The ORBIT Programming language

OFFICIAL DOCUMENTATION

Getting Started

Orbit is a custom programming inspired by languages such as Lua & JavaScript. It was created in early 2022, and its backend is written in Java.

It is used on the NightRealm Minecraft Server, on the Nations gamemode.

- Regarding Nations, computers with NightRealm OS (Build Jasmine or above) come with both a compiler and an integrated development environment preinstalled on them.
- Regarding Java Development, an official library allowing you to run your own orbit code is currently in the works, so stay tuned!
- Regarding Visual Studio Code, an official extension is planned, but any information regarding it still has to be announced.

Comments

Ironically, the first thing we are going to do is tell the computer to ignore a part of the written code. Text written in a program that isn't run by the computer is called a comment.
Comments in orbit are marked by `//` at the start of the line.

Comments can be used in many ways such as helping other people to understand the code better or ignoring a line of code to see how the program would work without it.

```
// The line below prints a message
print("Hello, World!")
```

Output

The `print()` function can be used to output text on the screen or, respectively, to the player.

```
print("Orbit is awesome!")
```

```
Orbit is awesome!
```

It also has one additional use. When used on NightRealm, you can also specify to what players to send the output text to.

```
print("your text goes here.", aPlayer, anotherPlayer)
```

Variables

Variables are containers used for storing data and values.
The types of variables existent in orbit are:

- `int` - Represents integers / whole numbers.
- `float` - Represents floating point numbers.
- `bool` - Has only two possible states: true and false.
- `string` - Represents written text.

Creating Variables

To create a variable, also known as declaring a variable, you must specify a name, a type, and, optionally, a value:

```
type variableName = value
```

Operators

Operators are used to perform operations on variables and values.

Arithmetic Operators:
As the name suggests, arithmetic operators are used to perform mathematical operations.

- + - Addition. Adds together two values.
- - - Subtraction. Subtracts one value from another.
- * - Multiplication. Multiplies two values.
- / - Division. Divides one value by another.
- % - Modulus. Returns the division remainder.
- ++ - Increment. Increases the value of a variable by 1.
- -- - Decrement. Decreases the value of a variable by 1.

Assignment Operators:

assignment operators are used when assigning values to variables.

- = - X = Y
- += - X += Y | Same as X = X + Y
- -= - X -= Y | Same as X = X - Y
- *= - x *= Y | Same as X = X * Y
- /= - X /= Y | Same as X = X / Y
- %= - X %= Y | Same as X = X % Y

Comparison Operators:

Comparison operators are used to compare values.

- == - Equal to
- != - Not equal
- > - Greater than
- < - Less than
- >= - Greater than or equal to
- <= - Less than or equal to

Logical Operators:

Logical operators are used to perform logical operations.

- && - AND. Returns true if both statements are true.
- || - OR. Returns true if any statement is true.
- ! - NOT. Negates the statement (From true to false, respectively from false to true)

Lists

Lists are used to store multiple values in a single variable, instead of needing to declare a variable for each individual value.

To declare a list, you must specify a name and a type:

```
list<type> listName
```

When declaring it, you can also insert values into it, separated by commas, inside square brackets:

```
list<type> listName = [var1, var2, var3]
```

Accessing Elements

You can access the elements of a list by referring to the index.

Note: List indexes start at 0. The first element is [0], the second is [1], the third is [2] and so on...

```
print(listName[0])
```

Changing Elements

To change an element of a list, you must again, refer to its index.

```
listName[0] = var
```

Note: Lists are dynamic, meaning that if an element does not exist, it will be created.

List Length

You can find out how many elements a list has using the `List#size()` function.

```
list<int> numbers = [1,2,3,4,5]
print(numbers.size())
```

The Player Type

Note: The following section applies only for Nations.

The `player` type is a special variable type that can hold information about any player that is currently online.

