Trident Language Documentation

by Energyxxer

[Link to HTML mirror]
[Link to complementary documentation]

Table of Contents

What's Trident?

Trident is a Minecraft Functions language by *Energyxxer* that builds upon the vanilla mcfunction syntax to add more and more quality-of-life features any data pack aimed toward a small project or a fully-fledged map may need. All Trident code gets compiled into a data pack you can then distribute for your creations or maps.

What does Trident do that regular functions can't?
A lot of mcfunction's syntax is verbose and repetitive, so often times you will end up writing a lot of function structures just to organize what should be a basic module. Trident takes care of that by adding compile-time variables, constants, and even custom entities and items.

Trident Project Structure

To work with Trident you'll need to organize your files in a specific directory layout similar to that of data packs and resource packs. Inside the project directory, there will usually be three files: a datapack folder, a resources folder, and a *.tdnproj* file. The first two are self-explanatory: the contents of the data pack, and the contents of the resource pack, respectively. The *.tdnproj* is a json file with several properties you may want to edit to fit with your project.

All of the Trident code will go in the datapack folder, inside the functions folder of any namespace. As a refresher, a data pack structure goes as follows:

```
datapack
    pack.mcmeta
    data
        [namespace]
            functions
            advancements
            tags
                blocks
                fluids
                items
                functions
                entity_types
            loot_tables
            structures
```

Inside the `[namespace]>functions` folders, you can place your function files as you normally would. The only exception is that functions you want to process with Trident will have the extension .tdn instead of .mcfunction.

The Format

The Trident language uses the .tdn file extension to represent code files. In a Trident file, you can add commands just as you would normally do in a .mcfunction file. Most commands that are valid in .mcfunction are also valid in .tdn, with a few exceptions:

1. Fake player names starting with the $ symbol won't mean the same thing (this is due to a feature of the Trident language that uses that symbol).
2. The /particle command's delta arguments (dx,dy,dz) only accept real numbers - not coordinates.

A difference between Trident functions and Minecraft functions is whitespace tolerance. Commands in Trident accept extra whitespace and even newlines between arguments (as long as there is no ambiguity). This means you can lay out things such as execute subcommands or NBT in their own indented lines, and it would be no different to keeping them all in the same line. Example:

```
execute
    as @a
    at @s
    align xyz
    run
        summon armor_stand ~ ~ ~ {
            Tags: [
                "position_history",
                "new"
            ],
            Invisible: 1b,
            Marker: 1b
        }
```

An Important Note

The features of the Trident Language are better understood if Trident is seen as an interpreted language, and not a function pre-processor. A file is composed of entries – which may be either commands or instructions.
For every command in a function, its grammar is interpreted, and the final command text is then appended to the function currently being written. Instructions are similar, but instead of appending a command, they change the state of the program (for example, change a variable).
If there are any variables used in a command, that command will only see the state of the variables at the moment it is being interpreted. This makes the order in which Trident files are read very important, and so you should use the language's ordering features to solidify the order of the files.

File Directives

You can specify certain attributes of a .tdn file at the very beginning via directives. Directives are prefixed by a @ symbol. Here are all the available directives:

1. @ on compile Specifies that the current file should be compilation-only. This means that it should not export itself as a .mcfunction file, and that it should run before all other non-compilation files. This is generally used to define objectives and other setup purposes.
2. @ tag <tag name> Marks the current function export with the given function tag. The tag name should be a resource location. If the namespace is omitted, it defaults to minecraft. Cannot be used on compile-only files. Examples:
```
@ tag load
@ tag tick
@ tag custom:tag
```
3. @ meta_tag <tag name> Marks the current file with the given meta tag. The tag name should be a resource location. Unlike the regular @ tag directive, this does not create a function tag, and can be used on compile-only files. Useful for Reflection (see Reflection.getFilesWithMetaTag). Examples:
```
@ meta_tag trident:advancement_reward
```
4. @ require <file> Specifies that this function should run after the specified file. This also imports all the local variables in the specified file into the current file (see Variable Visibility). The file argument should be a resource location, as you would normally access the function file generated by it.
5. @ priority <number> Sets this file's priority number to the real number specified. Higher priority files run before lower priority files. Note that @ on compile and @ require directives all have higher precedence over the @ priority directive. A file with priority 2 will run before a priority 6 if the former is compile-only and the latter is not. A file with priority 9 will run before a file with priority 12 if the former requires the latter.
6. @ language_level <level> Overrides the project language level for this particular file (see Language Levels).
7. @ metadata <object> Sets this file's metadata to the given object (see Reflection.getMetadata).
8. @ breaking If this directive is set, any uncaught exceptions in this file will halt the execution of the rest of the file, similarly to how a non-recovering try block behaves. (see Control Flow>try-catch). This is particularly useful for files that primarily contain compile-time instructions.
9. @ using_plugin <plugin name> If this directive is set, all custom commands from the specified plugin can be used in this file without the namespace. (see Custom Commands)

Language Levels

Trident is designed to be very close to vanilla functions, and avoids standing in the way between the user and the commands it outputs. However, some of the language features are too abstract and far from Minecraft commands to still allow that level of control. In order to logically separate these features, Language Levels come into play.
The language level is a numerical setting that can be changed in the project setting or in a per-file basis. The lowest language level only contains the features that are the most straight-forward, and doesn't allow features that will generate a large volume of commands without user control. The highest language level enables all features of the language, including those which involve many commands, functions and files that the user doesn't have direct control over.
Following is a list of language levels and their features:

1. Basic pre-processing of functions (vanilla commands, variables, objectives, custom entities, basic custom items, etc.)
2. Basic Trident commands (tag update) and **using** instruction. These will usually turn into more than one command each, and short extra functions may be created to properly implement the expected behavior.
3. Custom item events and game logger. These will usually create several functions with many commands each, which may introduce some execution time and space overhead. However, these functions will only generate once the first usage of the feature is found, and further uses of the feature will either reference a generated function or append to one.

The language level can be specified using the `language-level` property in the project configuration file `.tdnproj`, or per file, using the **@ language_level** file directive.

Implicit execute

Trident lets you omit execute and run keywords from your commands entirely. The previous snippet can be simplified to the following command:

```
as @a at @s align xyz
    summon armor_stand ~ ~ ~ {
        Tags: [
            "position_history",
            "new"
        ],
        Invisible: 1b,
        Marker: 1b
    }
```

Note that this comes with a limitation: Omitting the execute and run keywords will require you to follow up with a command; If you need to store a value obtained from the if or unless subcommand alone, you must include execute. Example:

```
execute store result score PLAYER_COUNT global if entity @a
```

The execute keyword cannot be omitted in this case!

Objectives

Trident allows you to declare scoreboard objectives on the fly, and it takes care of the commands that add them. Define objectives using the `define objective` instruction:

```
define objective global
```

This will create the objective named global, and the command to create it will then be added to a separate objective initialization function marked with the `minecraft:load` tag, which runs whenever the data pack is loaded. By default, the defined objective will be of type *dummy*, and have the same display name as its internal name.
You can also specify the criteria and display name of the objective in the definition, both sequence-optional. Example:

```
define objective health health {"text": "Current Health"}
```

You may use objectives not previously defined in commands, and the compiler will silently register them, but it will give you a warning if you do so, asking you to define the objective first. Note that a `scoreboard objectives add` command will also register the objective.

Verbatim Commands

The implementation of the Trident language requires it to receive an update every time a new command feature is added to Minecraft. This is why Trident features verbatim commands, which are a way to export a command as-is, without any processing from Trident. These are commands prefixed by a forward slash (/). Anything after the slash and before the end of the line will be included in the output function as-is. Example:

```
/futurecommand @a minecraft:future_block
```

You may also use interpolation blocks in verbatim commands (see Interpolation Blocks):

```
var raw_command = "futurecommand @a"
/${raw_command + " minecraft:future_block"}
```

Raw Modifiers

You may also want to output an execute modifier as-is, while using Trident syntax for the chained command. This is why you can use the raw execute modifier for that, which takes a string literal or an interpolation block. Example:

```
raw "if block ~ ~-1 ~ minecraft:future_block" tp @s ~ 127 ~
```

This also works with interpolation blocks and lists of strings.

Inner Function Syntax

Creating a file for every function you want in a datapack can often be tedious, and difficult to maintain. This is why Trident implements several ways you can create functions within functions.

Via Define Instruction:

You can define an inner function via the define instruction, with the specified name. The name is a resource location, which can be relative to the parent function, or an absolute path in a non-minecraft namespace. Example:

```
#at: tdndemo:inner_functions.tdn
define function inner {
      if block ~ ~ ~ minecraft:hopper summon armor_stand ~ ~ ~ {Tags:["block_marker", "new"],
Invisible:1b, Marker:1b}
      scoreboard players add BLOCKS_CHECKED global 1
}
#inner is created at tdndemo:inner_functions/inner
```

Via function command:

You can define an inner function inside the function command, which will create the function and immediately call it. Example:

```
# at: tdndemo:inner_functions.tdn
function inner {
      if block ~ ~ ~ minecraft:hopper summon armor_stand ~ ~ ~ {Tags:["block_marker", "new"],
Invisible:1b, Marker:1b}
      scoreboard players add BLOCKS_CHECKED global 1
}
#inner is created at tdndemo:inner_functions/inner
```

In this example, inner is created at tdndemo:inner_functions/inner, and the command is resolved to `function tdndemo:inner_functions/inner`
You may also omit the name, when creating a function via the function command:

```
# at: tdndemo:inner_functions.tdn
function {
      setblock ~ ~ ~ stone
}
```

Here, the inner function is created at tdndemo:inner_functions/_anonymous0
The number at the end will increase for every anonymous inner function created in the parent file.

There is also another way to create inner functions, but we'll discuss it in the Interpolation Values section.

Relative Resource Locations

Wherever you can use a resource location (typically function references), Trident lets you use function paths relative to the function currently being written, starting with a forward slash (/). Example:

```
#at: tdndemo:relative_functions
function /inner
function /
```

The above example resolves into:

```
function tdndemo:relative_functions/inner
function tdndemo:relative_functions
```

The second example (with a single slash) is particularly useful for recursive functions.

Introduction to Custom Objects

Trident allows you to abstract the process of creating a custom entity or item, using its custom object features. This section will describe features common to both entity and items.

Object Inner Functions

Inside a custom entity or item body you may define functions that are logically tied to it.

An object's directory

An object's inner functions need to be created under a directory, and so both the path and name of the file that declares that object affects where those functions will be.
For objects whose name is the same as the file that created it, inner functions will be created relative to the function that created it. Example:

```
# file: mypack:entities/guard.tdn
define entity guard minecraft:iron_golem {
    function idle {
        # ...
    }
}
```

The function idle will be created in function whose resource location is: mypack:entities/guard/idle.

For objects whose name is different from the file that created it, a subfolder will be created relative to the original function, whose name is that of the entity or item. Example:

```
# file: mypack:entities/guards.tdn
define entity town_guard minecraft:iron_golem {
    function idle {
        # ...
    }
}
```

The function idle will be created in function whose resource location is: mypack:entities/guards/town_guard/idle.

Function declarations

The most basic form of object inner functions is the following:

```
function {
    # function file contents here...
}
```

This will declare a function relative to this object's directory. Since this function declaration doesn't have a name, an anonymous function will be created, similarly to the function command (See Inner Function Syntax > Via function command).

You can specify a name for the function, with the following syntax:

```
function <name> {
    # function file contents here...
}
```

This will create the function with the path specified by the name, relative to the object's folder.
Example:

```
# file: mypack:items.tdn
define item key minecraft:tripwire_hook {
    function validate/block {
        # ...
    }
}
```

The function validate/block will be created at mypack:items/key/validate/block.

Object inner functions may have extra syntax options depending on the object type, which are specified in the Custom Entity Functions and Custom Item Functions sections.

Custom Entities

Trident lets you simplify the process of creating "new" entity types. A custom entity starts with a definition specifying its base type (what vanilla entity type it uses), its default NBT data, its passengers, and functions it may want to run. After declaration, you can use its name in place of the summon command entity type, or type= selector argument. Here's an example declaration:

```
# file: mypack:entities/guard.tdn
define entity guard minecraft:iron_golem {

    default name [
            "",
            {"text":"Guard","color":"red"},
            "[",
            {"text":"Lv. ","color":"gold"},
            {"text":"1","color":"yellow"},
            "]"
    ]

    default nbt {
            Silent: 1b,
            CustomNameVisible: 1b,
            DeathLootTable: "minecraft:empty"
    }

    default health 30

    function animation/hurt {
        particle damage_indicator ~ ~ ~ 0.5 0.5 0.5 0.5 8

        playsound minecraft:block.anvil.place master @a

        # ...
    }

    ticking function tick {
        if entity @s[nbt={HurtTime:10s}] function ${this["animation/hurt"]}

        # ...
    }
}
```

That declaration defines an entity named 'guard' that is based on an iron golem. It defines a default name, default NBT and health that will be assigned upon summoning.
As well as these default values, the guard declaration also defines two functions: tick and animation/hurt.
The ticking keyword before the function tells Trident to run that function on every entity of the type **guard**, every tick (using `as @e[...] at @s`).

Inside the tick function, it calls another function called animation/hurt. Notice how that function is accessed via the this identifier, which is set to whatever custom entity type being declared. (see Interpolation Values > Data Types > Custom Entities)

To spawn this custom entity later in the program, all you need is a summon command with an interpolation block retrieving the custom entity. Example:

```
summon $guard ~ ~ ~

# Equivalent to:
summon minecraft:iron_golem ~ ~ ~ {
    Tags: ["trident-entity.mypack.guard"],
    CustomName: '["",{"text":"Guard","color":"red"},"[",{"text":"Lv. ","color":"gold"},{"text":"1","color":"yellow"},"]"]',
    Silent: 1b,
    CustomNameVisible: 1b,
    DeathLootTable: "minecraft:empty",
    Health: 30.0f,
    Attributes:[
        {Name:"generic.maxHealth",Base:30.0d}
    ]
}
```

Trident uses tags to distinguish custom entity types apart. They are `"trident-entity"`, followed by the namespace it was declared in, and the name of the custom entity, all separated by dots.

Custom Entity Declaration

Custom entities are defined by two identifying properties: a name, and a base type. Depending on whether these properties are set, the custom entity will behave differently.

1. **Named custom entities** (Has a name and a base type)
   Named custom entities define a new spawnable entity that doesn't alter the behavior of other vanilla entities. The declaration header looks like this:
   `define entity <name> <base_type>`
   Example:
   `define entity guard minecraft:iron_golem`
   Note: The base type can also be another named custom entity. Example:
   `define entity town_guard $guard`

2. **Default custom entities** (Has no name but has a base type)
   Default custom entities serve as a way to add functionality to a specific vanilla entity type. The declaration header looks like this:
   `define entity default <base_type>`
   Example:
   `define entity default minecraft:iron_golem`
   Behavior defined inside this example entity declaration will affect **all iron golems**.

3. **Wildcard custom entities** (Has no name and no base type)
   Wildcard custom entities serve as a way to add functionality to **all** entities.
   The declaration header looks like this:
   `define entity default *`

4. **Entity Components** (Has a name but no base type)
   Entity components serve as a way to add functionality to specific entities, regardless of their entity type. More about Entity Components in another section.

Custom Entity Body

- Variable Declarations.
- `default name <text_component>`
  (Named entities and components only)
  Changes the entity's base NBT to contain the given CustomName.
- `default nbt <tag_compound>`
  (Named entities and components only)
  Changes the entity's base NBT to contain the tags in the given compound (via a merge operation).
- `default health <real>`
  (Named entities and components only)
  Changes the entity's base NBT to reflect the given max health (Changes the Health tag, as well as the generic.maxHealth attribute).
- `default passengers [<new_entity_literal>, <new_entity_literal>, ...]`
  (Named entities and components only)
  Adds the given entities as passengers of this entity. See Command Syntax Additions > New-Entity literal.
  Example:

```
define entity free_block minecraft:armor_stand {
    default passengers [
        shulker[invisible]{NoAI:1b},
        falling_block[everlasting]{Time:1}
    ]
    # ...
}
```

- Inner functions (See next section)

Custom Entity Functions

Custom entities and entity components support inner functions, as described in the Object Inner Functions section.

Ticking Functions

Within custom entity bodies, you may specify that an inner function should run every tick on entities of that type. This is done via the `ticking` keyword before the `function` keyword.
Example:

```
# file: mypack:entities/guard.tdn
define entity guard minecraft:iron_golem {
    ticking function tick {
        if entity @s[nbt={HurtTime:10s}] function ${this["animation/hurt"]}

        # ...
    }
}
```

That roughly translates to the following command in a Trident-generated function tagged with `#minecraft:tick`:

```
as @e at @s if entity @s[type=$guard] function mypack:entities/guard/tick
```

You may also specify other modifiers that should be added in that command. Extra modifiers go after the `ticking` keyword and before the `function` keyword.
Example:

```
# file: mypack:entities/guard.tdn
define entity guard minecraft:iron_golem {
    ticking if entity @s[nbt={HurtTime:10s}] function animation/hurt {
        particle damage_indicator ~ ~ ~ 0.5 0.5 0.5 0.5 8
        playsound minecraft:block.anvil.place master @a

        # ...
    }
}
```

That roughly translates to the following command:

```
as @e at @s if entity @s[type=$guard] if entity @s[nbt={HurtTime:10s}] function mypack:entities/guard/animation/hurt
```

This is done for the sake of optimization with several custom entity types, so that all entities are only iterated through once per tick. The result is optimized more if there is only one ticking entity function.

NOTE: Since this uses the @e selector to iterate through entities, this will not run for dead players.
You may also specify an interval between executions. Write a time literal right after the 'ticking' keyword.
Example:

```
# file: mypack:entities/guard.tdn
define entity guard minecraft:iron_golem {
    ticking 5s function {
        playsound minecraft:block.anvil.use master @a ~ ~ ~ 1.0 2.0 0.0
    }
}
```

This will create a self-scheduling function tagged with #minecraft:tick, one for each unique time interval used in the project.

Entity Components

Entity components are an **extension** of the custom entity system, which allows adding functionality to arbitrary entities via tags.
Entity components have all the features and functionality of a named entity.
Entity components can be added to an entity's definition or after the summon command entity type, as well as a Trident-exclusive component= selector argument. Here's an example declaration:

```
# file: mypack:components.tdn
define entity component invisible {

    default nbt {
        ActiveEffects: [
            {Id:14b,Duration:200000,Amplifier:0b,ShowParticles:0b}
        ]
    }

    function reapply {
        @tag load
        effect give @e[component=invisible] invisibility 10000 0 true
        schedule function / 10000s
    }
}
```

That declaration defines an entity component named 'invisible'. It defines a default NBT containing the invisibility effect that will be applied upon summoning.
The `invisible` declaration also defines a reapply function that runs every 10,000 seconds that reapplies the effect to all entities with the component.

Just like named custom entities, entity components are represented by a tag on the entity. For entity components, they are `"trident-component"`, followed by the namespace it was declared in, and the name of the component, all separated by dots.

Usage

Summon

There are many ways to use an entity component. The most simple way is through the summon command. To give a summoned entity components, you may put the names of the components, separated by commas, between square braces after the entity name.
Example:

```
summon pig[invisible]

# Equivalent to:
summon minecraft:pig ~ ~ ~ {
    Tags: ["trident-component.mypack.invisible"],
    ActiveEffects: [
        {Id:14b,Duration:200000,Amplifier:0b,ShowParticles:0b}
    ]
}
```

Component add/remove command

Another way to use entity components is by adding/removing them, just like tags. To do so, use the Trident-exclusive component command:

```
component @e[type=pig] remove invisible
component @e[type=cow] add invisible

# Equivalent to:
tag @e[type=pig] remove trident-component.mypack.invisible
tag @e[type=cow] add trident-component.mypack.invisible
```

Component selector argument

You can filter entities based on which components they have, using the Trident-exclusive component selector argument:

```
as @e[component=invisible] say I'm invisible
as @e[component=!invisible] say I'm not invisible

# Equivalent to:
as @e[tag=trident-component.mypack.invisible] say I'm invisible
as @e[tag=!trident-component.mypack.invisible] say I'm not invisible
```

Entity Inheritance

You may also use components in the declaration of a named custom entity, or even another entity component. Do this by using the implements keyword at the end of the entity/component declaration header with the names of the components, separated by commas:

```
# file: mypack:entities/rclick_agent.tdn
define entity rclick_agent minecraft:villager implements invisible {
    default nbt {Silent:1b, Offers: {Recipes: [] }}
}

define entity component outline implements invisible {
    default nbt {Glowing:1b}
}

summon $rclick_agent
summon minecraft:iron_golem[outline]

# Equivalent to:
summon minecraft:villager ~ ~ ~ {
    Tags: [
        "trident-entity.mypack.rclick_agent",
        "trident-component.mypack.invisible"
    ],
    ActiveEffects: [
        {Id:14b,Duration:200000,Amplifier:0b,ShowParticles:0b}
    ],
    Silent: 1b,
    Offers: {Recipes:[]}
}
summon minecraft:iron_golem  ~ ~ ~ {
    Tags: [
        "trident-component.mypack.outline",
        "trident-component.mypack.invisible"
    ],
    ActiveEffects: [
        {Id:14b,Duration:200000,Amplifier:0b,ShowParticles:0b}
    ],
    Glowing: 1b
}
```

Abstract Entity Events

Trident offers a form of polymorphism for entity functions, in the form of Abstract Entity Events.
Abstract entity events are special functions whose behavior is incomplete by default. The function's behavior is entirely dependent on the type of entity that executes the function. The event can then be called through a custom event command, which simply tells the entities to run their own implementation of the event.

Each entity that decides to implement the event can do so through `on <event> <optional modifiers> function {...}`

Here's an example:

```
define event identify

define entity boar minecraft:pig {
    default nbt {Glowing:1b}

    on identify function identify_impl {
        say I'm a boar
    }
}

define entity default minecraft:pig {
    on identify if entity @s[type=!$boar] function identify_impl {
        say I'm a pig
    }
}

define entity default minecraft:sheep {
    on identify function identify_impl {
        say I'm a sheep
    }
}

event @e[distance=..10] identify



# Equivalent to:

define function trident_dispatch_event_identify {
    if entity @s[type=$boar] function /boar/identify_impl
    if entity @s[type=minecraft:pig] if entity @s[type=!$boar] function /default_pig/identify_impl
    if entity @s[type=minecraft:sheep] function /default_sheep/identify_impl
}

function /trident_dispatch_event_identify
```

The event command has the following syntax:
`event <entity> <event>`
which simply gets transformed into a function command as the provided entity
`as <entity> function <event_dispatch_function>`

Custom Items

Trident lets you simplify the process of creating "new" item types and modifying existing ones. A custom item starts with a definition specifying its base type (what vanilla item type it uses), its default NBT data, its name, lore, and functions it may want to run. After declaration, you can use its name in place of the give/clear/replaceitem/etc. commands' item type. Here's an example declaration:

```
# file: mypack:items/teleport_wand.tdn
define objective tp_dist

define item teleport_wand minecraft:carrot_on_a_stick#1 {

    default name {"text":"Teleport Wand","color":"aqua","italic":false}

    default lore [
            [
                {"keybind":"key.use","color":"green"},
                " to teleport to where you're looking"
            ]
    ]

    default nbt {Enchantments:[{}]}

    function raytrace {
        scoreboard players remove @s tp_dist 1
        unless block ^ ^ ^1 minecraft:air
            tp @s ~ ~ ~
        unless score @s tp_dist matches ..0
            positioned ^ ^ ^1
            if block ~ ~ ~ minecraft:air
            function /
    }

    on used function {
        set @s->tp_dist = 64
        anchored eyes
            positioned ^ ^ ^
            anchored feet
            function ${this.raytrace}
    }
}
```

The code above is for an item that teleports the player to the block they're looking at when right-clicking with it.
Its declaration defines an item named 'teleport_wand' that is based on a carrot on a stick with CustomModelData 1. It defines a default name, default lore and default NBT that will be assigned when given.
As well as these default values, the teleport wand declaration also defines two functions: raytrace and an anonymous function.
The on keyword before the function tells Trident to run that function whenever a player fires an event using that item (using scoreboard objective criteria, more on that in Custom Item Functions > Item Event Functions) (using `as @a[...] at @s` on the player that fired the event).

