Minr Scripts 2.0 SPECIFICATION

Minr

February 15, 2019

Contents

1 Introduction

A short introduction on the document and structure, including notation and practices maintained within the document.

1.1 Introduction

Previous versions of the scripting language (Scriptblock, MSC 1) have shown that scripts are extremely powerful and often simpler than command blocks. However, these versions also had shortcomings. Scriptblock ended up being outdated, causing all interact scripts to be fired twice upon interact, and some operators such as @delay and @cooldown could break by a player logging out. To circumvent this, we developed a Minr-specific version, Minr Scripts (MSC), which initially aimed to solve these issues. Since the codebase was ours, we were able to add additional operators, features and more, which made scripting even more powerful.

With this expansion of features the correct usage, functionality, shortcomings, bugs, and dangers with implementing became obscured, causing unwanted behaviour in scripts and confusion among scripters.

Additionally, variables were all globally stored, dynamically typed, and rather verbose to manipulate (each operation takes one line). Scripts were unable to be reused, nor was it easy to mass-edit a script, requiring third-party mods.

MSC 2 attempts to solve these shortcomings in variables and scripts by the addition of typed variables, namespaces, functions, hastebin-based import and export of scripts, and the addition of expressions in scripts, allowing for easier manipulation of variables.

This document attempts to clarify the features, shortcomings and dangers of MSC 2, combined with good practice and examples of use cases.

1.2 Structure

MSC 2 contains a lot of new features. Additionally, to remain compatible with major features and standards from MSC 1, many features of MSC 1 have been leveraged, which means that the core concept of creating and editing scripts remains the same.

Each feature will be extensively handled in its own chapter, with the chapters slowly building up the required knowledge.

For your first read it may be best to read from the top to bottom, because the document is structured keeping this in mind. For future reference, the Appendix can be used, which contains a summary of all tables, commands, functions, script operators, types, and more features present in the current implementation of MSC 2. If you are unsure how a specific element works, you can always refer back to the table of contents and search it in the main document.

If you feel this document is missing important pieces of information, feel free to post on discord or on the forums.

1.3 Notation

In examples and command definitions, arguments contain brackets or less than and greater than signs. Arguments in brackets ([ ]) can be optionally defined. Arguments between the less than and greater than signs (<>) are required.

For example:

```
/variable define <namespace> [qualifier [...]]  <Type> <name> [= expression]
```

Requires the *namespace*, *Type* and *name* arguments, and optionally provides the *qualifier* and = *expression* arguments. Any amount of qualifiers can be passed, indicated by the [...].

Scripts are represented as:

```
@player Hello
@bypass /rocket
@if true
     @player True
@else
     @player False
```

The result of a script is represented as:

```
Hello
True
```

2 Namespaces

Namespaces are a way of separating project-specific variables, functions and types. In previous versions, variables tended to cause collisions: frequently used names such as x or y could not have different values in different scripts. Namespaces allow encapsulation to prevent these collisions. A variable named x can coexist within multiple namespaces at any given time, without causing a collision. Generally a namespace would encapsulate a given project, a module within a project, or a logical module that can be reused in different projects (such as the math namespace).

2.1 The Namespace

A namespace consists of variables, functions and types. A user can define a namespace using a unique name. Within the namespace, variables can be defined to make them usable within the namespace. These are the *persistent* variables and will be stored to file.

Before being able to use a variable, function or type, they should be defined as element of a namespace. The definition of the variable requires a type. Persistent variables have to be defined in advance so that the compiler knows where to look and what type they are. See Variables for more details.

When using variables, functions or types in a script, the script should know what namespace the variable, function or type is in. If the namespace is undefined, it automatically defaults to the **local namespace** which contains variables that are *not persistent*. All variables within the local namespace should therefore be defined locally (that is, within the script). If using an undefined variable, the compiler will throw an error.

The local namespace is also present when using a specified namespace, and always takes precedence over the specified namespace. Any variables defined within the local namespace will shadow - that is overrule - the variable in the specified namespace. This is to make sure that an addition of a same-named variable in the namespace will not change the functionality of the script. By explicitly accessing the namespace, as described in Best Practice, you can still target the shadowed variable in the specified namespace, by using the temporary namespace specifier (**::**).

The local namespace is created at the start of a script, and deleted when the script terminates. Therefore the variables and types stored within the local namespace are *not persistent*.

2.2 Using Namespaces

When a script starts, it starts out with the **local namespace**. Unless defined, the local namespace contains no variables, and only the built-in functions and types.

A script can switch namespaces at any time using the **@using** script operation. Only one namespace will be active at any time, with the exception of the local namespace, which is always active and its variables always shadow the active namespace.

A variable or function can be temporarily accessed from a different namespace using the namespace specifier. A variable can be accessed using **namespace::variable**, a function using **namespace::function()**, and a type using **namespace::Type**.

Note that when executing a function, the function's namespace will be used. Variables can still be supplied from a different namespace through the parameters, and the function can still access other variables using the @using operator and temporary namespace specifier.

2.3 Best Practice

When using multiple variables, functions and/or types from one namespace, it is best to use **@using**.

Sometimes it is required to use multiple namespaces at the same time, such as using a variable from one namespace in a function of a different namespace. In this case you can use **@using** for the most frequently used namespace, to limit unnecessary words and characters.

When using a namespace once, it is best to use the temporary namespace specifier (**::**) since adding additional lines would be less clear than adding the specifier.

It can occur that a variable with the same name is both defined in the namespace that is currently set as **@using**, and in the local namespace (that is, earlier in the script). To access the local variable, there is no need to (and you cannot) use a namespace specifier to access the variable. To access the namespace variable, you are required to use the temporary namespace specifier (**::**) to access the variable, because the local namespace shadows the variable within the specified namespace, unless explicitly using the variable from the namespace.

In general, it is best to use the most readable approach. **@using** is intended for repeated usage, while **::** is intended for one-time-use of namespaces.

Table 2.1: Summary of namespaces

None	Uses the local namespace. Variables are not persistent and disappear when the script is terminated.
@using	Sets the current namespace and allows variables, functions and types from this namespace to be used. Only one namespace can be enabled at the same time. Local variables override namespace variables.
namespace::	Grabs the specified variable, function or type from the specified namespace. Can be used at any given instance. Is not shadowed by local variables due to the specificity.

3 Variables

Variables are the objects that can be adjusted to make a process behave differently. A process without variables yields the same result every time. Variables have certain characteristics defined by their type. The type defines what a variable can do and restricts what variable can passed to functions. Variables can be manually set and/or changed through operators.

3.1 Types

Variables do not have a strict feature set. They are simply name tags for a value, to which a new value can be attached. The Type is what determines in what format the value is stored and what operations can be performed on the value.

Whenever a variable is defined, the Type is always the word immediately preceding the variable's name. For example, the variable *name* defined as:

```
@define String name
```

has the type *String* (see Built-in Types for more information on the String type).

The type both represents how the variable can be instantiated and how it can be used.

3.1.1 Built-in Types

MSC 2 comes with a set of predefined types which can be used at any time from any namespace. User defined types can build on top of these to further expand functionality, or represent an entire different structure.

As of version 2.0, MSC contains:

Table 3.1: List of built-in-types

String	Plain text. Commands passed to @bypass, @command, @console and @player must be of this type.
Int	A (signed) integer. Represents whole positive and negative numbers. Can be used to define an amount among other things. Can represent values from -2^{31} through $2^{31} - 1$.
Long	A (signed) integer. Represents more values than an Int can. Can represent values from -2^{63} through $2^{63} - 1$. Generally not necessary until the Int does not suffice.
Float	A single-precision floating point number. Can represent a wide range of decimals (but sometimes suffers from being unable to represent a number). Can be used for a lot of things.
Double	A double-precision floating point number. Can represent a wider range of decimals than Float can (but still not all). For general purposes, Float is likely enough, but if it is not, Double can represent more precise state when needed, such as when doing precise maths.
Boolean	Can either be *true* or *false*. Used to keep track of conditions.
Player	Represents the Minecraft Player. Contains a wide range of utility functions and access to player statistics and variables. Can be used to directly read and alter a Player state, to some extent.
Entity	Represents a Minecraft Entity. Contains a wide range of utility functions to read and alter the Entity state, to some extent.
Block	Represents a Minecraft Block. Contains information about a block and ways to alter it, to some extent.

For a more detailed list on what functions and variables each of these types expose, take a look at the appendix: Built-in Types.

3.1.2 Literals

It may not always be preferable to first instantiate a variable and then pass it to a function or operator. Sometimes it is useful to just add a constant number or pass a constant piece of text to a function directly. For this reason literals are an option, serving as a piece of text that is converted to the correct variable type.

Table 3.2: List of literals

String	"Content of the string" - text contained between two " characters. If a String has to contain a " character, use the escape character: \. "This is in the string: \" "
Int	1 - any integer.
Long	1L - an integer followed by L.
Float	1.0 - any decimal.
Double	1.0D - any decimal followed by D.
Boolean	true or false.

Note that Block, Player and Entity have no literals. They always require constructors to instantiate the state. User-defined types can be instantiated in the same way, taking parameters as required.

3.2 Qualifiers

When defining a new type or namespace, sometimes it is useful to have variables that are player relative, or a variable that has a constant value. Persistent variables can be qualified by a qualifier keywords that determine their behaviour. Where the type determines what can be done with the value of the variable, the qualifier determines what properties the variable itself has. As of MSC 2.0 there are two qualifiers:

Table 3.3: List of qualifiers

final	A constant variable. Once initialized cannot be changed. Useful for more clear scripts, and makes changing values more maintainable.
relative	A variable that is player-bound. This is MSC 2's way of defining per-player variables, rather than shared variables.

3.3 Usage

As described in the previous sections, variables consist of one or more qualifiers, a type and a changeable value. Through commands, variables can be defined and operated upon. The main commands are:

```
/variable define <namespace> [qualifier [...]]  <Type> <name> [= expression]
```

```
/variable set <namespace> <name> = <expression>
```

In scripts this is can be written shorter by:

```
@define <Type> <name> [= expression]
```

and

```
@var [name =] <expression>
```

namespace is where you define which namespace is being altered.

[qualifier [...]] is where you define any amount of qualifiers. These are not present in scripts because variables in scripts are not persistent.