This type does not do much on it's own, but there are a few functions in the STD library which use it:

- `player::getPlayer(string name)`
- `list<player>::getPlayersInRange(float range)`
- `int::getX(player p)`
- `int::getY(player p)`
- `int::getZ(player p)`

These are all return-type functions, meaning that their result get assigned to a variable.

If... Else

The `if` statement is used to execute a block of code only if a certain condition is true. The condition is specified after the `if` keyword, and the code is placed between the `then` keyword and the `end` keyword.
If the condition is not met, the code in the `else` block is executed.

```
if condition then
  // the code to execute
end
```

```
if condition then
  // the code to execute
else
  // code if condition is false
end
```

While Loop

The while loop is a control flow statement that allows a block of code to be executed until a given condition has been satisfied. The while loop can be thought of as a repeating if statement.

```
while condition do
  // the code to execute
end
```

For Loop

The for loop is a control flow statement for specifying iteration. Specifically, a for loop functions by running a section of code repeatedly until a certain condition has been satisfied.

```
for assignment; condition; increment do
  // the code to execute
end
```

The assignment is used to initialize a loop variable. This can be any valid orbit expression.

The increment is a statement that is executed at the end of each iteration.

Break Statement

The break keyword is used to exit out of a loop early, before the loop condition is no longer true. It is often used in conjunction with an if statement to provide a way to exit the loop based on some condition.

```
for i = 1; i <= 10; i++ do
  if i == 5 then
    break
  end

  print(i)
end
```

Declaring Functions

In Orbit, you can create your own functions to reuse code and make your program more organized. To declare a function, you can use the following syntax:

```
function functionName() does
  // the code to execute
fend
```

Calling Functions

To call a function, you can use the function name followed by parentheses:

```
functionName()
```

Return Statement

The `return` keyword works very similarly to the `break` keyword. It is used to exit out of a function at any point. If the program isn't currently in a function, the `return` keyword will instead stop the execution of the program.

Returning Values

Currently, custom functions cannot return values. However, some of the built-in functions can return values and these are called return-type functions. Here are some examples on how they are properly used:

```
string input
input::getInput()
```

```
player andrej
andrej::getPlayer("AndrejFish_")
```

The Standard Library

The Standard (Also known as **STD**) Library contains the base and the most important variables and methods used in the whole language.

STD Methods

- `print(string message)`
  - Prints a message to the screen.
- `print(string message, player... players)`
  - Prints a message to the given players.
- `broadcast(string message)`
  - Broadcasts a message to all players in a 45 block radius.
- `broadcast(string message, float radius)`
  - Broadcasts a message to all players in a given radius.
  - Radius has to be smaller or equal to the default 45.
- `sleep(float ticks)`
  - Pauses the execution of the code for a given amount of ticks.
  - (1 Second = 20 Ticks).
- `sound(string sound, float pitch)`
  - Plays a sound, with the given pitch.
  - Here is a list of the sounds you can play.

- `string::getInput()`
  - Opens an input box, and assigns the value of the given text to the variable.

STD Variables

- `playerHealth` - The health of the player running the script.
- `playerFood` - The hunger of the player running the script.
- `nationsBalance` - The balance of the player running the script.
- `phoneNumber` - The phone number of the player running the script.

The Utility Library

The Utility Library contains multiple "sub libraries". One of them is Random, allowing you to generate random integers. The other being "Unsafe", giving you access to unsafe functions such as eval.

Utility Methods

Random:

- `setRand(int min, int max)`
  - Generates a random value to the **rand** variable.

Unsafe:

- `eval(string code)`
  - Allows you to dynamically run a line of code.
- `goto(int index)`
  - Allows the program to jump to a specific line in the code and continue execution from there.

Utility Variables

- `rand` - Used by the setRand() function.