Inside the anonymous function, it calls another function called raytrace. Notice how that function is accessed via the this identifier, which is set to whatever custom entity type being declared. (see Interpolation Values > Data Types > Custom Items)

To give, clear or use this custom item later in the program, all you need is an interpolation block retrieving the custom item. Example:

```
give @a $teleport_wand
clear @a $teleport_wand

# Equivalent to:
give @a minecraft:carrot_on_a_stick{
    TridentCustomItem: -2008252046,
    CustomModelData: 1,
    display: {
        Name: '{"text":"Teleport Wand","color":"aqua","italic":false}',
        Lore: ['[{"keybind":"key.use","color":"green"}," to teleport to where you\'re looking"]']
    },
    Enchantments: [{}]
}
clear @a minecraft:carrot_on_a_stick{
    TridentCustomItem: -2008252046
}
```

Trident uses a TridentCustomItem item tag to distinguish custom item types apart. The value of that tag is determined by hashing the string formed after combining the namespace and the item name together.

Custom Item Declaration

Just like Custom Entities, Custom Items are defined by two identifying properties: a name, and a base type. Depending on whether these properties are set, the custom entity will behave differently.

1. **Named custom items** (Has a name and a base type)
   Named custom items define a new item type that doesn't alter the behavior of other vanilla items. The declaration header looks like this:
   `define item <name> <base_type>`
   Example:
   `define item wand minecraft:stick`
   Note: Unlike entities, custom items' base type cannot be another named custom item.

2. **Default custom items**(Has no name but has a base type)
   Default custom entities serve as a way to add functionality to a specific vanilla item type. The declaration header looks like this:
   `define item default <base_type>`
   Example:
   `define item default minecraft:egg`
   Behavior defined inside this example item declaration will affect **all eggs** unless specified otherwise.

There are no item equivalents of entities' wildcard entities and entity components.

Custom Item Body

- Variable Declarations.
- `default name <text_component>`
  (Named items only)
  Changes the item's base NBT to contain the given display.Name.
- `default lore [<text_component>, <text_component>, ...]`
  (Named items only)
  Changes the item's base NBT to contain the given display.Lore.
  Each text component must represent a different line of lore.
- `default nbt <tag_compound>`
  (Named items only)
  Changes the entity's base NBT to contain the tags in the given compound (via a merge operation).
- Inner functions (See next section)

Custom Item Functions

Custom items support inner functions, as described in the Object Inner Functions section.

Item Event Functions

[LL3] *This feature may only be used if the project's language level is 3*
Within custom item bodies, you may specify that an inner function should run whenever a scoreboard criteria event is fired from that item. This is done via the `on` keyword, followed by the type of event it should be fired from.
Those events are: `used`, `dropped`, `picked_up`, `broken` and `crafted`*. The actions that trigger these events may vary for each item, depending on how the corresponding scoreboard objective criterion behaves.
** crafted is not supported for custom items, only default items support it.*
Example:

```
# file: mypack:items/demo.tdn
define item demo minecraft:snowball {
    on used function {
        say Used demo snowball
    }
    on dropped function {
        say Dropped demo snowball
    }
    on picked_up if entity @s[tag=pickup_enabled] function {
        say Picked up demo snowball
    }
}
```

The function marked with `on used` will say *Used demo snowball* in chat whenever a player uses (throws) a snowball internally marked with the item code associated with this custom item. It will create a scoreboard objective named `uitem.12c23c81` with criterion `minecraft.used:minecraft.snowball`. Whenever that score increases for a player, it runs the corresponding function only if they were holding the `demo` item a tick earlier (that check doesn't occur with **`default`** items).
A similar process occurs with the other two functions declared in the item body.

Pure Events

For default items, you may want to limit certain events to non-custom items. You can achieve this by adding the `pure` keyword after the event key. Example:

```
# file: mypack:items/demo.tdn
define item demo minecraft:snowball {
    # ...
}

define item default minecraft:snowball {
    on used pure function {
        say Used pure snowball
    }
    on used function {
        say Used snowball
    }
}
```

With this second item declaration, all snowballs thrown, custom or not, will print the message "Used snowball". Every non-custom snowball thrown will print the message: "Used pure snowball". Consider the following scenarios:

Player named Foo throws a default snowball (such as from the creative inventory). Chat:

```
[Foo] Used pure snowball
[Foo] Used snowball
```

Player named Foo throws a **$demo** snowball. Chat:

```
[Foo] Used demo snowball
[Foo] Used snowball
```

Command Syntax Additions

Blocks

You can use interpolation blocks wherever a block is required. However, you may also want to overwrite some of the values of the block in that variable. You can do so, simply by appending a blockstate and/or a tag compound after the first interpolation block.
Example:

```
var blockVar = block<chest[facing=east]{Lock:"Key"}>

setblock ~ ~ ~ $blockVar{CustomName:'"Treasure Chest"'}
setblock ~ ~ ~ $blockVar[waterlogged=true]{CustomName:'"Treasure Chest"'}
setblock ~ ~ ~ $blockVar[facing=north,waterlogged=true]
setblock ~ ~ ~ $blockVar{Lock:"Different Key"}

# Equivalent to:
setblock ~ ~ ~ chest[facing=east]{
        Lock:"Key",
        CustomName:'"Treasure Chest"'
}
setblock ~ ~ ~ chest[waterlogged=true,facing=east]{
        Lock:"Key",
        CustomName:'"Treasure Chest"'
}
setblock ~ ~ ~ chest[waterlogged=true,facing=north]{
        Lock:"Key"
}
setblock ~ ~ ~ chest[facing=north]{Lock:"Different Key"}
```

Just like in the vanilla syntax, blockstates or NBT segments in a block literal should be attached directly to the name, without whitespace in between.

Items

Trident adds a shorthand for CustomModelData for items. Simply append a hash sign (#), followed by the custom model data number, to the end of the item name, and before the NBT (if any). Example:

```
give @a stick#5

# Equivalent to:
give @a stick{CustomModelData:5}
```

You can use interpolation blocks wherever an item is required. Just like blocks, you may also want to overwrite some values in the NBT of the item in that variable. You can do so, simply by appending a tag compound after the first interpolation block. Example:

```
var itemVar = item<stick#5>

give @a $itemVar{HideFlags:63}
give @a $itemVar#8{HideFlags:63}

# Equivalent to:
give @a stick{CustomModelData:5,HideFlags:63}
give @a stick{HideFlags:63,CustomModelData:8}
```

Just like in the vanilla syntax, NBT segments in an item literal should be attached directly to the name, without whitespace in between.

New-Entity literal

Trident uses a common format to refer to an entity type to create. It contains information about the entity type to spawn, its entity components and its NBT data, where the latter two are optional. The entity type can also be a custom entity.
This format is used in the summon command, using summon and default passengers in a custom entity.
Examples:

```
summon armor_stand
summon armor_stand[componentA, componentB] ~ ~ ~
summon armor_stand{Glowing:1b} ~ ~ ~
summon $guard[componentA, componentB]{Glowing:1b} ~ ~ ~

define entity block_hitbox minecraft:armor_stand {
    default passengers [
        shulker[invisible]{NoAI:1b}
    ]
    # ...
}
```

New-entity-literals can also be NBT Compounds inside Interpolation Blocks, so long as there is an "id" tag inside, containing the entity type to spawn. Example:

```
summon ${nbt<{id:"minecraft:skeleton",Glowing:1b}>}
# Turns into:
summon minecraft:skeleton ~ ~ ~ {Glowing:1b}
```

Selectors

You can use interpolation blocks wherever an entity is required. However, you may want to add more selector arguments to the selector. You can do so, simply by appending selector arguments after the first interpolation block. **Note:** This will **add** the given selector arguments, and it will not override old ones if that selector type is repeatable. It will also try to merge scores and advancements arguments. Example:

```
var entityVar = entity<@e[type=pig,tag=selected,scores={scoreA=5}]>

tp $entityVar[tag=new] @s
tp $entityVar[tag=new,scores={scoreB=3}] @s

# Equivalent to:
tp @e[type=pig,tag=selected,scores={scoreA=5},tag=new] @s
tp @e[type=pig,tag=selected,tag=new,scores={scoreA=5,scoreB=3}] @s
```

Just like in the vanilla syntax, a selector argument block in a selector should be attached directly to the selector header, without whitespace in between.

score: isset

Trident adds a shorthand to the score execute argument that evaluates to true if a score is set, by matching it against the entire integer range. Example:

```
if entity @s[scores={id=isset}] say ID is set

# Equivalent to:
if entity @s[scores={id=..2147483647}] say ID is set
```

Commands

set

Trident adds a shorthand command that unifies operations to and from scores and NBT.
The syntax is `set <pointer: target> <operator> <pointer: source>`.
Alternatively, the source pointer can be replaced with an NBT value.
Refer to Interpolation Values > Pointers to learn about pointers and their syntax.

This is a table showing what each combination of pointer/value type translates into:

Target Type	Operator	Source Type	Transformed into
SCORE	=	SCORE	`scoreboard players operation A = B`
SCORE	=	VALUE	`scoreboard players set A B`
SCORE	=	NBT	`store result score A data get B`
NBT	=	SCORE	`store result A scoreboard players get B`
NBT	=	VALUE	`data modify A set value B`

NBT	=	NBT	`# When: scale A * scale B == 1:`
			`data modify A set from B`
			
			`# When: scale A * scale B != 1:`
			`store result A * data get B *`

SCORE	+=	SCORE	`scoreboard players operation A += B`
SCORE	+=	VALUE	`scoreboard players add A B`
SCORE	-=	SCORE	`scoreboard players operation A -= B`
SCORE	-=	VALUE	`scoreboard players remove A B`
SCORE	*=	SCORE	`scoreboard players operation A *= B`
SCORE	*=	VALUE	`#[LL2]` `scoreboard players operation A *= #B trident_const`
NBT	*=	VALUE	`store result A data get A *B`
SCORE	/=	SCORE	`scoreboard players operation A /= B`
SCORE	/=	VALUE	`#[LL2]` `scoreboard players operation A /= #B trident_const`
SCORE	%=	SCORE	`scoreboard players operation A %= B`
SCORE	%=	VALUE	`#[LL2]` `scoreboard players operation A %= #B trident_const`
SCORE	<	SCORE	`scoreboard players operation A < B`
SCORE	<	VALUE	`#[LL2]` `scoreboard players operation A < #B trident_const`
SCORE	>	SCORE	`scoreboard players operation A > B`
SCORE	>	VALUE	`#[LL2]` `scoreboard players operation A > #B trident_const`
SCORE	<>	SCORE	`scoreboard players operation A <> B`
SCORE	=	null	`scoreboard players reset A`
NBT	=	null	`data remove A`

For arithmetic operations between a score and a value, an objective `trident_const` is created and initialized with all the constants used in the project. That way, scoreboard operations can simply use the fake players to get a score with a constant. This requires a language level of at least 2.

Example:

```
set @s->altitude = @s->prev_altitude
set @s->altitude = @s.Pos[1] * 100
set #OPERATION->global -= 64
set #OPERATION->global *= 8

set (-30000 0 0).RecordItem.tag.nextVariant (int) = #RANDOM->global
set @s.Variant = (-30000 0 0).RecordItem.tag.nextVariant
set @s.HandItems[0] = {id:"minecraft:bow",Count:1b}

# Equivalent to:
scoreboard players operation @s altitude = @s prev_altitude

store result score @s altitude
    data get entity @s Pos[1] 100

scoreboard players remove #OPERATION global 64
scoreboard players operation #OPERATION global *= #8 trident_const

store result block -30000 0 0 RecordItem.tag.nextVariant int 1
    scoreboard players get #RANDOM global

data modify entity @s Variant set from block -30000 0 0 RecordItem.tag.nextVariant

data modify entity @s HandItems[0] set value {id:"minecraft:bow",Count:1b}
```

tag update

[LL2] *This feature may only be used if the project's language level is at least 2*
Trident adds a shorthand for removing a tag from all entities and reapplying it only to specific entities. Example:

```
tag @e[nbt={OnGround:1b}] update onGround

# Equivalent to:
tag @e remove onGround
tag @e[nbt={OnGround:1b}] add onGround
```

component

This component command is similar in use to the tag command, but its purpose is to add and remove custom entity components.
See Entity Components > Component add/remove command.

event

This event command is used to invoke an event on certain entities. See Abstract Entity Events.

summon

The summon command syntax has been modified so that it uses New-Entity literals. As a result, you can append a list of entity components to add to the summoned entity, as well as specify the NBT of the spawned entity while omitting the coordinates. See Command Syntax Additions > New-Entity literal. The default syntax for the summon command is still valid, however.

expand

The expand command applies the same set of modifiers to multiple commands in a block.
Syntax:

```
expand {
    # Commands go here:
}
```

Example:

```
if entity @e[tag=!assigned] expand {
    function try_assign_0
    function try_assign_1
    function try_assign_2
    function try_assign_3
}
# Equivalent to:
if entity @e[tag=!assigned] function try_assign_0
if entity @e[tag=!assigned] function try_assign_1
if entity @e[tag=!assigned] function try_assign_2
if entity @e[tag=!assigned] function try_assign_3
```

It's advised you only use this when you expect the condition to be invalidated somewhere in the block to stop the other commands from running. If you want all the commands in the block to run if the condition is met, consider using functions instead.

gamelog

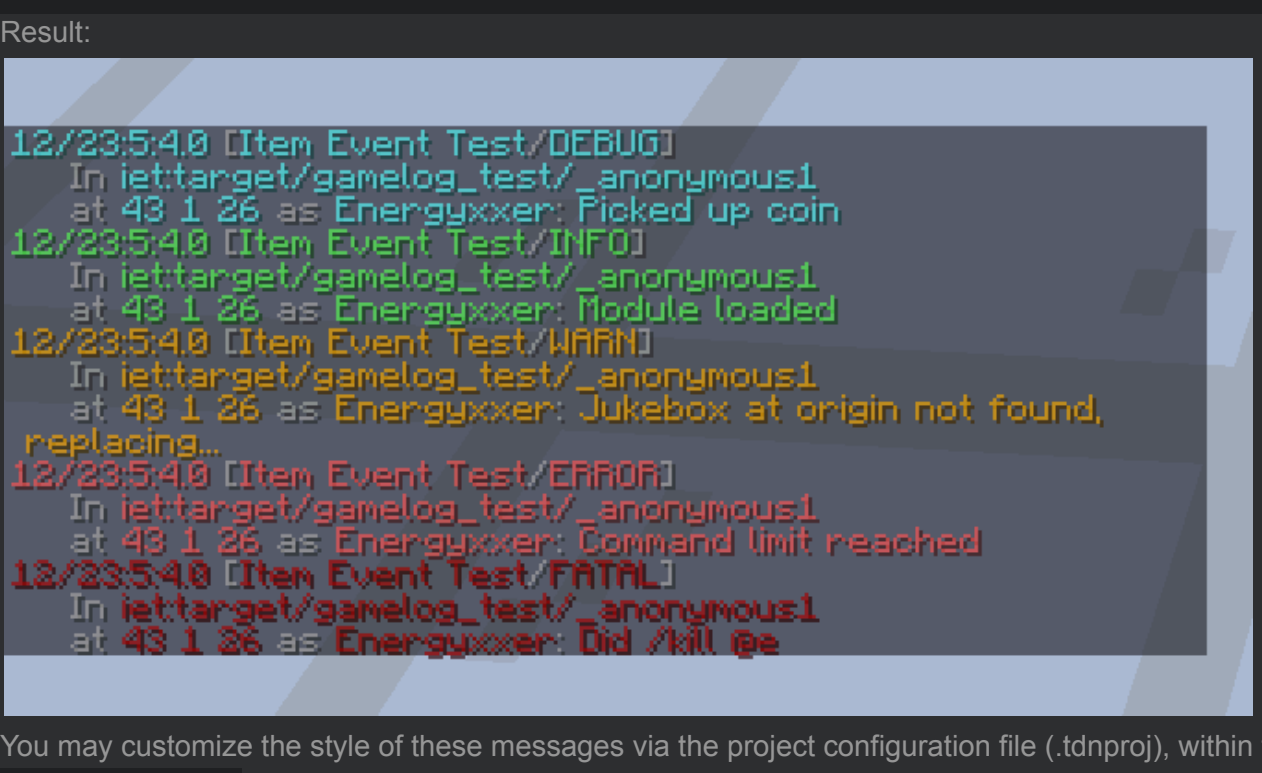
[LL3] *This feature may only be used if the project's language level is 3*
Trident adds a runtime equivalent of the log instruction. This lets you send debug messages to select people, containing information about the execution context of the command. The syntax is `gamelog <info|debug|warning|error|fatal> <value>`.
The different severities correspond to an integer, and a message of a certain severity will only be shown to players whose `tdn_logger` score is at least the severity number.
This is a table showing which type of log each player sees, depending on the `tdn_logger` score:

```
tdn_logger 0  : none
tdn_logger 1  : fatal
tdn_logger 2  : fatal, errors
tdn_logger 3  : fatal, errors, warnings
tdn_logger 4  : fatal, errors, warnings, info
tdn_logger 5  : fatal, errors, warnings, info, debug
tdn_logger 6  : debug
```

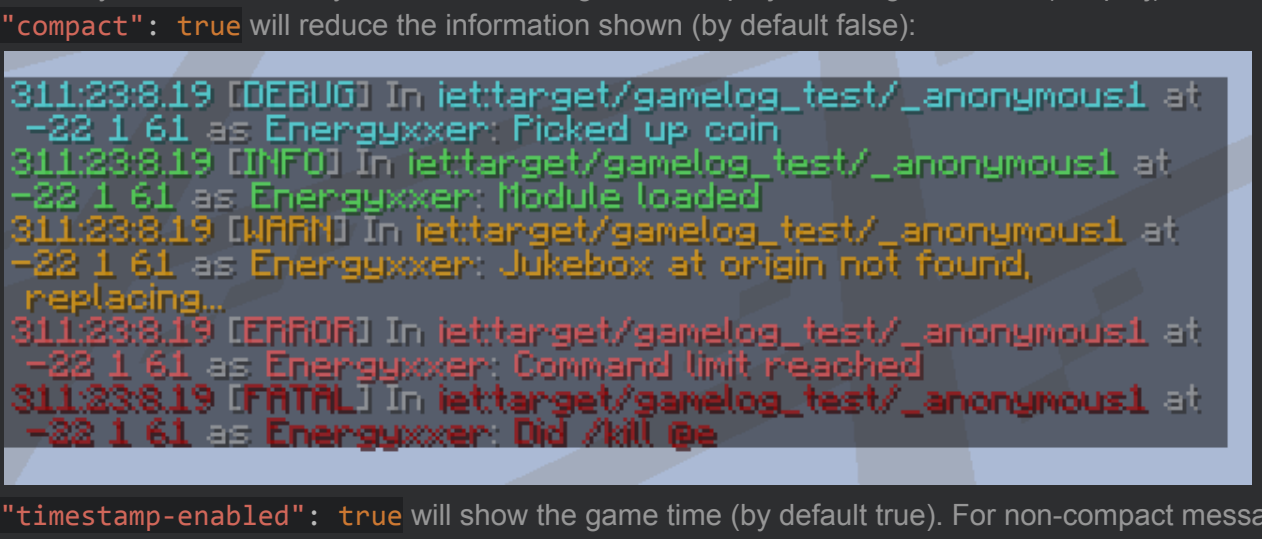
Example:

```
gamelog debug "Picked up item"
gamelog info "Module loaded"
gamelog warning "Jukebox at origin not found, replacing..."
gamelog error "Command limit reached"
gamelog fatal "Did /kill @e"
```

Result:



You may customize the style of these messages via the project configuration file (.tdnproj), within the game-logger object.
`"compact": true` will reduce the information shown (by default false):



`"timestamp-enabled": true` will show the game time (by default true). For non-compact messages, it will be in the format: `dd/hh:mm:ss.t` where dd is days, hh is hours, mm is minutes, ss is seconds and t is ticks. For compact messages, the format is `hh:mm:ss.t`.
`"pos-enabled": true` will show the block at which the command was executed (by default true).
`"line-number-enabled": true` will show the line number where the gamelog command is located. **Note that this line number is for the Trident file the command is located in, not the function it's in**. By default false.

The gamelog messages show a variety of information in addition to the description, such as the execution time, message severity, function file, command sender, execution position and project name. Note that some of these may or may not be present depending on the project configuration.

Execute Modifiers

if/unless score: isset

Trident adds a shorthand to the score condition modifier that evaluates to true if a score is set, by matching it against the entire integer range. Example:

```
if score @s id isset say ID is set

# Equivalent to:
if score @s id matches ..2147483647 say ID is set
```

if/unless selector

Trident adds a shorthand to the entity condition modifier. You can omit the entity keyword if you use a selector. Example:

```
if @s[tag=valid] say Valid

# Equivalent to:
if entity @s[tag=valid] say Valid
```

raw

The execute modifier version of verbatim commands. Takes a string literal or an interpolation block (containing either a string or a list of strings) as a parameter. See Raw Modifiers. Example:

```
raw "if block ~ ~-1 ~ minecraft:future_block" tp @s ~ 127 ~
```

Instructions

Instructions in Trident typically alter the state of the program during compilation, or generate many commands or functions at once. As such, they cannot be used with execute modifiers, as they are not commands.

log

The log instruction is used to display information after the project has finished compiling. In command-line based systems, this may be in the form of text in the console. In the official Trident UI, these logs will appear in a panel at the bottom of the program, which allow you to quickly jump to the location it was logged from.
There are different types of messages you can log: `info`, `warning` and `error`.
None of these will halt the execution of the program, but the latter (`error`) will prevent the output data pack and resource pack from generating.
Syntax:

```
log <info|warning|error> <value>
```

Example:

```
log info "Added  " + i + " to the list"
```

using

using tag

[LL2] *This feature may only be used if the project's language level is at least 2*
Tags on entities are often used temporarily to target a specific entity without changing the context. This requires adding and removing the tag from the entity. Trident adds an instruction that makes this process easier. The using tag instruction runs a specific block after adding a tag to the entity given, and automatically adds the tag remove command at the end. Note that the tag is removed from **all** entities at the end, so don't use a tag that is used permanently somewhere else.
Syntax:

```
using tag <tag_to_use> <entity_to_tag> <optional_modifiers> {
    # Commands in the middle go here:
}
```

Examples:

```
using tag needs_id @e[scores={id=0},limit=1] {
    scoreboard players operation @e[tag=needs_id] id > * id
    tellraw @a ["ID assigned to ", {"selector":"@e[tag=needs_id]"}]
}
# Equivalent to:
tag @e[scores={id=0},limit=1] add needs_id
scoreboard players operation @e[tag=needs_id] id > * id
tellraw @a ["ID assigned to ", {"selector":"@e[tag=needs_id]"}]
tag @e remove needs_id
```

```
using tag needs_id @e[scores={id=0},limit=1] if block ~ ~-1 ~ air {
    scoreboard players operation @e[tag=needs_id] id > * id
    tellraw @a ["ID assigned to ", {"selector":"@e[tag=needs_id]"}]
}
# Equivalent to:
as @e[scores={id=0},limit=1] if block ~ ~-1 ~ air tag @s add needs_id
scoreboard players operation @e[tag=needs_id] id > * id
tellraw @a ["ID assigned to ", {"selector":"@e[tag=needs_id]"}]
tag @e remove needs_id
```

using summon

[LL2] *This feature may only be used if the project's language level is at least 2*
Another common use of temporary tags is targeting a newly summoned entity. Trident makes this much easier by adding a function block that runs `as` the summoned entity. The selector targeting the entity with the tag is optimized whenever possible so that it only targets entities close to the summoning location.
Syntax:

```
using <summon_command> with <tag_to_use> <optional_modifiers> {
    # Commands in the middle go here:
}
```

Example (without modifiers):

```
using summon armor_stand ~-3 0 ~4 {
        CustomName:'"guard_model"',
        CustomNameVisible:1b,
        Tags:["entity_model"]
    }
    with new {
        set @s->id = NEXT_ID->global
        scoreboard players add NEXT_ID global 1
        set @s->cooldown = 0
        set @s->jump_time = 0
    }

# Equivalent to:
summon armor_stand ~-3 0 ~4 {
    CustomName:'"guard_model"',
    CustomNameVisible:1b,
    Tags:["entity_model","new"]
}

as @e[type=armor_stand,tag=new,limit=1,y=0,distance=4.99..5.01] function {
    tag @s remove new
    set @s->id = NEXT_ID->global
    scoreboard players add NEXT_ID global 1
    set @s->cooldown = 0
    set @s->jump_time = 0
}
```