Type is where you define the Type of the variable. The Type has to be an already defined Type within the namespace. (If using an external type, use :: to indicate the namespace it comes from). Type names always start with an uppercase character.

name is where you define the name of the variable. Choose a descriptive name that makes clear what the variable is used for. Variable names may not begin with an uppercase character.

expression is how you first initialize the variable. Note that when using a final variable, this field is required. Otherwise, this can be left blank, to initialize the variable to their default state. (See Built-in Types for the default states of each type). For user-defined variables this will be null. See Expressions for more information on how to build an expression.

3.4 Null

Types that do not have a default state can sometimes be null. Null means multiple things, taking the form of 'unrepresentable', 'undefined', and 'non-existent'. As became apparent in the previous section a variable can be defined without expression, automatically taking on the default state. User-defined variables do not have a default state, and therefore automatically take the value null.

Some functions are unable to return a meaningful result. For example the Player() constructor can only return a Player if the player exists. If the Player is not online, it cannot return a meaningful result and thus returns null.

The reader should be aware that this case can occur. Performing operations on and with null variables will cause the script to fail with a NullPointerException. It is wise to keep track of the variables that can become null and script defensively. The Script cannot make assumptions to what behaviour is wanted when the value is undefined, and therefore it should always be explicitly stated.

4 Expressions

Expressions describe manipulations on variables with functions, operators and variables. They follow a few rules, but can be completely freely made as seen fit to the context. Expressions range from very short assignments to complex logic that can be stored or used further in the script, or used to perform an action themselves. They are the foundation of variables, giving the user many ways to manipulate and handle data.

4.1 The Expression

In the simplest sense, an expression is a piece of text that describes what to do. In the context of scripts, expressions are used to perform actions (functions) or manipulate variables (operators), or these two combined.

Expressions always evaluate toward one or no value. An expression ending in a function call with no return type will result in no value. Other expressions will always result in one value of a static type.

This static type can in turn be used in a different part of the expression, and so on. They can also be stored in a variable of the correct type.

Some examples of simple expressions are:

```
@player {{5 + 5}}
@player {{10 / 5}}
@player {{"Hello" + "World"}}
```

```
10
2
HelloWorld
```

Expressions can also be chained or nested:

```
@player {{"Hello" + "World" + 5 + 5}}
@player {{(5 + 5) * (5 + 5)}}
@var player.setMaxHealth(5 + 5)
```

```
HelloWorld55
100
```

As described in Execution Order, some operators take precedence over others, such as the * operator taking precedence over the + operator. If operators have the same precedence, the expression is evaluated from left to right. The above example: "Hello" + "World" + 5 + 5 therefore evaluates to "HelloWorld55".

You may be confused why it is not "HelloWorld10". The + operators has equal precedence whether it is used for a String or an Int. Reading left to right, first Hello and World are concatenated by the + operator, then HelloWorld + 5 is evaluated, which results in HelloWorld5 (because String + Int = String), then HelloWorld5 + 5 is evaluated, resulting in HelloWorld55.

Before a function is called, each of the parameters are first processed. This happens the same way as a normal expression. Thus player.setMaxHealth(5 + 5) evaluates to player.setMaxHealth(10) after which the function is successfully called.

Sometimes you may accidentally write an expression that does not work. For example, you write:

```
@var Block(5, 5, 5, "Zero") + 5
```

```
Operator '+' is not applicable on types:  Block, Int
```

Because Block has no + operator, this expression cannot complete, and will error. Sometimes this may be less apparent because of chained expressions. In general it is smart to keep your expressions as simple as possible, often preferring the most readable solution.

4.2 Execution Order

Chained expressions have an execution order. You are probably used to this in maths as well: * comes before +, but + and - happen at the same time (in our case from left to the right). Expressions also follow these simple rules. The operators with higher precedence are executed first, and operators with same precedence are executed left to right.

Expressions have a few more operators than those generally used in maths however, and we will list the execution order here. From top to bottom, top executes first, and bottom executes last.

Table 4.1: Precedence of operators

()	Parentheses around an expression prioritize this sub-expression.
!	Negation of Booleans.
*, %, /	Multiplication, modulo, division.
+, -	Addition, concatenation and subtraction.
<, <=, >, >=	Relational operators.
==, !=	Equality or non-equality.
&&	Logical AND.
||	Logical OR.
=	Assignment.

For example:

```
@player {{5 - 5 * 5}}
@player {{5 > 10 && 4 != 4 || 5 == 5}}
```

```
-20
true
```

The first one is fairly logical following basic math. The second may be harder to see at first. Due to the operator precedences, the expression is evaluated as ((5 >10) && (4 != 4)) || (5 == 5). This in turn evaluates to ((false) && (false)) || (true), which corresponds with false || true, which is true.

4.3 Short Circuit

In the case of the logical operators, the expression will short circuit whenever the expression has gathered enough info about the result. For example, assuming function() returns a Boolean:

```
@var true || function()
```

will never execute *function*, because the first operand was already true.

One of the most important reasons for this feature is the usage in if statements:

```
@define String var
@if var != null && var.toLowerCase() == "this is a string"
     @player Incorrect.
@else
     @player Correct!
```

```
Correct!
```

It will never evaluate the right side of the logical AND, because the left side was already false, saving you from a NullPointerException being thrown during the execution of the script. This will save some lines of code to check if something is null, and almost always results in predictable behaviour.

4.4 Syntax

Expressions follow fairly strict, but very logical syntax rules. We will list all syntax rules here with examples. For a more summarized list, refer to Syntax.

4.4.1 Define

The syntax for the define operator and command is as follows:

```
[qualifiers [...]]  <Type> <name> [= expression]
```

qualifiers can be any amount of qualifiers handled in Qualifiers. These always precede the rest of the definition and are always in lowercase. The qualifiers are **keywords** and can therefore not be used as a variable, function or type name. Qualifiers can only be used on persistent variables, and therefore not in scripts.

Type is the type of the variable, which always starts with an uppercase character. This makes it easier to distinguish the type from the variable and qualifiers.

name can be any word that is not a keyword or literal. The name can consist of the following characters: a-z, A-Z, 0-9, _. The name cannot start with a number, _, or an uppercase character.

expression has to be a valid expression resulting in a value of Type. See Expression.

```
@define String correctName = "5"
@player {{correctName}}
@define String 0invalidName
@define String InvalidName
@define InvalidType correctName
@define String true
@define String final
```

```
5
Variable does not start with a lowercase character:  '0'
Variable does not start with a lowercase character:  'I'
Type 'InvalidType' could not be found in namespace local
Collision with a keyword:  'true'
Collision with a keyword:  'final'
```

```
/variable define namespace final String example = "Unchangeable."
```

```
@var namespace::example = "5"
```

```
Variable 'final String example = Unchangeable' is declared final and
can therefore not be assigned a new value.
```

4.4.2 Var

The syntax for the var operator and command is as follows:

```
[name] <op> <expression>
```

name can be any predefined variable, or field that is available.

```
@define String example = "This is an example of good things."
@var example = "5"
```

```
@define String example = "This is an example of bad things."
@var undefined = "5"
```

```
Variable 'undefined' could not be found in namespace local
```

op can be either '=' or any of the numerical operators followed by '='. The former case sets the variable to the result of the expression, the latter case performs the operation on the variable itself and the expression, and saves the result in the variable. Available operators are: =, +=, -=, *=, /=, %=.

```
@define Int example = 5
@var example += 5
@player example
```

```
10
```

expression has to be a valid expression resulting in a value of the correct type. See Expression.

4.4.3 String Formatting

The String literal supports a formatting context in which all expressions are allowed. This is useful for both debugging and readability.

Within any String literal, an expression is started with a '{{' and closed with a '}}'. The resulting value is automatically converted to a String. If this is not possible, it will result in an error.

```
@define String hello = "I can do math:  {{5 + 5}}!"
@player {{hello}}
```

```
I can do math:  10!
```

4.4.4 Expression

The expression syntax allows any variables, literals and functions to be used.

Variables are just referred to by their name.

```
@define String name = "Hello"
@player {{name}}
```

```
Hello
```

Literals follow the syntax rules of their type.

```
@player {{"This is a literal."}}
@player {{true}}
@player {{5.0D}}
```

```
This is a literal
true
5.0
```

Function names are always immediately followed by an opening parenthesis '(', after which the parameters come, separated by a comma, and closes with a ')'. Functions always start with a lowercase character to distinguish from a constructor.

```
@player {{"Hello".contains("e")}}
```

```
true
```

To chain variables, results of functions and literals, operators are required.

```
@player {{5 + (5 / 5)}}
@player {{!true}}
@player {{!(true && true)}}
```

```
6
false
false
```

The resulting type is decided by the last remaining object after all sub-expressions have been evaluated, and has to fit the context. If any sub-expressions can not perform an

operation with an operator, or be assigned to a given type, the expression fails and an error is thrown.

```
@define Int x = 5 + 5
@define String y = "5" + 5
@define Block z = Block(0, 0, 0, "Zero")
@player {{x}}
@player {{y}}
@player {{z}}
@player {{z + 5}}
```

```
10
55
0 0 0
Operator '+' is not applicable on types:  Block, Int
```

5 Scripts

A script is an ordered list of script operations. A script will execute each script operation in succession to accomplish a certain objective. Within a script commands can be executed, variables manipulated, game state manipulated, script flow can be controlled and more. Scripts differ from functions in that they are triggered in specific situations, while a function has to be called explicitly, from a script, command block or through a command.

5.1 Script Operators

Every line within a script contains exactly one operator. The operator gives meaning to the line, because it determines what has to be done with the arguments. There are operators to execute commands, control the script flow and manipulate variables. Each such type has a dedicated section. Additionally, a full summary is available in Script Operators.

5.1.1 Command Operators

There are three operators to execute a command in a script: @command, @bypass and @console.

@command <command>
Executes the command with the permissions of the player. A greenie can use /warp, /rocket, while a whitie cannot. @command keeps in mind these differences in rank, and executes a command normally as if it was typed in chat.

@bypass <command>
Elevates the permissions of a player to semi-admin rank. It allows the script to perform all Minr admin commands and most Minecraft op commands (including specifiers @a, @e, @p, @s, @r).

@console <command>
Executes the command from the console, which means all commands can be executed, but it has the drawback of having to explicitly state the player, world, or sometimes not being able to perform a command at all.

