(`*int* ⟨*itemID*⟩,*int* ⟨*count*⟩, *dataTable* ⟨*data*⟩`)`

⟨*count*⟩ item with ⟨*itemID*⟩ (=number!) are erased from the ⟨*character*⟩ inventory. You have no influence on which items are deleted, you can just determine ID and number. The return value contains the amount of items that could not be deleted. In case the optional ⟨*data*⟩ parameter is set, only items that include these data values are deleted. If the data table is empty however, only items without data are erased.

int ⟨*character*⟩:`countItem(`*int* ⟨*itemID*⟩`)`
int ⟨*character*⟩:`countItemAt(`*text* ⟨*location*⟩,*int* ⟨*itemID*⟩`)`
int ⟨*character*⟩:`countItemAt(`*text* ⟨*location*⟩,*int* ⟨*itemID*⟩,*dataTable* ⟨*data*⟩`)`

Counts only at a certain position; ⟨*character*⟩:countItemAt("all",...) is the same as ⟨*character*⟩:`countItem(...)`
⟨*location*⟩ can be "all", "belt", "body", "backpack". The variant with ⟨*data*⟩ does only count items that include these data values. If the data table is empty however, only items without data are counted.

void ⟨*character*⟩:`increaseAtPos(`*int* ⟨*Position_body*⟩,*int* ⟨*count*⟩`)`
void ⟨*character*⟩:`swapAtPos(`*int* ⟨*Position_body*⟩,*int* ⟨*itemID*⟩,*int* ⟨*quality*⟩`)`

Position_body: `BACKPACK=0, HEAD=1, NECK=2, BREAST=3, HANDS=4, LEFT_TOOL=5, RIGHT_TOOL=6, FINGER_LEFT_HAND=7, FINGER_RIGHT_HAND=8, LEGS=9, FEET=10, COAT=11, LAST_WEARABLE=11`
To be combined with ⟨*Item*⟩:`getType( )`.
See fig.(8.1).
If quality=0, then the quality remains the same.

scrItem ⟨*character*⟩:`getItemAt(`*int* ⟨*Position_body*⟩`)`

⟨*Position_body*⟩: `Character.backpack=0, Character.head=1, Character.neck=2, Character.breast=3, Character.hands=4, Character.left_tool=5, Character.right_tool=6, Character.finger_left_hand=7, Character.finger_right_hand=8, Character.legs=9, Character.feet=10, Character.coat=11, Character.belt_pos_1=12, Character.belt_pos_2=13, Character.belt_pos_3=14, Character.belt_pos_4=15, Character.belt_pos_5=16, Character.belt_pos_6=17`
This returns a ScriptItemStruct. See fig. (8.1).

list (scrItem) ⟨*character*⟩:`getItemList(`*int* ⟨*ItemID*⟩`)`

Returns a list with all items of this ⟨*ItemID*⟩.

conStruct ⟨*character*⟩:`getBackPack()`

Returns a container-item (which is different from scriptitem and commonitem). Container-items can be used to pick out items which are placed in it. See ⟨*Container*⟩:`takeItemNr(`⟨*itempos*⟩,⟨*count*⟩`)`.

conStruct ⟨*character*⟩:`getDepot(`*int* ⟨*depotId*⟩`)`

Returns a container-item (the depot of that Character). Containeritems can be used to pick out items which are placed in it. See ⟨*Container*⟩:`takeItemNr(`⟨*itempos*⟩,⟨*count*⟩`)`.

5.1.5. All the rest

boolean ⟨*character*⟩:`isNewPlayer()`

Returns whether ⟨*character*⟩ is a new player or not. The exact behaviour is defined in the database function is_new_player.

boolean ⟨*character*⟩:`isInRange(`*Character* ⟨*character2*⟩,*int* ⟨*Distance*⟩`)`

Returns true if ⟨*character2*⟩ is within ⟨*Distance*⟩ of ⟨*character*⟩, else false.

int ⟨*character*⟩:`distanceMetric(`*Character* ⟨*character2*⟩`)`

Returns distance.
Very similar to isInRange, but much more flexible. Better use distanceMetric.

int ⟨*character*⟩:`distanceMetricToPosition(`*posStruct* ⟨*Position*⟩`)`

Returns the distance from ⟨*character*⟩ to ⟨*Position*⟩.

boolean ⟨*character*⟩:`isInRangeToPosition(`*posStruct* ⟨*Position*⟩,*int* ⟨*distance*⟩`)`

Returns ***true*** when the ⟨*character*⟩ is within the ⟨*distance*⟩ to ⟨*position*⟩ and ***false*** otherwise.

void ⟨*character*⟩:`warp(`*posStruct* ⟨*Position*⟩`)`

"Position" is a position-structure as described above.

void ⟨*character*⟩:`forceWarp(`*posStruct* ⟨*Position*⟩`)`

"Position" is a position-structure as described above. This command works exactly as warp, but it ignores any non-passable flags on the target position. That means that you can warp onto e.g. water using this command.

void ⟨*character*⟩:`startMusic(`*int* ⟨*Number*⟩`)`

Starts music ⟨*Number*⟩ for ⟨*character*⟩, with 0 meaning silence. This overrides the default music given by the map until logout or a call of ⟨*character*⟩:`defaultMusic()`

void ⟨*character*⟩:`defaultMusic()`

Plays the default music for ⟨*character*⟩ as defined by the map.

boolean ⟨*character*⟩:`isAdmin()`

Returns ***true*** if that character is admin (GM) and ***false*** otherwise.

void ⟨*character*⟩:`setClippingActive(`*boolean* ⟨*status*⟩`)`

⟨*status*⟩ must be either ***true*** (walking through walls disabled) or ***false***; this enables the character to walk on fields where he usualy can't walk (water, walls, ...). Please use with care: This has to be turned OFF again!

boolean ⟨*character*⟩:`getClippingActive()`

Returns ***true*** or ***false***.

int ⟨*character*⟩:`idleTime()`

If ⟨*character*⟩ is a player, returns the number of seconds they are idle. Returns 0 otherwise.

boolean `isValidChar(`*Character* ⟨*char*⟩`)`

Returns ***true*** iff ⟨*char*⟩ is still valid and safe to use. Validity has to be checked if ⟨*char*⟩ is used in another entrypoint call than the one where it was originally obtained, since a player might have logged out, an NPC might have been deleted and a monster might have been killed in the meantime.

boolean ⟨*character*⟩:`pageGM(`*text* ⟨*ticket*⟩`)`

Returns ***true*** iff ⟨*ticket*⟩ is successfully logged as message to the GM team. Differing from the normal !gm command, the originating player is not informed about success.

void ⟨*character*⟩:`logAdmin(`*text* ⟨*message*⟩`)`

Writes ⟨*message*⟩ for the given user into the server log using the admin facility. Only used to record GM actions.

5.2. Variables

r: *text* ⟨*character*⟩.`lastSpokenText`

Returns this characters last spoken line of text

r: *posStruct* ⟨*character*⟩.`pos`

Position-structure

r: *text* ⟨*character*⟩.`name`
r: *int* ⟨*character*⟩.`id`
r: *boolean* ⟨*character*⟩.`attackmode`

true if character currently attacks, *false* otherwise.

rw: *int* ⟨*character*⟩.`activeLanguage`

"common language"=0, "human language"=1, "dwarf language"=2, "elf language"=3, "lizard language"=4, "orc language"=5, "halfling language"=6, "fairy language"=7, "gnome language"=8, "goblin language"=9, "ancient language"=10

rw: *int* ⟨*character*⟩.`movepoints`

A character has usually (when being idle) 21 movepoints. Every action like talking, fighting, using etc. needs at least ca. 6 movepoints. The reduction of movepoints depends on the character's agility. The regeneration is for all the same: +10 per second. A greater amount than 21 movepoints is not possible as it is immediately set again to 21.

6. Containers

Some items can contain other items. These are for example a character's backpack and depot. There are multiple functions to get access to such a container variable. Here is what you can do with those:

6.1. Functions

boolean , *scrItem* , *conStruct* ⟨*Container*⟩:`viewItemNr(`*int* ⟨*itempos*⟩`)`

> Returns three values in that specific order: `bool` ⟨*success*⟩, `structitem` ⟨*item*⟩, `containeritem` ⟨*container*⟩. ⟨*success*⟩ is ***true*** if Lua was able to get the item, ⟨*item*⟩ holds the item at that position number and ⟨*container*⟩ holds the containerstruct in case the item at that position was a container. This can be used together with ⟨*Container*⟩:`takeItemNr(`⟨*itempos*⟩,⟨*count*⟩`)`.

boolean , *scrItem* , *conStruct* ⟨*Container*⟩:`takeItemNr(`*int* ⟨*itempos*⟩,*int* ⟨*count*⟩`)`

> Returns three values in that specific order: `bool` ⟨*success*⟩, `structitem` ⟨*item*⟩, `containeritem` ⟨*container*⟩ and deletes this item (⟨*count*⟩ of them). ⟨*success*⟩ is ***true*** if Lua was able to get the item, ⟨*item*⟩ holds the item at that position number and ⟨*container*⟩ holds the containerstruct in case the item at that position was a container.
> Example:

```
TheDepot=User:getDepot(1);
for i=0,30 do
    worked,theItem,theContainer=TheDepot:takeItemNr(i,1);
    if (worked==true) then
        if (theContainer==nil) then
            User:inform("This is no container. It's item-ID is "..theItem.id);
        else
            User:inform("This is a container. It's item-ID is "..theItem.id);
        end
    end
end
```

void ⟨*Container*⟩:`changeQualityAt(`*int* ⟨*itempos*⟩,*int* ⟨*amount*⟩`)`

> Changes the quality of an item at a given position inside a container. Returns ***true*** if it worked.

boolean ⟨*Container*⟩:`insertContainer(`*scrItem* ⟨*item*⟩, *conStruct* ⟨*container*⟩`)`
boolean ⟨*Container*⟩:`insertContainer(`*scrItem* ⟨*item*⟩, *conStruct* ⟨*container*⟩, *int* ⟨*itempos*⟩`)`

> If ⟨*itempos*⟩ has been provided, tries to insert a container at that position. If not, or if that position is not free, inserts the container at the first free position. Returns ***true*** if successful.

void ⟨*Container*⟩:`insertItem(`*scrItem* ⟨*Item*⟩, *boolean* ⟨*merge*⟩`)`

Inserts an item into a container. Collects identical items which are stackable together to a stack if ⟨*merge*⟩ is ***true***. If there already is an item it will probably be overwritten!

void ⟨*Container*⟩:`insertItem(`*scrItem* ⟨*Item*⟩`)`

Inserts an item which is then placed on the last slot in that container.

int void ⟨*Container*⟩:`countItem(`*int* ⟨*itemid*⟩`)`
int void ⟨*Container*⟩:`countItem(`*int* ⟨*itemid*⟩, *dataTable* ⟨*data*⟩`)`

Counts the number of items in a container of a given ID. It works recursively, which means that if there is a container in that container containing items of that ID, they are counted as well. In case the optional ⟨*data*⟩ parameter is set, only items that include these data values are counted. If the data table is empty however, only items without data are counted.

int void ⟨*Container*⟩:`eraseItem(`*int* ⟨*itemid*⟩, *int* ⟨*count*⟩`)`
int void ⟨*Container*⟩:`eraseItem(`*int* ⟨*itemid*⟩, *int* ⟨*count*⟩, *dataTable* ⟨*data*⟩`)`

⟨*count*⟩ item with ⟨*itemid*⟩ (=number!) are erased from the ⟨*Container*⟩ inventory. You have no influence on which items are deleted, you can just determine ID and number. The return value contains the amount of items that could not be deleted. In case the optional ⟨*data*⟩ parameter is set, only items that include these data values are deleted. If the data table is empty however, only items without data are erased.

int void ⟨*Container*⟩:`increaseAtPos(`*int* ⟨*pos*⟩, *int* ⟨*value*⟩`)`

Increases the number of items at a given position. Supposedly returns the number of items afterwards.

boolean void ⟨*Container*⟩:`swapAtPos(`*int* ⟨*pos*⟩, *int* ⟨*newid*⟩, *int* ⟨*newquality*⟩`)`

Changes an item to another one with a new ID, returns ***true*** on success.

int void ⟨*Container*⟩:`weight();`

Returns the total weight of that container.