Example (with modifiers):

```
using summon armor_stand ~-3 0 ~4 {
        CustomName:'"guard_model"',
        CustomNameVisible:1b,
        Tags:["entity_model"]
    }
    with new
    if score ID_ENABLED global matches 1.. {
        set @s->id = NEXT_ID->global
        scoreboard players add NEXT_ID global 1
        set @s->cooldown = 0
        set @s->jump_time = 0
    }

# Equivalent to:
summon armor_stand ~-3 0 ~4 {
    CustomName:'"guard_model"',
    CustomNameVisible:1b,
    Tags:["entity_model","new"]
}

as @e[type=armor_stand,tag=new,limit=1,y=0,distance=4.99..5.01] function {
    tag @s remove new
    if score ID_ENABLED global matches 1.. function {
        set @s->id = NEXT_ID->global
        scoreboard players add NEXT_ID global 1
        set @s->cooldown = 0
        set @s->jump_time = 0
    }
}
```

Trident will create at least one extra function for the using summon block, and it will use two only if necessary (e.g. if one of the modifiers specified may cause the command not to run, like conditionals if and unless). It will only create one extra function if modifiers likely wouldn't change the execution condition (e.g. `at @s`)

eval

The eval instruction is used to evaluate an interpolation value or expression that might change the state of the program.
Syntax:

```
eval <value_or_expression>
```

Example:

```
var count = 0
eval count += 4
eval someFunction(count)
```

within

The within instruction makes it easy to write commands that run at every block in a volume. This is done by assigning coordinates to a variable inside a loop, similar to a foreach loop in many languages.
Syntax:

```
within <identifier> <coordinates:from> <coordinates:to> {
    # Commands using the coordinates go here:
}
```

Example:

```
within pos ~-1 ~ ~-1 ~1 ~ ~1 {
    positioned $pos if block ~ ~ ~ hopper summon armor_stand
}
# Equivalent to:
positioned ~-1 ~  ~-1 if block ~ ~ ~ hopper summon armor_stand
positioned ~-1 ~  ~   if block ~ ~ ~ hopper summon armor_stand
positioned ~-1 ~  ~1  if block ~ ~ ~ hopper summon armor_stand
positioned ~   ~  ~-1 if block ~ ~ ~ hopper summon armor_stand
positioned ~   ~  ~   if block ~ ~ ~ hopper summon armor_stand
positioned ~   ~  ~1  if block ~ ~ ~ hopper summon armor_stand
positioned ~1  ~  ~-1 if block ~ ~ ~ hopper summon armor_stand
positioned ~1  ~  ~   if block ~ ~ ~ hopper summon armor_stand
positioned ~1  ~  ~1  if block ~ ~ ~ hopper summon armor_stand
```

You may also specify the step by which to increase the coordinates. By default, the step is 1, but you can overwrite it by adding `step <real>` after the coordinates.
Example:

```
within pos ~-16 ~ ~-16 ~15 ~ ~15 step 16 {
    positioned $pos fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
}
# Equivalent to:
positioned ~-16 ~   ~-16 fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
positioned ~-16 ~   ~    fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
positioned ~-16 ~   ~16  fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
positioned ~    ~   ~-16 fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
positioned ~    ~   ~    fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
positioned ~    ~   ~16  fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
positioned ~16  ~   ~-16 fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
positioned ~16  ~   ~    fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
positioned ~16  ~   ~16  fill ~ ~ ~ ~14 ~ ~14 white_stained_glass
```

Coordinates used in the within instruction may have mixed coordinates, but the coordinate types of each axis must match.

```
# This is allowed:
# Relative Absolute Absolute - Relative Absolute Absolute
within pos ~-1 0 10 ~1 15 12 {
    # ...
}

# This is NOT allowed:
# Relative Relative Absolute - Relative Absolute Absolute
within pos ~-1 ~ 10 ~1 15 12 {
    # ...
}
```

Control Flow

if - else

(Not to be confused with the if/unless execute modifiers)
The **if instruction** runs an entry block (a list of commands and/or instructions), if and only if a given compile time expression evaluates to the boolean value true. You may also specify that, in case the expression above evaluates to false, another entry block should be run instead.
Syntax:

```
if(<expression>) {
    <commands and instructions...>
} else {
    <commands and instructions...>
}
```

...where the else keyword and the following block are optional.
Examples:

```
var someVar = 5
if(someVar == 5) {
    summon cow
    log info "First block run"
} else {
    summon sheep
    log info "Second block run"
}
log info "Out of if/else blocks"

# Evaluates the following lines:
summon cow
log info "First block run"
log info "Out of if/else blocks"
```

You may omit the curly braces if there is only one entry in the block. Example:

```
if(expr == 5) summon cow
else summon sheep
```

You may see older Trident code using `do if` instead of `if`. It was previously required for compile-time `if`s to have a do keyword before them, to distinguish them from the execute modifier `if`. This restriction has been lifted so if there's an open parenthesis right after the `if`, it is interpreted as a compile-time `if`. For backwards compatibility, `do if` is still allowed, although discouraged.

while

The while instruction runs an entry block repeatedly based on a compile time expression, if it evaluates to the boolean value true.
Syntax:

```
while(<expression>) {
    <commands and instructions...>
}
```

Examples:

```
var counter = 3
while(counter > 0) {
    give @s stick $counter
    eval counter--
}

# Equivalent to the following:
var counter = 3
give @s stick $counter
eval counter--              # counter is now 2
give @s stick $counter
eval counter--              # counter is now 1
give @s stick $counter
eval counter--              # counter is now 0

# Ultimately evaluates to:
give @s stick 3
give @s stick 2
give @s stick 1
```

for

The for instruction runs an entry block repeatedly based on a compile time expression, just like while, except for supports different headers for different uses.
Syntax:

```
for(<loop_header>) {
    <commands and instructions...>
}
```

Classic Header

The classic header for the for instruction has three expressions separated by semicolons: the initializer, the condition and the iterator. The **initializer** expression is evaluated once at the beginning of the loop. The **condition** expression is evaluated before every loop and it should evaluate to a boolean: true if the loop should continue, false if the loop should end. The **iterator** expression is evaluated after every loop. Syntax:

```
for(<initializer>;<condition>;<iterator>) {
    <commands and instructions...>
}
```

Example:

```
for(var i = 0; i < 5; i++) {
    worldborder add $i
}

# Equivalent to the following:
var i = 0
worldborder add $i
eval i++                # i is now 1
worldborder add $i
eval i++                # i is now 2
worldborder add $i
eval i++                # i is now 3
worldborder add $i
eval i++                # i is now 4
worldborder add $i
eval i++                # i is now 5

# Ultimately evaluates to:
worldborder add 0
worldborder add 1
worldborder add 2
worldborder add 3
worldborder add 4
```

Iterator Header

The iterator header for the for instruction runs the entry block for every element in an iterable value (list, dictionary). It creates a temporary variable that will contain the elements of the iterator. Syntax:

```
for(<temp_var_name> in <iterable>) {
    <commands and instructions...>
}
```

Example:

```
var items = [item<stick>,item<glass>,item<bow>,item<arrow>]
for(itm in items) {
    give @s $itm
}

# Ultimately evaluates to:
give @s stick
give @s glass
give @s bow
give @s arrow
```

switch

The switch instruction takes a single value, and associates case values to the entry block that should run if the original expression matches that value. Note that once a case has been matched, all the entry blocks after it will run, unless the break instruction is used. The optional default case will match any expression value and run the associated block.
Syntax:

```
switch(<expression>) {
    case <match_0>: {
        <commands and instructions...>
    }
    case <match_1>: {
        <commands and instructions...>
    }
    case <match_n>: {
        <commands and instructions...>
    }
    default: {
        <commands and instructions...>
    }
}
```

Example:

```
var someVar = 0
switch(someVar) {
    case 0:
    case 1: {
        log info "someVar is 0 or 1"
        break
    }
    case 2: {
        log info "someVar is 2"
        break
    }
    case 3: {
        log info "someVar is 3"
        break
    }
    default: {
        log info "someVar is not between 0 and 3"
    }
}
# Evaluates the following lines:
log info "someVar is 0 or 1"
```

break

The break instruction is used in conjuction with looping instructions or switch.
In loops, break will forcefully exit out of the innermost loop, even if the condition still matches.
In switch, it will forcefully exit the innermost switch instruction. Any entries after the break, and still inside the structure it exits, will not be evaluated.
Syntax:

```
break
```

Example:

```
var counter = 3
while(counter > 0) {
    give @s stick $counter
    eval counter--
    break
}

# Equivalent to the following:
var counter = 3
give @s stick $counter
eval counter--              # counter is now 2

# Ultimately evaluates to:
give @s stick 3
```

continue

The continue instruction is used in conjuction with looping instructions.
continue will end the current iteration and begin the next, if possible. Conditions will still determine if the next iteration runs or not. In the classic for header, the iteration expression will still run after a continue.
Any entries after the continue, and still inside the loop it affects, will not be evaluated.
Syntax:

```
continue
```

Example:

```
for(var i = 0; i < 5; i++) {
    if(i == 3) {
        continue
    }
    worldborder add $i
}

# Equivalent to the following:
var i = 0
worldborder add $i
eval i++                # i is now 1
worldborder add $i
eval i++                # i is now 2
worldborder add $i
eval i++                # i is now 3
eval i++                # i is now 4
worldborder add $i
eval i++                # i is now 5

# Ultimately evaluates to:
worldborder add 0
worldborder add 1
worldborder add 2
worldborder add 4
```

return

The return instruction is used in conjuction with dynamic functions, and is used to transfer the result of an operation to the environment that called it.
return will forcefully exit from the dynamic function that contains it.
Any entries after the return, and still inside the dynamic function it's in, will not be evaluated.
Syntax:

```
return <optional_value>
```

If the return value is omitted, it defaults to `null`.
Example:

```
var min = function(a, b) {
    if(a < b) {
        return a
    } else {
        return b
    }
    # This line never runs
}

log info min(12, 4) # Logs 4
```

Exceptions

try-catch

The try-catch instruction is used to handle exceptions thrown inside an entry block. Exceptions in the try block will halt execution of that block and will jump to the catch block.
The catch block will not run if no exception was caught in the try block.
Syntax:

```
try {
    <commands and instructions...>
} catch(<identifier: exception_var_name>) {
    <commands and instructions...>
}
```

Examples:

```
try {
    eval 1 / 0
    eval null.property
} catch(ex) {
    # ex contains an exception
    log info ex

    # Prints the following:
    # Arithmetic Error: / by zero
    #     at stt:documentation_test ~ <body>
}
```

The standard try-catch block stops execution after the first exception has been caught, and holds that exception in the variable specified in the catch block.

Also see: Data Types > Exceptions

try-recovering-catch

The try-recovering-catch instruction is a variation of the try-catch instruction, where the only difference is, after catching an exception in the try block, it will continue execution until it reaches the end. After that, the variable specified in the catch block will contain a list of exceptions, collected from the try block.
The catch block will not run if no exceptions were caught in the try block.
Syntax:

```
try recovering {
    <commands and instructions...>
} catch(<identifier: exception_var_name>) {
    <commands and instructions...>
}
```

Examples:

```
try recovering {
    eval 1 / 0
    eval null.property
} catch(gx) {
    # gx contains a list of exceptions
    log info gx

    # Prints the following:
    # [Arithmetic Error: / by zero, Type Error: Unexpected null value]
}
```

try-recovering-catch blocks behave more similarly to entries outside any exception handling block, as an error in a single command will not stop later commands from logging their errors. However, unlike entries outside of it, try-recovering blocks will

Also see: Data Types > Exceptions

throw

The throw instruction is used to inform the calling environment that an error occurred in the execution of a piece of code, using a string to describe the error, or another exception that has been caught.
throw will forcefully exit from the dynamic function that contains it, unless it was called from a try-recovering block.
Uncaught exceptions will prevent the output data pack and resource packs from being generated.
Syntax:

```
throw <string: message>
throw <exception>
```

Example:

```
var sqrt = function(x) {
    if(x < 0) {
        throw "Illegal argument 'x', expected non-negative"
    } else {
        # ...
    }
}
```

Variables

You can declare variables to use in commands with the following syntax:

```
[<visibility>] [<final>] var <identifier> [ : [<constraint>]] = <value>
```

Where visibility is optional, and specifies the visibility of the variable to other files (see Variable Visibility). If not specified, defaults to *local*.
The identifier is the name of the variable it is creating.
The value is an Interpolation Value (such as integers, real numbers, strings, booleans, null, etc.) to assign to the variable (see Interpolation Values)
You can access variables from commands using Interpolation Blocks, which we'll cover later (see Interpolation Blocks).

If you specify `final` before the **var** keyword, the variable will only be able to change its value once. Any further attempts to change the value in this variable will fail.

Variable Visibility

There are three main modes of variable visibility, which affect how files other than the declaring one can access the variable. They are:

1. `private`: Only the declaring file can access the symbol.
   In the context of Class members, private visibility means only the defining class can access it.
2. `local`: Only the declaring file and files which @require the declaring file can use it. This is the default for variables declared with *var*.
   In the context of Class members, local means only the declaring file and subclasses can access this symbol.
3. `global`: Any file can see the variable without any @require directives, as long as the declaring file is read before. This is the default for custom items and entities.
   This cannot be used for Class members.
4. `public`: Only usable by Class members. Public visibility means any file can access this symbol.

Type Constraints

Variables and other symbols (such as function parameters and return values) can be constrained to be a certain type.
A type constraint has two properties:

1. The Type. This is the type definition that the values MUST be an instance of. If a type constraint has no type, it will accept any type of value.
   The type may be a primitive or a user-defined class.
2. Nullability: If the type constraint is nullable, it will accept null values as well as whatever type is assigned to it. If the type constraint is not nullable, attempting to assign a null value to it will result in an exception. In a type constraint literal, this is represented as a question mark after the type. Omitting the question mark will make the type constraint not nullable.

Assigning to a type-constrained symbol will check the type of the value before assigning, and throw an exception if the value doesn't conform to the constraints.

A type constraint is declared by using a colon `:`, followed by the type to constraint to and, optionally, a question mark `?` to mark it nullable.
Example using variables:

```
var someInt : int = 3                   # OK
var someString : string = "a"           # OK
var someNullableString : string? = "b"  # OK

eval someInt = 5                # OK
eval someInt = ""               # ERROR

eval someString = "c"           # OK
eval someString = null          # ERROR

eval someNullableString = "d"   # OK
eval someNullableString = null  # OK
```

Assignment will also check if the value can be implicitly converted into the target type, and automatically perform the conversion. Some primitive types have implicit conversions defined, such as int to real. Example:

```
var someReal : real = 3.0  # OK
eval someReal = 5          # OK
```

If a type constraint isn't defined for a symbol, it defaults to a constraint that can accept any value of any type, null or not.

```
var anything  = 3.0   # OK
eval anything = "ab"  # OK
eval anything = null  # OK
```

Inferred Constraints

Some symbol definitions allow the constraints to be inferred by the initial value, such as variable definitions and field definitions. This is done by using a colon `:` but omitting the type. A type constraint will be created using the type definition of the initial value assigned to it. Note that constraints cannot be inferred for null initial values.
Example:

```
var symbolA : = 3       # Constrained to int
var symbolB : = 3.0     # Constrained to real
var symbolC : = "ab"    # Constrained to string
var symbolD : = null    # ERROR
```

Type Checking and Type Conversion

You may come across a situation where you'd like to check the type of a variable before doing any operations with it. Trident has a special operator specifically for this.

`is` operator

The `is` operator is a simple binary operator that takes a value on the left, and a type on the right (can be a primitive or a custom class). This operator will return `true` if the value given is of the specified type, either the exact type or another type that extends the one specified. Otherwise it will return `false`. Null values will always return `false`.
Note that this operator will **not** check whether the value can be coerced into the given type.
Example:

```
log info 5    is int       # Logs true
log info "a"  is string    # Logs true
log info 5    is real      # Logs false
log info 5.0  is real      # Logs true
log info null is string    # Logs false
```

Another way that might be useful is the `type_definition.of()` function, which will return the exact type of the value given.

You might also need to convert a value into a certain type, and some types may have ways to convert a value into another of a different type. For instance, you may want to convert an integer into a string, a string into a text component, a real number into a real number range, etc.
This can be done via **casting**, and there are two ways to do it:

Strict casting

This is done by prepending the type in parenthesis to the value you want to cast. Strict casting will attempt to convert the value into the target type, and it will throw an exception if the type does not support casting to the target type.
Example:

```
log info (real) 5           # 5.0
log info (tag_byte) 1       # 1b
log info (int) 2.8          # 2
log info (nbt_path) null    # null
log info (tag_byte) "foo"   # ERROR
```

Weak casting (`as` operator)

The as operator is a binary operator that does almost exactly the same as casting, except that if the value is unable to be cast to the target type, it will return null instead of throwing an error.
Example:

```
log info 5     as real       # 5.0
log info 1     as tag_byte   # 1b
log info 2.8   as int        # 2
log info null  as nbt_path   # null
log info "foo" as tag_byte   # null
```

Interpolation Values

Data Types

Integers

Identifier: `int`
Integer values represent any kind of 4-byte integer number with a range from -2^{31} to $2^{31}-1$, inclusive. An integer value can be created via a number literal like so:

```
var someInt = 3
var anotherInt = -72
clear @s minecraft:stone $someInt
```

Integers can be used in interpolation blocks in commands or instructions, in places where an integer is typically needed (such as the give command item count, effect give duration, etc.).

Integer literals may also be specified in binary form as well as hexadecimal, with 0b and 0x prefixes respectively. Example:

```
var someInt = 0b0101  # 5
var anotherInt = 0xFF # 255
```

Conversions

- `real` (explicit & **implicit**): A real with the same value as this integer.
- `int_range` (explicit & **implicit**): An integer range with min and max set to this integer.
- `real_range` (explicit & **implicit**): A real range with min and max set to this integer.
- `nbt_value` & `tag_int` (explicit): An int tag with the same value as this integer.
- `tag_byte` (explicit): A byte tag with the same value as this integer (adjusted into the byte range with modulo).
- `tag_short` (explicit): A short tag with the same value as this integer (adjusted into the short range with modulo).
- `tag_float` (explicit): A float tag with the same value as this integer.
- `tag_double` (explicit): A double tag with the same value as this integer.
- `tag_long` (explicit): A long tag with the same value as this integer.

Reals

Identifier: `real`
Real values represent any real number that can be stored in an 8-byte double-precision floating-point number. A real value can be created via a number literal with a decimal point like so:

```
var pitch = 0.5
var someReal = 1000.04
playsound minecraft:ui.button.click master @a ~ ~ ~ 1 $pitch 0
```

Real numbers can be used in interpolation blocks in commands or instructions, in places where a real number is typically needed (such as the playsound volume and pitch, spreadplayers distance, etc.).

Real literals may also be specified in engineering notation
Example:

```
var someReal = 1.5e+3    # 1500.0
var anotherReal = 1.5e-3 # 0.0015
```

Conversions

- `int` (explicit): An integer value with the result of truncating this real number.
- `real_range` (explicit & **implicit**): A real range with min and max set to this real number.
- `nbt_value` & `tag_double` (explicit): A double tag with the same value as this real number.
- `tag_float` (explicit): A float tag with the same value as this real number.

Booleans

Identifier: `boolean`
Real values represent true/false values. A boolean value can be created via a boolean literal like so:

```
var on = true
var off = false
```

Strings

Identifier: `string`
String values represent any sequence of characters, which together form text. A string value can be created via a string literal like so:

```
var str = "This is a string"
var anotherString = "A string with \"escaped\" characters"
var singleQuotedString = 'this is a string too, and the single quote (") needs no escaping'
var someObjective = "field0"
define objective $someObjective
scoreboard players add @s[name=$str] $someObjective 1
```

Strings can be used in interpolation blocks in commands or instructions, in places where a string literal is expected (name selector argument) or an identifier is expected (objective names, tag command tag names, etc.).

Members

Indexer: `string[index : int]`
readonly
Returns a single-character string containing the character at the given index.
Example:

```
log info "Mojang"[3]  #logs "a"
```

`length: int`
readonly
Contains the length of this string; that is, the number of characters in this string.
Example:

```
log info "Mojang".length  #logs 6
```

`substring: function(beginIndex : int, endIndex : int?) : string`
readonly
Returns a string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`.
Thus the length of the substring is `endIndex-beginIndex`.
If `endIndex` is omitted, it defaults to the length of the string.
Example:

```
log info "stonecutter".substring(5, 8)  #logs "cut"
log info "replaceitem".substring(7)     #logs "item"
```

`indexOf: function(str : string) : int`
readonly
Returns the first index at which the given parameter substring occurs in this string. Returns -1 if the string doesn't contain the given parameter.
Example:

```
log info "a b c d e".indexOf("d e")  #logs 6
```

`lastIndexOf: function(str : string) : int`
readonly
Returns the last index at which the given parameter substring occurs in this string. Returns -1 if the string doesn't contain the given parameter.
Example:

```
log info "this appears twice in this string".lastIndexOf("this") #logs 22
```

`split: function(delimiter : string, limit : int?) : list`
readonly
Splits this string around matches of the given substring, and returns the result as a list of strings. The limit, if greater than 0, limits the number of elements in the return list, and only splits the string $limit times. Additionally, if the limit is set to zero, consecutive occurrences of the delimiter will split the string once, avoiding empty strings in the result list.
If `limit` is omitted, it defaults to 0.
Example:

```
log info "foo:and:boo".split(":")  #logs ["foo", "and", "boo"]
```

`splitRegex: function(regex : string, limit : int?) : list`
readonly
Splits this string around matches of the given regular expression, and returns the result as a list of strings. The limit, if greater than 0, limits the number of elements in the return list, and only splits the string $limit times. Additionally, if the limit is set to zero, consecutive occurrences of the delimiter will split the string once, avoiding empty strings in the result list.
If `limit` is omitted, it defaults to 0.
Example:

```
log info "foo:and:boo".splitRegex(".(?=:)") #logs ["foo:", "and:", "boo"]
```

`replace: function(target : string, replacement : string) : string`
readonly
Returns this string after replacing all occurrences of the target string with the replacement string.
Example:

```
log info "foo:and:boo".replace(":",", ")  #logs "foo, and, boo"
```

`replaceRegex: function(target : string, replacement : string) : string`
readonly
Returns this string after replacing all matches of the target regex within this string with the replacement string.
Example:

```
log info "foo:and:boo".replaceRegex(":\w+",".")  #logs "foo.."
```

`replaceFirst: function(target : string, replacement : string) : string`
readonly
Returns this string after replacing the first match of the target regex within this string with the replacement string.
Example:

```
log info "foo:and:boo".replaceFirst(":\w+","")  #logs "foo:boo"
```

`toLowerCase: function() : string`
readonly
Returns this string after replacing all characters with their lowercase counterpart in the English locale.
Example:

```
log info "This is a String".toLowerCase()  #logs "this is a string"
```

`toUpperCase: function() : string`
readonly
Returns this string after replacing all characters with their uppercase counterpart in the English locale.
Example:

```
log info "This is a String".toUpperCase()  #logs "THIS IS A STRING"
```

`trim: function() : string`
readonly
Returns this string after replacing all whitespace characters at the beginning and the end of this string.
Example:

```
log info "   This is a String ".trim()  #logs "This is a String"
```

`startsWith: function(str : string) : boolean`
readonly
Tests if this string starts with the specified prefix.
Example:

```
log info "This is a String".startsWith("This")  #logs true
```

`endsWith: function(str : string) : boolean`
readonly
Tests if this string ends with the specified suffix.
Example:

```
log info "This is a String".endsWith("String")  #logs true
```

`contains: function(str : string) : boolean`
readonly
Tests if the given string exists as a substring of this string.
Example:

```
log info "setblock ~ ~ ~ stonecutter".contains("stone") #logs true
```

`matches: function(str : string) : boolean`
readonly
Tests if this string completely matches the given regular expression.
Example:

```
log info "minecraft:stone".matches("[a-z0-9_\\.-]+?:[a-z0-9_/\\.-]+")
#logs true
log info "mine/craft:stone".matches("[a-z0-9_\\.-]+?:[a-z0-9_/\\.-]+")
#logs false
```

`isEmpty: function() : boolean`
readonly
Returns true if this string contains no characters; that is, its length is zero.
Example:

```
log info "".isEmpty()  #logs true
```

`isWhitespace: function() : boolean`
readonly
Returns true if this string's first character is a whitespace character (space, tabulation, newline). See Java's Character.isWhitespace(char) method. Throws an exception if the string is empty.
Example:

```
log info " ".isWhitespace()  #logs true
```

`isDigit: function() : boolean`
readonly
Returns true if this string's first character is a digit character. See Java's Character.isDigit(char) method. Throws an exception if the string is empty.
Example:

```
log info "9".isDigit()  #logs true
```

`isLetter: function() : boolean`
readonly
Returns true if this string's first character is a letter character. See Java's Character.isLetter(char) method. Throws an exception if the string is empty.
Example:

```
log info "F".isLetter()  #logs true
```

`isLetterOrDigit: function() : boolean`
readonly
Returns true if this string's first character is a letter character or a digit character. See Java's Character.isLetterOrDigit(char) method. Throws an exception if the string is empty.
Example:

```
log info "F".isLetterOrDigit()  #logs true
```

Iterator

Using a for-each loop on a string will create single-character string entries for each character in the original string. The entries in the iterator will be in the order they appear in the string.
Example:

```
for(char in "ABC") {
    log info char
}
# logs:
# A
# B
# C
```

Conversions

- `nbt_value` & `tag_string` (explicit): A string tag with this string as its content.
- `entity` (explicit): Turns this entity into a fake player that can be used as an entity. The string may not be empty and must not contain whitespace.
- `resource` (explicit): Parses this string into a resource location. Must be a valid resource location.
- `text_component` (explicit): Turns this string into a string text component.