To prevent lag and potential server hiccups, all command operators introduce a one-tick delay between their execution and the rest of the script. When a script or function uses a substantial amount of commands, the script may take a while to execute (remember, 20 ticks is one second, so 20 commands already takes one second).

```
@command /say hi
@bypass /say hi
@console /say hi
```

Assuming a non-admin executes this script:

```
You do not have the permission to execute this command.
Machete:  hi
Machete:  hi
```

```
@command /rocket
@bypass /rocket
@console /rocket
```

Assuming a whitie or a blue executes this script:

```
You do not have the permission to execute this command.
You rocketed yourself.
Invalid command:  missing player parameter.
```

5.1.2 Branching Operators

Sometimes a script needs to conditionally execute a part of the script. For this reason we have branching operators, which provide ways to cause different execution flows using variables. The branching operators can be nested, causing more and more possible execution paths. Be warned, as increasing the amount of execution paths greatly complexifies the script.

@if <Boolean expression>
Takes an expression that evaluates to a Boolean. If the Boolean is true, the following section is executed, if it is false, the section is skipped until reaching an @elseif, @else or @fi of the same level.

@else
Executes the following section if the preceding @if and @elseif operators of the same level were false.

@elseif <Boolean expression>
Executes the following section if the preceding @if and @elseif operators of the same level were false, and the expression of this @elseif evaluates to true.

@fi

Ends the conditional section. Any @if, @else or @elseif operators of the same level will no longer apply after this operator.

@return
Stops the execution of the current script or function, and optionally returns a value.

Because the branching operators can be nested, the script maintains an 'if level' to keep track of which @if has impact on which @else and @elseif operators. This level is demonstrated visually through the use of indentation in both this document and any script viewings (such as /scripts view).

```
@if true
     @player 1
     @return
@fi
@player 2
```

```
1
```

```
@if true
     @if false
          @player 1
     @else
          @player 2
     @fi
@elseif true
     @player 3
@else
     @player 4
@fi
@player 5
```

```
2
5
```

5.1.3 Control Operators

There are also operators that provide control on the execution of a script.

@delay <time>
Allows an arbitrary delay in the midst of a script, making the rest of the script wait with execution until the delay is over.

@cooldown <time>
Takes an arbitrary time that controls when the script can be re-executed by the same

player. If used in a function, a function will terminate the calling script when the function is on cooldown.

@global_cooldown <time>
Takes an arbitrary time that controls when the script can be executed again by any player. If used in a function, a function will terminate the calling script when the function is on cooldown.

@cancel
Disables the interaction between player and the object the script is bound to. Only has effect in interact scripts and before any delays introduced by other operators (such as @delay, the command operators and other halting operators).

Do note that any @cooldown and @global_cooldown operators only have effect once they are executed. Due to these constraints, @cancel, @cooldown and @global_cooldown have to be used before any delay because we cannot turn back time to stop an interaction after it has already happened. Therefore an interaction should always be cancelled while it is still happening, thus before any delays. Cooldowns are locked to the beginning in order to ensure proper usage.

The time parameter is explained in Time.

5.1.4 Variable Operators

To simplify the definitions of local variables and altering of local and global variables, MSC 2 introduces new operators that can readily alter the variable state.

@define <Type> <name> [= expression]
Defines a new variable and sets the value to an optionally defined expression. The expression has to match the type of the variable. Refer to Define for more information on the parameters.

@var [name =] <expression>
Executes an expression. This can be an assignment, function call, or any valid expression. For more information, refer to Var.

@using <namespace>
Switches the namespace of lines following this line. For more information, refer to Using Namespaces.

5.1.5 Chat Operators

To interface with the player chat, there are operators that send a message, send a clickable message or store a player's input in a variable.

@player <message>
Sends a message to the player. Supports color codes prefixed by &. Supports String Formatting by using {{ and }}.

@chatscript <group> <time> <expression>
Binds a function to the first following @player script operation. The function can be activated by the player at any time upon clicking the chat message.

Only one of the chatscripts in the same *group* can be executed. This means that when binding a chatscript to multiple messages with the same group, only one chatscript can be executed.

Once time runs out, the chatscript expires and the expression can no longer be executed by clicking the text in chat. The chatscript also expires once the chatscript has been executed once.

```
/function define example Void one()
```

```
/function define example Void two()
```

```
/function define example Void three()
```

```
/s c f example one @player one
```

```
/s c f example two @player two
```

```
/s c f example three @player three
```

```
@chatscript same example::one()
@player Option 1
@chatscript same example::two()
@player Option 2
@chatscript other example::three()
@player Option 3
```

```
Option 1
Option 2
Option 3
```

If the player clicks Option 1:

```
one
```

Then, if the player clicks Option 2:

```

```

Then, if the player clicks Option 3:

```
three
```

two was not displayed because it shares the *same* group with *one*, and since *one* was already executed, *two* could no longer be executed. *three* was a separate group, and therefore was able to be executed after *one* executed.

@prompt <time> <variable> [message]
Halts the script until the player types something. If time runs out, the script ends here, sending the message the optional message, or 'Prompt expired' otherwise. Message supports color codes with &.

If the player types something in time, the text the player typed is stored in the passed variable. Therefore, variable has to be of type String.

5.2 Anatomy of Scripts

5.2.1 Script Types

As mentioned before, scripts are triggered, while functions are explicitly called. Scripts have to be bound to a block, entity, ground, or area in order to function. A script can be triggered by walking over a block, interacting with an entity, entering an area, or interacting with a block.

Each of these triggers come with a type.

interact
The interact script type triggers when the player interacts with a block (stone, button, or anything else). There is no vanilla counterpart to this, except triggering a button or causing a block update.

The interact script type is often used for passwords, submit buttons, NPC (wool-type) dialogue, and much more.

walk
The walk script type triggers when the player walks over the block containing the script. If the script was bound to a block that is now removed, the script still triggers when the player is in the space just above the block.

The walk script type is often used for traps, story elements, resets, and much more.

ground
The ground script type triggers only when the player walks over the block containing the script. It only triggers once the player is **on** the block, and not while jumping over it, or when the block is air.

The ground script type can be used for crumbling pathways and other effects that require the player to stand on the block.

entity
The entity script type triggers when a player interacts with an entity. The script gets removed once the entity dies or despawns. When a script is applied to an entity, the server tries its best to keep the entity from despawning, but sometimes the inevitable occurs. Therefore, be prepared to respawn the entity with all scripts in place (by for example creating a function that correctly restores the entity may it be missing, and calling it whenever the entity is needed).

The entity script type is often used for dialogue.

area
The area script type is a new script type in MSC 2.0, triggering a script once when a player enters the set area.

function
To create content in a function, the function type is used. A function is always explicitly called from a script or other function. When adding script lines to a function, the function has to be defined using the function command. See Function Commands for details on defining a function

5.2.2 Lines

Every script consists of script lines, which are the actual content of the script. Each line is prefixed with the Script Operator, described in Script Operators. The Script Operator takes parameters that make up the rest of the script line.

A script is executed from top to bottom, waiting, delaying and executing commands as necessary. A script may not execute fully when a *@return* operator is used. @return cancels the script upon being run, halting further execution. Any resets should therefore always be done before the script terminates.

Once a script starts, a local namespace is created. In this namespace temporary variables can be declared using *@define*. Since the script executes from top to bottom, the script cannot use a variable before it is defined. The local namespace of the script always overrides the global namespace. Even if a used namespace contains a variable of the same name as the local namespace, the variable in the local namespace will always be used, unless a namespace specifier is used (::).

Once the script ends (due to a @return, expired @prompt or the end of the script), the local namespace is deleted, including any variables stored in it.

5.2.3 Parameters

Alongside default types and variables, a script can also contain parameters. Script Parameters are set by the 'system'. Parameters can be accessed much like any other

variable.

A script can have the following basic parameters (situationally, use /s v *type* to see which ones are present):

player
A Player type. Represents the player executing the script. Is not present within functions.

block
A Block type. Represents the block the script is bound to (if any). Is not present within functions, entity scripts or area scripts.

entity
An Entity type. Represents the entity the script is bound to (if any). Is only present in entity scripts.

5.3 Commands

The scripts command always has the following format:

```
/script <action> <type> [typeparameters] [actionparameters]
```

5.3.1 Action

A summarized version can be found in the appendix: Supported actions for script commands

In this section, all <type> [typeparameters] are replaced by ... for easier overview. Do note that the type is still required, and the type parameters come **before** the action parameters.

action can be one of the following:

create ... [line] <@operator> <script>

Adds a line to the end of the script. When *line* is passed, it adds the line on the given line number instead.

Example:

```
/script create interact @player hi!
```

```
@player hi!
```

```
/script create interact @player hi2!
```

```
@player hi!
@player hi2!
```

```
/script create interact 1 @player hi3!
```

```
@player hi3!
@player hi!
@player hi2!
```

view ...

View the lines of the script in chat.

Example: (viewing the script created above).

```
/script view interact
```

```
@player hi3!
@player hi!
@player hi2!
```

remove ... [line]

Remove the entire script. When *line* is passed, it removes only the line instead.

Example: (editing the script created above).

```
/script remove interact 1
```

```
@player hi!
@player hi2!
```

```
/script remove interact
```

```

```

info ...

List metadata and comments about the script.

export ...

Export the script to hastebin. (See Hastebin for more information).

import ... <id>

Import the script from hastebin. *id* is the identifier of your hastebin script, and should be passed. (See Hastebin for more information).

copy

Copy all scripts in a World-Edit selected region to the players' clipboard, relative to player position.

paste <type>

Pastes all scripts of type previously copied to clipboard relative to player position.

wipe <type>

Removes all scripts of type in a World-Edit selected region.

count <type>

Counts all scripts of type in a World-Edit selected region.

undo

Undoes a previously executed Script command.

5.3.2 Type

Type is one of the triggers described in Script Types. Each type has their own set of optional type parameters to select a block, entity, area or function. Some types also support leaving this blank, allowing the player to interact with a block, entity or area to define it afterwards.

interact [x y z] [world]

x y z are the coordinates the script should be bound to. *world* is the world in which the block should be found. If world is undefined, it will take the player's current world. If x y z are undefined, the player will be asked to interact with a block to bind the script.

walk [x y z] [world]

x y z are the coordinates the script should be bound to. *world* is the world in which the block should be found. If world is undefined, it will take the player's current world. If x y z are undefined, the player will be asked to interact with a block to bind the script.

ground [x y z] [world]

x y z are the coordinates the script should be bound to. *world* is the world in which the block should be found. If world is undefined, it will take the player's current world. If x y z are undefined, the player will be asked to interact with a block to bind the script.

entity [uuid] [world]

uuid is the UUID of the entity the script should be bound to. *world* is the world in which the entity should be found. If world is undefined, it will take the player's current world. If uuid is undefined, the player will be asked to interact with an entity to bind the script. If no entity exists with the given UUID, the command will fail.

area [world] <region>

world is the world in which the block should be found. If world is undefined, it will take the player's current world. *region* is the WorldGuard region the script should be bound to. The script is executed upon entering the region.

function <namespace> <function>

Binds the script to the corresponding *function* in *namespace*. If no such function exists in the namespace, the command will fail.

method <namespace> <Type> <method>

Binds the script to the corresponding *method* in *Type*. If no such method exists, the command will fail.

constructor <namespace> <Constructor Signature>

Binds the script to the corresponding constructor. The Constructor Signature serves to distinguish multiple constructors with different signatures, such as:

```
String(Player)
String(Int)
```

These, while having the same type, have different signatures. To access these constructors (note that built-in constructors cannot be edited), you would use the full definition, in contrast to functions and methods, where only the name suffices.