7. Dialogs

Dialogs are a more sophisticated approach to aquire user input than e.g. User.lastSpokenText. Each dialog serves a specific purpose as displaying bulk text, interfacing with a merchant or with the crafting system. Dialogs should be the preferred interaction method. If necessary new types should be implemented rather than abusing old variants or falling back to lastSpokenText. Please note, that as with many functions dialog work with all types of characters (players, monsters, npcs) but only make sense with players. Using the other two types will work but do nothing at all. Creating a dialog instance always consists of three descrete steps:

1. Create a callback function to be triggered automatically when the user closes the dialog. This function has to have a single parameter to which the dialog will be passed with obtained results.
2. Invoke the constructor of a specific dialog to create a dialog instance, passing required parameters and callback.
3. Have a player object request the dialog in the user's client. Script execution does not stop after this request. The callback with results is called whenever the user closes the dialog.

Usually creating an object would be described before talking about results. However, here we chose a different order to follow the three steps mentioned above, which reflect the order in which you would write a dialog, and to review the most interesting details first, namely what each dialog actually contributes to a particular script.

7.1. MessageDialog

Use this dialog to display bulk text in an on-screen window. The user can close the window at any time.

7.1.1. Results

This type of dialog by it's very nature has no results to be accessed. Still a callback makes sense, because you might want to react on the dialog being closed.

7.1.2. Construction

MessageDialog `MessageDialog(`*text* ⟨*title*⟩, *text* ⟨*text*⟩, *function* ⟨*callback*⟩`)`

Creates a *MessageDialog* with specific ⟨*title*⟩ and message ⟨*text*⟩.

7.1.3. Request

void ⟨*character*⟩`:requestMessageDialog(`*MessageDialog* ⟨*dialog*⟩`)`

7.2. InputDialog

This dialog requests alphanumeric input from the user. If you want to further restrict the input, make that clear to the user with the description and enforce it inside the callback. For different kinds of input (e.g. items) use or request development of a different type of dialog.

7.2.1. Results

boolean ⟨*InputDialog*⟩:`getSuccess()`

The result is ***true*** if the dialog was confirmed and ***false*** if it was aborted.

text ⟨*InputDialog*⟩:`getInput()`

Returns the user's input if the dialog was successful. Otherwise the result is undefined.

7.2.2. Construction

InputDialog `InputDialog(`*text* ⟨*title*⟩, *text* ⟨*description*⟩, *boolean* ⟨*multiline*⟩, *int* ⟨*maxChars*⟩, *function* ⟨*callback*⟩`)`

Creates an *InputDialog* with ⟨*title*⟩ and ⟨*description*⟩. It allows linebreaks iff ⟨*multiline*⟩ is set to ***true***. The input can be up to ⟨*maxChars*⟩ characters long.

7.2.3. Request

void ⟨*character*⟩:`requestInputDialog(`*InputDialog* ⟨*dialog*⟩`)`

7.3. SelectionDialog

With this dialog you can prompt the user to select one of multiple choices. You can use an item graphic with your choices to illustrate them.

7.3.1. Results

boolean ⟨*SelectionDialog*⟩:`getSuccess()`

The result is ***true*** if the dialog was confirmed and ***false*** if it was aborted.

int ⟨*SelectionDialog*⟩:`getSelectedIndex()`

Returns the user's selection if the dialog was successful. Otherwise the result is undefined.

7.3.2. Construction

SelectionDialog `SelectionDialog(`*text* ⟨*title*⟩, *text* ⟨*description*⟩, *function* ⟨*callback*⟩`)`

Creates a *SelectionDialog* with specific ⟨*title*⟩ and ⟨*description*⟩.

void ⟨*SelectionDialog*⟩:`setCloseOnMove()`

If invoked on a SelectionDialog, the dialog will be closed by the server when the player, who owns the dialog, moves.

void ⟨*SelectionDialog*⟩:`addOption(`*int* ⟨*itemId*⟩, *text* ⟨*name*⟩`)`

Adds an option to the dialog. The ⟨*itemId*⟩ stands for an item graphic displayed along with the option described by ⟨*name*⟩. If ⟨*itemId*⟩ is 0 at least once, no graphics are displayed at all. The first option to be added has index 0, increasing from there.

7.3.3. Request

void ⟨*character*⟩:`requestSelectionDialog(`*SelectionDialog* ⟨*dialog*⟩`)`

7.4. MerchantDialog

This dialog is used as an interface to merchant NPCs. All trading with NPCs should be done with this type of dialog. The dialog is kept open until aborted.

7.4.1. Results

int ⟨*MerchantDialog*⟩:`getResult()`

The result is one of MerchantDialog.playerAborts, MerchantDialog.playerSells, MerchantDialog.playerBuys, depending on player action.

int ⟨*MerchantDialog*⟩:`getPurchaseIndex()`

Returns the user's purchase selection if getResult() equals MerchantDialog.playerBuys. Otherwise the result is undefined.

int ⟨*MerchantDialog*⟩:`getPurchaseAmount()`

Returns the amount the user wants to buy if getResult() equals MerchantDialog.playerBuys. Otherwise the result is undefined.

scrItem ⟨*MerchantDialog*⟩:`getSaleItem()`

Returns the item the user wants to sell if getResult() equals MerchantDialog.playerSells. Otherwise the result is undefined.

7.4.2. Construction

MerchantDialog `MerchantDialog(`*text* ⟨*title*⟩, *function* ⟨*callback*⟩`)`

Creates a *MerchantDialog* with a specific ⟨*title*⟩.

void ⟨*MerchantDialog*⟩:`addOffer(`*int* ⟨*itemId*⟩, *text* ⟨*name*⟩, *int* ⟨*price*⟩, *int* ⟨*stack*⟩=1`)`

Adds an offer to the dialog, i.e. something that can be sold to a player. The ⟨*itemId*⟩ stands for an item graphic displayed along with the offer described by ⟨*name*⟩. The first offer to be added has index 0, increasing from there. The ⟨*price*⟩ is given in copper. Optionally a ⟨*stack*⟩ can be given so that only stacks of this amount can be purchased by a player.

void ⟨*MerchantDialog*⟩:`addSecondaryRequest(`*int* ⟨*itemId*⟩, *text* ⟨*name*⟩, *int* ⟨*price*⟩`)`

Adds a secodary request to the dialog, i.e. something that can be sold to an NPC for the regular rate. The ⟨*itemId*⟩ stands for an item graphic displayed along with the request described by ⟨*name*⟩. The ⟨*price*⟩ is given in copper.

void ⟨*MerchantDialog*⟩:`addPrimaryRequest(`*int* ⟨*itemId*⟩, *text* ⟨*name*⟩, *int* ⟨*price*⟩`)`

Adds a primary request to the dialog, i.e. something that can be sold to an NPC for a premium rate. The ⟨*itemId*⟩ stands for an item graphic displayed along with the request described by ⟨*name*⟩. The ⟨*price*⟩ is given in copper.

7.4.3. Request

void ⟨*character*⟩:`requestMerchantDialog(`*MerchantDialog* ⟨*dialog*⟩`)`

7.5. CraftingDialog

This dialog is used as an interface for crafting. All crafting should be done with this type of dialog. The dialog is kept open until aborted.

7.5.1. Results

int ⟨*CraftingDialog*⟩:`getResult()`

The result is one of CraftingDialog.playerAborts, CraftingDialog.playerCrafts, CraftingDialog.playerLooksAtCraftable, CraftingDialog.playerLooksAtIngredient, CraftingDialog.playerCraftingComplete, CraftingDialog.playerCraftingAborted, depending on player action and crafting progress.

int ⟨*CraftingDialog*⟩:`getCraftableId()`

Returns the user's product selection if getResult() equals CraftingDialog.playerCrafts, the product the user looks at if getResult() equals CraftingDialog.playerLooksAtCraftable or the product including an ingredient the user looks at if getResult() equals CraftingDialog.playerLooksAtIngredient. Otherwise the result is undefined.

int ⟨*CraftingDialog*⟩:`getCraftableAmount()`

Returns the amount the user wants to craft if getResult() equals CraftingDialog.playerCrafts. Otherwise the result is undefined.

scrItem ⟨*CraftingDialog*⟩:`getIngredientIndex()`

Returns the ingredient index the user looks at if getResult() equals CraftingDialog.playerLooksAtIngredient. Otherwise the result is undefined.

7.5.2. Construction

CraftingDialog `CraftingDialog(`*text* ⟨*title*⟩, *int* ⟨*sfx*⟩, *int* ⟨*sfxDuration*⟩, *function* ⟨*callback*⟩`)`

Creates a *CraftingDialog* with a specific ⟨*title*⟩ and sound effect ⟨*sfx*⟩ to be played repeatedly while crafting. The duration of one single playback of that sound effect has to be specified in ⟨*sfxDuration*⟩.

void ⟨*CraftingDialog*⟩:`clearGroupsAndProducts()`

Removes all groups and products from the dialog. Can be used e.g. when a user gains skill and the product list has to be created from scratch.

void ⟨*CraftingDialog*⟩:`addGroup(`*text* ⟨*title*⟩`)`

Adds a group with ⟨*title*⟩ to the dialog. Each group is associated with an id, starting at 0.

void ⟨*CraftingDialog*⟩:`addCraftable(`*int* ⟨*index*⟩, *int* ⟨*groupId*⟩, *int* ⟨*itemId*⟩, *text* ⟨*name*⟩, *int* ⟨*deciseconds*⟩, *int* ⟨*stack*⟩=1`)`

Adds a product to the dialog, i.e. something that can be crafted. The ⟨*index*⟩ will be returned by getCraftableId. The ⟨*itemId*⟩ stands for an item graphic displayed along with the product described by ⟨*name*⟩. The first product to be added has index 0, increasing from there. The produt needs a certain amount of ⟨*deciseconds*⟩ to be crafted. Optionally a ⟨*stack*⟩ can be given if more than one item should be created each time this product is crafted.

void ⟨*CraftingDialog*⟩:`addCraftableIngredient(`*int* ⟨*itemId*⟩, *int* ⟨*stack*⟩=1`)`

Adds an ingredient to the product added last. The ⟨*itemId*⟩ stands for an item graphic displayed along with the ingredient. Optionally a ⟨*stack*⟩ can be given if more than one piece of this ingredient is required for crafting the corresponding product.

7.5.3. Request

void ⟨*character*⟩:`requestCraftingDialog(`*CraftingDialog* ⟨*dialog*⟩`)`

7.6. Examples

7.6.1. MessageDialog

```
local callback = function(dialog)
    User:inform("Dialog closed")
end
local lyrics = [[
O Fortuna, velut Luna
statu variabilis,
...
]]
local dialog = MessageDialog("O Fortuna", lyrics, callback)
User:requestMessageDialog(dialog)
```

7.6.2. InputDialog

```
local callback = function(dialog)
    if not dialog:getSuccess() then
        User:inform("You canceled! How dare you?")
    else
        User:inform("You wrote: " .. dialog:getInput())
    end
end
local dialog = InputDialog("Insert some text!", false, 255, callback)
User:requestInputDialog(dialog)
```

8. Items (scriptItem)

There are two kinds of items in lua. This is of the type scriptItem. These types of item-variables are the parameters in the entry point functions (TargetItem etc.). This kind of item variable holds the individual information about the item (position, ...), but not the general ones (weight, ...). It refers to an individual item (stack). You can, however, identify the commonStruct of an individual item, which can be achieved with the `world:getItemStats(`⟨*scriptItem*⟩`)`. Clearly, the other direction is not possible (gaining knowledge about an individual item via a general item). If you change item properties, you have to propagate the changes to the server:

```
item.quality = 284;
item.setData("name", "John's Item");
world:changeItem(item);
```

8.1. Functions

int ⟨*Item*⟩`:getType()`

Return values: `notdefined=0, showcase1=1, showcase2=2, field=3, inventory=4, belt=5`

boolean ⟨*Item*⟩`:isLarge()`

Returns true iff the item is large enough to block the view.

void ⟨*Item*⟩`:setData(`*text* ⟨*key*⟩, *text* ⟨*value*⟩`)`
void ⟨*Item*⟩`:setData(`*text* ⟨*key*⟩, *int* ⟨*value*⟩`)`

Sets the customizable data with the key ⟨*key*⟩ of an item to ⟨*value*⟩. *Example:*

```
...
SourceItem:setData( "magicbonus", 25 );
SourceItem:setData( "prefix", "very strong " );
...
if ( SourceItem:getData("prefix") ~= "" ) then
    if ( tonumber( SourceItem:getData(magicbonus) ) < 30 ) then
        ...
```

Note that a number gets automatically converted into the corresponding string (25).

text ⟨*Item*⟩`:getData(`*text* ⟨*key*⟩`)`

Returns the customizable data for the key ⟨*key*⟩ of an item if that key exists and an empty string otherwise.