Integer Ranges

Identifier: `int_range`
Integer ranges consist of one or two integers, which represent all possible numbers between them. (In the case only one integer, the other defaults to positive or negative infinity). They can be created either via a integer range literal or the int_range constructor:

```
var range0 = int_range<5..10>
var range1 = int_range<5..>
var range2 = int_range<..10>
var range3 = new int_range(5, 10)      # same as range0
var range4 = new int_range(5, null)    # same as range1
var range4 = new int_range(null, 10)   # same as range2
```

Integer ranges can be used in interpolation blocks in commands, in places where an integer range is typically needed (such as selector arguments).

Constructor

`new int_range(min : int?, max : int?) : int_range`
Creates an integer range with the specified min and max bounds.
Example:

```
var range0 = new int_range(5, 10)
var range1 = new int_range(5, null)
var range2 = new int_range(null, 10)
```

Members

`min: int?`
readonly
Contains the lower bound of this range. Returns null if the lower bound is not specified.
Example:

```
log info int_range<5..10>.min  #logs 5
log info int_range<..10>.min   #logs null
```

`max: int?`
readonly
Contains the upper bound of this range. Returns null if the upper bound is not specified.
Example:

```
log info int_range<5..10>.max  #logs 10
log info int_range<5..>.max    #logs null
```

`range: int`
readonly
Contains the difference between the upper and lower bounds, as if null values represented min and max values of the integer data type. (Note: may overflow for ranges larger than the max int)
Example:

```
log info int_range<5..15>.range  #logs 10
log info int_range<5..>.range    #logs 2147483642
```

`deriveMin: function(newMin : int?) : int_range`
readonly
Returns a new range whose lower bound is the one specified.
Example:

```
log info int_range<5..10>.deriveMin(2)    #logs 2..10
log info int_range<5..10>.deriveMin(null) #logs ..10
```

`deriveMax: function(newMax : int?) : int_range`
readonly
Returns a new range whose upper bound is the one specified.
Example:

```
log info int_range<5..10>.deriveMax(20)    #logs 5..20
log info int_range<5..10>.deriveMax(null)  #logs 5..
```

Real Ranges

Identifier: `real_range`
Real ranges consist of one or two real numbers, which represent all possible numbers between them. (In the case only one real number, the other bound is undefined). They can be created either via a real range literal or the real_range constructor:

```
var range0 = real_range<5.5..10.2>
var range1 = real_range<5.5..>
var range2 = real_range<..10.2>
var range3 = new real_range(5.5, 10.2)    # same as range0
var range4 = new real_range(5.5, null)    # same as range1
var range5 = new real_range(null, 10.2)   # same as range2
```

Real ranges can be used in interpolation blocks in commands, in places where a real range is typically needed (such as the distance selector argument).

Constructor

`new real_range(min : real?, max : real?) : real_range`
Creates an integer range with the specified min and max bounds.
Example:

```
var range0 = new real_range(5.5, 10.2)
var range1 = new real_range(5.5, null)
var range2 = new real_range(null, 10.2)
```

Members

`min: real?`
readonly
Contains the lower bound of this range. Returns null if the lower bound is not specified.
Example:

```
log info real_range<5.5..10.2>.min  #logs 5.5
log info real_range<..10.2>.min     #logs null
```

`max: real?`
readonly
Contains the upper bound of this range. Returns null if the upper bound is not specified.
Example:

```
log info real_range<5.5..10.2>.max  #logs 10.2
log info real_range<5.5..>.max      #logs null
```

`range: int`
readonly
Contains the difference between the upper and lower bounds. This will be Infinity if either the upper or lower bounds are unspecified.
Example:

```
log info real_range<5.5..15.2>.range  #logs 9.7
log info real_range<5.5..>.range      #logs Infinity
```

`deriveMin: function(newMin : real?) : real_range`
readonly
Returns a new range whose lower bound is the one specified.
Example:

```
log info real_range<5.5..10.2>.deriveMin(2.5)    #logs 2.5..10.2
log info real_range<5.5..10.2>.deriveMin(null)   #logs ..10.2
```

`deriveMax: function(newMax : real?) : real_range`
readonly
Returns a new range whose upper bound is the one specified.
Example:

```
log info real_range<5.5..10.2>.deriveMax(20.5)   #logs 5.5..20.5
log info real_range<5.5..10.2>.deriveMax(null)   #logs 5.5..
```

Entities

Identifier: `entity`
Values of type entity represent ways to target an entity, either through a selector or a player name. They can be created via a entity literal:

```
var fakePlayer = entity<#RETURN>
var selector = entity<@e[type=cow,tag=onGround]>

kill $selector
```

Entity values can be used in interpolation blocks in commands, in places where an entity is typically needed.
NOTE: UUIDs are internally treated as player names by the following functions.

Members

`isPlayerName: function() : boolean`
readonly
Returns true if this entity representation is a player name (not a selector)
Example:

```
log info entity<#RETURN>.isPlayerName()  #logs true
log info entity<@a>.isPlayerName()       #logs false
```

`isPlayer: function() : boolean`
readonly
Returns whether all the entities represented by this object are guaranteed to be of type minecraft:player.
Example:

```
log info entity<#RETURN>.isPlayer()   #logs true
log info entity<@a>.isPlayer()        #logs true
log info entity<@e>.isPlayer()        #logs false
log info entity<@s>.isPlayer()        #logs false
```

`isUnknownType: function() : boolean`
readonly
Returns whether the type of the entities represented by this object cannot be known just from this object's values.
An example of this is the @s selector, where @s could mean any entity type, even if it only returns one entity.
Returns false if it expected to return more than one entity type.
Example:

```
log info entity<#RETURN>.isUnknownType()      #logs false
log info entity<@a>.isUnknownType()           #logs false
log info entity<@e>.isUnknownType()           #logs true
log info entity<@e[type=cow]>.isUnknownType() #logs false
log info entity<@s>.isUnknownType()           #logs true
```

`getLimit: function() : int`
readonly
Returns this entity's count limit; that is, how many entities, at maximum, this entity object represents. -1 if indeterminate.

Example:

```
log info entity<#RETURN>.getLimit()     #logs 1
log info entity<@a>.getLimit()           #logs -1
log info entity<@e[limit=5]>.getLimit()  #logs 5
log info entity<@s>.getLimit()           #logs 1
```

Blocks

Identifier: `block`
Values of type block group together a block type, blockstate and block data, which may represent the tiles at any position in a Minecraft world. They can be created via a block literal:

```
var block0 = block<minecraft:prismarine_wall>
var block1 = block<minecraft:prismarine_wall[east=true,up=false]>
var block2 = block<command_block{auto:1b,Command:"tp @a ~ ~1 ~"}>
var block3 = block<chest[facing=east]{Lock:"Key"}>
var block4 = block<#wool>

setblock ~ ~ ~ $block2
```

Block values can be used in interpolation blocks in commands, in places where a block is typically needed.

Constructor

`new block(type : resource?) : block`
Creates a block value with the specified block type. May throw an exception if the type is not a valid block type.
If type is omitted, it defaults to `minecraft:air`.
Example:

```
var wall  = new block(resource<minecraft:prismarine_wall>)
var walls = new block(resource<#minecraft:walls>)
var air   = new block()
```

Members

`blockType: resource`
get-set
Holds the type or ID of this block.
Example:

```
log info block<stone>.blockType  #logs minecraft:stone
log info block<#wool>.blockType  #logs #minecraft:wool
```

`blockState: dictionary?`
get-set
Holds a Dictionary object containing all the key-value pairs of this block's state (may be null).
Example:

```
log info block<stone>.blockState
#logs null
log info block<brick_wall[east=true,up=false]>.blockState
#logs {east: "true", up: "false"}
```

`blockTag: tag_compound?`
editable
Contains the tag compound containing this block's NBT data (may be null).
Example:

```
log info block<stone>.blockTag
#logs null
log info block<command_block{auto:1b,Command:"tp @a ~ ~1 ~"}>.blockTag
#logs {auto:1b,Command:"tp @a ~ ~1 ~"}
```

Items

Identifier: `item`
Values of type item group together an item type and item data, which may represent the item at any entity or container's slot in a Minecraft world. They can be created via an item literal, using explicit type and NBT, or a reference to a custom item type:

```
var item0 = item<minecraft:prismarine_wall>
var item1 = item<$someCustomItem>
var item2 = item<bow{Enchantments:[{id:"flame",lvl:1s}]}>
var item3 = item<#minecraft:music_discs>

give @a $item1
clear @a $item3
```

Item values can be used in interpolation blocks in commands, in places where an item is typically needed.

Constructor

`new item(type : resource?) : item`
Creates an item value with the specified item type. May throw an exception if the type is not a valid item type.
If type is omitted, it defaults to `minecraft:air`.
Example:

```
var stick   = new item(resource<minecraft:stick>)
var records = new item(resource<#minecraft:music_discs>)
var air     = new item()
```

Members

`itemType: resource`
get-set
Holds the type or ID of this item.
Example:

```
log info item<stick>.itemType  #logs minecraft:stick
log info item<#wool>.itemType  #logs #minecraft:wool
```

`itemTag: tag_compound?`
editable
Contains the tag compound containing this item's NBT data (may be null).
Example:

```
log info item<stick>.itemTag
#logs null
log info item<bow{display:{Name:'"PSG-1"'}}>.itemTag
#logs {display:{Name:'"PSG-1"'}}
```

`getSlotNBT: function() : tag_compound`
readonly
Returns an NBT compound representing this item in a slot in NBT.
Example:

```
log info item<bow{display:{Name:'"PSG-1"'}}>.getSlotNBT()
#logs {tag:{display:{Name:"\"PSG-1\""}},id:"minecraft:bow",Count:1b}
```

Text Components

Identifier: `text_component`
Values of type text component represent JSON text with formatting, which can be shown in-game. They can be created via a text component literal or the text_component constructor:

```
var text0 = text_component<"Winner">
var text1 = text_component<{"selector":"@s"}>
var text2 = text_component<["Winner: ",{"selector":"@s"}]>
var text3 = new text_component(["First Round's ", text2])

tellraw @a $text3
```

Item values can be used in interpolation blocks in commands or instructions, in places where an item is typically needed.

Constructor

`new text_component(object : *?, ignoreIncompatibleTypes : boolean?) : text_component`
Creates a block text component out of the given object, recursively. If `ignoreIncompatibleTypes` is false (or unset), an exception will be thrown if a value that cannot be converted to a text component is found.
Valid `object` types are: `string`, `boolean`, `dictionary`, `list`, `entity`, `resource`, `coordinate`, `pointer` or `text_component`. If null, defaults to an empty string.
If `ignoreIncompatibleTypes` is omitted, it defaults to false.
Example:

```
var text3 = new text_component(["First Round's ", text2, " (", pointer<@s -> score>, " points)"])

log info text3
#logs ["First Round's ",["Winner",{"selector":"@s"}]," (",{"score":{"name":"@s","objective":"score"}}," points)"]
```

NBT Values

Identifier: `nbt_value`
grouping type

Values of type NBT Value represent data storage elements for Minecraft's entities, items and blocks. There are many subtypes of nbt values.

Constructor

`new nbt(object : *?, ignoreIncompatibleTypes : boolean?) : nbt_value`
Creates an nbt value out of the given object, recursively. If `ignoreIncompatibleTypes` is false (or unset), an exception will be thrown if a value that cannot be converted to an NBT value is found.
Valid `object` types are: `int`, `real`, `string`, `boolean`, `resource`, `dictionary`, `list` or `nbt_value`. If null, defaults to an empty dictionary.
If `ignoreIncompatibleTypes` is omitted, it defaults to false.
Notes: Any nbt value passed to this constructor will be cloned.
Any integers passed to this constructor will be turned into `tag_int`
Any reals passed to this constructor will be turned into `tag_double`
Any booleans passed to this constructor will be turned into `tag_byte` (1b->true, 0b->false)
Example:

```
var nbt2 = new nbt({CustomModelData:120,Unbreakable:nbt_value<1b>})
var nbt3 = new nbt({Item:{id:resource<stone>,Count:nbt_value<3b>,tag:nbt2}})

log info nbt2    #logs {Unbreakable:1b,CustomModelData:120}
log info nbt3    #logs {Item:{id:"minecraft:stone",Count:3b,tag:{Unbreakable:1b,CustomModelData:120}}}
```

Tag Compounds

Identifier: `tag_compound`
alias: `nbt`
extends: `nbt_value`

Values of type Tag Compound represent key-value pairs, where keys are strings, and values are exclusively NBT values. They can be created via a nbt literal, nbt_value literal or the nbt constructor:

```
var nbt0 = nbt<{CustomModelData:120,Unbreakable:1b}>
var nbt1 = nbt_value<{CustomModelData:120,Unbreakable:1b}>
var nbt2 = new nbt({CustomModelData:120,Unbreakable:nbt_value<1b>})

give @a minecraft:carrot_on_a_stick$nbt0
```

Tag Compound values can be used in interpolation blocks in commands, in places where a tag compound is typically needed. They can also be used in item or block literals to add or merge the two NBT compounds.

Members

Indexer: `tag_compound[key : string] : nbt_value?`
readonly
Returns the nbt value associated with the specified key, or null if it doesn't exist.
Example:

```
log info nbt<{Age:0s,PickupDelay:10s,Tags:["a"]}>["Age"]  #logs 0s
```

`merge: function(other : tag_compound) : tag_compound`
readonly
Creates a new tag compound that contains elements of both this compound and the compound given by the parameter. In case of conflict between tags, the tags from the other compound overwrite those of this compound.
Example:

```
var nbt0 = nbt<{Age:0s,PickupDelay:10s,Tags:["a"]}>
var nbt1 = nbt<{PickupDelay:40s,Tags:["b"]}>

log info nbt0.merge(nbt1) #logs {Age:0s,PickupDelay:40s,Tags:["a","b"]}
```

`remove: function(key : string)`
readonly
Removes a value from the tag compound using the given key, if it exists.
Example:

```
var nbt0 = nbt<{Age:0s,PickupDelay:10s,Tags:["a"]}>

log info nbt0 #logs {Age:0s,PickupDelay:10s,Tags:["a"]}
eval nbt0.remove("Age")
log info nbt0 #logs {PickupDelay:10s,Tags:["a"]}
```

`toDictionary: function() : dictionary`
readonly
Creates a dictionary representation of this NBT tag tree and its nested values. This will convert all number tags into integers and reals, string tags into strings, list tags into lists, compounds into dictionaries and so on.
Example:

```
var nbt0 = nbt<{Age:0s,PickupDelay:40s,Tags:["a","b"]}>

log info nbt0.toDictionary()
# logs {PickupDelay: 40, Age:0, Tags: ["a", "b"]}
# Note that what is returned is of type dictionary and is editable, unlike tag_compound
```

Iterator

Using a for-each loop on a tag compound will create one value for each key-value pair. The iterator's values are dictionaries with two entries each: "key", which holds the tag's name, and "value", which holds the entry's value as an NBT tag. The entries in the iterator are not guaranteed to be in any particular order.
Modifying the dictionary during iteration is not allowed and will throw a concurrent modification exception.
Example:

```
var comp = nbt<{Age:0s,PickupDelay:40s,Tags:["a","b"]}>
for(entry in comp) {
    log info entry
}
# logs:
# {value: 0s, key: "Age"}
# {value: 40s, key: "PickupDelay"}
# {value: ["a","b"], key: "Tags"}
```

Tag Lists

Identifier: `tag_list`
extends: `nbt_value`

Values of type Tag List represent sequences of nbt_values **of the same type**. They can be created via a nbt_value literal or the nbt constructor:

```
var tagList0 = nbt_value<[0.0f,180.0f,90.0f]>
var tagList1 = new nbt(["a","b","new"])

summon minecraft:skeleton ~ ~ ~ {Tags:$tagList1}
```

Tag List values can be used in interpolation blocks in commands or instructions, in places where a tag compound is typically needed. They can also be used in item or block literals to add or merge the two NBT compounds.

Members

Indexer: `tag_list[index : int] : nbt_value`
readonly
Returns the nbt value at the specified index.
Example:

```
log info nbt_value<[0.0f,180.0f,90.0f]>[2]  #logs 90.0f
```

`length: int`
readonly
Contains the length of this tag_list; that is, the number of elements in this tag list.
Example:

```
log info nbt_value<["a","b","new","needs_id"]>.length  #logs 4
```

`merge: function(other : tag_list) : tag_list`
readonly
Creates a new tag list that contains elements of both this tag list and the one given by the parameter. Conflicting tag lists with the same type will be merged, with elements from the parameter tag list appearing after those of this tag list. In case of conflict between tag list content types (e.g. trying to merge a float tag_list and a string tag_list), an exception will be thrown.
Example:

```
var tagList0 = nbt_value<[0.0f,180.0f,90.0f]>
var tagList1 = new nbt(["a","b"])
var tagList2 = new nbt(["new","needs_id"])

log info tagList1.merge(tagList2) #logs ["a","b,"new","needs_id"]
log info tagList0.merge(tagList1)
#Error: Unable to merge tag lists: Incompatible types: TAG_String and TAG_Float
```

`toList: function() : list`
readonly
Creates a list representation of this NBT tag tree and its nested values. This will convert all number tags into integers and reals, string tags into strings, list tags into lists, compounds into dictionaries and so on.
Example:

```
var tagList0 =
nbt_value<[{},{},{id:"minecraft:leather_chestplate",Count:1b},{id:"minecraft:pumpkin",Count:1b}]>

log info tagList0.toList()
# logs [{}, {}, {id: "minecraft:leather_chestplate", Count: 1}, {id: "minecraft:pumpkin", Count: 1}]
# Note that what is returned is of type list and is editable, unlike tag_list
```

Iterator

Using a for-each loop on a tag list will yield each element in the list to the iterating variable, as an NBT tag. The entries in the iterator will be in the order they appear in the list.
Modifying the list during iteration is not allowed and will throw a concurrent modification exception.
Example:

```
var list = nbt_value<[1s,2s,4s,8s]>
for(entry in list) {
    log info entry
}
# logs:
# 1s
# 2s
# 4s
# 8s
```

Other NBT Tag Types

extends: `nbt_value`

1. `tag_byte` *constructor:* `new tag_byte(n : int)`
2. `tag_short` *constructor:* `new tag_short(n : int)`
3. `tag_int` *constructor:* `new tag_int(n : int)`
4. `tag_float` *constructor:* `new tag_float(n : real)`
5. `tag_double` *constructor:* `new tag_double(n : real)`
6. `tag_long` *constructor:* `new tag_long(v)` *where argument 0 is:* `int`, `real` *or* `string`
7. `tag_string` *constructor:* `new tag_string(str : string)`
8. `tag_byte_array` *constructor:* `new tag_byte_array(list : list)` *where list contains* `int`*s*
9. `tag_int_array` *constructor:* `new tag_int_array(list : list)` *where list contains* `int`*s*
10. `tag_long_array` *constructor:* `new tag_long_array(list : list)` *where list contains* `int`*s,* `real`*s or* `string`*s.*

As a reminder of the syntax, tag_byte_array, tag_int_array and tag_long_array can be created through the nbt_value literal, just like values of type tag_list, except adding a prefix before all the elements. Note that for each element in the array (excluding int arrays), the proper suffix must be added, or else it will be read as an integer, an incompatible type.
Example:

```
var byteArr = nbt_value<[B; 0b, 1b, 2b, 3b]>
var intArr  = nbt_value<[I; 0,  1,  2,  3 ]>
var longArr = nbt_value<[L; 0L, 1L, 2L, 3L]>
setblock ~ ~ ~ minecraft:campfire{CookingTime:$intArr}
```

Conversions

<any numeric NBT type> to `int` (explicit): Gets the truncated integer value of this NBT value.
<any numeric NBT type> to `real` (explicit): Gets the real value of this NBT value.
<any numeric array NBT type> to `list` (explicit): Retrieves all the tags of this array into a list of nbt values.

NBT Paths

Identifier: `nbt_path`
Values of type NBT Path represent routes through an NBT tree that point to one or many nbt value within it. They can be created via the nbt_path literal:

```
var path = nbt_path<Items[0].tag.display.Name>

data modify block ~ ~ ~ $path set value '"New Item"'
```

NBT Path values can be used in interpolation blocks in commands, in places where a path is typically needed.

Constructor

4 overloads:
`new nbt_path(key : string, compoundMatch : tag_compound?) : nbt_path`
Creates a NBT path root out of the given key, of the form `key{compoundMatch}` where the compound match is optional.
`new nbt_path(compoundMatch : tag_compound, wrapInList : boolean?) : nbt_path`
Creates a NBT path root out of the given compound, of the form `{compoundMatch}`. If wrapInList is true, it'll be wrapped in a list match, making it `[{compoundMatch}]`.
`new nbt_path(index : int) : nbt_path`
Creates a NBT path root out of the given list index, of the form `[index]`.
`new nbt_path() : nbt_path`
Creates a NBT path root of the form `[]`
Examples:

```
var path0 = new nbt_path("CustomModelData")
var path1 = new nbt_path("display",nbt<{Lore:[]}>)
var path2 = new nbt_path(nbt<{HurtTime:10s}>)
var path3 = new nbt_path(nbt<{HurtTime:10s}>, true)
var path4 = new nbt_path(2)
var path5 = new nbt_path()

# Paths 3, 4 and 5 are never used in practice, though still valid in-game

log info path0    #logs CustomModelData
log info path1    #logs display{Lore:[]}
log info path2    #logs {HurtTime:10s}
log info path3    #logs [{HurtTime:10s}]
log info path4    #logs [2]
log info path5    #logs []
```

Members

`resolveKey: function(key : string, compoundMatch : tag_compound?) : nbt_path`
readonly
Creates a new NBT path with the given key segment (and optionally, a compound match) added to the end.
Example:

```
log info nbt_path<RecordItem>.resolveKey("id") #logs RecordItem.id
log info nbt_path<RecordItem>.resolveKey("tag", nbt<{Damage:0s}>)
#logs RecordItem.tag{Damage:0s}
```

`resolveIndex: function(index : int) : nbt_path`
readonly
Creates a new NBT path with the given index segment added to the end.
Example:

```
log info nbt_path<Items>.resolveIndex(0) #logs Items[0]
```

`resolveListMatch: function(match : tag_compound) : nbt_path`
readonly
Creates a new NBT path with the given list match segment added to the end.
Example:

```
log info nbt_path<Items>.resolveListMatch(nbt<{id:"minecraft:snowball"}>)
#logs Items[{id:"minecraft:snowball"}]
log info nbt_path<Items>.resolveListMatch()
#logs Items[]
```

Coordinate Sets

Identifier: `coordinates`
Values of type Coordinate Set represent points in 3D space in a given coordinate system. They can be created via the coordinate literal:

```
var origin = coordinates<0.0 0.0 0.0> # parsed as 0.0 0.0 0.0
var pos0 = coordinates<0 0 0>         # parsed as 0.5 0.0 0.5
var pos1 = coordinates<~ ~-8 ~>       # parsed as ~ ~-8 ~
var pos2 = coordinates<^ ^ ^.1>       # parsed as ^ ^ ^0.1
var pos3 = coordinates<15 3 -7>       # parsed as 15 3 -7
var pos4 = coordinates<~ 255 ~>       # parsed as ~ 255 ~

setblock $origin jukebox         # setblock 0 0 0 minecraft:jukebox
setblock $pos0   jukebox         # setblock 0 0 0 minecraft:jukebox
summon pig $origin               # summon pig 0.0 0.0 0.0
summon pig $pos0                 # summon pig 0.5 0.0 0.5
spreadplayers $pos3 1 2 true @a  # spreadplayers 15.5 -6.5 1 2 true @a
```

Coordinate values can be used in interpolation blocks in commands, in places where a coordinate set is typically needed.
NOTE: Through the coordinate literal, the coordinates are parsed as if they were going to be used for an entity position. That means 0.5 is added to X and Z axes if they are absolute. Note that for every command that uses coordinates, the output is adjusted for each type of coordinate mode it uses.