5.4 Hastebin

Minecraft has a pretty terrible way of inputting scripts. There's the option through chat, but that gets unreadable fast, and does not support multiple lines. We could use books, but they have limited horizontal space, which means most lines would wrap. Signs are no option either. There must be a better way to type scripts, right?

MSC 2 supports hastebin, which is an online coding pastebin. You can write text, press save, and a link will be generated that you can share with everyone. MSC 2.0 takes this raw text line by line, and converts it to a script.

Script can be imported from hastebin using:

```
/script import ...  <id>
```

and exported using

```
/script export ...
```

When you save your piece of text on hastebin, your URL will be appended by an identifier (a few random characters). You should use this identifier as the id when importing.

Exporting will upload the current script to hastebin, after which you can clone and edit the script, and import the edited script.

Hastebin uses automatically detected programming languages, resulting in MSC lines being picked up as some programming language. Hastebin will automatically include the programming language's extension. Whether you include the extension, or even the entire URL, or not, it will work regardless.

Example

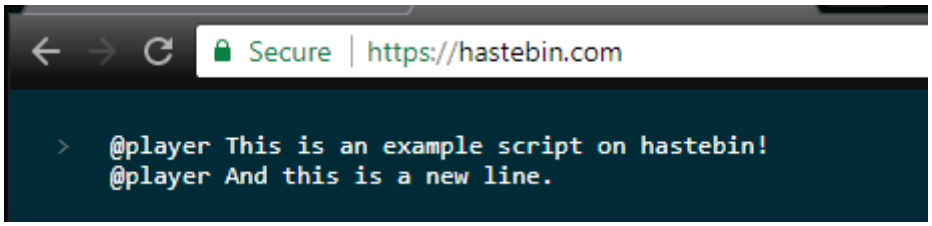

Figure 5.1: Write a script in hastebin

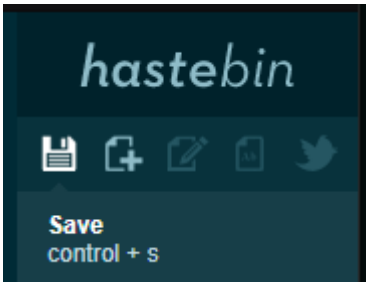

Figure 5.2: Save the script.

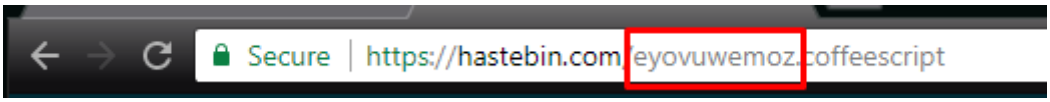

Figure 5.3: Find the identifier.

```
/script import interact eyovuwemoz
```

Figure 5.4: Run the import command, and press the block. That's it!

Exporting a script is as easy as running

```
/script export interact
```

and clicking the block, after which a link to the hastebin will be generated. To edit this script, you can press the edit button:

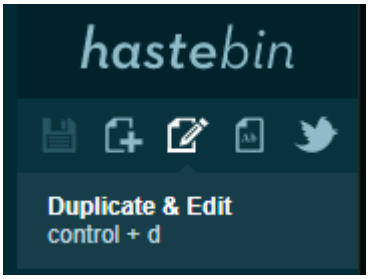

Figure 5.5: Click the edit button, and start editing. Then follow the instructions above to import the script again.

6 Functions

Functions are a way to reuse scripts. They can take parameters that allow one function to do many things, depending on the input. Functions can simply run preprogrammed lines, but they can also return a value.

6.1 The Function

A function is a script that can be explicitly called at any point, but is never triggered implicitly by the server. Because they are explicitly called, they can be called from within another function, from within a script, or even triggered as a chatscript.

Functions take parameters, which allows them to do a similar operation on different values, or simply change the operation when passed a different value.

Because it may be useful to grab a value from a type, convert a series of input variables to a single output variable, or anything else you can come up with, functions can return a value as well. Returning a value will allow the caller to store or use the result in a different script or function, further allowing for re-usability. For this reason functions have a return type, alongside parameter types. We will go a little more in depth later.

Functions are primarily meant to write parts of a script that is repetitive or can be used somewhere else. They are also used to condense a long script into one call (such as the use of functions in @chatscript).

Do keep in mind that functions are very much able to stall your script using @delay, @prompt and similar script operations. Cooldowns (@cooldown and @global_cooldown) terminate the calling script with an error when the function is on cooldown.

The definition of a function looks like:

```
ReturnType functionname(Type_1 name_1, ..., Type_n name_n)
```

Them the syntax of calling this function is as follows:

```
functionname(parameter_1, ..., parameter_n)
```

The ReturnType is discussed in Return Type. The parameters are discussed in Parameters.

6.1.1 Parameters

Functions are optionally defined with a series of parameters. Each parameter acts as a simple variable, but in reality they are a little more complex. A variable entering a function is a pass-by-value, which means that changing the value of the parameter will not change the value of the variable in the caller's script. The values of the call are copied over into the function, where they act as local variables.

Because they act as variables, they have to also be defined with a Type. However, because they are not persistent variables, qualifiers have no impact and are therefore not allowed.

Because they are copied over with value, variables with the relative qualifier will contain the player's value, the types passed will remain the same, and therefore any actions on a player within the function will also succeed. The only difference between these parameters and default (global) variables is that they do not change the value on the label of the variables that the function was called with.

Variables passed to a function should always match the type of the definition. If this is not the case, an error will occur.

6.1.2 Return Type

Functions can optionally return a value. The type of the value returned is governed by the definition of the function. Any @return operators should therefore return a value of the given type, and it is best practice to always explicitly return a value (or null). Implicitly, the function will return null when no return value is defined.

When no type is passed, the function internally takes the type 'Void', which is a type that has no functions, no operators, no literals and no constructors. Thus, when a function has no (or Void) return type, any expressions involving the function will fail, unless the function is a standalone expression.

When the function does have a return type, it can be freely used in expressions following the rules of the type. Functions are evaluated in the same order of expression rules, so keep in mind that short circuiting can occur.

Returning is always done using the @return operator, which takes an optional expression. The expression should always evaluate to the return type defined for the function, and can be left empty if the return type is Void or not present.

6.2 Syntax

A function can either be a standalone function, or a function bound to a type (often known as *methods*).

6.2.1 Definition

Functions are always defined with a name, parameter list and optionally a return type in the following format:

```
ReturnType name(Type parameter1, Type parameter2, ..., Type parametern)
```

The amount of parameters is completely variable. It is possible to declare a function without parameters by simply ending the declaration without defining any parameters:

```
ReturnType name()
```

When a function should merely perform an action, and should not result in a value, ReturnType can be removed as well:

```
name()
```

Do note that when a function is defined without a return type, any @returns taking variables will error, as it is probably a mistake.

In general, it is best to keep the amount of parameters within a reasonable range. Having a large amount of parameters not only makes it hard to read, but may also be hard to remember, document, and is often better split up into multiple functions. Some personal judgment is required to see what fits the situation best.

6.2.2 Function Calls

The functions can be called as follows, assuming the function is in the currently used namespace. Given a function called *sum* taking two arguments:

```
sum(parameter1, parameter2)
```

If the function is not in the current namespace, you can either use a different namespace using @using, or using the local namespace specifier:

```
math::sum(parameter1, parameter2)
```

In a script situation, this can look like:

```
@define Int result = math::sum(1, 2)
@player {{result}}
```

```
3
```

Most types (such as String, Player, Block and Entity) provide functions. For example, the function *closeInventory()* contained in the *Player* type can be called as follows:

```
@define Player player = Player("rickyboy320")
@var player.closeInventory()
```

7 User defined Types

User defined types are a way to reuse types. Common constructs can be grouped into a type, and be used like any other type. This allows users to build abstraction layers on top of the basic types, which should reduce overhead in scripting and greatly simplify commonly used constructs.

7.1 User defined Types

User defined types combine most previously handled concepts and allow them to be grouped together into a reusable type. A Type is defined in a Namespace, so that name conflicts should not occur. A Type can contain fields (*variables*) and methods (*functions*). The difference between a Namespace and a Type however, is that a Type can be instantiated. There can exist multiple variables of any given Type at any given moment. The values of the fields are not shared, unlike the Namespace. A Type can also be used as a Function argument, and can itself be used as a field-Type for a Type.

Because the explanation uses quite a bit of jargon, let us be a bit more concrete by using an example throughout this chapter. Say we find ourselves constantly using the variables x, y and z together, we can decide to create a Type. For the sake of this chapter, we are creating the Type **Location**.

Types are created using the type command:

```
/type define <namespace> <Type>
```

For example:

```
/type define world Location
```

7.2 Fields

A Type can contain fields. These are effectively variables, but are bound to a given instance of the Type. Since a Type is merely a *label* of what a Variable can do, the Type defines what the Variable contains as fields.

Our example Type should contain the fields x, y and z, because we want to unify these three variables into one structure: the Location Type.

We can add these fields to a Type using the type command:

```
/type variable define <namespace> <Type> <Type> <name>
```

For example:

```
/type field define world Location Int x
```

will bind the newly made field Int x to the Type Location.

Fields can be accessed in expressions using a dot. For example, after having added x, y and z in our Location example, we can use:

```
@using world
@define Location location = Location(5, 6, 7)
@player {{location.x}}
@player {{location.y}}
@player {{location.z}}
```

```
5
6
7
```

The first line is a constructor, which will be handled in Constructors. Assuming the constructor is defined as Location(Int x, Int y, Int z) and sets these variables respectively, the above will be the result.