Figure 8.1.: Illustration for positions of items. Red: `itempos`, Green: `getType`

8.2. Variables

r: *Character* ⟨*Item*⟩.`owner`

has the type of ⟨*character*⟩.

r: *posStruct* ⟨*Item*⟩.`pos`

has the type of ⟨*position*⟩, this means that the item lies on the floor.

r: *int* ⟨*Item*⟩.`itempos`

Returns the position of an item if it is at a character.

rw: *int* ⟨*Item*⟩.`id` *rw*: *int* ⟨*Item*⟩.`wear`

Measures how long it will take until the item decays.

rw: *int* ⟨*Item*⟩.`quality`

The quality of an item is a combination of actual quality (0-9) and durability (0-99). A quality of 872 stands for actual quality 8 and durability 72. The item decays shortly after durability hits zero. A quality below 100 describes an unfinished item (e.g. for crafting purposes). Quality 64 denotes a 64% completed item for example.

rw: *int* ⟨*Item*⟩.`number`

The number of items on that stack.

8.3. Constants

If an item has an entry "itemName" in itm_name in the items table, you can get its id with Item.itemName.

9. Items (ItemStruct)

Note that there are important functions for items in the chapter "World". This kind of item variable holds general information about an item (weight, ID,...), not individual ones like, for example, the current position or things like that. It is, so to say, a generalized item.

9.1. Variables

r: *int* ⟨*Item*⟩.`id`
r: *int* ⟨*Item*⟩.`AgeingSpeed`
r: *int* ⟨*Item*⟩.`Weight`
r: *int* ⟨*Item*⟩.`ObjectAfterRot`
r: *int* ⟨*Item*⟩.`MaxStack`
r: *boolean* ⟨*Item*⟩.`rotsInInventory`
r: *int* ⟨*Item*⟩.`Brightness`
r: *int* ⟨*Item*⟩.`Worth`
r: *text* ⟨*Item*⟩.`English`
r: *text* ⟨*Item*⟩.`German`
r: *text* ⟨*Item*⟩.`EnglishDescription`
r: *text* ⟨*Item*⟩.`GermanDescription`
r: *int* ⟨*Item*⟩.`Rareness`
r: *int* ⟨*Item*⟩.`Level`

> These variables are accessible for common struct items and script items (where they refer to the corresponding common struct item!).
> *Usage*:
> `MyItem.id`, `MyItem.AgeingSpeed`, ...

10. Weapons and Armor

10.1. WeaponStruct

r: *int* ⟨*weaponstruct*⟩.`Attack`
r: *int* ⟨*weaponstruct*⟩.`Defence`
r: *int* ⟨*weaponstruct*⟩.`Accuracy`
r: *int* ⟨*weaponstruct*⟩.`Range`
r: *int* ⟨*weaponstruct*⟩.`WeaponType`
r: *int* ⟨*weaponstruct*⟩.`AmmunitionType`
r: *int* ⟨*weaponstruct*⟩.`ActionPoints`
r: *int* ⟨*weaponstruct*⟩.`MagicDisturbance`
r: *int* ⟨*weaponstruct*⟩.`PoisonStrength`

WeaponType can be one of the following: WeaponStruct.{slashing, concussion, puncture, slashingTwoHand, concussionTwoHand, punctureTwoHand, firearm, arrow, bolt, stone, stave, shield}

10.2. ArmorStruct

r: *int* ⟨*armorstruct*⟩.`BodyParts`
r: *int* ⟨*armorstruct*⟩.`PunctureArmor`
r: *int* ⟨*armorstruct*⟩.`StrokeArmor`
r: *int* ⟨*armorstruct*⟩.`ThrustArmor`
r: *int* ⟨*armorstruct*⟩.`MagicDisturbance`
r: *int* ⟨*armorstruct*⟩.`Stiffness`
r: *int* ⟨*armorstruct*⟩.`Type`

Type can be one of the following: ArmorStruct.{clothing, general, light, medium, heavy, jewellery}

10.3. NaturalArmor

Monsters can have these intrinsic armor properties:
r: *int* ⟨*naturalarmor*⟩.`strokeArmor`
r: *int* ⟨*naturalarmor*⟩.`thrustArmor`
r: *int* ⟨*naturalarmor*⟩.`punctureArmor`

11. World

11.1. Functions

tleStruct `world:getField(`*posStruct* ⟨*position*⟩`)`

⟨*position*⟩ is a position-structure. The function returns a reference to a field.
Example:

```
Field=world:getField(position(22,10,-3));   -- get reference to "Field"
TileID=Field.tile;                          -- Determine the Tile-ID of that field
```

int `world:getTime(`*text* "⟨*time*⟩"`)`

⟨*time*⟩ can be "year", "month", "day", "hour", "minute", "second" or "unix". The last is the amount of seconds since 1th January 1970 00:00. The others are ingame dates.

void `world:erase(`*scrItem* ⟨*Item*⟩,*int* ⟨*amount*⟩`)`

Example 1:
`world:erase(`TargetItem,3`)`
erases 3 items on the TargetItem-Stack if possible
Example 2:
`world:erase(`TargetItem,0`)`
erases the whole TargetItem-Stack. (NOTE: Temporarily *DISABLED*!) If there are not enough of the items to erase, this function returns "false" and does not delete anything.

void `world:increase(`*scrItem* ⟨*Item*⟩,*int* ⟨*count*⟩`)`

Increases the item number of ⟨*Item*⟩ (⟨*SourceItem*⟩, ⟨*TargetItem*⟩, ...) by ⟨*count*⟩.

void `world:swap(`*scrItem* ⟨*Item*⟩,*int* ⟨*newItemID*⟩,*int* ⟨*quality*⟩`)`

Exchanges ⟨*Item*⟩ (ScriptItem!) with a new one with ⟨*newItemId*⟩ and ⟨*quality*⟩.

scrItem `world:createItemFromId(`*int* ⟨*ItemID*⟩,*int* ⟨*count*⟩,*posStruct* ⟨*position*⟩,*boolean* ⟨*always-flag*⟩,*int* ⟨*quality*⟩,*dataTable* ⟨*data*⟩`)`

where ⟨*position*⟩ is a position-structure and the always-flag is ***true*** (create also when there is already something on that field) or ***false***, depending on how to create the item. It returns a script item sctruct.

void `world:createItemFromItem(`*scrItem* ⟨*Item*⟩,*posStruct* ⟨*Position*⟩,
varalways-flag`)`

where ⟨*Item*⟩ is of the scriptitem-structure. (NOT of the common-structure! Therefore this IS usable with TargetItem!). It creates an identical copy of a scriptitem.

Character `world:createMonster(`*int ⟨monsterID⟩*,*posStruct ⟨position⟩*,*int ⟨movepoints⟩*`)`

Summons a monster with the given monster-ID at the given location. For a list of monster-IDs, please consult the database or a fellow developer.

Character `world:createDynamicNPC(`*text ⟨name⟩*, *int ⟨race⟩*, *posStruct ⟨position⟩*, *int ⟨sex⟩*, *text ⟨scriptname⟩*`)`

Summons an NPC with the given parameters, using the script that is stated.

list `world:LoS(`*posStruct ⟨start position⟩*, *posStruct ⟨end position⟩*`)`

Returns a list that contains lists that contain the type of the list entry and the corresponding items and characters that block the way between *⟨start position⟩* and *⟨end position⟩* and is *nil* otherwise. They can easily be referenced by e.g. `list[1].TYPE`, which returns either `"ITEM"` or `"CHARACTER"` and `list[1].OBJECT` which contains either the item-struct or the character-struct. For the following example, imagine that an item with the item-ID 100 and after that, a character with the character-ID 666 block the way between `startPos` and `endPos`:

```
...
list=world:LoS(startPos, endPos);
if (list ~= nil) then
    for key, listEntry in pairs(list) do
        if (listEntry.TYPE == "ITEM") then
            User:inform("Item with the ID: "..listEntry.OBJECT.id);
        elseif (listEntry.TYPE == "CHARACTER") then
            User:inform("Character with the ID: "..listEntry.OBJECT.id);
        end
    end
else
    User:inform("Nothing blocks the way!");
end
...
```

This will produce the output:

```
Item with the ID: 100
Character with the ID: 666
```

void `world:makeSound(`*int ⟨Number⟩*,*posStruct ⟨position⟩*`)`

Starts soundeffect. 1=scream, 2=sheep, 3=sword hit, 4=thunder, 5=bang, 6=chopping wood, 7=fire, 8=smithing, 9=water splash, 10=pouring in (bottle), 11=saw, 12=drink, swallow, 13=snaring noise

void `world:gfx(`*int ⟨Number⟩*,*posStruct ⟨position⟩*`)`

Starts graphicseffect on *⟨position⟩*.

void `world:changeTile(`*int ⟨TileID⟩*,*posStruct ⟨position⟩*`)`
comItem `world:getItemStats(`*scrItem ⟨Item⟩*`)`

Returns an commonitem-struct from an ⟨*Item*⟩ that is a scriptitem (like TargetItem). Example:

```
myItem=world:getItemStats(TargetItem)
if (myItem.Weight<100) then
    ...
end
```

comItem `world:getItemStatsFromId(`*int* ⟨*ItemID*⟩`)`

Returns an item-struct like `world:getItemStats(`⟨*Item*⟩`)`

scrItem `world:getItemOnField(`*posStruct* ⟨*Position*⟩`)`

Returns a scriptItem on that field.

boolean `world:isItemOnField(`*posStruct* ⟨*Position*⟩`)`
boolean `world:isCharacterOnField(`*posStruct* ⟨*Position*⟩`)`

Returns ***`true`*** for a Character standing on that position and ***`false`*** otherwise.

Character `world:getCharacterOnField(`*posStruct* ⟨*Position*⟩`)`

Returns a character-struct. See chapter "Characters". Example:

```
...
myPosition=position(122,12,3);
if world:isCharacterOnField(myPosition) then
    myPerson=world:getCharacterOnField(myPosition);
    myPerson:talk(Character.say,"You found me!");
end
...
```

Character `world:getPlayersInRangeOf(`*posStruct* ⟨*Position*⟩, *int* ⟨*Range*⟩`)`

Returns a list of character-structs who are in the ⟨*Range*⟩ of ⟨*Position*⟩. See chapter "Characters" and lists in lua.