Members

`getMagnitude: function(axis : trident-util:native@Axis) : real`
readonly
Retrieves the magnitude of the coordinate set along a given axis (in entity position mode). The axis parameter should be one of the X, Y and Z constants of the Axis class.
Example:

```
log info coordinates<0.0 1.0 2.0>.getMagnitude(Axis.Z) # logs 2.0
log info coordinates<0 1 2>.getMagnitude(Axis.X)       # logs 0.5
log info coordinates<~ ~1 2>.getMagnitude(Axis.Y)      # logs 1
```

`getCoordinateType: function(axis : trident-util:native@Axis) : trident-util:native@CoordinateType`
readonly
Retrieves the coordinate type of the coordinate along a given axis. The axis parameter should be one of the X, Y and Z constants of the Axis class. The returned value is one of the ABSOLUTE, RELATIVE and LOCAL constants of the CoordinateType class.
Example:

```
log info coordinates<~ ~ 2.0>.getCoordinateType(Axis.Z)
#logs CoordinateType.ABSOLUTE
log info coordinates<~ ~ 2.0>.getCoordinateType(Axis.Y)
#logs CoordinateType.RELATIVE
log info coordinates<^ ^ ^.1>.getCoordinateType(Axis.X)
#logs CoordinateType.LOCAL
```

`deriveMagnitude: function(newMagnitude : real, axis : trident-util:native@Axis?) : coordinates`
readonly
Creates a new coordinate set after changing one or all axes' magnitudes to the one given. The axis parameter is optional, and should be one of the X, Y and Z constants of the Axis class. If the axis is not specified, all axes' magnitudes will be changed; otherwise, only the specified axis is changed.
Example:

```
log info coordinates<~ ~ 2.0>.deriveMagnitude(1, Axis.X)
#logs ~1 ~ 2.0
log info coordinates<~ ~ 2.0>.deriveMagnitude(1)
#logs ~1 ~1 1.0
```

```
deriveCoordinateType:
function(
      newType : trident-util:native@CoordinateType,
      axis : trident-util:native@Axis?
) : coordinates
```

readonly
Creates a new coordinate set after changing one or all axes' coordinate types to the one given. The axis parameter is optional, and should be one of the X, Y and Z constants of the Axis class. Only the type of the axis specified will be changed to the new type, unless at least one of the following conditions is met:

1. The axis is not specified.
2. The specified coordinate type is CoordinateType.LOCAL
3. The coordinate type of the original coordinate set is CoordinateType.LOCAL

If one of these conditions is met, all axes will be changed to the given type, regardless of the axis specified.
Example:

```
log info coordinates<~ ~ 2.0>
         .deriveCoordinateType(CoordinateType.ABSOLUTE, Axis.X)
#logs 0.0 ~ 2.0
log info coordinates<1.0 2.0 3.0>
         .deriveCoordinateType(CoordinateType.RELATIVE, Axis.Y)
#logs 1.0 ~2 3.0
log info coordinates<1.0 2.0 3.0>
         .deriveCoordinateType(CoordinateType.RELATIVE)
#logs ~1 ~2 ~3
log info coordinates<1.0 2.0 3.0>
         .deriveCoordinateType(CoordinateType.LOCAL)
#logs ^1 ^2 ^3
log info coordinates<1.0 2.0 3.0>
         .deriveCoordinateType(CoordinateType.LOCAL, Axis.X)
#logs ^1 ^2 ^3
log info coordinates<^1 ^2 ^3>
         .deriveCoordinateType(CoordinateType.RELATIVE, Axis.X)
#logs ~1 ~2 ~3
```

Rotations

Identifier: `rotation`
Values of type Rotation represent an Euler rotation formed by yaw (rotation along the Y axis) and pitch (rotation along the X axis). They can be created via the coordinate literal:

```
var origin = rotation<0 0>
var rot0 = rotation<45 90>
var rot1 = rotation<~ 0>
var rot2 = rotation<~180 ~12.5>

tp @s ~ ~ ~ $origin              # tp @s ~ ~ ~ 0 0
tp @s ~ ~ ~ $rot0                # tp @s ~ ~ ~ 45 90
rotated $rot1 tp @s ^ ^ ^1 $rot2 # execute rotated ~ 0 run tp @s ^ ^ ^1 ~180 ~12.5
```

Rotation values can be used in interpolation blocks in commands, in places where a rotation is typically needed.

Members

`getMagnitude: function(axis : trident-util:native@Axis) : real`
readonly
Retrieves the magnitude of the coordinate set along a given axis (in entity position mode). The axis parameter should be one of the X and Y constants of the Axis class. Remember that rotations are written in YX order.
Example:

```
log info rotation<~180 ~12.5>.getMagnitude(Axis.X)     # logs 12.5
log info rotation<0 90>.getMagnitude(Axis.Y)           # logs 0
```

`getRotationType: function(axis : trident-util:native@Axis) : trident-util:native@RotationType`
readonly
Retrieves the rotation type of the rotation along a given axis. The axis parameter should be one of the X and Y constants of the Axis class. The returned value is one of the ABSOLUTE and RELATIVE constants of the RotationType class.
Example:

```
log info rotation<~ 90>.getCoordinateType(Axis.X)
#logs RotationType.ABSOLUTE
log info rotation<~ 90>.getCoordinateType(Axis.Y)
#logs RotationType.RELATIVE
```

`deriveMagnitude: function(newMagnitude : real, axis : trident-util:native@Axis?) : rotation`
readonly
Creates a new rotation after changing one or both axes' magnitudes to the one given. The axis parameter is optional, and should be one of the X and Y constants of the Axis class. If the axis is not specified, both Y and X axes' magnitudes will be changed; otherwise, only the specified axis is changed.
Example:

```
log info rotation<~ 90>.deriveMagnitude(-90, Axis.X)
#logs ~ -90
log info rotation<~ 90>.deriveMagnitude(-45)
#logs ~-45 -45
```

```
deriveRotationType:
function(
      newType : trident-util:native@RotationType,
      axis : trident-util:native@Axis?
) : rotation
```

readonly
Creates a new rotation after changing one or both axes' rotation types to the one given. The axis parameter is optional, and should be one of the X and Y constants of the Axis class. If the axis is not specified, both Y and X axes' types will be changed; otherwise, only the specified axis is changed.
Example:

```
log info rotation<~ 90>.deriveRotationType(CoordinateType.ABSOLUTE, Axis.X)
#logs 0 -90
log info rotation<0 90>.deriveRotationType(CoordinateType.RELATIVE)
#logs ~ ~90
```

Resource Locations

Identifier: `resource`
Values of type Resource Location represent locations in data packs or resource packs where any asset or resource may be located within a namespace. They can be created via the resource literal, or cast from a string. If a namespace is not specified, it defaults to the minecraft namespace. Can be prefixed with a # symbol to denote a tag. They also support Relative Resource Locations.

```
# file: mypack:demo/resources

var res0 = resource<tick>                 # contains: minecraft:tick
var res1 = resource<mypack:tick>          # contains: mypack:tick
var res2 = resource<#wool>                # contains: #minecraft:wool
var res3 = resource<#mypack:load>         # contains: #mypack:load
var res4 = (resource) "mypack:util/id"    # contains: mypack:util/id
var res5 = resource<mypack:tile.step>     # contains: mypack:tile.step
var res6 = resource</>                    # contains: mypack:demo/resources
var res7 = resource</a>                   # contains: mypack:demo/resources/a

playsound $res5 master @a
define function $res4 {
    # ...
}
function $res4
```

Resource Locations can be used in interpolation blocks in commands and instructions, in places where a resource location is typically needed.

Constructor

`new resource(whole : string) : resource`
Assembles a resource location entirely out of the given string.
`new resource(namespace : string, parts : list, delimiter : string?) : resource`
Assembles a resource location out of the given namespace and the specified parts, separated by a delimiter (by default, the forward slash `"/"`).
The parts may be strings or resource locations - in the latter case, only the body of the resource location is used in the output resource location.
Examples:

```
log info new resource("#mypack:utils/random")
# logs #mypack:utils/random

log info new resource("mypack", ["utils", "random", "get"])
# logs mypack:utils/random/get

var rain = new resource("minecraft", ["weather", "rain"], ".")
log info rain
# logs minecraft:weather.rain

log info new resource("minecraft", [rain, "above"], ".")
# logs minecraft:weather.rain.above
```

Members

Indexer: `resource[index : int] : string`
readonly
Returns the string which represents the name at the specified position of the body.
Example:

```
log info resource<mypack:util/random/get>[0]   # logs util
log info resource<mypack:util/random/get>[1]   # logs random
log info resource<mypack:util/random/get>[2]   # logs get
```

`namespace : string`
readonly
Contains the namespace of this resource.
Example:

```
log info resource<tick>.namespace              # logs "minecraft"
log info resource<#minecraft:tick>.namespace   # logs "minecraft"
log info resource<mypack:tick>.namespace       # logs "mypack"
```

`body : string`
readonly
Contains the body of this resource.
Example:

```
log info resource<ui.toast.in>.body       # logs "ui.toast.in"
log info resource<minecraft:load>.body    # logs "load"
log info resource<mypack:tiles/step>.body # logs "tiles/step"
```

`isTag : boolean`
readonly
Contains whether this resource is tagged.
Example:

```
log info resource<ui.toast.in>.isTag       # logs false
log info resource<#minecraft:load>.isTag   # logs true
```

nameCount : int
readonly
Retrieves the number of "names" in this resource location - that is, the number of parts separated by forward slashes in the body.
Example:

```
log info resource<mypack:util/random/get>.nameCount   # logs 3
log info resource<mypack:tiles/step>.nameCount        # logs 2
log info resource<ui.toast.in>.nameCount              # logs 1
```

parent: string?
readonly
Retrieves the resource location that is the parent of this resource location. That is, the resource location that has the same namespace and isTag as this one, and its body is made from all the "names" from this location excluding the last one. If there is no parent to this resource location, this member is `null`.
Example:

```
log info resource<mypack:util/random/get>.parent  # logs mypack:util/random
log info resource<ui.toast.in>.parent             # logs null
```

subLoc: function(beginIndex : int, endIndex : int) : resource
readonly
Returns a similar resource location whose body contains only a specified range of names, such that the resulting resource location's nameCount will be `endIndex - beginIndex`.
The names that will be kept will range from beginIndex (inclusive) to endIndex (exclusive)
Example:

```
log info resource<mypack:util/rand/get>.subLoc(1, 3) # logs mypack:rand\get
log info resource<mypack:util/rand/get>.subLoc(0, 2) # logs mypack:util\rand
```

resolve: function(other : string, delimiter : string?) : resource
readonly
Returns a similar resource location, with the specified name appended to the end. The `other` parameter may be a string or a resource location - in the latter case, only the location's body will be added to the end. The delimiter used to append the resource location will be, by default, the forward slash `"/"`, unless specified by the second optional argument.
Example:

```
log info resource<mypack:util>.resolve("rand")
# logs mypack:util/rand
log info resource<mypack:util>.resolve("rand", ".")
# logs mypack:util.rand

log info resource<mypack:util>.resolve(resource<rand>)
# logs mypack:util/rand
log info resource<mypack:util>.resolve(resource<rand>, ".")
# logs mypack:util.rand
```

Iterator

Using a for-each loop on a resource location will create string entries for names in the resource location body. The entries in the iterator will be in the order they appear in the resource location.
Example:

```
for(name in resource<mypack:util/random/get>) {
    log info name
}
# logs:
# util
# random
# get
```

Pointers

Identifier: pointer
Values of type Pointer represent places where data is typically stored in-game. They have four properties:

1. Target: The block, entity or resource location that holds the information
2. Member: The place they hold the information; either a scoreboard objective (entities only) or an NBT path.
3. Scale: Used in the set custom command. Stores how much to scale the value when getting or setting from this pointer. May only exist for NBT path Members.
4. Numeric type: Used in the set custom command, when the member is an NBT path. Specifies what the data type of the NBT tag is. If unspecified, Trident will attempt to determine the data type.

They can be created using the pointer literal.
Block targets should be
The separator between the target and the member depends on the type of target and member; score members will require an arrow `->`, NBT paths require a dot `.`, and storage requires a tilde `~`.

```
var ptr0 = pointer<#RETURN->global>    # player #RETURN, score global
var ptr1 = pointer<@s->id>             # sender, score id
var ptr2 = pointer<custom:storage~Tmp (double)>
# storage "custom:storage", Tmp
var ptr3 = pointer<@s.Pos[1]>          # sender, tag Pos[1], double (guess)
var ptr4 = pointer<(~ ~-1 ~).auto (byte)>
# block at ~ ~-1 ~, tag auto, byte
var ptr5 = pointer<@p.Motion[0] * 0.1 (double)>
# nearest player, tag Motion[0], scale 0.1, double

set $ptr1 = $ptr0
# scoreboard players operation @s id = #RETURN global
set $ptr2 = $ptr5
# execute store result storage custom:storage Tmp double 1 run data get entity @p Motion[0] 0.1
scoreboard players operation deref $ptr1 += deref $ptr0
# scoreboard players operation @s id += #RETURN global
```

Pointers can be used in interpolation blocks in the set command, as well as the scoreboard command by using the deref keyword in place of the player name, and an interpolation block to the pointer in place of the objective name.

Note: The target and member properties of pointer objects can hold any data type; regardless of whether they make up a valid pointer. However, if an invalid pointer is used on any command or instruction that uses pointers, an error will be thrown.

Members

target : *
editable
Contains the target of this pointer. Becomes an illegal pointer if it's anything other than an entity, a coordinate set or a resource location.
Example:

```
log info pointer<#RETURN->global>.target # logs #RETURN (type entity)
log info pointer<@s->id>.target          # logs @s (type entity)
log info pointer<custom:storage~Pos>.target
# logs custom:storage (type resource)
log info pointer<(~ ~ ~1).auto>.target   # logs ~ ~ ~1 (type coordinates)
```

member : *
editable
Contains the target of this pointer. Becomes an illegal pointer if it's anything other than a string or an NBT path, if the string is an invalid objective name, or if the target is a coordinate set or resource while the member is a string.
Example:

```
log info pointer<@s->global>.member       #logs "global"
log info pointer<@s.Pos[1]>.member        #logs Pos[1]
```

scale: real
get-set
Contains the scale of this pointer. It is 1 by default.
Example:

```
log info pointer<@s->global>.scale             #logs 1
log info pointer<@s.Pos[1] * 100>.scale        #logs 100
log info pointer<@s.Pos[1] * 0.01>.scale       #logs 0.01
```

numericType : string
get-set
Contains the numeric type of this pointer. null by default.
Must be one of: null, "byte", "short", "int", "float", "double", "long"
If attempting to set to an invalid value, no changes will be made to this field.
Example:

```
log info pointer<@s.Pos[1] * 100>.numericType           #logs null
log info pointer<@s.Pos[1] * 100 (double)>.numericType  #logs double
log info pointer<@s.FallDistance (float)>.numericType   #logs float
```

Dictionaries

Identifier: dictionary
Dictionaries are collections of named values. These contain key-value pairs where keys are strings, and values are any type of interpolation value. They can be created via a dictionary literal:

```
var MapProperties = {
    name: "The Aether II Map",
    version: "1.1.2",
    final: true,
    idle: resource<aether:aemob.moa.say>,
    mainFunc: resource</main>,
    "Not an identifier": entity<@s>,
    nested: {
        level: 2
    }
}

log info MapProperties.name            # logs "The Aether II Map"
log info MapProperties.nested.level    # logs 2
```

Dictionaries are particularly useful to store large amounts of related information together.

Members

Note: This data type features dynamic members (user-made member keys), as well as static members (those given by the language). User-made members with the same name as language-provided members are valid, but they will hide the language-defined member for that object.
Example:

```
var dict0 = {}
var dict1 = {map: "overwritten"}

log info dict0.map
# logs <internal function>
log info dict1.map
# logs "overwritten"
```

Indexer: dictionary[key : string] : *?
variable
Returns the value associated with the specified key.
Example:

```
var MapProperties = {
    idle: resource<aether:aemob.moa.say>,
    mainFunc: resource</main>,
    "Not an identifier": entity<@s>
}
log info MapProperties["idle"]              # logs aether:aemob.moa.say
log info MapProperties["Not an identifier"] # logs @s
eval MapProperties["new"] = true            # sets property "new" to true
```

*** : *?**
variable
Returns the value associated with the specified key. This can only access values named as identifiers; to access values with keys of any string, use the accessor instead.
New entries can be added and changed from the dictionary, using the assignment operator, as well as through the dictionary literal upon creation.
Example:

```
var MapProperties = {
    name: "The Aether II Map",
    version: "1.1.2",
    final: true
}
log info MapProperties.name              # logs "The Aether II Map"
log info MapProperties.version           # logs "1.1.2"
log info MapProperties.final             # logs true
log info MapProperties.unset             # logs null
eval MapProperties.newProp = entity<@e>  # sets property "newProp" to @e
```

merge: function(other : dictionary) : dictionary
readonly
Creates a new dictionary that contains elements of both this dictionary and the dictionary given by the parameter. In case of conflict between keys, the values from the other dictionary overwrite those of this dictionary.
Example:

```
var dict0 = {
    a: "Original A",
    b: "Original B"
}
var dict1 = {
    b: "New B",
    c: "New C"
}

log info dict0.merge(dict1)
# logs {a: "Original A", b: "New B", c: "New C"}
```

remove: function(key : string)
readonly
Removes the key-value pair with the given value, if it exists. If it doesn't exist, no action is taken.
Example:

```
var dict0 = {
    a: "Original A",
    b: "Original B"
}

log info dict0
# logs {a: "Original A", b: "Original B"}
eval dict0.remove("b")
log info dict0
# logs {a: "Original A"}
```

hasOwnProperty: function(key : string)
readonly
Returns whether the dictionary has a value associated with the given key. Does not include native functions.
Example:

```
var dict0 = {
    a: "A",
    b: "B"
}

log info dict0.hasOwnProperty("a") # logs true
log info dict0.hasOwnProperty("b") # logs true
log info dict0.hasOwnProperty("c") # logs false
log info dict0.hasOwnProperty("hasOwnProperty") # logs false
```

clear: function()
readonly
Removes all elements from this dictionary.
Example:

```
var dict0 = {
    a: "Original A",
    b: "Original B"
}

log info dict0
# logs {a: "Original A", b: "Original B"}
eval dict0.clear()
log info dict0
# logs {}
```

map: function(filter : function[(key : string, value : *?) : *?]) : dictionary
readonly
Creates a new dictionary with the same keys as this one, but with the values given by the filter function passed to this function. For each key-value pair, the filter function should take as its first parameter, the key, and as its second parameter, the original value. Whatever is returned by the function at each key-value pair will be used as the value for that key in the new dictionary.
Example:

```
var dict0 = {
    a: "Original A",
    b: "Original B",
    c: "Original C"
}

log info dict0.map(function(key, value) {
    if(key != "c") {
        return value.replace("Original", "New")
    } else {
        return value
    }

})
# logs {a: "New A", b: "New B", c: "Original C"}
```

Iterator

Using a for-each loop on a dictionary will create one value for each key-value pair. The iterator's values are dictionaries with two entries each: "key", which holds the entry's key, and "value", which holds the entry's value. The entries in the iterator are not guaranteed to be in any particular order.
Example:

```
var dict0 = {
    a: "Original A",
    b: "Original B",
    c: "Original C"
}

for(entry in dict0) {
    log info entry
}
# logs:
# {value: "Original A", key: "a"}
# {value: "Original B", key: "b"}
# {value: "Original C", key: "c"}
```

Lists

Identifier: list
Lists are ordered collections of interpolation values of any type. They can be created via a list literal using square braces:

```
var list0 = [
    "A String",
    true,
    resource<minecraft:weather.rain>,
    entity<@s>,
    {min: 0, max: 100},
    ["another", "list"]
]
```

Members

Indexer: list[index : int] : *?
variable
Returns the value located at the specified index.
Example:

```
var list0 = [
    "A String",
    true,
    resource<minecraft:weather.rain>,
    entity<@s>,
    {min: 0, max: 100},
    ["another", "list"]
]

log info list0[0] # logs "A String"
log info list0[2] # logs minecraft:weather.rain
log info list0[4] # logs {min: 0, max: 100}
log info list0[5] # logs ["another", "list"]
```

length : int
readonly
Contains the length of this list; that is, the number of elements in it.
Example:

```
log info ["a","b","new","needs_id"].length  # logs 4
```

add: function(elem : *?)
readonly
Appends the given element to the end of this list.
Example:

```
var list0 = [
    "A",
    "B"
]

log info list0         # logs ["A", "B"]
eval list0.add("C")
log info list0         # logs ["A", "B", "C"]
```

insert: function(elem : *?, index : int)
readonly
Inserts the given element at the specified index, shifting all elements previously at that index of higher, one index up.
Example:

```
var list0 = [
    "A",
    "C",
    "D"
]

log info list0               # logs ["A", "C", "D"]
eval list0.insert("B", 1)
log info list0               # logs ["A", "B", "C", "D"]
```

remove: function(index : int)
readonly
Removes the element at the specified index.
Example:

```
var list0 = [
    "A",
    "B",
    "C",
    "D"
]

log info list0          # logs ["A", "B", "C", "D"]
eval list0.remove(2)
log info list0          # logs ["A", "B", "D"]
```

clear: function()
readonly
Removes all elements from this list.
Example:

```
var list0 = [
    "A",
    "B",
    "C"
]

log info list0        # logs ["A", "B", "C", "D"]
eval list0.clear()
log info list0        # logs []
```

contains: function(value : *?)
readonly
Returns true if the given value exists in the list, false otherwise.
Example:

```
var list0 = [
    "Original A",
    "Original B",
    "Original C"
]

log info list0.contains("Original A")    # logs true
log info list0.contains("Original D")    # logs false
```

indexOf: function(value : *?) : int
readonly
Returns the index at which the specified value first occurs in the list. Returns -1 if the list does not contain it.
Example:

```
var list0 = [
    "Original A",
    "Original B",
    "Repeat",
    "Repeat"
]

log info list0.indexOf("Repeat")        # logs 2
log info list0.indexOf("Original C")    # logs -1
```

lastIndexOf: function(value : *?) : int
readonly
Returns the index at which the specified value last occurs in the list. Returns -1 if the list does not contain it.
Example:

```
var list0 = [
    "Original A",
    "Original B",
    "Repeat",
    "Repeat"
]

log info list0.lastIndexOf("Repeat")        # logs 3
log info list0.lastIndexOf("Original C")    # logs -1
```

isEmpty: function() : boolean
readonly
Returns true if the list contains no elements, false otherwise.
Example:

```
var list0 = [
    "Original A",
    "Original B",
    "Repeat",
    "Repeat"
]

log info list0.isEmpty()    # logs false
log info [].isEmpty()       # logs true
```

map: function(filter : function[(value : *?, index : int) : *?]) : list
readonly
Creates a new list, where each element corresponds to one of the original list, after going through a filter. The filter function should take as its first parameter the original value, as second parameter, the index it resides in, and return its corresponding value in the new list.
Example:

```
var list0 = [
    "Original A",
    "Original B",
    "Original C"
]

log info list0.map(function(value, index) {
    return value.replace("Original", "New at " + index + ":")
})
# logs ["New at 0: A", "New at 1: B", "New at 2: C"]
```

filter: function(filter : function[(value : *?, index : int) : boolean]) : list
readonly
Creates a duplicate of this list, keeping only elements that, when run through the filter function, return true. The filter function should take as its first parameter the original value, as second parameter, the index it resides in, and return true for elements that should be kept in the new list.