We can also set fields of a type after the initial definition, by also using the dot.

```
@define Location location = Location(5, 5, 5)
@player {{location.x}}
@var location.x = 10
@player {{location.x}}
```

```
5
10
```

This allows retroactively changing the values of the variable.

7.3 Methods

As with Built-in Types, User defined Types can also contain Methods. The goal of these methods generally has to do with the state of the Type. They are to manipulate the instance, or give information about it.

They are defined using the /type command:

```
/type method define <namespace> <Type> <name>([Type name[, ...]])
```

For example, we can define the method getX() on the Location type as follows:

```
/type method define world Location Int getX()
```

To add lines to the body of this function, we use the script command:

```
/script create method world Location getX @return this.x
```

As with built-in types, these methods can be called on a type with the dot.

```
@define Location location = Location(5, 6, 7)
@player {{location.x}}
@player {{location.getX()}}
```

```
5
5
```

They work exactly the same as any other Function described in the Functions chapter, except that they have access to the variables' state directly, through the *this* keyword, and that they have to be called on an instance.

7.3.1 This keyword

The **this** keyword is the handle to access the state of the current instance. Normally an instance is constructed and acted upon as a value bound to a variable. However, as the instance yourself, there is no way to access yourself using any other means, therefore the this keyword exists.

In our example, if our Location Type needs a function that sets the current coordinates to the given coordinates, the type needs to reference its own fields. This can be done using the this keyword, so that the instance can manipulate itself.

7.4 Constructors

Constructors serve to build the initial state of an instance. For example, it would be weird to have an uninitialized Location, since its fields would all be 0. We want to set up a Location with the right coordinates out of the box, and not wait until it is instantiated.

We can achieve this using Constructors. As described with Built-in Types, the Player, Entity and Block Types each have constructors to initialize the state. We can define constructors on our built-in types as well, even allowing for multiple constructors with different definitions (as also seen in Built-in Types).

A constructor is defined much like a Type:

```
/type constructor define <namespace> Type([Type name[, ...]])
```

For example, our Location constructor taking x, y and z:

```
/type constructor define world Location(Int x, Int y, Int z)
```

We can define an *overloaded* constructor with the same command:

```
/type constructor define world Location(Float x, Float y, Float z)
```

The constructors' body can be defined with the Script command:

```
/script create constructor <namespace> <constructor signature> <script>
```

For example, our Location constructor:

```
/script create constructor world Location(Float, Float, Float) <script>
```

Clearly this is a bit verbose, so look at Hastebin for more information on how to simplify definitions.

Constructors can be chained by using @return in the constructors' body. Of course the Constructor should return the type that is being constructed. @return can also be left out, returning the currently constructed instance.

In constructors the **this** keyword can be used to access methods and fields of the instance.

8 Examples

To be filled in once we have a stable version.

9 Appendix

9.1 Built-in Namespaces

9.1.1 system

The system namespace handles all types of miscellaneous behaviour typically found in the system, such as time.

Variables

The system namespace contains no variables.

Functions

Table 9.1: Supported Functions for the system namespace

Long **currentTimeMillis**()	Returns the current time in milliseconds. Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in units of tens of milliseconds.

9.1.2 math

The math namespace contains a series of common math operations.

Variables

The math namespace contains no variables.

Functions

Table 9.2: Supported Functions for the math namespace

Double **sqrt**(Double value)	Returns the correctly rounded positive square root of a double value.
Double **abs**(Double value)	Returns the absolute value of a double value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Double **pow**(Double value, Double exponent)	Returns the value of the first argument raised to the power of the second argument.
Int **randomInt**()	Returns the next pseudorandom, uniformly distributed Int value.
Long **randomLong**()	Returns the next pseudorandom, uniformly distributed Long value.
Float **randomFloat**()	Returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0.
Double **randomDouble**()	Returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0.
Double **random**(Double min, Double max)	Returns the next pseudorandom, uniformly distributed double value between min and max.

Most of these functions have special cases with special arguments. View https://docs.oracle.com/javase/10/docs/api/java/lang/Math.html for these cases.

9.2 Built-in Types

9.2.1 String

A String represents plain text. Any piece of text surrounded with " is considered a String. Script operators that take exactly one string (such as @player, @bypass, @console, @command) do not require this (for backwards compatibility and less clutter).

```
@player Hey
@player "Hey"
```

```
Hey
"Hey"
```

A String will be null when it is referenced before initialization.

Constructors

A string can be created in one of two ways. The first one is using the String literal, and the other is the String constructor. The string literal is any piece of text surrounded with ". If the String needs to contain a ", use the backslash to escape the double quotation marks, as follows: "This is escaped: \". Cool."

The second way is through a constructor. Available constructors are:

Table 9.3: Supported constructors for the String type

String(String value)	Clone a String.
String(Int value)	Get the textual value of an Int.
String(Long value)	Get the textual value of a Long.
String(Float value)	Get the textual value of a Float.
String(Boolean value)	Get the textual value of a Boolean.
String(Double value)	Get the textual value of a Double.
String(Player value)	Get the Player name in textual form.
String(Entity value)	Get the Entity UUID in textual form.
String(Block value)	Get the Block coordinates in textual form.
String(Item value)	Get the Item in textual form.

The String literal has an additional property for easier formatting. Within the quotation marks it supports string formatting using {{ and }}. Any expression or value represented within these double curly brackets will be evaluated and converted to a String. If any other type remains within the curly brackets, the appropriate constructor is automatically called to convert it into a String, if any. Admins can use these curly brackets in chat to quickly evaluate an expression (for example to see the contents of a variable). Do keep in mind that expressions in chat will require the local namespace specifiers to specify the namespace, as there is no @using in chat.

For example:

```
@player {{Player("rickyboy320")}}
```

```
rickyboy320
```

This works because the String(Player) constructor defaults to the player name in textual form. Additionally, script operators that take exactly one string do not take quotation marks.

(If required, {{ and }} can be escaped like the quotation marks, using a backslash: \)

Operators

Table 9.4: Supported operators for the String type

+	String	Concatenates two Strings together.
	Boolean	Concatenates String and Boolean together, as if the value were a string. ($"true" + true = "truetrue"$)
	Int, Double, Float, Long	Concatenates String and the textual value of the other together.
	Player	Concatenates String and the name of the Player together.
	Entity	Concatenates String and the UUID of the Entity together.
	Block	Concatenates String and the coordinates of Block together.
	Item	Concatenates String and Item together.
==	String	Checks for equality between Strings. This is case-sensitive. For case-insensitive equality, use .equalsIgnoreCase(). (Returns Boolean with the result: true if equal).
!=	String	Checks for inequality between Strings. (Returns Boolean with the result: true if not equal).

Methods

Table 9.5: Supported Methods for the String type

Boolean **contains**(String sequence)	Returns true if the String contains *sequence*, false otherwise.
Boolean **equalsIgnoreCase**(String other)	Returns true if the String is equal except for case to *other*, false otherwise.
Int **indexOf**(String sequence)	Returns the index the first occurrence of *sequence* starts at. If the String does not contain *sequence*, returns -1.
String **replace**(String old, String new)	Replaces all occurrences of *old* with *new* in the String.
String **substring**(Int start, Int end)	Returns a substring starting (inclusive) at *start* and ending (exclusive) at *end*. Throws IndexOutOfBoundsException when *start* or *end* are invalid indices within the string. Throws InvalidParameterException when *end* is smaller than *start*.
String **toLowerCase**()	Returns the String in lowercase.
String **toUpperCase**()	Returns the String in uppercase.
String **trim**()	Returns the String with leading and trailing whitespace omitted.

9.2.2 Int & Long

The Integer represents whole numbers (-1, 0, 1, 2, etc). Within a computing environment, not all numbers can be represented.

The Java standard upholds a max Integer value of $2^{31} - 1$ and a minimum Integer value of -2^{31}. Any number outside of this range will overflow, resulting in a sign flip and counting the opposite way. Roughly said: $2^{31} - 1 + 1 = -2^{31}$ (note that this is unsupported and can change at any time).

If you need to represent a discrete number outside of this range, you can use Long instead. Long has a max value of $2^{63} - 1$ and a min value of -2^{63}.

Int and Long are *recessive* types. Any operation with a Float, Double or String will take priority and converts the Int or Long to the correct type. The resulting type will always be that of the operand. This is exactly why Integer division does not occur when using a Double or Float as the operand.

An Int and Long will be 0 when it is referenced before initialization.

Constructors

Integers and Longs can be created in one of two ways. The first one is using the Int or Long literal, and the other is a constructor.

The Int literal is any whole number: 1, 2, 4, 10, -5.

The Long literal is any whole number followed by L: 1L, 2L, 4L, 10L, -5L.

The second way is through a constructor. Available constructors are:

Table 9.6: Supported constructors for the Int and Long type

Int(Int value)	Make an Int from another Int. (Clone operation)
Int(Long value)	Cast a Long down to an Int. (Precision loss)
Int(Float value)	Discard the decimals and convert a Float to Int.
Int(Double value)	Discard the decimals and convert a Double to Int.
Int(String value)	Attempt to parse a String into an Int. Only succeeds if the entire String can be represented as an Int. Throws NumberFormatException otherwise.
Long(Int value)	Upcast an Int to a Long.
Long(Long value)	Clone a Long.
Long(Float value)	Discard the decimals and convert a Float to Long.
Long(Double value)	Discard the decimals and convert a Double to Long.
Long(String value)	Attempt to parse a String into an Long. Only succeeds if the entire String can be represented as a Long. Throws NumberFormatException otherwise.

Operators

Table 9.7: Supported operators for the Int and Long type

+	String	Concatenates Int and String together, as if the value were a string. $(2 + "2" = "22")$
	Int, Double, Float, Long	Adds the value to the numerical value of the operand.
-	Int, Double, Float, Long	Subtracts the operand value from the value.
*	Int, Double, Float, Long	Multiplies the value with the operand value.
/	Int, Long	Integer divides the value and the operand. $(5/2 = 2)$
	Double, Float	Divides the Integer value and the operand. $(5/2.0 = 2.5)$
%	Int, Double, Float, Long	The modulo operation. Finds the remainder after division. $(5\%2 = 1)$
==	Int, Double, Float, Long	Returns whether this numerical value and the other numerical value are *exactly* the same.
!=	Int, Double, Float, Long	Returns whether this numerical value and the other numerical value are not *exactly* the same.
<	Int, Double, Float, Long	Returns whether this numerical value is less than the other numerical value.
>	Int, Double, Float, Long	Returns whether this numerical value is more than the other numerical value.
<=	Int, Double, Float, Long	Returns whether this numerical value is less than or equal to the other numerical value.
>=	Int, Double, Float, Long	Returns whether this numerical value is more than or equal to the other numerical value.