Character `world:getCharactersInRangeOf(`*posStruct* ⟨*Position*⟩, *int* ⟨*Range*⟩`)`
Character `world:getNPCSInRangeOf(`*posStruct* ⟨*Position*⟩, *int* ⟨*Range*⟩`)`
Character `world:getMonstersInRangeOf(`*posStruct* ⟨*Position*⟩, *int* ⟨*Range*⟩`)`
Character `world:getPlayersOnline()`

Returns a list of character-structs of all players online. See chapter "Characters" and lists in lua.

void `world:changeQuality(`*scrItem* ⟨*ScriptItem*⟩,*int* ⟨*amount*⟩`)`

Changes the quality of a scriptitem (TargetItem, ...) for ⟨*amount*⟩.

void `world:changeTile(`*int* ⟨*tileid*⟩,*posStruct* ⟨*position*⟩`)`

Changes the tile on position-struct "position" to tileid.

void `world:makePersistentAt(`*posStruct ⟨position⟩*`)`

Makes the map persistent over server shutdowns and map imports at ⟨*position*⟩. If no field exists at ⟨*position*⟩, it is created empty.

boolean `world:isPersistentAt(`*posStruct ⟨position⟩*`)`

Returns ***`true`*** if there is a persistent field at ⟨*position*⟩ and ***`false`*** otherwise.

boolean `world:removePersistenceAt(`*posStruct ⟨position⟩*`)`

Removes persistence from a field at ⟨*position*⟩. If no regular map exists there, the field is removed. In this case the server will clean up all characters on this field. Monsters and NPCs are removed, while players will be warped to a nearby field on the same level if possible, or to the starting position otherwise. The script is responsible for cleaning up the field before removal, since those defaults are generally not desirable for players.

text `world:getItemName(`*int ⟨Itemid⟩*,*int ⟨PlayerLanguage⟩*`)`

Returns string that represents the itemname of the item with this id in playerlanguage according to table "itemnames".

void `world:changeItem(`*scrItem ⟨ScriptItem⟩*`)`

Changes a scriptitem against a new one. Handle with care! Example:

```
function UseItem(User, SourceItem)
      SourceItem.id = 1               -- we change the source Item to a sword
      SourceItem.quality = 699        -- a really good sword.
      SourceItem.wear = 10            -- a sword wich rots in a very long time
      world:changeItem(SourceItem)    -- now the item is changed
end
```

boolean , *wpnStruct* `world:getWeaponStruct(`*int ⟨itemID⟩*`)`

Returns two values: bool (true if it is a weapon) and the weaponstruct of the given item (if there is any).
Example:

```
...
foundWp,MyWeapon=world:getWeaponStruct(1);
if (foundWp==true) then
    User:inform("Attack: " .. MyWeapon.Attack .. " def: " .. MyWeapon.Defence);
end
...
```

boolean , *armStruct* `world:getArmorStruct(`*int ⟨itemID⟩*`)`

Returns two values: bool (true if it is an armor) and the armorstruct of the given item.

boolean , *natarmStruct* `world:getNaturalArmor(`*int ⟨raceID⟩*`)`

Returns two values: bool (true if that race has natural armor) and the naturalarmorstruct of the given race.

void `world:broadcast(`*text* ⟨*germanText*⟩, *text* ⟨*englishText*⟩`)`

Sends a broadcast to all players in the game. Players receive the text fitting their account language.

11.2. Variables

`r,w`: *weatherStruct* `weather`

Returns the current weather.

12. Fields

12.1. Functions

In general, these functions will be combined with `world:getField(`*posStruct* ⟨*position*⟩`)` most of the time.

int ⟨*field*⟩`:countItems()`

Returns the number of items that are placed on top of that field.

scrItem ⟨*field*⟩`:getStackItem(`⟨*stackpos*⟩`)`

Returns the item with position ⟨*stackpos*⟩ (0 being the bottom item) within the pile of items on this field. If ⟨*stackpos*⟩ exceeds the number of items on that field, a 0-item is returned (id=0), therefore it is a good idea to check the number of items on that field first.

boolean ⟨*field*⟩`:isPassable()`

Determines whether a field allows a character to pass over it or having items dropped on it.

12.2. Variables

`r`: *int* `tile`

Returns the tile ID of that field. Recommended use with `world:getField(`⟨*position*⟩`)`.

13. It's a kind of magic

13.1. Global variables

`thisSpell`

Refers to the ID of the spell that is currently casted.

13.2. Some words on magic

Casting a spell is done by selecting one or more runes and eventually selecting a target. We assign numbers to these runes like in fig(13.1). Every spell gets a unique spell-ID which is entirely determined

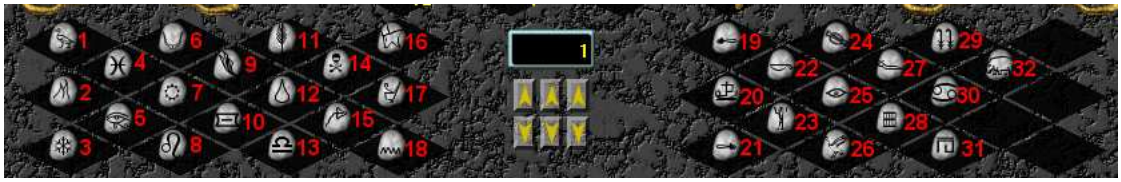

Figure 13.1.: The runes

by the used runes. Suppose we have to use the runes with the numbers $a_1 \dots a_n$ to cast that spell, the spellId can then be calculated by

$$I_{\text{spell}} = \sum_{k=1}^{n} 2^{a_k - 1} = 2^{a_1 - 1} + 2^{a_2 - 1} + \cdots + 2^{a_n - 1}. \tag{13.1}$$

To give a concrete example: Imagine for your spell you have to use runes 2 and 5. The spell Id then is

$$I_{\text{spell}} = \sum_{k=1}^{2} 2^{a_k - 1} = 2^{a_1 - 1} + 2^{a_2 - 1} = 2^{2-1} + 2^{5-1} = 2^1 + 2^4 = 2 + 16 = 17. \tag{13.2}$$

The caption of every spell script should include a brief description of the spell, the rune combination and the SQL insert statement (as comments, of course):
`INSERT INTO spells VALUES(`⟨*spellID*⟩`,`⟨*magicType*⟩`,'`⟨*scriptname.lua*⟩`')`
In our case, that might be: `INSERT INTO spells VALUES(17,0,'m_17_fireball.lua')`

14. Weather

Till now, weather is a global effect. Once you set the weather to a specific value, it's the same everywhere. Eventually there will be "areas" of weather in the future. A weatherStruct is a set of different variables, just like any other struct so far. Altering these variables changes the weather.

14.1. Variables

rw: *int* ⟨*weatherStruct*⟩.`cloud_density`

Varies between 0 (no clouds) and 100 (full clouds).

rw: *int* ⟨*weatherStruct*⟩.`fog_density`
rw: *int* ⟨*weatherStruct*⟩.`wind_dir`
rw: *int* ⟨*weatherStruct*⟩.`gust_strength`
rw: *int* ⟨*weatherStruct*⟩.`percipitation_strength`
rw: *int* ⟨*weatherStruct*⟩.`percipitation_type`
rw: *int* ⟨*weatherStruct*⟩.`thunderstorm`
rw: *int* ⟨*weatherStruct*⟩.`temperature`

14.2. Functions

14.3. Entry point

There is just one entry point for weather scripts: *function* `changeWeather()`

Is invoked everytime the weather should be changed.

15. Long time effects

15.1. Basic idea

Long time effects (LTEs) allow you to influence a character over a period of time. The values are saved in the database, so they endure even a server crash.

LTEs are bound to a character. They basically consist of:

- The name (only visible in the database) and the ID of the LTE
- A script that defines the LTE
- A counter that counts how often an effect had been called already
- A variable that controls when this effect on this character will be called again
- Several self defined variables that can be accessed by a key string and that can hold an integer.

First, one has to add the LTE to the database with a unique ID, a name and a script name, e.g. `42` is the ID, `myeffect` the name and `myeffect.lua` the script.

Then one has to add the effect to a character, e.g. inside an item script. How the effect works has to be defined in its script `myeffect.lua`. Everytime the LTE is called, its script is invoked; to be exact, the function `callEffect` is invoked, see the section about Entry Points. There one can change the character's attributes etc. Note that you should always save any change of fixed attributes as a value of the LTE, so you can restore everything when the LTE ends.

If the character logs out, all values are saved. If he logs in again, the function `loadEffect` in the LTE script is invoked. Any temporary change of attributes will be gone by now, so here you can read the value you have saved and do the change again.

The example at the end of this chapter will help you a lot.

15.2. Functions

void ⟨*effect*⟩:`addValue(`*text* ⟨*name*⟩,*int* ⟨*value*⟩`)`

> ⟨*name*⟩ is an arbitrary name for a variable that can be introduced and filled with ⟨*value*⟩ and is added to that effect. It can later (at one of the following calls, for example) be read or changed again. Note that ⟨*value*⟩ can be any integer x, but it will be saved as y, where $0 \leq y < 2^{32}$ and $x \equiv y \pmod{2^{32}}$.

void ⟨*effect*⟩:`removeValue(`*text* ⟨*name*⟩`)`

> ⟨*name*⟩ is the name of a value that will be removed from that effect.

boolean , *int* ⟨*effect*⟩:`findValue(`*text* ⟨*name*⟩`)`

> This function returns `true` if a value ⟨*name*⟩ is found plus its value and `false` otherwise. Note that these are two values!

boolean , effStruct ⟨*Character*⟩`.effects:find(`⟨*effect-ID*⟩`)`

This function returns `true` if an effect with ⟨*effect-ID*⟩ was found and the respective ⟨*effect*⟩, `false` otherwise.

void ⟨*Character*⟩`.effects:addEffect(LongTimeEffect(`⟨*effect-ID*⟩,*int* ⟨*nextCalled*⟩`))`

This function adds the effect ⟨*effect-ID*⟩ to a character. One also has to set the initial ⟨*nextCalled*⟩ value. Thus, right after the script, which contains this command, is completely executed, the function `addEffect` in the effect's script is called and, if not changed in `addEffect`, the function `callEffect` is called after ⟨*nextCalled*⟩$\cdot \frac{1}{10}$ seconds.

boolean ⟨*Character*⟩`.effects:removeEffect(`⟨*effect-ID*⟩`)`

This function removes the effect ⟨*effect-ID*⟩ from a character. It returns a boolean which indicates whether that worked or not.

15.3. Variables

`r`: *int* ⟨*effect*⟩`.effectId`
`r`: *int* ⟨*effect*⟩`.effectName`
`r,w`: *int* ⟨*effect*⟩`.nextCalled`
IMPORTANT: `nextCalled` must not exceed $2^{31} - 1$, otherwise the effect won't be safed properly.
`r`: *int* ⟨*effect*⟩`.lastCalled`
`r`: *int* ⟨*effect*⟩`.numberCalled`

15.4. Entry points for longtime effects

Inside that script which was invoked, there are several possible entry points that can be called:
function `callEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

MUST either return `true` if the effect should be called again or `false` if not! ⟨*Effect*⟩`.nextCalled` has to be set. It will be lowered by 1 every $\frac{1}{10^{\text{th}}}$ second and `callEffect` will be called as soon as it reaches 0.

function `addEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

Is invoked when an effect is newly created.

function `removeEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

Is invoked after an effect ended (by having `callEffect` return `false`.

function `doubleEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

Is invoked when an effect is added to a character that already has that effect. Note that a character can hold just one effect of one type at a time!
This function is currently bugged, as all effect values are deleted when the effect is re-added (see Mantis issue #451).

function `loadEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

Is invoked when a player character logs into the game. It should be used to set temporary stats changes and so on, which can be stored in effect-variables, using `findValue` and so on.

15.5. Example: Adding long time effects to characters

Imagine the following situation: You drink a potion of a fluid and after that, you get "drunk", that means that your perception and agility are lowered and you sometimes make uncontrolled steps for the next 4 minutes. The first thing to do is to create a table entry in `longtimeeffects` in the following way:

```
 lte_effectid | lte_effectname | lte_scriptname
--------------+----------------+----------------
          666 | alcohol        | lte_alcohol.lua
```

To start with, we need to script the bottle (`bottle.lua`) which adds the effect 666 (`alcohol`) to the character drinking that bottle.

```
function UseItem(User, SourceItem, LTstate)

    foundEffect, alcEffect = User.effects:find(666); -- does effect #666 already exist?

    if (not foundEffect) then              -- if that effect is not there...
        alcEffect = LongTimeEffect(666,10); -- create new effect
        User.effects:addEffect(alcEffect);    -- add effect #666
                            -- funct. "addEffect(...)" in the LT-script will be called.
                            -- 1 second until first call of "callEffect(...)"
        foundEffect = User.effects:find(666);   -- does effect #666 exist now?
        if (not foundEffect) then              -- effect not found (security check)
            User:inform("An error occured, inform a developer.");
            return;                              -- exit immediately if not found!
        end
    end

    alcEffect:addValue("alcLevel",10);      -- sets the alcLevel-value to 10.
    alcEffect:addValue("strMod",-5);        -- sets modifier for strength to -5.

end
```

So, this script simply adds `alcEffect` (666) to the `User` of the bottle and adds the value `alcLevel` to this effect and sets it to 10.