Example:

```
var list0 = [
    "Original A",
    "Original B",
    "New C",
    "New D"
]

log info list0.filter(function(value, index) {
    return value.startsWith("Original")
})
# logs ["Original A", "Original B"]
```

reduce:
function(function : function[(total : *?, currentValue : *?, index : int) : *?], initialValue : *?) : *?
readonly
Reduces all of the values of the list into a single value, using the given function and starting with the given initial value.

Example:

```
var list0 = [1, 2, 3, 4, 5]

log info list0.reduce(function(total, currentValue, index) {
    return total + currentValue
})
# logs 15
```

Iterator

Using a for-each loop on a list will yield each element in the list to the iterating variable. The entries in the iterator will be in the order they appear in the list.
Example:

```
var list0 = [
    "A",
    "B",
    "C",
    "D",
    "E"
]

for(entry in list0) {
    log info entry
}
# logs:
# "A"
# "B"
# "C"
# "D"
# "E"
```

Custom Entities

Identifier: `custom_entity`
Values of type custom_entity represent blueprints for an user-defined non-default entity or entity component. These typically contain the values defined in their body. They can only be created via the define entity instruction. More it the Custom Entities section.

```
define entity bear minecraft:polar_bear {
    var idleSound = resource<minecraft:entity.polar_bear.ambient>

    ticking function tick {
        # ...
    }
}

summon $bear
```

Members

Note: This data type features dynamic members (user-made member keys), as well as static members (those given by the language). User-made members with the same name as language-provided members are valid, but they will hide the language-defined member for that object.

Indexer: `custom_entity[key : string] : *?`
variable
Returns the value associated with the specified key.
Note: New properties can't be added through the accessor; only properties that have been declared inside the entity body can be edited.
Example:

```
# file: mypack:entities/bear.tdn

define entity bear minecraft:polar_bear {
    var idleSound = resource<minecraft:entity.polar_bear.ambient>

    ticking function tick {
        # ...
    }

    function animation/hurt {
        # ...
    }
}

log info bear["idleSound"]       # logs minecraft:entity.polar_bear.ambient
log info bear["tick"]            # logs mypack:entities/bear/tick
log info bear["animation/hurt"]  # logs mypack:entities/bear/animation/hurt
```

`* : *?`
variable
Returns the value associated with the specified key. This can only access values named as identifiers; to access values with keys of any string, use the accessor instead.
Note: New properties can't be added through dot notation; only properties that have been declared inside the entity body can be edited.
Example:

```
# file: mypack:entities/bear.tdn

define entity bear minecraft:polar_bear {
    var idleSound = resource<minecraft:entity.polar_bear.ambient>

    ticking function tick {
        # ...
    }

    function animation/hurt {
        # ...
    }
}

log info bear.idleSound         # logs minecraft:entity.polar_bear.ambient
log info bear.tick              # logs mypack:entities/bear/tick
```

`idTag : string`
readonly
Contains the tag that identifies entities of this type or component. This is added to the Tags tag_list of the entity when created, or added through the custom component command.
Example:

```
# file: mypack:entities.tdn

define entity bear minecraft:polar_bear {}
define entity component animatable {}

log info bear.idTag         # logs trident-entity.mypack.bear
log info animatable.idTag   # logs trident-component.mypack.animatable
```

`baseType : resource?`
readonly
Contains the resource location representing the base entity type of this entity. This is always null for entity components.
Example:

```
# file: mypack:entities.tdn

define entity bear minecraft:polar_bear {}
define entity component animatable {}

log info bear.baseType         # logs minecraft:polar_bear
log info animatable.baseType   # logs null
```

`getSettingNBT: function() : tag_compound`
readonly
Returns a tag_compound that can be used to create an entity of this type, or an entity with this component.
Example:

```
# file: mypack:entities.tdn

define entity bear minecraft:polar_bear {
    default nbt {Glowing:1b}
}
define entity component invisible {
    default nbt {ActiveEffects:[{Id:14b,Amplifier:0b,Duration:100000}]}
}

log info bear.getSettingNBT()
# logs {Tags:["trident-entity.mypack.bear"],Glowing:1b,id:"minecraft:polar_bear"}
log info invisible.getSettingNBT()
# logs
{Tags:["trident-component.mypack.invisible"],ActiveEffects:[{Id:14b,Amplifier:0b,Duration:100000}]}
```

`getMatchingNBT: function() : tag_compound`
readonly
Returns a tag_compound that can be used to match an entity of this type, or an entity with this component, using its idTag.
Example:

```
# file: mypack:entities.tdn

define entity bear minecraft:polar_bear {
    default nbt {Glowing:1b}
}
define entity component invisible {
    default nbt {ActiveEffects:[{Id:14b,Amplifier:0b,Duration:100000}]}
}

log info bear.getMatchingNBT()
# logs {Tags:["trident-entity.mypack.bear"]}
log info invisible.getMatchingNBT()
# logs {Tags:["trident-component.mypack.invisible"]}
```

Custom Items

Identifier: `custom_entity`
Values of type custom_item represent blueprints for an user-defined non-default item. These typically contain the values defined in their body. They can only be created via the define item instruction. More at the Custom Items section.

```
define item staff minecraft:snowball#3 {
    default nbt {Enchantments:[{}]}

    var useSound = resource<minecraft:item.totem.use>

    on used function use {
        playsound ${this.useSound} master @a
    }
}

give @a $staff
```

Members

Note: This data type features dynamic members (user-made member keys), as well as static members (those given by the language). User-made members with the same name as language-provided members are valid, but they will hide the language-defined member for that object.

Indexer: `custom_item[key : string] : *?`
variable
Returns the value associated with the specified key.
Note: New properties can't be added through the accessor; only properties that have been declared inside the item body can be edited.
Example:

```
# file: mypack:items/staff.tdn

define item staff minecraft:snowball#3 {
    default nbt {Enchantments:[{}]}

    var useSound = resource<minecraft:item.totem.use>

    on used function use {
        playsound ${this.useSound} master @a
    }

    on dropped function other/dropped {
        # ...
    }
}

log info staff["useSound"]       # logs minecraft:item.totem.use
log info staff["use"]            # logs mypack:items/staff/use
log info staff["other/dropped"]  # logs mypack:items/staff/other/dropped
```

`* : *?`
variable
Returns the value associated with the specified key. This can only access values named as identifiers; to access values with keys of any string, use the accessor instead.
Note: New properties can't be added through dot notation; only properties that have been declared inside the item body can be edited.
Example:

```
# file: mypack:items/staff.tdn

define item staff minecraft:snowball#3 {
    default nbt {Enchantments:[{}]}

    var useSound = resource<minecraft:item.totem.use>

    on used function use {
        playsound ${this.useSound} master @a
    }

    on dropped function other/dropped {
        # ...
    }
}

log info staff.useSound       # logs minecraft:item.totem.use
log info staff.use            # logs mypack:items/staff/use
```

`itemCode : int`
readonly
Contains the int used to identify this custom Trident item from those of a different type. This is added to a custom TridentCustomItem tag_int to the item tag upon creation or for testing.
Example:

```
# file: mypack:items/staff.tdn

define item staff minecraft:snowball#3 {}

log info staff.itemCode       # logs 1721657579
```

`baseType : resource`
readonly
Contains the resource location representing the base item type of this item.
Example:

```
# file: mypack:items/staff.tdn

define item staff minecraft:snowball#3 {}

log info staff.baseType       # logs minecraft:snowball
```

`getSlotNBT: function() : tag_compound`
readonly
Returns a tag_compound that can be used to create an item of this type. This includes the id, Count and tag tags of a typical item compound.
Example:

```
# file: mypack:items/staff.tdn

define item staff minecraft:snowball#3 {
    default nbt {Enchantments:[{}]}}
}

log info staff.getSlotNBT()
# logs
{tag:{TridentCustomItem:1721657579,CustomModelData:3,Enchantments:[{}]},id:"minecraft:snowball",Count:1b}
```

`getItemTag: function() : tag_compound`
readonly
Returns the item tag of this item, containing all its initial data. This does not include the base ID of the item.
Example:

```
# file: mypack:items/staff.tdn

define item staff minecraft:snowball#3 {
    default nbt {Enchantments:[{}]}}
}

log info staff.getItemTag()
# logs {TridentCustomItem:1721657579,CustomModelData:3,Enchantments:[{}]}
```

`getMatchingNBT: function() : tag_compound`
readonly
Returns a tag_compound that can be used to match an item of this type using its itemCode.
Example:

```
# file: mypack:items/staff.tdn

define item staff minecraft:snowball#3 {
    default nbt {Enchantments:[{}]}}
}

log info staff.getMatchingNBT()
# logs {TridentCustomItem:1721657579}
```

UUIDs

Identifier: `uuid`
Values of type UUID represent an immutable universally unique identifier. A UUID represents a 128-bit value, and is commonly used in Minecraft for identifying entities and attribute modifiers. They can be created via the uuid literal or constructor:

```
var id0 = uuid<0-0-0-0-0>
var id1 = uuid<1-2-3-4-5>

attribute @s generic.armor modifier add $id0 myAttribute0 5 add
attribute @s generic.armor modifier add $id1 myAttribute1 2 multiply
```

Constructors

`new uuid() : uuid`
Generates a random UUID using the `trident-util:native@Random.PROJECT_RANDOM` object, meaning it will be consistent between compilations yet unique for your project. UUIDs generated this way will be version 4 UUIDs that conform to RFC 4122.
`new uuid(raw : string) : uuid`
Parses the given string and creates a UUID out of it.
`new uuid(random : trident-util:native@Random) : uuid`
Creates a random UUID using the next 4 ints returned by the nextInt method of the given Random object. UUIDs generated this way will be version 4 UUIDs that conform to RFC 4122.
Examples:

```
log info new uuid("0-0-0-0-0")  # Fixed
log info new uuid("1-2-3-4-5")  # Fixed
log info new uuid()             # Random, consistent across compilations
log info new uuid(new Random()) # Random, different every compilation
```

Conversions

`nbt_value` & `tag_int_array` (explicit): Creates an array with 4 ints corresponding to this UUID. Example:

```
log info (tag_int_array) new uuid("1-2-3-4-5")
# logs [I;1,131075,262144,5]
```

Functions

Identifier: `function`
These values represent dynamic functions that can take parameters, return a value, and insert commands into the file currently being written. They can be called using the function name followed by parentheses.
Each parameter may or may not be constrained to a type, and the return value may also be constrained by placing the constraint after the parameter list and before the opening brace.
Example:

```
var rand = function(min : int, max : int) : pointer {
    function $generate_random
    set RANDOM_TEMP->global = ${max - min}
    scoreboard players operation RANDOM global %= RANDOM_TEMP global
    scoreboard players add RANDOM global $min
    return pointer<RANDOM->global>
}

scoreboard players operation @s offset += deref ${rand(-16, 17)}

# Result:
#  rand() call begin
function stt:utils/generate_random
scoreboard players set RANDOM_TEMP global 33
scoreboard players operation RANDOM global %= RANDOM_TEMP global
scoreboard players add RANDOM global 16
#  rand() call end
scoreboard players operation @s offset += RANDOM global
```

Parameters can be passed to a function by name, which allows them to be passed out of order. Here's an example:

```
var testFunction = function(a : int, b : int) {
    log info "a: " + a
    log info "b: " + b
}

eval testFunction(b: 1, a: 2)
# Logs:
#    a: 2
#    b: 1
```

Members

`formalParameters : list`
get
Retrieves a list of the parameter names defined for this function. Will be an empty list if the function does not define any parameters OR if the function is language-provided.
Example:

```
log info function(min, max : int) {}.formalParameters
# logs [{nullable: true, name: "min", type: null}, {nullable: false, name: "max", type:
type_definition<int>}]
```

`declaringFile : resource?`
get
Retrieves a resource location specifying the static function it was declared from. Will always be null for language-provided functions.
Example:

```
# file: mypack:function_tests
log info function(min, max) {}.declaringFile  # logs mypack:function_tests
```

Exceptions

Identifier: `exception`
These values represent exceptions or errors thrown by the program logic. Values of type exception are only created when an exception is thrown.
Example:

```
# file: mypack:exceptions
try {
    throw "An error occurred"
} catch(ex) {
    log info ex
}
```

Members

`message : string`
readonly
Contains a brief description of the exception.
Example:

```
# file: mypack:exceptions
try {
    throw "An error occurred"
} catch(ex) {
    log info ex.message # logs "An error occurred"
}
```

`extendedMessage : string`
readonly
Contains a long description of the exception, detailing the error type, message and stack trace.
Example (with line numbers):

```
# file: mypack:exceptions
1 try {
2     throw "An error occurred"
3 } catch(ex) {
4     log info ex.extendedMessage
      # logs An error occurred
      #   at mypack:exceptions ~ <body>
      #   at mypack:exceptions ~ <body> (exceptions.tdn:2)
5 }
```

`line : int`
readonly
Contains the line from the file at which the exception was thrown (starting from 1).
Example (with line numbers):

```
# file: mypack:exceptions
1 try {
2     throw "An error occurred"
3 } catch(ex) {
4     log info ex.line # logs 2
5 }
```

`column : int`
readonly
Contains the column from the file at which the exception was thrown (starting from 1).
Example (with line numbers):

```
# file: mypack:exceptions
1 try {
2     throw "An error occurred"
3 } catch(ex) {
4     log info ex.line # logs 5
5 }
```

`index : int`
readonly
Contains the index from the file at which the exception was thrown (starting from 0).
Example (with line numbers):

```
# file: mypack:exceptions
1 try {
2     throw "An error occurred"
3 } catch(ex) {
4     log info ex.line # logs 10
5 }
```

`type : string`
readonly
Contains a key of the exception type, that describes what could have caused it. Will always return one of the following:
"USER_EXCEPTION"
"TYPE_ERROR"
"ARITHMETIC_ERROR"
"COMMAND_ERROR"
"INTERNAL_EXCEPTION"
"STRUCTURAL_ERROR"
"LANGUAGE_LEVEL_ERROR"
"DUPLICATION_ERROR"
"IMPOSSIBLE"
Example:

```
# file: mypack:exceptions
try recovering {
    throw "An error occurred"
    eval 1 / 0
    summon ${1}
    summon player
} catch(exceptions) {
    for(ex in exceptions) {
        log info ex.type
        # logs:
        #     "USER_EXCEPTION"
        #     "ARITHMETIC_ERROR"
        #     "TYPE_ERROR"
        #     "COMMAND_ERROR"
    }
}
```

All exceptions thrown by the throw instruction will be of type USER_EXCEPTION. The type IMPOSSIBLE is reserved for language bugs, and typically signifies that something has gone wrong with the compiler and that the user should report it.

Type Definitions

Identifier: `type_definition`
Type Definitions are objects that represent each Interpolation Value type. They can be created via the type_definition literal, as well as the `type_definition.of()` function:

```
var int_def = type_definition<int>
var real_def = type_definition<real>
var string_def = type_definition<string>
var dict_def = type_definition.of({"a":"b"})
var type_def_def = type_definition<type_definition>
# This last one is the same as the type_definition global constant
```

The static value of Custom Classes are also type definitions and can be used as such.

Members

`is: function(value : *?) : boolean`
readonly
Checks whether the given value is instance of the given type. Null values always return false.
Example:

```
log info type_definition<int>.is(5)       # logs true
log info type_definition<string>.is("a")  # logs true
log info type_definition<real>.is(5)      # logs false
log info type_definition<real>.is(5.0)    # logs true
log info type_definition<string>.is(null) # logs false
```

This acts identically to the `is` operator.

```
log info 5    is int       # logs true
log info "a"  is string    # logs true
log info 5    is real      # logs false
log info 5.0  is real      # logs true
log info null is string    # logs false
```

`of: function(value : *?) : type_definition`
readonly
Returns the type definition of a given value.
The type definition object you call this on has no effect on the return value; this is only found in all type definition instances so it can be used via the global `type_definition` constant (which is the same as `type_definition<type_definition>`).
Example:

```
log info type_definition.of(5)      # logs type_definition<int>
log info type_definition.of(5.0)    # logs type_definition<real>
log info type_definition.of("a")    # logs type_definition<string>
log info type_definition.of(null)   # logs type_definition<null>
```

Custom Classes

Trident allows you to create custom interpolation data types. These types can be instantiated using a defined constructor, which will return an object of the class, containing custom data as defined by the class. They can have custom fields, methods, accessors and even casting definitions. They can also extend one or more classes, sharing their definitions.

```
# define [<access modifier>] [<symbol modifiers>] class <identifier> [: <superclasses...>]
define class MyClass {
    # fields, methods, accessors and cast definitions go here
}
```

Custom Class Declaration

Classes are declared using the define instruction. The declarations may have the following:

1. **Access modifier** (See Variable Visibility). One of: global, local, private. If omitted, defaults to local.
   Changes where the class symbol is initially accessible from.
2. **Symbol modifiers**. Any combination of:
   a. static: Means the class cannot be instantiated. A static class may only have static members. This also prevents other classes from extending this one.
   b. final: Means the class cannot be extended.
3. **Class Identifier**: The name of the class.
4. **Superclasses**: A list of classes that this new class will inherit from.

Incomplete Declarations

Often you may want to have multiple classes which reference each other, but because instructions need to be run one after another, you may run into an issue where one class is not defined when the other needs to reference it.
To circumvent this, Trident allows for incomplete declaration of classes. They're simply the class declaration without the body, and they must be declared in the same file as the complete definition.
Example:

```
define class B

define class A {
    public static var bReference : B
    # Without the incomplete B declaration, this would throw an error
}
define class B {
    public static var aReference : A
}
```

While a class definition is incomplete, you may use it in type constraints and class inherits, but you may not instantiate it or access static members.

There are certain restrictions when using incomplete definitions. The complete declaration header must be, more or less exactly the same as the incomplete declaration header:

1. The symbol modifiers must be the exact same. For instance, if the incomplete declaration had only the `static` modifier, the complete declaration must have the `static` modifier and the `static` modifier **only**.
2. The complete declaration must extend **all** the classes marked for extending by the incomplete declaration. The complete declaration, however, may extend other classes in addition to the ones promised by the incomplete declaration.

Fields

Classes may have fields, which are essentially variables. They may either be static fields (if the `static` modifier is present) or instance fields.
Static fields can be accessed without the need for an instance, and are attached to the type definition of the class.
Instance fields are variables created for each instance of the class, and all the instances may each have different values for that field.
Syntax:

```
[<access modifier>] [<symbol modifiers>] var <identifier> [: <constraints>] [= <initial value>]
```

Example:

```
define class A {
    public static var someStaticField : int = -5
    public var someInstanceField : int = 1
}

log info A.someStaticField      # -5
log info A.someInstanceField    # ERROR

var firstInstance : = new A()
var secondInstance : = new A()

log info firstInstance.someStaticField      # ERROR

log info firstInstance.someInstanceField    # 1

eval firstInstance.someInstanceField = 8
log info firstInstance.someInstanceField    # 8

log info secondInstance.someInstanceField   # 1
```

Field inheritance

When a class inherits from another, the new class may not create a field with the same name as a field in the inherited class - that field must be overridden to use that name.
To override a field, you must declare it in the new class, using the `override` keyword before the **var** keyword.
Overriding a field from an inherited class has the following requirements:

3. The inheriting class must have access to the field in the inherited class (e.g. the original field must not be private)
4. Both the original nor the new field must **not** be static
5. Both the original nor the new field must **not** be final
6. Both fields must have the **exact same** type constraints.

When a field is overridden, it simply changes what the initial value of the field becomes. Note that the original field's expression will **not** be evaluated. e.g. if the default value of the original field calls a function, that function **will not run** when the new class is instantiated.
Example:

```
define class A {
    public var someField : string? = "A"
}
define class B : A {
    public var someField : int = 5                  # ERROR

    public override var someField : string = "B"   # ERROR

    public override var someField : string? = "B"  # OK
}
```

Methods

Classes may also have methods, which are essentially dynamic functions, but with one key difference: **They can be overloaded**.
This also means that class methods are not retainable objects. You cannot take a class's method and store it in a variable, since you must call it for it to resolve into any function.
Static methods can be called without the need for an instance, and are attached to the type definition of the class.
Instance methods are methods created for each instance of the class. The `this` identifier can also be used inside instance methods to access the current object.

Syntax:

```
[<access modifier>] [<symbol modifiers>] <identifier>([<formal parameters...>]) [: <return constraints>]
{
    <commands and instructions...>
}
```

Example:

```
define class A {

    public someMethod() {
        log info "Got nothing"
    }

    public someMethod(number : int) {
        log info "Got integer " + number
    }

    public someMethod(number : real) {
        log info "Got real " + number
    }

    public someMethod(message : string) {
        log info "Got message \"" + message + "\""
    }
}

var someA : = new A()

eval someA.someMethod()          # 'Got nothing'
eval someA.someMethod(5)         # 'Got integer 5'
eval someA.someMethod(5.0)       # 'Got real 5.0'
eval someA.someMethod("5")       # 'Got message "message"'
eval someA.someMethod("5", null) # ERROR: Overload not found for parameter types: (string, null)
```

Static methods and instance methods are completely separated, meaning you can have methods with the exact same signature and different body, one as static and another as instance.

Picking an overload

Whenever a method is called, the best overload is picked based on the given parameter types and the type constraints. Each overload is given a rating from 0 to 4 based on the average "score" of its parameters. The parameters are rated with the following logic:

- 6 if the actual parameter matches perfectly the formal parameter (without coercion or inheritance)
- 5 if the actual parameter is an instance of the formal parameter type (without coercion)
- 3 if the actual parameter can be coerced into the formal parameter type
- 2 if the actual parameter is null and and the formal parameter constraint is nullable
- 1 if the actual parameter is omitted and and the formal parameter constraint is nullable
- 0 if the actual parameter does not match at all (and cannot be coerced) OR if the actual parameter is null and the formal parameter is not nullable

Any parameters with a rating of 0 automatically discard the overload.
Also, overloads with fewer former parameters than the actual parameters are automatically discarded.
Given the above scores, the single overload with the highest average score is chosen and called.
If there's a tie for first place, an error is thrown for an "ambiguous call"; and if there are none, an error is thrown for "overload not found for parameter types..."
Note that the calling context must have access to the chosen overload for it to be called correctly.