Methods

There are no methods contained in the Int and Long type.

9.2.3 Float & Double

The Float and Double represent decimal values (-0.1, 37.5, 42.0, etc.). Internally it uses an interesting notation, a bit like the scientific notation to represent numbers. Because of this way of representing the numbers (using a floating point), not all numbers are represented as accurately. A Float and a Double can both represent a wider range of values than the Integer or Long can, but not as precisely.

The Java standard upholds a max Float value of $(2 - 2^{-23}) \cdot 2^{127}$ and a minimum

(positive) Float value of 2^{-149}. All numbers that can be represented positively can also be represented negatively (including 0!). Do note that not all numbers in the range of the min and max value can be represented, and that there is more than often a case of precision loss.

The Double type can represent numbers more accurately, maintaining a maximum value of $(2 - 2^{-52}) \cdot 2^{1023}$ and a minimum value of 2^{-1074}. It can represent numbers more accurately than a Float, but can still have precision loss. In most cases this should not pose a problem.

On top of overflowing, much like the Integer and Long types, the Float and Double can also underflow. This occurs when it tries to represent a number between 0 and the minimum positive (or negative) value. In most cases this should not be a problem.

An Float and Double will be 0.0 when it is referenced before initialization.

Constructors

Floats and Doubles can be created in one of two ways. The first one is using the Float or Double literal, and the other is a constructor.

The Float literal is any decimal number: 1.0, 2.0, 4.0, 10.2342, -5.12.

The Double literal is any number followed by D: 1D, 2D, 4.0D, 10.2342D, -5.12D.

The second way is through a constructor. Available constructors are:

Table 9.8: Supported constructors for the Float and Double type

Float(Int value)	Cast an Int to a Float.
Float(Long value)	Cast a Long down to an Int. (Precision loss)
Float(Float value)	Clone a Float.
Float(Double value)	Cast a Double to a Float. (Precision loss)
Float(String value)	Attempt to parse a String into an Float. Only succeeds if the entire String can be represented as a Float. Throws NumberFormatException otherwise.
Double(Int value)	Cast an Int to a Double.
Double(Long value)	Cast a Long to a Double.
Double(Float value)	Upcast a Float to a Double.
Double(Double value)	Clone a Double.
Double(String value)	Attempt to parse a String into an Double. Only succeeds if the entire String can be represented as a Double. Throws NumberFormatException otherwise.

Operators

Table 9.9: Supported operators for the Float and Double type

+	String	Concatenates Float and String together, as if the value were a string. ($2.0 + "2" = "2.02"$)
	Int, Double, Float, Long	Adds the value to the numerical value of the operand.
-	Int, Double, Float, Long	Subtracts the operand value from the value.
*	Int, Double, Float, Long	Multiplies the value with the operand value.
/	Int, Double, Float, Long	Divides the value and the operand. ($5.0/2 = 2.5$)
%	Int, Double, Float, Long	The modulo operation. Finds the remainder after division. ($0.5\%0.2 = 0.1$)
==	Int, Double, Float, Long	Returns whether this numerical value and the other numerical value are *exactly* the same.
!=	Int, Double, Float, Long	Returns whether this numerical value and the other numerical value are not *exactly* the same.
<	Int, Double, Float, Long	Returns whether this numerical value is less than the other numerical value.
>	Int, Double, Float, Long	Returns whether this numerical value is more than the other numerical value.
<=	Int, Double, Float, Long	Returns whether this numerical value is less than or equal to the other numerical value.
>=	Int, Double, Float, Long	Returns whether this numerical value is more than or equal to the other numerical value.

Methods

There are no methods contained in the Float and Double type.

9.2.4 Boolean

The Boolean can either represent *true* or *false*. It is primarily used in branches (such as @if, @elseif) or conditions. Booleans contain some additional operators to perform boolean logic with.

A Boolean will be false when it is referenced before initialization.

Constructors

Booleans can be created in one of two ways. The first one is using the Boolean literal, and the other is a constructor.

The Boolean literal is either true or false.

The second way is through a constructor. Available constructors are:

Table 9.10: Supported constructors for the Boolean type

Boolean(Boolean)	Copy a Boolean.
Boolean(String)	Parse true or false in string format to a boolean. Defaults to false.

Operators

Table 9.11: Supported operators for the Boolean type

+	String	Concatenates Boolean and String together, as if the value were a string. ($true + "true" = "truetrue"$)
!	(Prefix)	Negates the boolean value. ($!true = false$)
&&	Boolean	ANDs the booleans together. Results in true only if both booleans are true. ($true \&\& true = true, true \&\& false = false, false \&\& false = false$)
||	Boolean	ORs the booleans. Results in true when either boolean is true. ($true \| \| true = true, true \| \| false = true, false \| \| false = false$)
==	Boolean	Returns whether two Boolean values are the same (both true, or both false).
!=	Boolean	Returns whether two Boolean values are not the same.

The logical operators && and || are short-circuiting. This means that when reading from left to right, one of the operands causes the result to always be true or false, the other operand is not evaluated. For example the expression

```
@if x != null && x.contains("blue")
```

will not throw a NullPointerException even if x is null, because the if statement short circuits before it reaches the substring expression.

Methods

There are no methods contained in the Boolean type.

9.2.5 Player

The Player represents an (online) Minecraft Player. There are a multitude of things you can accomplish through supported methods that are generally not directly available

through commands.

A Player will be null when it is referenced before initialization.

Constructors

Table 9.12: Supported constructors for the Player type

Player(String value)	Construct a player from their name or UUID. Null if the player does not exist.
Player(Int x, Int y, Int z, String world)	Find a player at these coordinates in the passed world. Null if the player does not exist. In the scenario that multiple Players are in the same location, nondeterministically returns one Player at that location.

Operators

Table 9.13: Supported operators for the Player type

+	String	Concatenates the name of Player and String together.
==	Player	Checks for equality between Players. (Returns true when the players are the same player).
!=	Player	Checks for inequality between Players. (Returns true when the players are not the same player).

Methods

Table 9.14: Supported Methods for the Player type

Float **getFallDistance**()	Returns the distance this entity has fallen.
Int **getFireTicks**()	Returns the entity's current fire ticks (ticks before the entity stops being on fire).
setFireTicks(Int ticks)	Sets the entity's current fire ticks (ticks before the entity stops being on fire).
Double **getX**()	Gets the entity's current x position.
Double **getY**()	Gets the entity's current y position.
Double **getZ**()	Gets the entity's current z position.
Float **getYaw**()	Gets the entity's current rotation around the y axis.
Float **getPitch**()	Gets the entity's current rotation around the x axis.
Double **getVelocityX**()	Gets the entity's current velocity in the x direction.
Double **getVelocityY**()	Gets the entity's current velocity in the x direction.
Double **getVelocityZ**()	Gets the entity's current velocity in the x direction.
String **getWorld**()	Gets the current world this entity resides in.
Boolean **isDead**()	Returns true if this entity has been marked for removal.
Boolean **isFlying**()	Checks to see if this player is currently flying or not.
Boolean **isOnGround**()	Returns true if the entity is supported by a block. This value is a state updated by the server and is not recalculated unless the entity moves.
Boolean **isSneaking**()	Returns if the player is in sneak mode.
Boolean **isSprinting**()	Gets whether the player is sprinting or not.
giveExp(Int amount)	Gives the player the amount of experience specified.
Float **getExp**()	Gets the players current experience points towards the next level.
setExp(Float exp)	Sets the players current experience points towards the next level.
giveExpLevels(Int amount)	Gives the player the amount of experience levels specified. Levels can be taken by specifying a negative amount.
Float **getLevel**()	Gets the players current experience level.
setLevel(Int level)	Sets the players current experience level.

Table 9.15: Supported Methods for the Player type (continued)

damage(Double amount)	Deals the given amount of damage to this entity.
Double **getHealth**()	Gets the entity's health from 0 to getMaxHealth(), where 0 is dead.
setHealth(Double health)	Sets the entity's health from 0 to getMaxHealth(), where 0 is dead. Throws IllegalArgumentException if the health is < 0 or > getMaxHealth().
Double **getMaxHealth**()	Gets the maximum health this entity has.
setMaxHealth()	Sets the maximum health this entity has. If the health of the entity is above the value provided it will be clamped to the max value. Only sets the 'base' max health value, any modifiers changing this value (potions, etc) will apply *after* this value. The value returned by getMaxHealth may deviate from the value set here.
Float **getFoodLevel**()	Gets the players current food level.
setFoodLevel(Int value)	Sets the players current food level.
Float **getSaturation**()	Gets the players current saturation level. Saturation is a buffer for food level. Your food level will not drop if you are saturated ¿ 0.
setSaturation(Float value)	Sets the players current saturation level.
Boolean **isInsideVehicle**()	Returns whether this entity is inside a vehicle.
Boolean **leaveVehicle**()	Leave the current vehicle. If the entity is currently in a vehicle (and is removed from it), true will be returned, otherwise false will be returned.
closeInventory()	Force-closes the currently open inventory view for this player, if any.
Long **getTimePlayed**()	Gets the player's playtime on the server in milliseconds.
String **getLocale**()	Gets the player's current locale. The value of the locale String is not defined properly. The vanilla Minecraft client will use lowercase language / country pairs separated by an underscore, but custom resource packs may use any format they wish.
String **getUniqueId**()	Gets the UUID of the entity (in string format).
Boolean **isOnline**()	Checks if this player is currently online.
Boolean **isOp**()	Checks if this Player is a server operator.
setResourcePack(String url, String hash)	Request that the player's client downloads and switches resource

Table 9.16: Supported Methods for the Player type (continued)

Item **getItem**(Int slot)	Returns the Item found in the slot at the given index.
Item **getItemInMainHand**()	Gets a copy of the item the player is currently holding in their main hand.
Item **getItemInOffHand**()	Gets a copy of the item the player is currently holding in their off hand.
Item **getBoots**()	Return the Item from the boots slot.
Item **getLeggings**()	Return the Item from the leg slot.
Item **getChestplate**()	Return the Item from the chestplate slot.
Item **getHelmet**()	Return the Item from the helmet slot.
setItem(Int slot, Item item)	Stores the Item at the given index of the inventory. Indexes 0 through 8 refer to the hotbar. 9 through 35 refer to the main inventory, counting up from 9 at the top left corner of the inventory, moving to the right, and moving to the row below it back on the left side when it reaches the end of the row. It follows the same path in the inventory like you would read a book. Indexes 36 through 39 refer to the armor slots. Though you can set armor with this method using these indexes, you are encouraged to use the provided methods for those slots. If you attempt to use this method with an index less than 0 or greater than 39, an ArrayIndexOutOfBounds exception will be thrown.
setItemInMainHand(Item item)	Sets the item the player is holding in their main hand.
setItemInOffHand(Item item)	Sets the item the player is holding in their off hand.
setBoots(Item item)	Put the given Item into the boots slot. This does not check if the Item is a boots.
setLeggings(Item item)	Put the given Item into the leg slot. This does not check if the Item is a pair of leggings.
setChestplate(Item item)	Put the given Item into the chestplate slot. This does not check if the Item is a chestplate.
setHelmet(Item item)	Put the given Item into the helmet slot. This does not check if the Item is a helmet.