The next thing to be done is to define this effect. This is done in the actual long time script we defined in the database before, `lte_alcohol.lua`:

```
function addEffect(myEffect, Character)     -- called only the first time
    Character:inform("You feel a little bit dizzy.");
    found, strMod = myEffect:findValue("strMod");
    if found then                           -- read the str modificator
        Character:increaseAttrib("strength",strMod);
    else                                    -- if modificator is not found
        Character:inform("Error, please inform a developer");
        myEffect:addValue("strMod",-5);     -- set to a default value
    end
end

function callEffect(myEffect, Character)    -- is called everytime
    found, alcLevel = myEffect:findValue("alcLevel");
```

```
                                        -- get value
    Character:talk(Character.say,"Hick!"); -- Hick!
        -- add some more effects
    if not found then
        Character:inform("Error, please inform a developer!")
        return false;                   -- bug occured! remove effect
    else
        if (alcLevel>0) then            -- alcohol still has effect
            myEffect.nextCalled=200;    -- next call in 20 s.
            myEffect:addValue("alcLevel",alcLevel-1);
                                        -- just override the value
            return true;
        else                            -- alcohol has no more effect
            myEffect:removeValue("alcLevel");
            return false;               -- return false and remove effect
        end
    end
end

function loadEffect(myEffect, Character)    -- is called when the character logs in
                                        -- do the STR change again
    Character:inform("You still feel a little bit dizzy.");
    found, strMod = myEffect:findValue("strMod");
    if found then                       -- read the str modificator
        Character:increaseAttrib("strength",strMod);
    else                                -- if modificator is not found
        Character:inform("Error, please inform a developer");
        myEffect:addValue("strMod",-5);     -- set to a default value
    end
end
```

15.6. Ideas for usage

Long time effects can be used for many effects, here are just some ideas:

- Illness, epidemy, infections and deseases
- Injuries
- Effects of potions (of all kinds)
- Effects of poison
- Punishment
- ...

16. Delayed execution and disturbation

There is a way to have a script being executed after some time. Of course, this could also be done using long time effects, which were described above. However, there is something that long time effects can't detect: If "something" happened between two invokations of an effect. Take, for example, a magician who casts a spell. Assume that there is a delay between casting that spell and having an effect (it needs some time of concentration). What happens if, for example, this mage is disturbed during the concentration phase (because he's under attack or alike)? There is no way to detect that in long time effect scripts.

16.1. Functions

void ⟨*Character*⟩:`startAction(`*int* ⟨*time*⟩, *int* ⟨*GFX-ID*⟩, *int* ⟨*GFX-interval*⟩, *int* ⟨*SFX-ID*⟩, *int* ⟨*SFX-interval*⟩`)`

> This starts an action for the character for ⟨*time*⟩ $\frac{1}{10th}$seconds. The GFX with ID ⟨*GFX-ID*⟩ is shown every ⟨*GFX-interval*⟩ $\frac{1}{10th}$seconds. The SFX with ID ⟨*SFX-ID*⟩ is played every ⟨*SFX-interval*⟩ $\frac{1}{10th}$seconds.

void ⟨*Character*⟩:`changeSource(`*Character* ⟨*item*⟩`)`
void ⟨*Character*⟩:`changeSource(`*scrItem* ⟨*item*⟩`)`
void ⟨*Character*⟩:`changeSource(`*posStruct* ⟨*position*⟩`)`

> Changes the source variable of the entrypoint used for subsequent calls of the current action. This has to be called explicitly to propagate changes of a source object to the action.

16.2. Constants

int `Action.none`
int `Action.abort`

16.3. Usage

In all "Use"-functions like `UseItem(...)`, the last parameter is the integer ⟨*ltstate*⟩. Its value is one of the constants above. So one can check the ⟨*ltstate*⟩ and eventually start an action.

There can only be one action at a time for a character. If multiple actions are started in the same script, only the last one counts (nevertheless the "first" sound is played and the "first" gfx is shown for every action immediately, if the interval is greater than 0).

After starting an action, the script is still executed until its end. Then after the ⟨*time*⟩ OR if the character got interrupted (was attacked, moved, used something...), the script is called again and is normally executed. So make sure to check the ⟨*ltstate*⟩!

16.4. Example

A simple example of a potion script is helpful:

```
function UseItem(User, SourceItem, ltstate)

    -- check for ltstate == Action.abort
    -- means the script got interrupted before the time needed was up
    -- (-> drinking was not finished!)
    if (ltstate == Action.abort) then

        -- Cast forced emotes from the Charakter who uses our potion
        -- (german for germans, english for the rest)
        User:talk(Character.say, "#me verschÃ¼ttet den Trank.", "#me spills the potion.")
        -- [...] possibly remove the bottle etc.
        -- since the user failed to drink the potion we are done now
        return
    end

    -- [...] possibly check if the character is in attackmode

    -- Now we check if the character is drinking the potion currently or not
    -- (since the drinking process needs some time)
    if (ltstate == Action.none) then
        -- Action.none so the character does nothing.
        -- Lets open the bottle and drink the potion

        -- Start the action!
        -- 2,0 seconds until the action is done
        -- GFX id 0 is shown while the waiting time ( so no gfx )
        -- every 0 seconds the GFX is shown ( so never )
        -- SFX id 15 is played ( drinking sound )
        -- every 2,5 seconds. So the sound is only played once (at the beginning)
        User:startAction(20,0,0,15,25);

        -- let's tell everyone that our user drank a potion with a forced emote
        User:talk(Character.say, "#me beginnt einen Trank zu trinken.", "#me starts to trink a potio

        -- And quit the script since we are done now and waiting for the next call
        return
    end

    -- since we are here the character started already to drink
    -- but got not interrupted. Now we offer some results

    -- [...] possibly remove the bottle, give healthpoints and foodpoints
    -- and slow down the character
    -- [...] etc.

end
```

17. Waypoints

The waypoint system allows us to use the pathfinding algorithm of the server for moving monsters and NPCs through the terrain. There is an internal list of position data which is handled like a queue, so first in - first out. The character always tries to reach the first waypoint, until the list is empty.

The algorithm checks for obstacles and a free way only in a certain radius. This process is quite expensive (and even more important: the server freezes until the algorithm finishes!), so the radius shouldn't be too big; around 15 tiles should suffice (the default value is currently unknown, but it won't be much more).

17.1. Functions

The waypoint list can be accessed by ⟨*character*⟩.`waypoints` and the following operations are possible:

void ⟨*character*⟩.`waypoints:addFromList(`⟨*waypointlist*⟩`)`

Adds a whole Lua list of position structs to the end of the internal list.

void ⟨*character*⟩.`waypoints:addWaypoint(`⟨*pos*⟩`)`

Adds a single position struct ⟨*pos*⟩ as last waypoint to the internal list.

list (posStruct) ⟨*character*⟩.`waypoints:getWaypoints()`

Returns the internal list as Lua list of position structs.

void ⟨*character*⟩.`waypoints:clear()`

Clears the internal list.

For controling the character there are these functions:

void ⟨*character*⟩:`setOnRoute(`*boolean* ⟨*toggleRoute*⟩`)`

The character starts the route if ⟨*toggleRoute*⟩ is ***true*** and stops if it is ***false***.

boolean ⟨*character*⟩:`getOnRoute()`

Returns if the character is currently on a route.

boolean , int ⟨*character*⟩:`getNextStepDir(`*posStruct* ⟨*pos*⟩, *int* ⟨*rangeToCheck*⟩`)`

Returns two values. The first is ***true*** if a way to ⟨*pos*⟩ is found considering only a radius of ⟨*rangeToCheck*⟩ tiles, ***false*** otherwise. The second is the direction of the next step for the found way.

This function does not move the character, it just returns the direction for the next step.

list (int) ⟨*character*⟩:`getStepList(`*posStruct* ⟨*pos*⟩, *int* ⟨*rangeToCheck*⟩`)`

Same as `getNextStepDir(`b`)`ut returns a complete step list to reach ⟨*pos*⟩.

17.2. Entry Points

If a character (Monster/NPC) is on a route, no other handling inside the server is done (no fighting) but all the script entry points like enemyNear or enemyInSight are called.

There is one special entry point for both monsters and NPCs:

function `abortRoute(`⟨*character*⟩`)`

> Is called if the ⟨*character*⟩ has reached the destination or if items block the way and the destination can't be reached.

18. Global Scriptvariables

These ScriptVars allow us to save a value in the database and access it by an identifier string. Use with caution as they have global effect. All ScriptVars are automatically saved upon a server shutdown.

18.1. Functions

void `ScriptVars:set(`⟨*identifier*⟩, ⟨*value*⟩`)`

Writes the integer or string ⟨*value*⟩ to the ScriptVar ⟨*identifier*⟩.

boolean , ⟨*value*⟩ `ScriptVars:find(`⟨*identifier*⟩`)`

Returns if there exists a ScriptVar ⟨*identifier*⟩. If ***true*** then its ⟨*value*⟩ is returned aswell.

boolean `ScriptVars:remove(`⟨*identifier*⟩`)`

Removes the ScriptVar ⟨*identifier*⟩, returns ***true*** if there was such a value.

void `ScriptVars:save()`

Saves all ScriptVars immediately. Use with caution as it could cause freezes.

19. Random

Most of the time the random number generator supplied by Lua should be enough. However, if you are looking for a statistically sound implementation of a random number generator, or need another distribution besides the uniform one, this is the class you are looking for.

19.1. Functions

double `Random.uniform()`

Returns a random floating-point value uniformly distributed in the range [0..1).

int `Random.uniform(`*int* ⟨*min*⟩, *int* ⟨*max*⟩`)`

Returns a random integer value uniformly distributed in the range [⟨*min*⟩..⟨*max*⟩].

double `Random.normal(`*double* ⟨*mean*⟩, *double* ⟨*standard_deviation*⟩`)`

Returns a random floating-point value normally distributed with given ⟨*mean*⟩ and ⟨*standard_deviation*⟩.

20. Debugging

While using luac -p for finding compilation errors is very straight forward, it is much more difficult to find runtime errors. The script error log which can be viewed via `http://illarion.org/~nitram/show_log.php` for the testserver and via `http://illarion.org/~vilarion/show_log.php` for the illarionserver shows all runtime errors and where they occured. These logs should be checked after every change and subsequent testing. Keeping the logs clean will help others to find their own bugs faster.

20.1. Functions

void `debug(`*text* ⟨*debugMessage*⟩`)`

> If used on the testserver prints ⟨*debugMessage*⟩ to the script log together with a stacktrace showing where the call of `debug(...)` originated. Has no effect on the illarionserver.

21. Entry Points

21.1. Items

function `UseItem(`⟨*User*⟩, ⟨*SourceItem*⟩, ⟨*ltstate*⟩`)`

When one item is used. If it is used with another item, ⟨*TargetItem*⟩ is the ID of that item, otherwise it is 0.

function *ItemLookAt* `LookAtItem(`⟨*User*⟩, ⟨*Item*⟩`)`

When someone looks at an item. Needs to return ItemLookAt as described in table 21.1.

function `MoveItemBeforeMove(`⟨*User*⟩, ⟨*SourceItem*⟩, ⟨*TargetItem*⟩`)`

Is invoked when someone tries to move an item before the move is commited. If this function returns *false* the move of the item will not be carried out; that can be used for cursed items. Basically, ⟨*SourceItem*⟩ is the item before it was moved, ⟨*TargetItem*⟩ is the item after it was moved.
`IMPORTANT:` It MUST return either *true* or *false*, otherwise the server crashes! (*return true;*)

function `MoveItemAfterMove(`⟨*User*⟩, ⟨*SourceItem*⟩, ⟨*TargetItem*⟩`)`

Is invoked after someone moved an item. See also `MoveItemBeforeMove(.)`

function `NextCycle()`

Is invoked every 10 seconds for commonitems.

function `CharacterOnField(`⟨*User*⟩`)`

Is invoked if someone steps on that item (which therefore lies on the floor); good for traps and fields. This function requires that the corresponding item has a specialitem-flag in the db-table tilesmodificators

21.2. NPC

For effective usage of NPCs and their scripts please read the section about string handling.
function `nextCycle(`⟨*npc*⟩`)`

Is invoked every few server cycles (=approximately constant time intervalls, $\frac{1}{10}$s). *IMPORTANT:* MUST exist in NPC scripts!

function `receiveText(`⟨*npc*⟩, ⟨*TextTyp*⟩, ⟨*Text*⟩, ⟨*Originator*⟩`)`

Is invoked if the NPC hears someone speaking (even himself!).

function `useNPC(`⟨*npc*⟩, ⟨*User*⟩`)`

Is invoked if the NPC is used (shift-click) by ⟨*User*⟩ without target.

function `lookAtNpc(`⟨*npc*⟩, ⟨*SourceCharacter*⟩, ⟨*mode*⟩`)`

Is invoked if the player ⟨*SourceCharacter*⟩ looks at the NPC. ⟨*mode*⟩ describes what kind of lookat is done: 0 means normal, 1 means close examination.