Symbols in context

In expressions within a class, static fields and methods of that class can be accessed without the need to use the class name.
Inside instance methods, you can access both static and instance methods without using the class name or `this`, unless a field is obscured by some variable or parameter in the current context.
Example:

```
define class A {

    private static var STATIC_FIELD : int = 2147483647
    private var instanceField : int = 255
    private var number : int = 128

    public someMethod() : string {
       return "Called someMethod()"
    }

    public someMethod(number : int) {
        log info someMethod()       # Called someMethod()

        log info STATIC_FIELD       # 2147483647
        log info A.STATIC_FIELD     # 2147483647

        log info instanceField      # 255
        log info this.instanceField # 255

        log info number             # 5 (from parameter)
        log info this.number        # 128 (from field)
    }
}

eval new A().someMethod(5)
```

Method inheritance

When a class inherits from another, the new class may not create an overload for a method with the same name. That overload must be overridden.
To override a method, you must declare it in the new class, using the `override` keyword before the function name.
Overriding a method from an inherited class has the following requirements:

1. The inheriting class must have access to the method in the inherited class (e.g. the original method must not be private)
2. Both the original nor the new method must **not** be static
3. Both the original nor the new method must **not** be final
4. The new method must not assign weaker access privileges than the original (can turn a `private` method `public`, but not a `public` method `private`)
5. The new method's return constraints must be entirely contained in the original method's return constraints. This means the new constraints should be **just as, if not more restrictive** than the original. e.g: `string?` to `string`

Example:

```
define class A {
    public someMethod() : string? {
        return null
    }
}

define class B : A {

    public someMethod(num : int) {}             # OK, different signature

    public someMethod() {}                      # ERROR

    public override someMethod() {}             # ERROR

    local override someMethod() : string? {     # ERROR
        return 0
    }

    public override someMethod() : string {     # OK
        return ""
    }
}
```

Virtual Methods

In some situations you may want a method to be overridden by a subclass. However, as class inheritance is flat, and not hierarchical, you may encounter issues trying to extend a class that overrides another class's method. The solution for this is the virtual keyword. Place the virtual modifier on the class that defines the method. This will make it so, if there is a clash between a virtual method and a non-virtual method, the non-virtual method will be chosen instead of throwing an error. Example:

```
define class A {
    public foo() {}
}
define class B : A {
    public override foo() {}
}
define class C : B {}
# Uncaught Structural Error: Method 'foo()' is defined in multiple inherited classes: B, A.
# Override it in the class body, or mark one of the two methods as virtual
```

```
define class A {
    public virtual foo() {} # virtual keyword added
}
define class B : A {
    public override foo() {}
}
define class C : B {}
# OK, B's foo is chosen over A's
```

Constructors

Constructors are methods that are used to instantiate objects of a class. They are called using the keyword **new**, followed by a reference to the class as a method call.
Constructors are defined just as a normal class method, but with the **new** keyword in place of the method name. Whenever a constructor is called, a new object of the class is created, its fields are initialized to their default values, and the appropriate constructor method's body is called as the newly constructed object.
At the end of a constructor, **all final fields must be initialized**, otherwise an exception will be thrown.
Note that Trident has no mechanism of ensuring that inherited classes run their own constructors appropriately. If you intend a class to require proper initialization beyond fields upon construction, and want it to be extendable, consider creating an initialization method and having all constructors (including in inheriting classes) call it.

Example:

```
define class A {

    private final var score : int

    public new(score : int) {
        eval this.score = score
    }
    private new(score : string) {
       eval this.score = score.length
    }
}


var a = new A(5)   # OK
var a = new A("")  # ERROR, private access
```

If a class does not explicitly declare any constructors, it will have an implicit empty public constructor.
Static classes may not declare any constructors.

Default Methods

All custom classes implicitly extend one base class, and it has the following methods defined, which can be overridden:

`public toString() : string`
Called whenever the object needs to be converted into a string (not to be confused with a cast). Called by language features such as the **log** instruction or the string concatenation operator +.

Example:

```
define class A {
    private final var score : int
    public new(score : int) {
        eval this.score = score
    }

    public override toString() : string {
        return "score=" + score
    }
}

log info new A(5)  # "score=5"
```

`public getIterator()`
Called whenever the object needs to be iterated through via a foreach loop. Must return a natively iterable object such as a list, dictionary or string, or another iterable class.

Example:

```
define class ListWrapper {
   private final var list : list = [0,1,2,3,4]

   public override getIterator() {
       return list
    }
}


for(element in new ListWrapper()) {
    log info element # Prints 0, 1, 2, 3, 4
}
```

Indexer

Custom classes may define the behavior of square brace `[]` indexers, for use similar to lists and dictionaries.
Indexers are essentially two methods: One for `setting`, and another for `getting`, and they both share a parameter type, which will be the value inside the square braces.

The syntax is as follows:

```
public this[<index parameter>] {
    get <return constraint> {
        # code for getting
    }
    set(<set parameter>) {
        # code for setting
    }
}
```

Where the index and set parameter may be constrained, the return constraint is optional, and the set block may be omitted.
Each get and set block may also have its own access modifier (before the `get` and `set` keywords).

Example:

```
define class BitField {
    private var field : int = 0

    public new(default : boolean) {
        if(default) eval field = ~0
    }

    public override toString() : string {
        return Integer.toUnsignedString(field, 2)
    }

    public this[bit : int] {
        get : boolean {
            return (this.field & (1 << bit)) != 0
        }
        set(value : boolean) {
            eval this.field &= ~(1 << bit)
            if(value) eval this.field |= (1 << bit)
        }
    }
}

var bitField : = new BitField(false)

#set
eval bitField[0] = true
eval bitField[1] = true
eval bitField[3] = true
eval bitField[31] = true

log info bitField # 10000000000000000000000000001011

#get
log info bitField[1]  #true
log info bitField[2]  #false
log info bitField[5]  #false
log info bitField[31] #true
```

Indexer inheritance

When a class inherits from another, the new class may not create an indexer in the inherited class without overriding the original indexer.
To override an indexer, you must declare it in the new class, using the `override` keyword before the **this** keyword.
Overriding an indexer from an inherited class has the following requirements:

6. The inheriting class must have access to both the getter and the setter (if it exists) in the inherited class (e.g. the original indexer must not be private)
7. The old indexer's index parameter constraints must be entirely contained in the new indexer's index parameter constraints. This means the new constraints should be **just as, if not less restrictive** than the original. e.g: `string` to `string?`
8. If the original indexer has a setter, the new indexer must also declare a setter.
9. The new indexer must not assign weaker access privileges than the original (can turn a `private` indexer `public`, but not a `public` indexer `private`)
10. The new indexer's getter return constraints must be entirely contained in the old indexer's getter return constraints. This means the new constraints should be **just as, if not more restrictive** than the original. e.g: `string?` to `string`
11. The old indexer's setter value constraints must be entirely contained in the new indexer's setter value constraints. This means the new constraints should be **just as, if not less restrictive** than the original. e.g: `string` to `string?`
    This does not apply if the old indexer doesn't have a setter.

Properties

Custom classes may define properties that have their own get and set methods, similarly to indexers, but accessed via dot notation, not bracket notation. Unlike accessors, properties can be static.

The syntax is as follows:

```
public <property identifier> {
    get <return constraint> {
        # code for getting
    }
    set(<set parameter>) {
        # code for setting
    }
}
```

where `<set parameter>` may be constrained, `<return constraint>` is optional, and the `set` block may be omitted.
Each get and set block may also have its own access modifier (before the `get` and `set` keywords).

Example:

```
define class PropertyTest {
    private var _count : int = 0

    public count {
        get : int {
            return this._count
        }
        set(value : int) {
            if(value < 0) throw "Count cannot be negative"
           eval this._count = value
        }
    }
}

eval new PropertyTest().count = -1 # ERROR: Count cannot be negative
eval new PropertyTest().count = 5 # OK
```

Property inheritance

To override a property, you must declare it in the new class, using the `override` keyword before the identifier.
Overriding an indexer from an inherited class has the following requirements:

1. The inheriting class must have access to both the getter and the setter (if it exists) in the inherited class (e.g. the original property must not be private)
2. If the original property has a setter, the new property must also declare a setter.
3. The new property must not assign weaker access privileges than the original (can turn a `private` property `public`, but not a `public` property `private`)
4. The new property's getter return constraints must be entirely contained in the old property's getter return constraints. This means the new constraints should be **just as, if not more restrictive** than the original. e.g: `string?` to `string`
5. The old property's setter value constraints must be entirely contained in the new property's setter value constraints. This means the new constraints should be **just as, if not less restrictive** than the original. e.g: `string` to `string?`
   This does not apply if the old property doesn't have a setter.

Conversion Definitions

Custom classes may define their behavior when objects of that class need to be cast or coerced into a type.
We call coercion "implicit conversion" and casting "explicit conversion". Coercion is what occurs when a value is passed through a type constraint that doesn't exactly match the value, but the value has an implicit conversion into that type.
Casting is what occurs when a value is explicitly cast to another type, whether through parenthesis casting or the `as` operator.

The syntax for conversion definitions is as follows:

```
override explicit <<target_type>> {
    # code for casting
}

override implicit <<target_type>> {
    # code for coercing
}
```

Note the lack of access modifiers, as the visibility of conversion functions cannot be limited.

Example:

```
define class A {
    private final var score : int
    public new(score : int) {
        eval this.score = score
    }

    override implicit <real> {
        # Need to explicitly cast to real,
        # as the return values of implicit
        # conversion functions are not coerced
        return (real)score
    }
    override explicit <real> {
        return 10 * score
    }
}

log info Math.pow(new A(5),2)  # Coercion, 25.0
log info new A(5) as real      # Casting,  50.0
```

Important note: the implicit conversion function is not allowed to coerce its return value into the target value; it must return the **exact** type.

Conversion inheritance

When a class inherits from another, the new class may freely change the conversion definitions, as the return constraints are already very strict and well defined.

Operator Overloading

Custom classes may define custom operator behavior for any combination of types and operators. They are not limited to the containing class, so a class may define operators for entirely different classes, or even primitives.
The operator definition to use is chosen the same way as overloaded methods.

The syntax for operator overloading is as follows:

```
operator <operator symbol>(<param 0>, <param 1>, ..., <param n>) {
    # code for operator
}
```

where:

<operator symbol> is the symbol of the operator to overload (e.g. `*`, `+`, `!`). For operators with more than 2 operands, only the first symbol needs to be used (For instance, the conditional ternary operator `?:`, which is grade 3, only needs the first symbol, `?`)

<param 0>, <param 1>, ..., <param n> are the operands. This number will depend on how many operands the operator you're overloading accepts. For example, if you're overloading the binary `+` operator, you'll need 2 operands. If you're overloading the unary (left) `+` operator, you'll need 1 operand.

If the operator you're overloading is unary, it will choose the left hand variant of the operator, if it exists. Otherwise it'll overload the right. At the moment though, there exists no right-hand unary operator in the language that doesn't also have a left-hand variant (`++` and `--` operators exist for both sides), so overloading a right unary operator is, in practice, impossible.

Compound assignment operators (such as `+=`, `*=`), as well as `??` cannot be overloaded. In the case of compound assignment operators, the corresponding simple operator will be used (so `a += b` behaves the same as `a = a + b`)

Short-circuiting operators such as `?:`, `||` and `&&` will use their default behavior if the first operand is a boolean. Otherwise, custom operator code will be used if all the arguments match the custom operator signature.

Note the lack of access modifiers, as the visibility of operators cannot be limited.

Example:

```
define static class CustomOperators {
    operator *(part : string, times : int) {
        var repeated = ""
        for(var i = 0; i < times; i++) {
            eval repeated += part
        }
        return repeated
    }
}

log info "ba" * 5 # Logs bababababa
```

Inheritance

Classes may inherit from from other classes, making objects of the new class behave as the inherited class. This is particularly useful for extending the functionality of one or more classes.

```
define class A {
    public var aField : int = 5
}
define class B : A {
    public var bField : string = "five"
}

var bInstance : B = new B()

log info bInstance.bField  # "five"
log info bInstance.aField  # 5
```

Note that, unlike in most object-oriented programming languages, fields and methods can only have one form. This means that a field declared in both A and B with the same name will effectively destroy the field in A, and only B's implementation will be accessible, even by methods of A. To avoid unintentionally causing issues like these, the `override` keyword must be used to resolve issues like these.

Also unlike in most object oriented programming languages, inheritance with Trident classes is iterative, not recursive. Because of this, circular inheritance is possible.

```
define class A
define class B : A
define class C : A

define class A : C {
}
define class B : A {
}
define class C : B {
}
```

As any conflicts between inherited members is required to be resolved by the inheriting class, classes in a cycle are effectively one large class with its members defined across three different classes.

Generics

Custom classes may define type parameters. Generic classes describe functionality that isn't specific to any particular data type, but needs the data type to be consistent across all the features of a single instance of that class. This can be useful for things like typed lists.

To define a generic class, the names of the type parameters need to be placed between angle brackets <>, separated by commas, after the class name. Then inside that class, the generic types can be referenced via those names.
To instantiate (or reference a type with explicit type parameters), the actual types should be placed in much the same way after the class name.
Example:

```
define class Pair<K, V> {
    public var key : K
    public var value : V

    public new(key : K, value : V) {
        eval this.key = key
        eval this.value = value
    }
}

var stringIntPair : Pair<string, int> = new Pair<string, int>("five", 5)
var itemBoolPair : Pair = new Pair<item, boolean>(item<stick>, true)

eval stringIntPair.key = "go" # OK
eval stringIntPair.key = true # ERROR

eval itemBoolPair.value = false # OK
eval itemBoolPair.value = 5 # ERROR
```

This describes a class that represents a pair of two values, where the type of each value is not known in advance, but upon instantiation the types will be resolved. Assigning values to the key and value fields of type that doesn't match the types bound to that object at instantiation is forbidden.

An object of a generic class cannot be instantiated without actual type parameters, and the number of type parameters must match that of the generic class.
However, you can use a generic class without type parameters in places such as type constraints and the `is` operator, where the objective is to verify the type of an object, not create one.

Native Libraries

~~`typeOf: function(obj : *?) : type_definition`~~
Deprecated, use [`type_definition.of(*?)`]
editable
Returns a string containing the type ID of the object passed into it.
Example:

```
log info typeOf("foo")
# logs "string"
log info typeOf(nbt_value<[B; 0b, 1b, 2b, 3b]>)
# logs "tag_byte_array"
log info typeOf(resource<minecraft:custom>)
# logs "resource"
log info typeOf(null)
# logs "null"
```

~~`isInstance: function(obj : *?, type : string) : boolean`~~
editable
Deprecated, use the `is` operator.
Checks whether the given object is of the type string passed into it.
Example:

```
log info isInstance(1, "int")
# logs false
log info isInstance("foo","real")
# logs false
log info isInstance(nbt_value<[0b, 1b, 2b, 3b]>,"nbt_value")
# logs true
log info isInstance(null,"string")
# logs false
```

Reflection

`trident-util:native@Reflection`
public static final class
This class provides several methods for dealing with Trident Files.

`getMetadata(loc : resource) : dictionary`
public static method
Returns the metadata object of the file represented by the resource location. If the specified file doesn't have metadata, an empty object is returned. Note that the file to search must be a top-level file.
Example:

```
# file: tdndemo:a.tdn
@ metadata {"foo": ["bar", "baz"]}

# file: tdndemo:reflection.tdn
log info Reflection.getMetadata(resource<tdndemo:a>)
# logs {"foo": ["bar", "baz"]}
```

`getFilesWithTag(loc : resource) : list`
public static method
Returns a list of all top-level Trident files whose functions are tagged with the given resource location. Whether the given resource location is a tag or not does not affect the result. The resulting list contains the resource locations for all the matching top-level functions.
Example:

```
# file: tdndemo:a.tdn
@ tag custom

# file: tdndemo:b.tdn
@ tag load
@ tag custom

# file: tdndemo:reflection.tdn
log info Reflection.getFilesWithTag(resource<minecraft:custom>)
# logs [tdndemo:a, tdndemo:b]
```

`getFilesWithMetaTag(loc : resource) : list`
public static method
Returns a list of all top-level Trident files with the given meta tag. The resulting list contains the resource locations for all the matching top-level functions.
Example:

```
# file: tdndemo:b.tdn
@ meta_tag custom

# file: tdndemo:a.tdn
@ tag custom

# file: tdndemo:reflection.tdn
log info Reflection.getFilesWithMetaTag(resource<minecraft:custom>)
# logs [tdndemo:a]
```

`insertToFile(loc : resource, writer : function)`
public static method
Runs the given dynamic function within the context of the top-level file specified by the given resource location.
Example:

```
# file: tdndemo:a.tdn
say Hello

# file: tdndemo:reflection.tdn
eval Reflection.insertToFile(resource<tdndemo:a>, function() {
    say World!
})

# output of file tdndemo:a
say Hello
say World!
```

`getCurrentFile() : resource`
public static method
Returns the resource location of the function this is called from.
Example:

```
# file: tdndemo:reflection.tdn
var testfunct = function() {
    log info Reflection.getCurrentFile()
}

eval testfunct()
# logs tdndemo:reflection

define function inner {
    log info Reflection.getCurrentFile()
    # logs tdndemo:reflection/inner

    eval testfunct()
    # logs tdndemo:reflection
}
```

`getWritingFile() : resource`
public static method
Returns the resource location of the function currently being written to.
Example:

```
# file: tdndemo:reflection.tdn
var testfunct = function() {
    log info Reflection.getWritingFile()
}

eval testfunct()
# logs tdndemo:reflection

define function inner {
    log info Reflection.getWritingFile()
    # logs tdndemo:reflection/inner

    eval testfunct()
    # logs tdndemo:reflection/inner
}
```

`getCurrentFilePath() : string`
public static method
Returns the path of the file this is called from
Example:

```
# file: tdndemo:reflection.tdn

log info Reflection.getCurrentFilePath()
# logs datapack/data/tdndemo/functions/reflection.tdn
```

File

`trident-util:native@File`
public static final class
Container class for File IO classes. The IO classes are defined all defined privately under `trident-util:native`, and the File class exposes them publicly through static final fields. Any access to each of the IO classes must be through the File class.

File.in

`trident-util:native@in`
private static final class
This class provides methods for reading files within the project directory.

`read(path : resource) : *`
public static method
Returns the contents of the file at the given location, from the project directory.
If the specified path represents a file, its text data is returned as a string. If the specified path represents a directory, its children's filenames are returned as a list of strings.
Example:

```
# file: /structure_gen/oak_tree.json
{
    "trunk_block": "minecraft:oak_log",
    "leaf_block": "minecraft:oak_leaves",
    "trunk_length": [4, 7],
    "leaf_shape": "bush"
}
# file: /structure_gen/birch_tree.json
{
    "trunk_block": "minecraft:birch_log",
    "leaf_block": "minecraft:birch_leaves",
    "trunk_length": [5, 9],
    "leaf_shape": "bush"
}
# file: /datapack/data/tdndemo/functions/io.tdn

log info File.in.read("structure_gen")
# logs ["oak_tree.json", "birch_tree.json"]

log info JSON.parse(File.in.read("structure_gen/birch_tree.json"))
# logs {leaf_block: "minecraft:birch_leaves", leaf_shape: "bush", trunk_length: [5, 9], trunk_block: "minecraft:birch_log"}
```

`exists(path : string) : boolean`
public static method
Returns whether the file at the specified path within the input project directory exists.
Example:

```
log info File.exists("datapack/data/test/functions")
```

`isDirectory(path : string) : boolean`
public static method
Returns whether the file at the specified path within the input project directory exists and is a directory.
Example:

```
log info File.isDirectory("datapack/data/test/functions")
```

`write(path : string, content : string)`
public static method
Writes the given string to the file at the specified path within the input project directory.
Example:

```
eval File.in.write(
    "datapack/data/test/functions/function_for_next_compilation.tdn",
    "@on compile\nlog info \"Hello from the past!\""
)
```

`wasFileChanged(path : string) : boolean`
public static method
Returns whether the file at the specified path within the input project directory was changed since the last successful compilation.
Example:

```
log info File.in.wasFileChanged(Reflection.getCurrentFilePath())
```

File.out

`trident-util:native@out`
private static final class
This object provides methods for writing to the output data pack and resource pack files.

`writeData(path : string, content : string)`
public static method
Writes the given string to the file at the specified path within the output data pack.
Example:

```
eval File.out.writeData(
    "data/custom/tags/blocks/unbreakable.json",
    JSON.stringify(
        {
            "values": [
                "minecraft:dark_oak_planks",
                "minecraft:chest",
                "minecraft:iron_block"
            ]
        },
        true
    )
)
```

`writeResource(path : string, content : string)`
public static method
Writes the given string to the file at the specified path within the output resource pack, if there is a resource pack in the project.
Example:

```
eval File.out.writeResource(
    "assets/minecraft/sounds.json",
    JSON.stringify(
        {
            "block.enderchest.open": {
                "category": "block",
                "replace": true,
                "sounds": [
                    "blank"
                ]
            }
        },
        true
    )
)
```

JSON

`trident-util:native@JSON`
public static final class
This class provides several methods for converting from JSON strings to complex Trident values and vice versa.

`parse(s : string) : *`
public static method
Parses the string argument and attempts to convert it into Trident values, in the following manner:

- JSON Objects become dictionaries
- JSON Lists become lists
- JSON Numbers become integers (if they can be expressed as such), otherwise, real numbers
- JSON Booleans become booleans
- JSON Strings become strings

Example:

```
log info JSON.parse('{"a": 1.8, "b": ["1", 2.0, true]}')
# logs {a: 1.8, b: ["1", 2, true]}
```

```
stringify (multiple overloads)
stringify(obj : string,     prettyPrinting : boolean?) : string
stringify(obj : boolean,    prettyPrinting : boolean?) : string
stringify(obj : int,        prettyPrinting : boolean?) : string
stringify(obj : real,       prettyPrinting : boolean?) : string
stringify(obj : list,       prettyPrinting : boolean?) : string
stringify(obj : dictionary, prettyPrinting : boolean?) : string
```

public static method
Converts the given value into a JSON string, in the following manner:

- Dictionaries become JSON Objects
- Lists become JSON Lists
- Reals and Integers become JSON Numbers
- Strings and Resource Locations become JSON Strings
- Booleans become JSON Booleans

Any other values will not be present in the return value.
If prettyPrinting is true, the output will be spread over multiple lines and indented. If not specified, prettyPrinting is not enabled.
Example:

```
log info JSON.stringify({a: 1.8, b: ["1", 2, true, entity<@a>], c: resource<tick>}, true)
# logs {
#   "a": 1.8,
#   "b": [
#     "1",
#     2,
#     true
#   ],
#   "c": "minecraft:tick"
# }
```

Text

`trident-util:native@Text`
public static final class
This class provides several methods for dealing with Text Components.

`parse(s : string) : text_component`
public static method
Parses the string argument and attempts to convert it into a Text Component. *Roughly* equivalent to passing the result of `JSON.parse(s)` to `new text_component(o)`
Example:

```
log info Text.parse('["Hello ", {"text": "World", "color": "green"}]')
# logs ["Hello ",{"text":"World","color":"green"}] (of type text_component)
```

Character

`trident-util:native@Character`
public static final class
This class provides methods for creating and getting info about characters.

`fromCodePoint(codePoint : int) : string`
public static method
Creates a single-character string with the unicode character of the given code point.
Example:

```
log info Character.fromCodePoint(230)
# logs æ
```

`toCodePoint(char : string) : int`
public static method
Retrieves the code point value of the first character of the given string. Throws an exception if the string is empty.
Example:

```
log info Character.toCodePoint("æ")
# logs 230
```

`getName(char : string) : string`
public static method
Returns the Unicode name of the first character in the given string. Throws an exception if the string is empty.
Example:

```
log info Character.getName("æ")
# logs LATIN SMALL LETTER AE
```

`isLetter(char : string) : boolean`
public static method
Returns a boolean dictating whether the first character in the given string is a letter. Throws an exception if the string is empty.
Example:

```
log info Character.isLetter("æ") # logs true
log info Character.isLetter("5") # logs false
```

`isDigit(char : string) : boolean`
public static method
Returns a boolean dictating whether the first character in the given string is a digit. Throws an exception if the string is empty.
Example:

```
log info Character.isDigit("æ") # logs false
log info Character.isDigit("5") # logs true
```

`isWhitespace(char : string) : boolean`
public static method
Returns a boolean dictating whether the first character in the given string is whitespace. Throws an exception if the string is empty.
Example:

```
log info Character.isWhitespace(" ")  # (space)     logs true
log info Character.isWhitespace("5")  #             logs false
log info Character.isWhitespace("\n") # (line feed) logs true
```

`isUpperCase(char : string) : boolean`
public static method
Returns a boolean dictating whether the first character in the given string is an uppercase letter. Throws an exception if the string is empty.
Example:

```
log info Character.isUpperCase("5") # logs false
log info Character.isUpperCase("a") # logs false
log info Character.isUpperCase("A") # logs true
```

`isLowerCase(char : string) : boolean`
public static method
Returns a boolean dictating whether the first character in the given string is a lowercase letter. Throws an exception if the string is empty.
Example:

```
log info Character.isLowerCase("5") # logs false
log info Character.isLowerCase("a") # logs true
log info Character.isLowerCase("A") # logs false
```

Tags

`trident-util:native@Tags`
public static final class
This class provides methods for creating and getting info about type tags.