Table 9.17: Supported Methods for the Player type (continued)

Boolean **isPlayingChallenge**()	Returns whether the player is playing a challenge.
String **getCurrentChallenge**()	Returns the challenge the player is playing. Returns null when player is not playing any challenge.
Int **getChallengePoints**()	Returns the amount of challenge points the player has.
Int **getHexaRecord**()	Returns the stage the player reached in hexa.
Boolean **hasCompletedChallenge**(String challengetag)	Returns whether the player has completed the specified challenge.
Long **getChallengeTime**()	Returns the current time the player has spent in the challenge.
Boolean **isPlayingMap**()	Returns whether the player is playing a map.
String **getCurrentCheckpoint**()	Returns the checkpoint the player has. Returns null when no checkpoint in the current checkpoint mode is set. Returns the checkpoint from the current checkpoint mode (HC or FFA).
Int **getPoints**()	Returns the amount of FFA points the player has.
Int **getGlobalPoints**()	Returns the amount of global points the player has.
Boolean **hasCompletedMap**(String maptag)	Returns whether the player has completed the specified map.
Long **getMapTime**()	Returns the current time the player has spent in the map.
Int **getAttempts**()	Get the amount of times a player has hit any starting checkpoint sign.
invalidate()	Invalidate the player's challenge and map run.
invalidateTime()	Invalidate the player's time on map and challenge, but allows them to complete the map and challenge.

9.2.6 Entity

An Entity is a move-able or dynamic object in the Minecraft world. Animals and monsters are Entities, but also arrows, item frames and paintings.

An Entity will be null when it is referenced before initialization.

Constructors

Table 9.18: Supported constructors for the Entity type

Entity(String uuid)	Construct an entity from its UUID. Returns null if it does not exist.
Entity(Int x, Int y, Int z, String world)	Find an entity in the passed world at these coordinates. Returns null if it does not exist. In the scenario that multiple entities are in the same location, nondeterministically returns any entity.

Operators

Table 9.19: Supported operators for the Entity type

+	String	Concatenates the UUID of Entity and String together.
==	Entity	Checks for equality between Entities. (Returns true when the entities are the same entity).
!=	Entity	Checks for inequality between Entities. (Returns true when the entities are not the same entity).

Methods

Table 9.20: Supported Methods for the Entity type

String **getEntityType**()	Gets the entity's type. Actual value returned is a 'magic value' and can change at any spigot or bukkit update.
Double **getX**()	Gets the entity's current x position.
Double **getY**()	Gets the entity's current y position.
Double **getZ**()	Gets the entity's current z position.
Float **getYaw**()	Gets the entity's current rotation around the y axis.
Float **getPitch**()	Gets the entity's current rotation around the x axis.
Double **getVelocityX**()	Gets the entity's current velocity in the x direction.
Double **getVelocityY**()	Gets the entity's current velocity in the x direction.
Double **getVelocityZ**()	Gets the entity's current velocity in the x direction.
String **getWorld**()	Gets the current world this entity resides in.
Boolean **isDead**()	Returns true if this entity has been marked for removal.
Boolean **isOnGround**()	Returns true if the entity is supported by a block. This value is a state updated by the server and is not recalculated unless the entity moves.
damage(Double amount)	Deals the given amount of damage to this entity.
Double **getHealth**()	Gets the entity's health from 0 to getMaxHealth(), where 0 is dead.
setHealth(Double health)	Sets the entity's health from 0 to getMaxHealth(), where 0 is dead. Throws IllegalArgumentException if the health is ¡ 0 or ¿ getMaxHealth().
Double **getMaxHealth**()	Gets the maximum health this entity has.
setMaxHealth()	Sets the maximum health this entity has. If the health of the entity is above the value provided it will be set to that value.
String **getUniqueId**()	Gets the UUID of the entity (in string format).

9.2.7 Block

A Block represents a Block in the Minecraft world. Any valid block (within reasonable bounds, $0 \leq y \leq 255$) can be represented, whether it is an empty (air) block, liquid, or a solid block.

A Block will be null when it is referenced before initialization.

Constructors

Table 9.21: Supported constructors for the Block type

Block(Int x, Int y, Int z, String world)	Get the block at these coordinates in the given world.

Operators

Table 9.22: Supported operators for the Block type

+	String	Concatenates the coordinates of Block and String together.
==	Block	Checks for equality between Blocks. (Returns true when the blocks are the same block).
!=	Block	Checks for inequality between Blocks. (Returns true when the blocks are not the same block).

Methods

Table 9.23: Supported Methods for the Block type

Int **getBlockPower**()	Returns the Redstone power being provided to this block.
Int **getLightLevel**(()	Returns the amount of light at this block.
Int **getLightFromBlocks**()	Returns the amount of light at this block from nearby blocks.
Int **getLightFromSky**()	Returns the amount of light at this block from the sky.
Block **getRelative**(Int modX, Int modY, Int modZ)	Gets the block at the given offsets.
String **getBlockType**()	Gets the type of this block. Actual value returned is a 'magic value' and can change at any spigot or bukkit update.
Int **getX**()	Returns the x-coordinate of this block.
Int **getY**()	Returns the y-coordinate of this block.
Int **getZ**()	Returns the z-coordinate of this block.
String **getWorld**()	Returns the world where this block resides in.
Boolean **isBlockIndirectlyPowered**()	Returns true if the block is being indirectly powered by Redstone.
Boolean **isBlockPowered**()	Returns true if the block is being powered by Redstone.
Boolean **isEmpty**()	Returns true if this block is Air.
Boolean **isLiquid**()	Returns true if this block is liquid.

9.2.8 Item

An Item represents an Item in the Minecraft world. Any valid item can be represented, along with the stack size.

An Item will be null when it is referenced before initialization.

Constructors

Table 9.24: Supported constructors for the Item type

Item(String item, Int amount)	Create an item from the passed name with a stack size of amount. Throws MaterialNotFoundException when passed an invalid name.

Operators

Table 9.25: Supported operators for the Item type

+	String	Concatenates the Item and String together.
==	Item	Checks for equality between Items. (Returns true when the items match and the stack size is equal).
!=	Item	Checks for inequality between Items. (Returns true when the blocks are not the same, and/or the stack size is unequal).

Methods

Table 9.26: Supported Methods for the Item type

Int **getAmount**()	Gets the amount of items in this stack.
String **getItemType**()	Gets the type of this item.
Int **getMaxStackSize**()	Get the maximum stacksize for the material hold in this ItemStack. (Returns -1 if it has no idea).
setAmount(Int amount)	Sets the amount of items in this stack.
setItemType(String item)	Sets the type of this item. Note that in doing so you will reset the extra data for this stack as well. Throws MaterialNotFoundException when passed an invalid name.
Boolean **isSimilar**(Item item)	Returns whether two items are equal, but does not consider stack size (amount).

9.3 Syntax

9.3.1 Define

The syntax for the define operator and command is as follows:

```
[qualifiers [...]]  <Type> <name> [= expression]
```

qualifiers can be any amount of qualifiers handled in Qualifiers. These always precede the rest of the definition and are always in lowercase. The qualifiers are **keywords** and can therefore not be used as a variable, function or type name. Qualifiers only work on persistent variables and can therefore not be used in scripts.

Type is the type of the variable, which always starts with an uppercase character. This makes it easier to distinguish the type from the variable and qualifiers.

name can be any word that is not a keyword or literal. The name can consist of the following characters: a-z, A-Z, 0-9, _. The name cannot start with a number, _, or an uppercase character.

expression has to be a valid expression resulting in a value of Type. See Expression.

9.3.2 Var

The syntax for the var operator and command is as follows:

```
[name] <op> <expression>
```

name can be any predefined variable or field that is available.

op can be either '=' or any of the numerical operators followed by '='. The former case sets the variable to the result of the expression, the latter case performs the operation on the variable itself and the expression, and saves the result in the variable. Available numerical operators are: =, +=, -=, *=, /=, %=.

expression has to be a valid expression resulting in a value of the correct type. See Expression.

9.3.3 String Formatting

The String literal supports a formatting context in which all expressions are allowed. This is useful for both debugging and readability.

Within any String literal, an expression is started with a '{{' and closed with a '}}'. The resulting value is automatically converted to a String. If this is not possible, it will result in an error.

9.3.4 Expression

The expression syntax allows any variables, literals and functions to be used. Variables are just referred to by their name. Literals follow the syntax rules of their type. Functions are always immediately followed by an opening parenthesis '(', after which the parameters come, separated by a comma, and closes with a ')'. Fields and methods are accessed from an instance by using a '.'.

To chain variables, results of functions and literals, operators are required.

The resulting type is decided by the last remaining object after all sub-expressions have been evaluated, and has to fit the context. If any sub-expressions can not perform an operation with an operator, or be assigned to a given type, the expression fails and an error is thrown.

9.3.5 Time

Some operators and commands take a time parameter. The parameter is the same as in a temporary /ban command: <amount>[s/m/h/d/w].

amount decides the amount of the unit that is used.

The unit can either be blank, s, m, h, d, or w. A blank unit means ticks, that is 1/20 of a second. s means seconds, m means minutes, h means hours, d means days, w means weeks.

Thus 5d means 5 days, 10 means 10 ticks, 7h means 7 hours, and so on.

9.4 Commands

9.4.1 Namespace

The parent command is /namespace. All subcommands in the table below will start with this parent command.