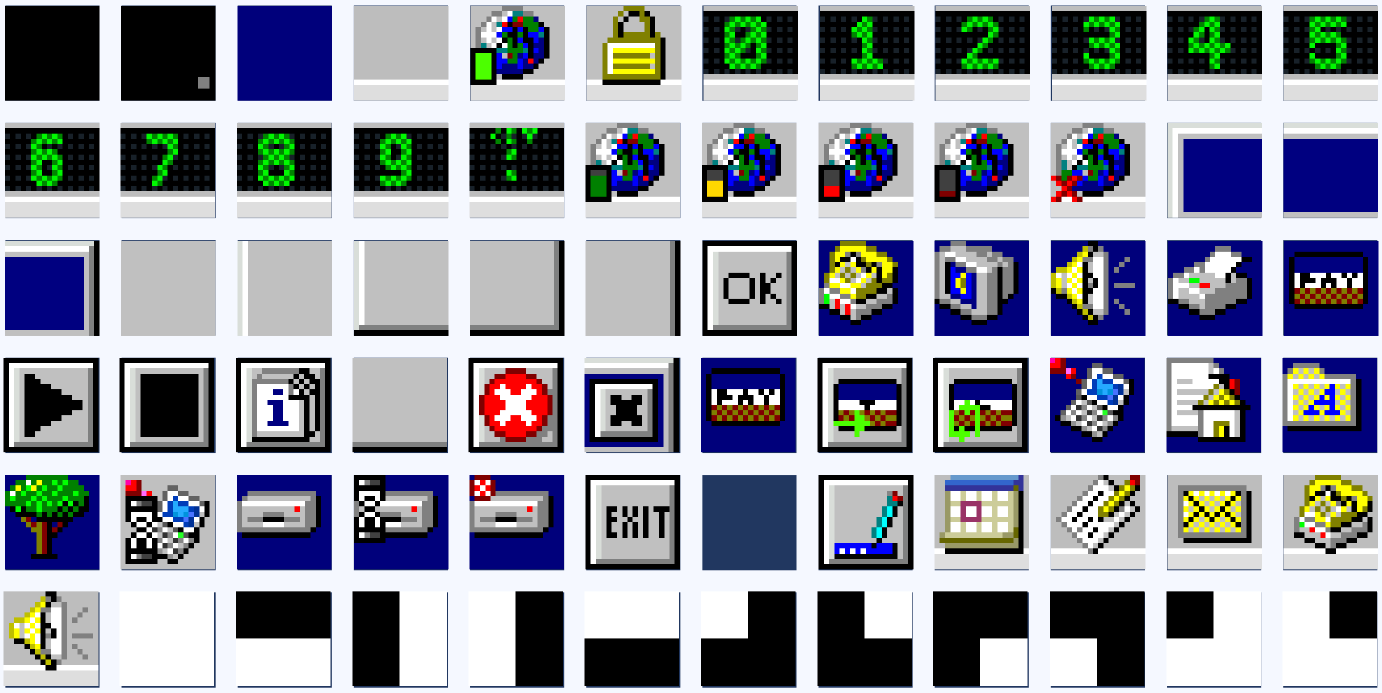
The GUI Library

The Graphical User Interface (Also known as **GUI**) Library contains the required tools to display a custom Screen to the player.

GUI Methods

- openGui(string title)
  - Opens a GUI with a given title.
- closeGui()
  - Forcefully closes the GUI (if any is open).
- poke(int slot, int data)
  - Changes a slot's graphics, based only on data.
- poke(int slot, int data, string color)
  - Changes a slot's graphics, based on both data and a hex color.
  - It is important to note to not put the # at the start of the color.
- fill(int data)
  - Changes the whole screen to a given data value.
- fill(int data, string color)
  - Changes the whole screen to a given data and hex color value.
- clear()
  - Clears the whole GUI, removing any data value.

Note: The usable slots of the GUI are 0-53. Any other values given will be ignored. Usable data goes from 1-74 and is represented by:



NOTE: There are 2 additional icons which did not fit on the table.