`createTag(category : string, location : resource, values : list[resource])`
public static method
Creates a tag of the given category at the given resource location, and adds the provided list of resource location values to that tag (all must be valid types of the specified category, that is, they must exist).
Refer to the complementary documentation (Definition Packs > Types) for category names and info.
Example:

```
eval Tags.createTag(
        "block",
        resource<#custom:ignore>,
        [
                resource<minecraft:glass>,
                resource<minecraft:sponge>,
                resource<#minecraft:stained_glass>
        ]
)
```

`exists(category : string, location : resource) : boolean`
public static method
Checks whether the specified tag in the given category exists.
Refer to the complementary documentation (Definition Packs > Types) for category names and info.
Returns true if a tag exists for it, false otherwise.
Example:

```
log info Tags.exists("entity", resource<#minecraft:skeletons>)
# logs true
log info Tags.exists("block", resource<#minecraft:skeletons>)
# logs false (unless a user-created #skeletons block tag exists in the project)
```

`tagContainsValue(category : string, tagLocation : resource, valueLocation : resource) : boolean`
public static method
Checks whether the specified tag in the given category contains the provided value.
Refer to the complementary documentation (Definition Packs > Types) for category names and info.
Example:

```
eval Tags.tagContainsValue(
        "block",
        resource<#minecraft:flower_pots>,
        resource<minecraft:potted_poppy>
)
# logs true
eval Tags.tagContainsValue(
        "block",
        resource<#minecraft:slabs>,
        resource<#minecraft:wooden_slabs>
)
# logs true
eval Tags.tagContainsValue(
        "block",
        resource<#minecraft:slabs>,
        resource<minecraft:oak_slab>
)
# logs true
eval Tags.tagContainsValue(
        "block",
        resource<#minecraft:wooden_slabs>,
        resource<minecraft:stone_slab>
)
# logs false
```

MinecraftTypes

`trident-util:native@MinecraftTypes`
public static final class
This class provides methods for gathering info about all minecraft-defined types.

`getDefinitionsForCategory(category : string) : dictionary`
public static method
Retrieves an object with all the names and properties of all types of a certain type category, provided by the definition pack used in Trident. Refer to the complementary documentation (Definition Packs > Types) for category names and info.
Example:

```
log info MinecraftTypes.getDefinitionsForCategory("effect")
# logs {"minecraft:speed": {"id": 1,"type": "positive"},"minecraft:slowness": {"id": 2,"type":
"negative"}, ...
```

`exists (multiple overloads)`
`exists(category : string, name : string) : boolean`
`exists(category : string, location : resource) : boolean`
public static method
Checks whether the given resource location or string represents a valid type of the given category. Refer to the complementary documentation (Definition Packs > Types) for category names and info.
Returns true if a type exists for it, false otherwise.
Example:

```
log info MinecraftTypes.exists("enchantment", resource<minecraft:mending>)
# logs true
log info MinecraftTypes.exists("gamemode", "hardcore")
# logs false
```

Block

`trident-util:native@Block`
public static final class
This class provides several methods for dealing with block types.

`exists(location : resource) : boolean`
public static method
Checks whether the given resource location represents a valid block type. Returns true if a block type exists for it, false otherwise.
Example:

```
log info Block.exists(resource<minecraft:fire>)    # logs true
log info Block.exists(resource<minecraft:string>)  # logs false
```

`getAll() : list`
public static method
Creates and returns a list containing resource locations for all the block types in no particular order.
Example:

```
log info Block.getAll()   # logs [minecraft:potted_acacia_sapling, minecraft:nether_brick_slab ...
minecraft:pink_concrete]
```

Item

`trident-util:native@Item`
public static final class
This class provides several methods for dealing with item types.

`exists(location : resource) : boolean`
public static method
Checks whether the given resource location represents a valid item type. Returns true if a item type exists for it, false otherwise.
Example:

```
log info Item.exists(resource<minecraft:fire>)    # logs false
log info Item.exists(resource<minecraft:string>)  # logs true
```

`getAll() : list`
public static method
Creates and returns a list containing resource locations for all the item types in no particular order.
Example:

```
log info Item.getAll()   # logs [minecraft:nether_brick_slab, minecraft:andesite_slab, minecraft:egg ...
minecraft:carrot]
```

Project

`trident-util:native@Project`
public static final class
This class provides several methods for dealing with the current Trident Project.

`getTargetVersion() : list`
public static method
Returns a list representing the target version of the running project. The first element is the Major release, the second is the Minor release, and the third number is the Patch. In practice, the Major number will always be 1 and the Patch number will always be 0.
Example:

```
log info Project.getTargetVersion()    # logs [1, 16, 0] (on a project targeting 1.16)
```

`getFeatureBoolean(key : string, defaultValue : boolean?) : boolean`
public static method
Checks the feature map provided to the project via the build configuration for the given feature key, and returns a boolean based on its value. If the given key corresponds to a boolean in the feature map, that value will be returned. Otherwise, defaultValue is returned (or false if defaultValue is omitted).
Example:

```
log info Project.getFeatureBoolean("command.attribute")
# logs true (on a project targeting 1.16)

log info Project.getFeatureBoolean("textcomponent.hex_color", false)
# logs true (on a project targeting 1.16)
```

`getFeatureInt(key : string, defaultValue : int?) : int`
public static method
Checks the feature map provided to the project via the build configuration for the given feature key, and returns an integer based on its value. If the given key corresponds to an integer in the feature map, that value will be returned. Otherwise, defaultValue is returned (or 0 if defaultValue is omitted).
Example:

```
log info Project.getFeatureInt("objectives.max_length")
# logs 16
```

`getFeatureString(key : string, defaultValue : string?) : string?`
public static method
Checks the feature map provided to the project via the build configuration for the given feature key, and returns a string based on its value. If the given key corresponds to a string in the feature map, that value will be returned. Otherwise, defaultValue is returned (or null if defaultValue is omitted).
Example:

```
log info Project.getFeatureString("objectives.regex")
# logs "[A-Za-z0-9_.\-+]+"
log info Project.getFeatureString("function.export_path")
# logs "data/$NAMESPACE$/functions/"
```

Math

`trident-util:native@Math`
public static final class
A port of java.lang.Math for Trident. The Math class contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

`PI : real`
public static final
The real value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.
Example:

```
log info Math.PI       #logs 3.141592653589793
```

`E : real`
public static final
The double value that is closer than any other to e (Euler's Number), the base of the natural logarithms.
Example:

```
log info Math.E        #logs 2.718281828459045
```

`pow(a : real, b : real) : real`
public static method
Returns the real value of the first argument raised to the power of the second argument.
Example:

```
log info Math.pow(2, 8)   # logs 256.0
log info Math.pow(2, -8)  # logs 0.00390625
```

`min (multiple overloads)`
`min(a : int, b : int) : int`
`min(a : real, b : real) : real`
public static method
Returns the smallest (closest to negative infinity) of two values. If both parameters are integers, the returned value will also be an integer. Otherwise a real number is returned.
Example:

```
log info Math.min(2, -8)   # logs -8
```

`max (multiple overloads)`
`max(a : int, b : int) : int`
`max(a : real, b : real) : real`
public static method
Returns the largest (closest to positive infinity) of two values. If both parameters are integers, the returned value will also be an integer. Otherwise a real number is returned.
Example:

```
log info Math.max(2, -8)   # logs 2
```

`abs (multiple overloads)`
`abs(x : int) : int`
`abs(x : real) : real`
public static method
Returns the absolute value of a value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
The return value will match the type of the parameter (int stays int, real stays real).
Example:

```
log info Math.abs(-8)   # logs 8
log info Math.abs(8)    # logs 8
```

`floor(x : real) : real`
public static method
Returns the largest (closest to positive infinity) real value that is less than or equal to the argument and is equal to a mathematical integer.
Example:

```
log info Math.floor(-0.1)   # logs -1
log info Math.floor(0.1)    # logs 0
```

`ceil(x : real) : real`
public static method
Returns the smallest (closest to negative infinity) real value that is greater than or equal to the argument and is equal to a mathematical integer.
Example:

```
log info Math.ceil(-0.1)   # logs 0
log info Math.ceil(0.1)    # logs 1
```

`round(x : real) : real`
public static method
Returns the closest real value that is equal to a mathematical integer, with ties rounding to positive infinity.
Example:

```
log info Math.round(-0.9)   # logs -1
log info Math.round(-0.1)   # logs 0
log info Math.round(0.5)    # logs 1
```

`floorMod(x : int, y : int) : int`
public static method
Returns the floor modulus of the int arguments.
The floor modulus is `x - (floorDiv(x, y) * y)`, has the same sign as the divisor `y`, and is in the range of `-abs(y) < r < +abs(y)`
The difference in values between floorMod and the % operator is due to the difference between floorDiv that returns the integer less than or equal to the quotient and the / operator that returns the integer closest to zero.
Example:

```
log info Math.floorMod(4, 3)   # logs 1
log info 4 % 3                 # logs 1

log info Math.floorMod(4, -3)  # logs -2
log info 4 % -3                # logs 1

log info Math.floorMod(-4, 3)  # logs 2
log info -4 % 3                # logs -1

log info Math.floorMod(-4, -3) # logs -1
log info -4 % -3               # logs -1
```

`floorDiv(x : int, y : int) : int`
public static method
Returns the largest (closest to positive infinity) int value that is less than or equal to the algebraic quotient.
Normal integer division operates under the round to zero rounding mode (truncation). This operation instead acts under the round toward negative infinity (floor) rounding mode. The floor rounding mode gives different results than truncation when the exact result is negative.
Example:

```
log info Math.floorDiv(4, 3)   # logs 1
log info 4 / 3                 # logs 1

log info Math.floorDiv(-4, 3)  # logs -2
log info -4 / 3                # logs -1
```

`sin(rad : real) : real`
public static method
Returns the trigonometric sine of an angle in radians.
Example:

```
log info Math.sin(0)             # logs 0.0
log info Math.sin(Math.PI / 2)   # logs 1.0
log info Math.sin(Math.PI)       # logs 1.2246467991473532E-16
```

`cos(rad : real) : real`
public static method
Returns the trigonometric cosine of an angle in radians.
Example:

```
log info Math.cos(0)             # logs 1.0
log info Math.cos(Math.PI / 2)   # logs 6.123233995736766E-17
log info Math.cos(Math.PI)       # logs -1.0
```

`tan(rad : real) : real`
public static method
Returns the trigonometric tangent of an angle in radians.
Example:

```
log info Math.tan(0)             # logs 0.0
log info Math.tan(Math.PI / 2)   # logs 1.633123935319537E16
log info Math.tan(Math.PI)       # logs -1.2246467991473532E-16
```

`sinh(x : real) : real`
public static method
Returns the hyperbolic sine of a real value.
The hyperbolic sine of x is defined to be $(e^x - e^{-x})/2$ where e is Euler's number.
Example:

```
log info Math.sinh(0)             # logs 0.0
log info Math.sinh(1)             # logs 1.1752011936438014
```

`cosh(x : real) : real`
public static method
Returns the hyperbolic cosine of a real value.
The hyperbolic cosine of x is defined to be $(e^x + e^{-x})/2$ where e is Euler's number.
Example:

```
log info Math.cosh(0)             # logs 1.0
log info Math.cosh(1)             # logs 1.543080634815244
```

`tanh(x : real) : real`
public static method
Returns the hyperbolic tangent of a real value.
The hyperbolic tangent of x is defined to be $(e^x - e^{-x})/(e^x + e^{-x})$ where e is Euler's number.
Example:

```
log info Math.tanh(0)             # logs 0.0
log info Math.tanh(1)             # logs 0.7615941559557649
```

`asin(a : real) : real`
public static method
Returns the arc sine of a value; the returned angle is in the range -pi/2 through pi/2 (in radians).
Example:

```
log info Math.asin(0)               # logs 0.0
log info Math.asin(1)               # logs 1.5707963267948966 (pi/2)
log info Math.asin(Math.sqrt(2)/2) # logs 0.7853981633974484 (pi/4)
```

`acos(a : real) : real`
public static method
Returns the arc cosine of a value; the returned angle is in the range -pi/2 through pi/2 (in radians).
Example:

```
log info Math.acos(0)               # logs 1.5707963267948966 (pi/2)
log info Math.acos(1)               # logs 0.0
log info Math.acos(Math.sqrt(2)/2) # logs 0.7853981633974484 (pi/4)
```

`atan(a : real) : real`
public static method
Returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2 (in radians).
Example:

```
log info Math.atan(0)               # logs 0.0
log info Math.atan(1)               # logs 0.7853981633974484 (pi/4)
log info Math.atan(Math.sqrt(2)/2) # logs 0.6154797086703874
```

`atan2(y : real, x : real) : real`
public static method
Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta). This method computes the phase theta by computing an arc tangent of y/x in the range of -pi to pi.
Example:

```
log info Math.atan2(1,Math.sqrt(3))    # logs 0.5235987755982989  (pi/6)
log info Math.atan2(-1,Math.sqrt(3))   # logs -0.5235987755982989 (-pi/6)
log info Math.atan2(1,-Math.sqrt(3))   # logs 2.6179938779914944  (5*pi/6)
log info Math.atan2(-1,-Math.sqrt(3))  # logs -2.6179938779914944 (-5*pi/6)
```

`toRadians(angdeg : real) : real`
public static method
Converts an angle measured in degrees to an approximately equivalent angle measured in radians. The conversion from degrees to radians is generally inexact; users should not expect `Math.cos(Math.toRadians(90.0))` to exactly equal 0.0.
Example:

```
log info Math.toRadians(0)         # logs 0.0
log info Math.toRadians(30)        # logs 0.5235987755982988
log info Math.toRadians(45)        # logs 0.7853981633974483
log info Math.toRadians(60)        # logs 1.0471975511965976
log info Math.toRadians(90)        # logs 1.5707963267948966
```

`toDegrees(angrad : real) : real`
public static method
Converts an angle measured in radians to an approximately equivalent angle measured in degrees. The conversion from radians to degrees is generally inexact.
Example:

```
log info Math.toDegrees(0)         # logs 0.0
log info Math.toDegrees(Math.PI/6) # logs 29.999999999999996
log info Math.toDegrees(Math.PI/4) # logs 45.0
log info Math.toDegrees(Math.PI/2) # logs 90.0
log info Math.toDegrees(Math.PI)   # logs 180.0
```

`log(a : real) : real`
public static method
Returns the natural logarithm (base e) of a real value.
Example:

```
log info Math.log(0)        # logs -Infinity
log info Math.log(1)        # logs 0.0
log info Math.log(Math.E)   # logs 1.0
```

`log10(a : real) : real`
public static method
Returns the base 10 logarithm of a real value.
Example:

```
log info Math.log10(0)        # logs -Infinity
log info Math.log10(1)        # logs 0.0
log info Math.log10(10)       # logs 1.0
log info Math.log10(100)      # logs 2.0
```

`log2(a : real) : real`
public static method
Returns the base 2 logarithm of a real value.
Example:

```
log info Math.log2(0)        # logs -Infinity
log info Math.log2(1)        # logs 0.0
log info Math.log2(2)        # logs 1.0
log info Math.log2(4)        # logs 2.0
log info Math.log2(8)        # logs 3.0
```

`sqrt(a : real) : real`
public static method
Returns the correctly rounded positive square root of a real value.
Example:

```
log info Math.sqrt(2)        # logs 1.4142135623730951
log info Math.sqrt(4)        # logs 2.0
log info Math.sqrt(16)       # logs 4.0
log info Math.sqrt(256)      # logs 16.0
```

`cbrt(a : real) : real`
public static method
Returns the correctly rounded positive cube root of a real value.
Example:

```
log info Math.cbrt(3)        # logs 1.4422495703074083
log info Math.cbrt(8)        # logs 2.0
log info Math.cbrt(27)       # logs 3.0
log info Math.cbrt(81)       # logs 5.0
```

`exp(a : real) : real`
public static method
Returns Euler's number e raised to the power of a real value.
Example:

```
log info Math.exp(0)        # logs 1.0
log info Math.exp(1)        # logs 2.718281828459045
log info Math.exp(2)        # logs 7.38905609893065
```

`signum(x : real) : real`
public static method
Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
Example:

```
log info Math.signum(-8)     # logs -1.0
log info Math.signum(0)      # logs 0.0
log info Math.signum(12)     # logs 1.0
```

`random (multiple overloads)`
`random() : real`
`random(min : real, max : real) : real`
public static method
Returns a real value, in a range between the given min (inclusive) and max (exclusive), or 0.0 and 1.0 if the bounds are not specified. Returned values are chosen pseudorandomly with (approximately) uniform distribution from that range.
Example:

```
log info Math.random()        # logs 0.40657443943172444
log info Math.random()        # logs 0.08285686880762
log info Math.random()        # logs 0.7925125273103358
# Don't expect to get the same numbers as this example
```

Integer

`trident-util:native@Integer`
public static final class
This class provides several methods for converting an int to a String and a String to an int, as well as other constants and methods useful when dealing with an int.

`MIN_VALUE : int`
public static final
A constant holding the minimum value an int can have, -2^{31}.
Example:

```
log info Integer.MIN_VALUE       #logs -2147483648
```

`MAX_VALUE : int`
public static final
A constant holding the maximum value an int can have, $2^{31} - 1$.
Example:

```
log info Integer.MAX_VALUE       #logs 2147483647
```

`parseInt (multiple overloads)`
`parseInt(s : string) : int`
`parseInt(s : string, radix : int) : int`
public static method
Parses the string argument as a signed integer in the radix specified by the second argument. The resulting integer value is returned. If the radix is left unspecified, it defaults to 10. Throws an exception if the resulting integer is outside of the valid signed integer range.
Example:

```
log info Integer.parseInt("48")      # logs 48
log info Integer.parseInt("42", 8)   # logs 34
log info Integer.parseInt("A0", 16)  # logs 160
```

`parseUnsignedInt (multiple overloads)`
`parseUnsignedInt(s : string) : int`
`parseUnsignedInt(s : string, radix : int) : int`
public static method
Parses the string argument as an unsigned integer in the radix specified by the second argument. The resulting integer value is returned. If the radix is left unspecified, it defaults to 10. Throws an exception if the resulting integer is outside of the valid unsigned integer range.
Example:

```
log info Integer.parseInt("80000000", 16)         # ERROR
log info Integer.parseUnsignedInt("80000000", 16) # -2147483648

log info Integer.parseInt("FFFFFFFF", 16)         # ERROR
log info Integer.parseUnsignedInt("FFFFFFFF", 16) # -1
```

`toString(multiple overloads)`
`toString(i : int) : string`
`toString(i : int, radix : int) : string`
public static method
Returns a string representation of the first argument in the radix specified by the second argument. If the radix is left unspecified, it defaults to 10. Calling this method with radix 10 is equivalent to casting the parameter to a string.
Example:

```
log info Integer.toString(48)        # logs 48
log info Integer.toString(34, 8)     # logs 42
log info Integer.toString(160, 16)   # logs a0
```

`toUnsignedString` *(multiple overloads)*
`toUnsignedString(i : int) : string`
`toUnsignedString(i : int, radix : int) : string`
public static method
Returns a string representation of the first argument as an unsigned integer value in the radix specified by the second argument. If the radix is left unspecified, it defaults to 10.
Example:

```
log info Integer.toUnsignedString(-1)     # 4294967295
log info Integer.toUnsignedString(-1, 16) # ffffffff
```

Real

`trident-util:native@Real`
public static final class
This class provides several methods for converting a real to a String and a String to a real, as well as other constants and methods useful when dealing with real numbers.

`Infinity : real`
public static final
A constant holding a positive infinity value.
Example:

```
log info Real.Infinity            #logs Infinity
```

`NaN : real`
public static final
A constant holding a Not-a-Number (NaN) value.
Example:

```
log info Real.NaN                 #logs NaN
```

`MIN_VALUE : real`
public static final
A constant holding the minimum finite value a real number can have, $-(2-2^{-52})\cdot 2^{1023}$.
Example:

```
log info Real.MIN_VALUE           #logs -1.7976931348623157E308
```

`MAX_VALUE : real`
public static final
A constant holding the maximum finite value a real number can have, $(2-2^{-52})\cdot 2^{1023}$.
Example:

```
log info Real.MAX_VALUE           #logs 1.7976931348623157E308
```

`MIN_POSITIVE_VALUE : real`
public static final
A constant holding the minimum positive value a real number can have, 2^{-1074}.
Example:

```
log info Real.MIN_POSITIVE_VALUE  #logs 4.9E-324
```

`parseReal(s : string) : real`
public static method
Returns a new real number initialized to the value represented by the specified String.
Example:

```
log info Real.parseReal("-0.3")            # logs -0.3
log info Real.parseReal("1")               # logs 1.0
log info Real.parseReal("15.0000001")      # logs 15.0000001
```

`isFinite(r : real) : boolean`
public static method
Returns true if the argument is a finite floating-point value; returns false otherwise (for NaN and infinity arguments).
Example:

```
log info Real.isFinite(48)            # logs true
log info Real.isFinite(Real.Infinity) # logs false
log info Real.isFinite(Real.NaN)      # logs false
```

`isInfinite(r : real) : boolean`
public static method
Returns true if the specified number is infinitely large in magnitude, false otherwise.
Example:

```
log info Real.isInfinite(48)            # logs false
log info Real.isInfinite(Real.Infinity) # logs true
log info Real.isInfinite(Real.NaN)      # logs false
```

`isNaN(r : real) : boolean`
public static method
Returns true if the specified number is a Not-a-Number (NaN) value, false otherwise.
Example:

```
log info Real.isNaN(48)               # logs false
log info Real.isNaN(Real.Infinity)    # logs false
log info Real.isNaN(Real.NaN)         # logs true
```

`toString(r : real) : string`
public static method
Returns a string representation of the real argument. Calling this method is equivalent to casting the parameter to a string.
Example:

```
log info Real.toString(-0.3)            # logs -0.3
log info Real.toString(1)               # logs 1.0
log info Real.toString(15.0000001)      # logs 15.0000001
```

Random

`trident-util:native@Random`
public class
A port of java.util.Random for Trident. This class provides several methods for generating random numbers.

`PROJECT_RANDOM : trident-util:native@Random`
public static final
A random number generator that is always consistent between compilations. It's seeded using the hash of the project's name or, alternatively, the `random-seed` key in the project properties file.

`new() : trident-util:native@Random`
public constructor
Creates a new random number generator, seeded with a value dependent on the current time.
`new(seed : int) : trident-util:native@Random`
public constructor
Creates a new random number generator with the given seed.

`nextInt` *(multiple overloads)*
`nextInt() : int`
public method
Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence. All 2^{32} possible int values are produced with (approximately) equal probability.
`nextInt(bound : int) : int`
public method
Returns the next pseudorandom, uniformly distributed int value between `0` (inclusive) and `bound` (exclusive), drawn from this random number generator's sequence. All `bound` possible int values are produced with (approximately) equal probability.

`nextReal() : real`
public method
Returns the next pseudorandom, uniformly distributed real value between `0.0` and `1.0` from this random number generator's sequence.

`nextGaussian() : real`
public method
Returns the next pseudorandom, Gaussian ("normally") distributed real value with mean `0.0` and standard deviation `1.0` from this random number generator's sequence.

`nextBoolean() : boolean`
public method
Returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence. The values `true` and `false` are produced with (approximately) equal probability.

CoordinateType

`trident-util:native@CoordinateType`
public final class
This class provides several constants used in the Coordinate Set methods.

`ABSOLUTE : trident-util:native@CoordinateType`
public static final
Represents the Absolute coordinate type.
`RELATIVE: trident-util:native@CoordinateType`
public static final
Represents the Relative coordinate type.
`LOCAL: trident-util:native@CoordinateType`
public static final
Represents the Local coordinate type.

RotationType

`trident-util:native@RotationType`
public final class
This class provides several constants used in the Rotation methods.

`ABSOLUTE : trident-util:native@RotationType`
public static final
Represents the Absolute rotation type.
`RELATIVE: trident-util:native@RotationType`
public static final
Represents the Relative rotationtype.

Axis

`trident-util:native@Axis`
public final class
This class provides several constants used in the Coordinate Set and Rotation methods.

`X : trident-util:native@Axis`
public static final
Represents the west-east axis.
`Y : trident-util:native@Axis`
public static final
Represents the down-up axis.
`Z : trident-util:native@Axis`
public static final
Represents the north-south axis.

Documentation Terms

1. *sequence-optional*: Arguments are sequence-optional if they can all be omitted - but in order to specify an argument, all arguments before it must also be specified, optional or not.
2. *get-set:* This means that internally, this member is not an editable symbol; but attempts to get and set the symbol may succeed. There is no guarantee that retrieving the value and editing it will change the actual value in the symbol it was retrieved from.
3. *top-level file*: A Trident function file that is not defined as an inner function of another, but rather as its own file.