ItemLookAt field	type	item data	description
name	string	"nameDe" "nameEn"	
rareness	ItemLookAt. commonItem uncommonItem rareItem epicItem	"rareness"	
description	string	"descriptionDe" "descriptionEn"	
craftedBy	string	"craftedBy"	
type	string	-	derived from WeaponStruct or ArmorStruct
level	number	-	0..100
usable	boolean	-	can the player use this?
weight	number	-	
worth	number	-	selling price in copper
qualityText	string	-	
durabilityText	string	-	
durabilityValue	number	-	0..100 in percent
diamondLevel	number	"magicalDiamond"	magic gem level: 0..10
emeraldLevel	number	"magicalEmerald"	magic gem level: 0..10
rubyLevel	number	"magicalRuby"	magic gem level: 0..10
sapphireLevel	number	"magicalSapphire"	magic gem level: 0..10
amethystLevel	number	"magicalAmethyst"	magic gem level: 0..10
obsidianLevel	number	"magicalObsidian"	magic gem level: 0..10
topazLevel	number	"magicalTopaz"	magic gem level: 0..10
bonus	number	-	gem bonus: 0..255

Table 21.1.: ItemLookAt member variables

21.3. Magic

function `CastMagic(`⟨*Caster*⟩`)`

Is invoked when ⟨*Caster*⟩ casts a spell without target.

function `CastMagicOnCharacter(`⟨*Caster*⟩, ⟨*TargetCharacter*⟩`)`

Is invoked when ⟨*Caster*⟩ casts a spell on another character/monster (⟨*TargetCharacter*⟩).

function `CastMagicOnField(`⟨*Caster*⟩, ⟨*pos*⟩`)`

Is invoked when ⟨*Caster*⟩ casts a spell on a field at the position ⟨*pos*⟩.

function `CastMagicOnItem(`⟨*Caster*⟩, ⟨*TargetItem*⟩`)`

Is invoked when a spell is casted on an item.

21.4. Monsters

function `onDeath(`⟨*Monster*⟩`)`

Is invoked as a monster dies.

function `receiveText(`⟨*Monster*⟩, ⟨*TextTyp*⟩, ⟨*Text*⟩, ⟨*Originator*⟩`)`

Is invoked when a monster (⟨*Monster*⟩) receives spoken text.

function `onAttacked(`⟨*Monster*⟩, ⟨*Attacker*⟩`)`

Is invoked when a monster is attacked.

function `onCasted(`⟨*Monster*⟩, ⟨*Caster*⟩`)`

Is invoked when a spell is casted on a monster.

function `useMonster(`⟨*Monster*⟩, ⟨*User*⟩`)`

Is invoked when a monster is used by ⟨*User*⟩

function `onAttack(`⟨*Monster*⟩,⟨*Enemy*⟩`)`

Is invoked every time when a monster would hit the enemy.

function `enemyOnSight(`⟨*Monster*⟩,⟨*Enemy*⟩`)`

Is invoked every time when a monster sees an enemy. *IMPORTANT*: MUST return true (did something) or false (did nothing)! It is not invoked when the monster stands on a field next to the enemy.

function `enemyNear(`⟨*Monster*⟩,⟨*Enemy*⟩`)`

Is invoked every time when a monster sees an enemy and stands next to it. Works exactly like `enemyOnSight(,)` MUST return true or false! But beware: If you plan to have `setTarget(...)` return 0 ("don't attack anyone"), `enemyNear(...)` must return false, otherwise the server is caught in an endless loop!

function `lookAtMonster(`⟨*SourceCharacter*⟩, ⟨*monster*⟩, ⟨*mode*⟩`)`

Is invoked if the player ⟨*SourceCharacter*⟩ looks at the ⟨*monster*⟩. ⟨*mode*⟩ describes what kind of lookat is done: 0 means normal, 1 means close examination.

function `onSpawn(`⟨*Monster*⟩`)`

Is called immediately after the ⟨*Monster*⟩ has spawned. There one can e.g. set it on a route using waypoints.

function `setTarget(`⟨*Monster*⟩, ⟨*CandidateList*⟩`)`

If setTarget exists, it is called whenever ⟨*Monster*⟩ need to decide what it should attack. ⟨*CandidateList*⟩ is a list of players who are possible targets. To set a target, it's index in that list has to be returned. The first enemy in the list has the index 1. If the function returns 0, the monster will ignore any enemy. If `setTarget()` returns 0 ("don't attack anyone"), `enemyNear()` must return false otherwise the server is caught in an endless loop!

If setTarget does not exist, the player with lowest health is chosen by default.

21.5. Fields

function `useTile(`⟨*User*⟩,⟨*Position*⟩`)`

Is invoked when a tile is shift-clicked (used).

function `MoveToField(`⟨*User*⟩`)`

Is invoked if a character moves on that triggerfield (entry in "triggerfields" necessary).

function `MoveFromField(`⟨*User*⟩`)`

Is invoked if a character moves away from that triggerfield.

function `PutItemOnField(`⟨*Item*⟩,⟨*User*⟩`)`

Is invoked if an item is put on that triggerfield.

function `TakeItemFromField(`⟨*Item*⟩,⟨*User*⟩`)`

Is invoked if an item is taken away from that triggerfield.

function `ItemRotsOnField(`⟨*oldItem*⟩,⟨*newItem*⟩`)`

Is invoked when ⟨*oldItem*⟩ rots into ⟨*newItem*⟩ on a triggerfield.

21.6. Quests

While you can create quests with setQuestProgress and getQuestProgress, it is much nicer to have an actual log of that progress in your client. This way you cannot forget about your quests and you always know what to do and where to do it. The client might even point you in the right direction. The entry-points in this section implement that behaviour.

text *function* `QuestTitle(`*Character* ⟨*user*⟩`)`

You should return the quest title here, depending on user language.

text *function* `QuestDescription(`*Character* ⟨*user*⟩, *int* ⟨*status*⟩`)`

You should return the quest description here, depending on user language and quest ⟨*status*⟩. You can be as extensive as you want, but make sure to cover the most important points. A good description should serve as a reminder where to go to complete the next step of the quest, let the player know how to get there and what to do there. Imagine a player continuing a quest after some time, they still need to know where they are at and how to go on.

posStruct ***function*** `QuestTargets(`*Character* ⟨*user*⟩, *int* ⟨*status*⟩`)`

Here you should return the position where the quest continues, i.e. where a new quest status can be obtained. The client will receive this position and direct the player towards it. You can also return ***nil*** or an empty list if no such positions should be displayed in the client. Omit positions with care. Even if you think it would be cool to let players search on their own, most of the time it is just annoying. You can also return a list of positions here if the quest can continue in more than one location.

int ***function*** `QuestFinalStatus()`

Return the final status of your quest here. This is used by the server to determine whether the end of a quest has been reached. If so, the client is notified to display the quest as completed.

21.7. Scheduled Scripts

Scheduled scripts allow the execution of functions in arbitrary scripts in certain intervals.
function ⟨*functionname*⟩()

This script is invoked by having an entry in the database table "`scheduledscripts`" after some given time intervall.
Example:

```
  sc_scriptname   | sc_mincycletime | sc_maxcycletime | sc_functionname
------------------+-----------------+-----------------+----------------
 scheduletest.lua |              14 |              16 | makeeffect
```

This will invoke the function `makeeffect()` in the script `scheduletest.lua` every 14-16 seconds (a random time between 14 and 16 seconds).

21.8. Server Scripts

Some scripts provide a special global service for the server. These scripts are not attached to a specific item, character or field, but define general behaviour. All of these scripts are located in server/. Server scripts should be edited with great caution, since breaking one of those scripts would break the related behaviour for the server.

21.8.1. Combat (standardfighting.lua)

Everything related to physical combat is handled by standardfighting.lua. It is called everytime a character tries to hit another character.

function `onAttack(`*Character* ⟨*Attacker*⟩, *Character* ⟨*Defender*⟩`)`

Is invoked for every attack by ⟨*Attacker*⟩ against ⟨*Defender*⟩.

21.8.2. Login (login.lua)

This script is invoked when a player logs in. Here you can e.g. display login information or tax players.

function `onLogin(`*Character* ⟨*player*⟩`)`

Called when ⟨*player*⟩ logs in.

21.8.3. Logout (logout.lua)

This script is invoked when a player logs out. Here you can e.g. substitute GM faction leaders by their NPC equivalent.

function `onLogout(`*Character* ⟨*player*⟩`)`

Called when ⟨*player*⟩ logs out.

21.8.4. Learning (learn.lua)

function `learn(`*Character* ⟨*char*⟩, *Skill* ⟨*skill*⟩, *int* ⟨*actionPoints*⟩, *int* ⟨*opponent*⟩`)`

Called when Character:learn is invoked. See there for a description of paramters. Implements the learning system.

function `reduceMC(`*Character* ⟨*player*⟩`)`

Called every 10s for every player who is online. Use to reduce mental capacity necessary for learning.

21.8.5. Death (playerdeath.lua)

function `playerDeath(`*Character* ⟨*deadPlayer*⟩`)`

When a player dies this function is called by the server. It should handle e.g. penalties for dying.

21.8.6. Depot Access (depot.lua)

function `onOpenDepot(`*Character* ⟨*player*⟩, *scrItem* ⟨*depot*⟩`)`

When a player tries to open a depot, this function is called. It has to return either ⟨*true*⟩ if opening the depot is successful or ⟨*false*⟩ otherwise.

21.8.7. Player LookAt (playerlookat.lua)

function `lookAtPlayer(`*Character* ⟨*player*⟩, *Character* ⟨*targetPlayer*⟩, *int* ⟨*mode*⟩`)`

Is invoked when the ⟨*player*⟩ looks at the ⟨*targetPlayer*⟩. ⟨*mode*⟩ describes the type of lookat performed: 0 means normal, 1 means close examination.

21.8.8. Item LookAt (itemlookat.lua)

function *ItemLookAt* `lookAtItem(`*Character* ⟨*player*⟩, *scrItem* ⟨*item*⟩`)`

Handles basic item lookat if there is no lookat script attached to the given item. Has to return an ItemLookAt as described in table 21.1.

21.8.9. Reloading Scripts (reload.lua)

function `onReload()`

Invoked after server start as well as after the !fr command has been issued. Possibility to set defaults or initialize global variables.

22. Lua

22.1. Important commands

For a good summary of the importand commands and how they work look at http://lua-users.org/wiki/TutorialDirectory
Of special interest are: *for, if, function, while* and the concept of lists.

22.2. Built in functions

math.random()

Returns a random number between 0 (incl.) and 1 (excl.).

math.random(⟨*Upper*⟩)

Returns a random integer between 1 and ⟨*Upper*⟩ (inclusive).

math.random(⟨*Lower*⟩,⟨*Upper*⟩)

Returns a random integer between ⟨*Lower*⟩ and ⟨*Upper*⟩ (inclusive).

math.abs(⟨*number*⟩)

Returns the absolute value of a number. Example: *math.abs*-4.2 -¿ 4.2

math.ceil(⟨*number*⟩)

Rounds ⟨*number*⟩ to the next higher integer.

math.floor(⟨*number*⟩)

Rounds ⟨*number*⟩ to the next lower integer.

table.getn(⟨*List*⟩)

Returns the number of entries in a table.

string.find(⟨*text1*⟩,⟨*text2*⟩)

Returns *nil*, if ⟨*text2*⟩ was not found in ⟨*text1*⟩. If, however, ⟨*text1*⟩ contains ⟨*text2*⟩, it returns the position of the starting character of ⟨*text2*⟩ in ⟨*text1*⟩, the end position of ⟨*text2*⟩ in ⟨*text1*⟩ and, in case one uses so called *captures*, all the captures found. *Captures* are a powerfull concept for stings to analyze them by pattern matching. See the lua wiki.

22.3. Binary operators

`A=B;`

A will get the value of B.