Table 9.27: Namespace Commands

define <name>	Define a new namespace with label/name *name*.
remove <name>	Delete a namespace and all variables and functions attached to it.
info <name>	View metadata about a namespace.
variables <name>	View the definitions and current values of variables in this namespace.
functions <name>	View the definitions of the functions in this namespace.
types <name>	View the types defined in this namespace

9.4.2 Variable

The parent command is /variable. Possible aliases: /var. All subcommands in the table below will start with this parent command.

Table 9.28: Variable Commands

define <namespace> [qualifiers [...]] <Type> <name> [= expression]	Define a new variable with optional qualifiers, a given name and Type and a possible initial value, supplied by the expression. The expression should resolve to the Type parameter.
remove <namespace> <name>	Delete a variable definition.
set <namespace> <name> <= expression>	Set a variable to the result of an expression. The expression should resolve to the Type of the variable, or **null**.
info <name>	View metadata about a variable.

9.4.3 Function

The parent command is /function. Aliases: /func. All subcommands in the table below will start with this parent command.

Table 9.29: Function Commands

define <namespace> [ReturnType] <functionName([Type name[, ...]])>	Define a function in *namespace* returning a value of the type *ReturnType* (Void if empty). The function has the name *functionName* and takes any amount of parameters, defined in sets of *Type name*. Type defines the type of the parameter and name defines the name on which the variable can be addressed. Fails when a function with functionName already exists in the namespace.
remove <namespace> <functionName>	Delete a function definition in a given namespace. This removes the attached scripts.
redefine <namespace> [ReturnType] <functionName([Type name[, ...]])>	Redefine a function. This keeps the associated script, but allows changing the calling parameters or the return type. Will fail when functionName has not been defined yet.
info <name>	View metadata about a function.

9.4.4 User Types

The parent command is /type. All subcommands in the table below start with this parent command. The method and field subcommands have their own tables.

Table 9.30: Type Commands

define <namespace> <Type>	Define a new Type in the *namespace*. Should start with an uppercase character, contain no spaces and only alphanumeric characters.
remove <namespace> <Type>	Deletes a Type, with its associated fields and methods.
info <namespace> <Type>	View metadata about a type.
fields <namespace> <Type>	Display a list of all fields in this Type.
methods <namespace> <Type>	Displays a list of all methods in this Type. Since built-in types are part of every namespace, a built-in type can be inspected too.
constructors <namespace> <Type>	Displays a list of all constructors in this Type. Since built-in types are part of every namespace, a built-in type can be inspected too.

Fields

The parent command is /type field. All subcommands in this table start with this parent command.

Table 9.31: Field Commands

define <namespace> <Type> <Type> <name>	Define a field for *Type*. The field has the given Type and name. Fails when a field with the same name already exists in the type.
remove <namespace> <Type> <name>	Delete a field in Type with the given name.
info <namespace> <Type> <name>	View metadata about a field.

Methods

The parent command is /type method. All subcommands in this table start with this parent command.

Table 9.32: Method Commands

define <namespace> <Type> <ReturnType> methodName([Type name[, ...]])>	Define a method in *Type* returning a value of the type *ReturnType*. The method has the name *methodName* and takes the specified amount of parameters, defined in sets of *Type name*. Type defines the type of the parameter and name defines the name on which the variable can be addressed. Fails when a function with methodName already exists in the type.
remove <namespace> <Type> <methodName>	Delete a method definition in a given Type. This removes the attached scripts.
redefine <namespace> <Type> [ReturnType] <methodName([Type name[, ...]])>	Redefine a method. This keeps the associated script, but allows to change the calling parameters or the return type. Will fail when methodName has not been defined yet.
info <name>	View metadata about a method.

Constructors

The parent command is /type constructor. All subcommands in this table start with this parent command.

Table 9.33: Constructor Commands

define <namespace> Type([Type name[, ...]])	Define a constructor for *Type*. The constructor takes the specified amount of parameters, defined in sets of *Type name*. Type defines the type of the parameter and name defines the name on which the variable can be addressed. Fails when a constructor with the same parameter signature already exists in the type.
remove <namespace> Type([Type name[, ...]])	Delete the constructor with the given parameter signature in the associated Type. This removes the attached scripts.
info <namespace> Type([Type name[, ...]])	View metadata about a constructor.

9.4.5 Script

The script command has the following syntax:

```
/script <action> <type> [typeparameters] [actionparameters]
```

action and *[actionparameters]* are defined in Supported actions for script commands. *type* and *[typeparameters]* are defined in Supported Script types. Script operators that can be used in script lines are defined in Script Operators.

9.5 Scripts

9.5.1 Script Actions

Table 9.34: Supported actions for script commands

create ... [line]	Add a line to the end of the script. When *line* is passed, it adds the line on the given line number instead.
view ...	View the lines of the script in chat.
remove ... [line]	Remove the entire script or a given line.
info ...	List metadata and comments about the script.
export ...	Export the script to hastebin. (See Hastebin for more information).
import ... <id>	Import the script from hastebin. *id* is the identifier of your hastebin script, and must be passed. (See Hastebin for more information).
copy	Copies all scripts in a WorldEdit selected area to the players clipboard, relative to the position of the player. Scripts in the copied area that are removed or not present upon pasting, will not be pasted.
wipe <type>	Removes all scripts of the given script *type* in a WorldEdit selected area.
paste <type>	Pastes all scripts of the given script *type* relative to the new location. (Offsets are calculated from the copy position and then reapplied from the new position).
count <type>	Counts the amount of scripts of the given script *type* in the WorldEdit selected area.
undo	Undos the last script creation, removal, edit, import or export. Currently not supported for any commands involving Functions, Constructors or Methods. Stores up to 10 actions.

9.5.2 Script Types

Table 9.35: Supported Script types

interact [x y z] [world]	Binds to a script triggered when the player interacts with a block. Optionally attached to x, y, z in world.
walk [x y z] [world]	Binds to a script triggered when the player walks over a block. Optionally attached to x, y, z in world.
ground [x y z] [world]	Binds to a script triggered when the player is on the ground. Optionally attached to x, y, z in world.
entity [uuid] [world]	Binds to a script triggered when the player interacts with an entity. Script is removed once the entity dies. Optionally attached to a specific UUID in world.
area <region>	Binds to a script triggered once when a player enters an area. Attached to a WorldGuard region.
function <namespace> <function>	Binds to a function explicitly called from within other scripts or expressions.
method <namespace> <Type> <method>	Binds to a method explicitly called with an instance of *Type*.
constructor <namespace> <Constructor Signature>	Binds to a constructor explicitly called when constructing an instance. *Constructor Signature* serves to distinguish multiple constructors with different signatures.

9.5.3 Script Operators

Table 9.36: Command Script operators

@command <command>	Execute a command as the player. Can only execute the commands the player can also execute.
@bypass <command>	Execute a command as the player in an elevated position. Allows the execution of most admin commands.
@console <command>	Execute a command as the console. Allows the execution of all admin commands, but not those relative to the player.

Table 9.37: Chat Script operators

@chatscript <group> <time> <function>	Binds a function to the following @player message. When the message is clicked in chat, it will be executed. Chatscript runs out when *time* runs out, or if a chatscript of *group* was already clicked.
@player <message>	Sends a message to the player in chat. Supports color codes prefixed with the character '&'.
@prompt <time> <variable> [message]	Stores the next message the player types in chat in the variable. Prompt ends when time runs out, with the given optional message. Defaults to Prompt expired. Message supports color codes with &.

Table 9.38: Variable Script operators

@using <namespace>	Sets the namespace for the following lines. The script can then use the variables and functions from the namespace. Note that the variables in the local namespace will always override variables from an @using namespace.
@define <Type> <name> [= expression]	Defines a variable in the local namespace.
@var [name =] <expression>	Performs an expression or assigns a variable to the result of an expression.

Table 9.39: Control Script operators

@delay <time>	Delays the execution of the rest of the script by a specified amount.
@cooldown <time>	Disallows the executor to re-execute the script for a specified amount of time. When used in functions, terminates the calling script when the function is on cooldown.
@global_cooldown <time>	Disallows all players to execute the script for a specified amount of time. When used in functions, terminates the calling script when the function is on cooldown.
@cancel	Cancels the interaction between player and the object the script is bound to. Only has effect before any @delay, @prompt, @command, @console or @bypass lines.
@return [expression]	Stops the execution of the current script/function, and optionally returns a value, if required.

Table 9.40: Branching Script operators

@if <expression>	Conditionally evaluate the following section of the script if the operand is (or evaluates to) true.
@else	Evaluate the following section of the script if the preceding @if was false.
@elseif <expression>	Conditionally evaluate the following section of the script if the preceding @if was false, and the operand is (or evaluates to) true.
@fi	Ends a conditional section.

Table 9.41: Misc Script operators

@undefined	No operation. May sometimes appear on legacy scripts. Can be used as a comment for complex lines or scripts.

10 Version History

2.0

Additions

Namespaces

Grouping variables and functions so that variables with the same names over different projects do not clash.

User Types

Players can define their own variable Types, including constructors, methods and fields. This also supports the this keyword to select the current instance.

Qualifiers

To support strict variables and still support per-player variables, qualifiers were introduced. The two available qualifiers currently are: relative and final.

Null

To support the absence of a value (due to failed computation or other), null is supported as a substitute for a variable on unambiguous functions, or as the value of a variable.

Expressions

To allow multiple operations on one line (and not separate lines as was previously the case in MSC 1.0), full expression support was added to provide easier operations, function calls, assignment and more.

Functions

Functions have been added to greatly favor reuse of scripts. Functions can take input and can output values, and can be called from any context.

Hastebin

To support easier script writing, MSC 2.0 supports Hastebin. This allows the user to write scripts in a notepad-like environment, and import the scripts to the server from there. Exports are also supported.

Var Script Operators

Variable related script operators have been added: @var supports any assignment or simply an expression. @define supports variable definitions within the local namespace. @using supports switching namespaces for the rest of the script.

Script Metadata

In order for the user to trace back who the script originally belongs to, how many times the script has been executed and more metadata, scripts now store metadata to keep track of sometimes quintessential information while maintaining maps.

Modifications

Variable Typing

Variables are now typed. MSC 2.0 supports built-in types: String, Int, Long, Float, Double, Player, Block, Entity. Variables are strictly typed, and a String can no longer be implicitly used as an Int, as was possible before.

Literals

Literals have been modified to support the new built-in types, and automatically change to the corresponding type. Previously everything was parsed as a String.

@chatscript

Chatscript now only takes a function call as argument, instead of the previously supported script lines.