`A==B`

A is compared with B; *true* for A=B else *false*. Used in *if* statements and alike.

`A~=B`

A is compared with B; *false* for A=B else *true* (true if A and B are not equal).

22.4. Lists

Lists are collections of variables. Lists can be created by the following simple procedure: # Start a Lua-list and insert the entries you want (itemIDs etc.) # Run through the list (with a loop) and do what you need (add them to a menue etc.) # Start the process defined by 1. and 2. (send the menue to the player) Creating a list is easy:

`ListA`=⟨*value1*⟩,⟨*value2*⟩,⟨*value3*⟩,...

You can access elements of that table by

`ListA`[⟨*number of element*⟩]

for instance

`ListA[2]`

would be

⟨*value2*⟩

Creating a loop is easy as well:

```
for i = 1,5 do
      ...
end
```

runs through "..." 5 times, the first time with i=1, then i=2, ... to i=5. Combining that with a list would give:

```
ListA={value1,value2,value3,value4,value5}
for i = 1,5 do
    -- do something with ListA[i]
end
```

Example 1: Lets say we want to have a list of items added to a menue.

```
ItemList={45,54,67,81,110,145,215}       -- create list
UserMenu=MenuStruct()                    -- make new menu
for i = 1,7 do                           -- start loop
    UserMenu:addItem ItemList[i]         -- add Item to menu
end
User:sendMenu UserMenu                   -- send menu
```

Example 2: Lets say we want to have a list of items which are only accessible for the player for certain skills. We create a difficulty list.

```
ItemList={45,54,67,81,110,145,215}                        -- create list
DiffList={ 1, 7,45,90, 25, 45, 65}                        -- list of difficulties
UserMenu=MenuStruct()                                     -- make new menu
for i = 1,7 do                                            -- start loop
    if (User:getSkill("smithing")>=DiffList[i]) then      -- if User has enough skill
        UserMenu:addItem(ItemList[i])                     -- add Item to menu
    end
end
User:sendMenu UserMenu                   -- send menu
```

If you are filling a list, you have to take care about the following:
Example 3: Filling a list with entries

```
MyList={};         -- initialize list (IMPORTANT!)
MyList[1]=12;
MyList[2]="Hello";
MyList[3]=56;
...
```

The important part is the first line: Without it, the script would not work. Lets now look to multidimensional lists (tables, for example):
Example 4: Tables

```
MyTable={};
MyTable[1]={};
MyTable[2]={};
MyTable[1][1]=23;
MyTable[1][2]=45;
MyTable[1][3]=34;
MyTable[2][1]="Maoam";
MyTable[2][2]="Hello";
MyTable[2][3]="Hi there!";
```

22.5. Modules: Using functions and variables of other lua files

Each script is in its own module. This is defined by:

```
local M = {}
return M
```

Now we can use this module, say npc/base/myscript.lua in another script:

```
local myscript = require("npc.base.myscript")
```

and we can use its variables like this: `myscript.myfunction()`

Example:

`base/file1.lua`:

```
local M = {}

M.testNumber = 3

function M.DoSomething(User,text)
    User:inform(text)
end

return M
```

`npc/file2.lua`:

```
local M = {}
local file1 = require("base.file1");

function M.TestingModules(Character)
    if (file1.testNumber == 3) then
        file1.DoSomething(Character, "Testing this feature!")
    end
end

return M
```

22.6. A note on namespaces, ambiguities and variable declaration

Each module has its own namespace, so there won't be any ambiguities with other scripts. But note that every variable or function that is not declared as *`local`* are global and will be present in the whole namespace of the module. At that point it does not matter if the variable is declared inside or outside a function body. All variables that are declared outside all function bodies are loaded only once.
There exists only one instance of every module, so e.g. all items of one kind (or that are bound to the same script) will use the very same instance of the module and therefore share all global variables. So if you use a variable only inside one function, declare it as `local`, so it will be dumped after the function call has ended and there is no possibility to interfere with other variables and to create any ambiguities. However, if there are two variables or functions (or a function and a variable) with the exact same name, the last definition will always override all previous ones. Note that Lua does not provide any mean for function overloading.

23. String handling

This is an important topic, as it is relevant for the use of Lua for NPCs (and eventually monsters). The seemably most important function is:
string.find(⟨*text1*⟩, ⟨*text2*⟩)

> Returns a number that indicates the position in text1 of the beginning of the first occurence of text2 in text1 and another number that indicates the last position of the last occurence.

Example:

```
a,b=string.find("Hello world","llo");
   -> a=3, b=5

a,b,c,d=string.find("I buy 20 shoes",".*buy (%d+) (.+)");
   -> a=0, b=14, c="20", d="shoes"
```

- Expressions in brackets "(...)" are returned to the variables. Without them, we would just have a and b.
- "`.`" means: any character, digit, just anything.
- "`*`" means: repetition of the previous, including 0 repetitions; '.*' therefore could mean any string, including an empty one ("").
- "`+`" means nearly the same as '*', except that it has to have at least 1 repetition, therefore the empty string is not included.
- "`%s`" simply means a space (" "). "`a%sb`" therefore means "`a b`".
- "`%d`" means any digit. Together with "tt +" we have "`%d+`", which means: at least one digit, but it can be more.
- "`[Ff]`" would mean: The character must be a "`F`" or a "`f`". "`[Hh][Ee][Ll]+[Oo]`" therefore can be "`hello`" or "`helo`" or "`HeLlo`" or "`heLLLlO`" or...

Therefore the above "`.*buy (%d+) (.+)`" means: Search for a string where you have:

1. any characters or nothing
2. followed by "buy"
3. followed by a space (" ")
4. followed by (at least) one or more digits
5. followed by space
6. followed by one or more characters of any type (could be "shoes", but could also be "!!98(jj" or "hallo" or "9982")

Beware: c and d are both strings, even if they contain a number like "23". If you perform mathematical operations with them (c*2), they behave like numbers, if you compare them (`if c==23`), they behave like strings, meaning that (`c="23"; if c==23 then...`) will NOT work, whereas (`c="23"; if c*1==23 then...`) WILL work, because c*1 is converted into a number.

If a string is not found inside another string, it returns *nil*.

23.1. File I/O

It is possible to read and write data from/into files. It is important to use files and directories where the scripts are permitted to read and write.

Example:

```
filepoint,errmsg,errno=io.open("/home/martin/scrdata/testing.luadat","r");
thisline=filepoint:read("*line");
User:inform("This line reads as: "..thisline);
filepoint:close();

filepoint,errmsg,errno=io.open("/home/martin/scrdata/testing.luadat","w+");
filepoint:write("User "..User.name.." called that script!");
filepoint:close();
```

For further information see the official lua documentation (`http://www.lua.org`)

24. Examples

24.1. Items

Let us first begin with something simple. Say we want to have a script for a sword with the item-ID 27 (fictional) which, when shift-clicked should simply be deleted. The first thing to do is to create an empty file like "simple_sword.lua" in some text editor (be sure that it uses unix-style end-of-lines!). This file needs a UseItem-function, because the sword should disappear when it is used (shift-clicked). Then we need to write down the command for deleting that item. That's it.

```
-- simple_sword.lua
function UseItem(User, SourceItem)
   world:erase(TargetItem,1);
end
```

That will do the job. Now we only need to copy this script to /usr/share/testserver/scripts/ (via svn!) and make an entry in the commons-table of the database into the com_script colum for item 27 which reads simple_sword.lua. Only do a #r inside Illarion's testserver and it works. Let's say that we want to extend our script a little. The character should know that he has deleted something. We add an extra line that informs the player:

```
-- simple_sword.lua
function UseItem(User, SourceItem)
   world:erase(TargetItem,1);
   User:inform("You have deleted that damn sword!");
end
```

Copy that file over the old one, do a #r and here we go. As soon as you shift-click the sword, the sword disappears and you get the message "You have deleted that damn sword!". We are still not satisfied with that. Nono. We want to give out some information about that sword, too. It's weight for example. Now, how to do that? This is a little more complex (only a little) because there are two "types" of items that lua knows: the one is the kind of variable like "TargetItem", which does not know anything about it's weight or other properties. The other one knows everything about itself. So we first have to convert TargetItem into such an object. Then we need to get the weight of that. That is done as follows:

```
-- simple_sword.lua
function UseItem(User, SourceItem)
   world:erase(TargetItem,1);
   MyItem=world:getItemStats(TargetItem);
   MyWeight=MyItem.Weight;
   User:inform("You have deleted that damn sword which weights "..MyWeight);
end
```

Proceed as before. Let's go on. Next step is: We want to create a new item as soon as the old is destroyed. Let's say the item with the ID 28. Not just one, but, say, 5 of them.

```
-- simple_sword.lua
function UseItem(User, SourceItem)
   world:erase(TargetItem,1);
   User:createItem(28,5,333);
   MyItem=world:getItemStats(TargetItem);
   MyWeight=MyItem.Weight;
   User:inform("You have deleted that damn sword which weights ".MyWeight);
end
```

Now, how simple is THAT? Wow. We are still not satisfied. For one reason or another, we do not want to delete that sword in any case. We only want to delete it if the corresponding character does NOT carry a magic key (fictional ID: 30) anywhere on his body. How about that then?

```
-- simple_sword.lua
function UseItem(User, SourceItem)
   keys=User:countItem(30);
   if (keys==0) then
      world:erase(TargetItem,1);
      User:inform("You have deleted that damn sword which weights ".MyWeight);
   end
   User:createItem(28,5,333);
   MyItem=world:getItemStats(TargetItem);
   MyWeight=MyItem.Weight;
end
```

This can easily be extended with all the functions and commands listed above. However, let us now turn to more advanced examples. Take, for example, a lockable door. There are several possibilities to lock a door, only limited by the scripters creative mind. The most basic one seems the following: Asume you have two versions of a door: an open one (fictional ID: 50) and a closed one (fictional ID: 51). This door stands on the fictional coordinates (30,30,0). The principle works as follows: closed=locked, open=unlocked, you replace the closed door (50) by the open one (51) if someone uses the correct key (fID: 60) with the closed door. This means that the opening and closing of that door entirely lies in the script of the key (as it is the first object to be shift-clicked!).

```
-- key_door.lua
function UseItem(User, SourceItem)
   MyDoor=world:getItemStats(TargetItem);
   if (MyDoor.id==50) then
      MyDoorPosition=TargetItem.pos;
      DesiredPosition=position(30,30,0);
      if (MyDoorPosition=DesiredPosition) then
         world:erase(TargetItem,1);
         world:createItemFromId(51,1,DesiredPosition,true,333)
      end
   elseif (MyDoor.id==51) then
      MyDoorPosition=TargetItem.pos;
      DesiredPosition=position(30,30,0);
      if (MyDoorPosition=DesiredPosition) then
         world:erase(TargetItem,1);
```

```
          world:createItemFromId(50,1,DesiredPosition,true,333)
       end
    end
 end
```

This can of course be done in a more elegant way, as the same structure appears twice. However, for learning purposes, this is more obvious: First we check the items ID; if it fits, we check the items position; if that fits, we delete it and create the opened (closed) version instead.

24.2. NPCs

IMPORTANT: NPCs MUST have a `nextCycle()` function, even if it is empty!
Simple NPCs are as simple as simple item scripts. However, they can grow rather large and be arbitrary complex. Lets start with a simple one: He should simply react on "Greetings" or "greetings".

```
function receiveText(npc, texttype, message, originator)
   if string.find(message,"[Gg]reetings") ~= nil then
       npc:talk(Character.say, "Greetings, my friend.");
   end
end
```

Note that we will need string operations intensively. The first one is hidden in "[Gg]reetings", which means that the first letter can be both, a "G" or a "g". If you implement that and test it, the NPC will not react. The reason is rather simple: He does not understand you. In fact, he does not understand any language at all. So we need to increase his language skill, common language preferably. But once he learnd that, he does not need to learn it again, so he only needs to learn it one time. Here comes one thing in quite handy: a script does not forget variables once set; if a variable was never set before, it is nil. So, to check if the variable was set ever before, we need only to check if it is nil.

```
function receiveText(npc, texttype, message, originator)
   if iniVar == nil then
       iniVar=1;
       npc:increaseSkill(1,"common language",100);
   end
   if string.find(message,"[Gg]reetings") ~= nil then
       npc:talk(Character.say, "Greetings, my friend.");
   end
end
```

However, this one will only react on the second string he "hears", because after the first one, he learns common language and doesn't understand anything. Afterwards, he will understand common language. However if we want to add several keywords, we would have an endless and unelegant sequence of if...elseif...elseif...elseif...elseif...end. To avoid that, we can use simple lists where we store the keywords and the reactions and then just loop through them. We do not need to "load" the lists everytime someone talks to our NPC but only once for each server restart, meaning that we could initialize the lists like we increase the language skill of the NPC.

```
function receiveText(npc, texttype, message, originator)
```

```
    if iniVar == nil then
        iniVar=1;
        npc:increaseSkill(1,"common language",100);
        NpcTrig=();
        NpcAnsw=();

        NpcTrig[1]="[Gg]reetings";
        NpcAnsw[1]="Greetings, my friend.";
        NpcTrig[2]="[Hh]ello";
        NpcAnsw[2]="Hello. How are you?";
    end
    for i=1,table.getn(NpcTrig) do
        if string.find(message,NpcTrig[i])~=nil then
            npc:talk(Character.say, NpcAnsw[i]);
        end
    end
end
```

To summarize: We fill the trigger texts and the answers into a list and then search in a loop the received message for a trigger text in that list; if we find one, we let the NPC speak the corresponding answer. We can create very simple dialog. It might be the case, and this is hoped much, that you want to create more complex NPCs than just "question" "answer" things. For example, lets add another thing to this NPC: We want him to do a simple calculation and add together two numbers we tell him. Furthermore we note that, once this NPC found a trigger in a received message, we do not want to search if there is another trigger in the message. We will therefore restructure the for-loop and make a repeat..until-loop instead, which is easier to stop once we found something.

```
function receiveText(npc, texttype, message, originator)
    if iniVar == nil then
        iniVar=1;
        npc:increaseSkill(1,"common language",100);
        NpcTrig={};
        NpcAnsw={};

        NpcTrig[1]="[Gg]reetings";
        NpcAnsw[1]="Greetings, my friend.";
        NpcTrig[2]="[Hh]ello";
        NpcAnsw[2]="Hello. How are you?";
    end
    i=0;
    foundTrig=false;
    repeat
        i=i+1;
        if string.find(message,NpcTrig[i])~=nil then
            npc:talk(Character.say, NpcAnsw[i]);
            foundTrig=true;
        end
    until (i==table.getn(NpcTrig) or foundTrig==true)
    if (foundTrig==false) then
        if(string.find(message,"%d++%d")~=nil then
```

```
            StartsAt,EndsAt,numberOne,numberTwo=string.find(message,"(%d+)+(%d+)";
            npc:talk(Character.say,"This is "..(numberOne+numberTwo));
        end
    end
end
```

25. Common bugs

- Missing `end`, missing `(` or `)`, script name and db-entry do not match (take care of invisible characters! Try a search in the db with e.g. `SELECT FROM testserver.spells WHERE spl_scriptname='spells.p_28')`
- A "." instead of the seperator ":" or vice versa. ("." is for variables, ":" for functions)
- Missing `()` for functions that don't need parameters.
- Incorrect number of parameters for functions.
- Mispelled function names (use syntax highlighting!).
- Forgot !fr in game to reload tables and scripts.
- = instead of == or vice versa.
- != instead of ~=.
- Missing conversion of a string to a number (when reading from a string).
- Using a variable that does not exist in this function (e.g. originator in `function nextCycle`...
- Beware of endless loops; they freeze the server. Always ensure that the script terminates.
- Program parts after return statement.
- Forgotten "then" in if-commands, forgotten "end" in inline-if's.

A. Versions

Version 5.22 (20 05 28)
* Add map persistence
* Add world::makePersistentAt
* Add world::isPersistentAt
* Add world::removePersistenceAt
* Remove world::sendMapUpdate

Version 5.21 (15 06 23)
* Add item id constants
* Move item level from weapons and armors to item struct

Version 5.20 (15 06 07)
* Remove obsolete bit manipulation functions
* Update section about modules

Version 5.19 (15 01 15)
* Add colour class, which represents RGBA values
* Rename colour related methods and change their signature to use colour objects

Version 5.18 (13 12 14)
* Add Item:isLarge

Version 5.17 (29 07 14)
* Add world:broadcast

Version 5.16 (02 07 14)
* Remove world:itemInform

Version 5.15 (20 06 14)
* Added CommonStruct member: English
* Added CommonStruct member: German
* Added CommonStruct member: EnglishDescription
* Added CommonStruct member: GermanDescription
* Added CommonStruct member: Rareness

Version 5.14 (23 05 14)
* Added Character:logAdmin

Version 5.13 (15 09 13)
* Added Character:isNewPlayer
* Updated ItemLookAt

Version 5.12 (08 08 13)

* Added Character:isBaseAttributeValid
* Added Character:getBaseAttributeSum
* Added Character:getMaxAttributePoints
* Added Character:saveBaseAttributes
* Added Character:setBaseAttribute
* Added Character:getBaseAttribute
* Added Character:increaseBaseAttribute

Version 5.11 (05 08 13)
* Added Character:setRace
* Added Character:turn

Version 5.10 (23 07 13)
* Removed isTestserver

Version 5.9 (05 04 13)
* Added ArmorStruct.Type, ArmorStruct.Level
* Added global constants for ArmorStruct.Type
* Added WeaponStruct.Level
* Added global constants for WeaponStruct.Type
* Added ItemLookAt.level, ItemLookAt.armorType, ItemLookAt.weaponType

Version 5.8 (25 03 13)
* Removed obsolete function Character:changeQualityItem
* Removed obsolete function Container:changeQuality

Version 5.7 (05 03 13)
* Added time as return value of getQuestProgress

Version 5.6 (01 03 13)
* Added numeric constants for char:getType()
* Added a note about how LTE values are saved

Version 5.5 (26 02 13)
* Removed deprecated talkLanguage command
* Added localised overload for talk command

Version 5.4 (01 02 13)
* Added quest entrypoints

Version 5.3 (25 01 13)
* Added setCloseOnMove for SelectionDialog

Version 5.2 (12 01 13)
* Removed counter from all entrypoints
* Removed param from all entrypoints
* Removed target item from UseItem entrypoint

Version 5.1 (08 01 13)

* Added isTestserver() for debugging

Version 5.0 (27 12 12)
* New major version for server version 0.9 (VBU)
* Added overload of insertContainer with item position
* Added Character:changeSource
* Added CraftingDialog

Version 4.24 (20 11 12)
* Changed Character:learn parameter name
* Changed all data based item search functions
* Changed all data based item delete functions
* Removed isStackable from commonStruct
* Added MaxStack to commonStruct
* Added Character:getSkillName
* Removed French from Character:getPlayerLanguage
* Removed magic numbers from Character:getPlayerLanguage

Version 4.23 (27 10 12)
* Removed deprecated function equapos (since 4.10)
* Removed deprecated old item data (since 4.18) and corresponding functions
* Added overload of Container:countItem with data
* Updated world:itemInform
* Renamed scheduled scripts chapter
* Removed Use*With* entrypoints
* Added all server entrypoints
* Added separate section for Container
* Added SelectionDialog
* Added MerchantDialog
* Adjusted everything skill related to new skill handling

Version 4.22 (12 10 12)
* Added random number generators

Version 4.21 (02 10 12)
* Added description to InputDialog

Version 4.20 (01 10 12)
* Removed Character:LTIncreaseHP
* Removed Character:LTIncreaseMana
* Removed Character:tempChangeAttrib
* Added Field:isPassable

Version 4.19 (13 09 12)
* Added new character:inform overload with NLS

Version 4.18 (12 09 12)
* Marked all functions using old data as deprecated
* Added functions with new data parameters where necessary

* Added integer overload for Item:setData
* Added integer values for dataTable type

Version 4.17 (11 09 12)
* Removed strange parameter from Container:weight

Version 4.16 (31 08 12)
* Updated Character:inform to include the priority parameter

Version 4.15 (20 08 12)
* Removed characterOnSight entrypoint for NPCs
* Removed characterNear entrypoint for NPCs
* Removed global thisNPC
* Changed syntax of NPC entrypoints

Version 4.14 (19 07 12)
* Added playerDeath entry point
* Removed Character.death_consequences property

Version 4.13 (04 07 12)
* Added pageGM command for players
* Removed everything UserMenu related, esp. the Menus chapter
* Added Dialogs chapter
* Added MessageDialog
* Added InputDialog

Version 4.12 (17 02 12)
* Changed incorrect function name getMonType to getMonsterType

Version 4.11 (09 01 12)
* Removed volume property of containers and items

Version 4.10 (06 08 11)
* Added operator == for position
* Marked function equapos as deprecated
* Added lua standard function tostring for position

Version 4.9 (04 08 11)
* Added CommonStruct member: isStackable
* Added CommonStruct member: rotsInInventory
* Added CommonStruct member: Brightness

Version 4.8 (24 07 11)
* Added skin/hair/color variation commands
* Added set/getData

Version 4.7 (04 07 11)
* Removed C prefix of server classes

Version 4.6 (20 04 11)
* Added isValidChar
* Added chapter on debugging

Version 4.5 (15 04 11)
* Added CommonStruct.Worth
* Added Character:sendBook
* Added entrypoint setTarget for Monsters
* Added and corrected data based item removal
* Added Character:defaultMusic
* Added Character:idleTime
* Modified Character:learn
* Brought Common Bugs up to date.

Version 4.4 (15 04 10)
* Added Waypoints
* Added ScriptVars
* Reworked LTEs
* Completed Delayed execution and disturbation
* Many other minor changes and additions

Version 4.3 (30 04 06)
* Added QuestProgress functions
* Added scheduledscripts
* Marked Longtimeeffects as active
* Corrected viewItemNr

Version 4.2 (10 11 05)
* Minor changes and additions (data)

Version 4.1 (23 09 05)
* Added new weapon struct variable
* Added combat functions

Version 4.0.0 (13 08 05)
* Added container commands
* Added variable types
* Removed some minor bugs

Version 3.2.0 (16 07 05)
* Updated dofile
* Added file io
* Updated index
* Changed layout
* Added new graphic

Version 3.1.1 (11 06 05)
* Deleted wrong version of itemInform
* Added index

Version 3.0.1 (11 06 05)
* Deleted wrong version of getItemName

Version 3.0.0 (10 06 05)
* Converted to LaTeXformat
* Deleted unnecessary chapters
* Some additions, correcting mistakes, ...

Version 2.6.2 (02 06 05)
* Minor additions

Version 2.6.0 (29 05 05)
* Added new entry points
* Small corrections
* New commands

Version 2.5.0 (20 04 05)
* Added bugs
* Minor corrections and adaptions

Version 2.4.1 (18 04 05)
* Added further NPC examples
* Corrected minor formating bug
* Minor additions

Version 2.4 (14 04 05)
* Corrected minor errors
* Added new commands
* Added better description of items
* Added starts of NPC tutorial

Version 2.3.1 (06 04 05)
* Minor additions and corrections

Version 2.3 (01 04 05)
* Added a new section for a tutorial
* Minor corrections

Version 2.2.8 (29 03 05)
* Minor corrections
* Minor additions

Version 2.2.7 (27 02 05)
* Converted to WIKI-format
* Some language corrections
* Added some chapters from other versions

Version 2.2.6 (19 11 04)

* Corrected world:makeSound(...)

Version 2.2.5 (15 11 04)
* Corrected world:gfx(...)

Version 2.2.4 (11 11 04)
* Minor additions
* Minor regrouping of skill/attribute-commands

Version 2.2.3 (10 11 04)
* Added chapter "Built in functions"

Version 0.2.2 (09 11 04)
* Minor correction on "world:erase".
* Deleted/Clearified last ?-lines.

Version 0.2.1 (07 11 04)
* Added chapter "Item" and some content

Version 0.2 (07 11 04)
* Added Entry points

Version 0.1.1 (04 11 04)
* Corrected sendMenu-command

Version 0.1 (03 11 04)
* Roughly organized character-commands by topic
* Deleted ⟨*character*⟩:`depot(-)`command
* Changed sendMessage() to inform()
* Added some short descriptions
* German to english translations
* Changed world:erase-command