# The TOO Programming Language Specification

TOO, acronym for Things Object Oriented, is a general purpose programming language (GPL) designed for citizen developers and professionals. It is tiny, but at the same time immensely powerful. TOO is strictly typed, event-driven, component-based, and object-oriented.

1. The following are the 9 syntax rules that define the grammar of TOO. Note that names in the EBNF below that start with a capital letter such as Id and Const are tokens and hence not outlined here:

```
thing ::= Id “{” (decl|when|event|func)* “}”
event ::= Id params
when ::= (Id [“[” decl “]”])+ Id params “{” act* “}”
func ::= Id params “➔” params [“{” act* “}”]
act ::= [“*”] expr [“?” “{” act* “}” [“{” act* “}”]]
expr ::= ((Const|Id|call) [“[” (expr|“*”) “]”] [“➔” (decl|params)])+
call ::= Id “(” [((expr|“*”) “,”)* (expr|“*”)] “)”
params ::= "(" [(decl “,”)* (["..."] decl)] ")"
decl ::= [“[]”] [Id “:”] Id+ [“=” Const] [Const]
```

2. The semantics behind the syntax rules.
   a. `when`: in the definition of a “`when`” syntax rule, the last identifier (`Id`) in the row denotes the “signal”, while the first identifiers with their optional subscripts denote the "source".
   b. `func`: “func” stands for function. The first “`params`” rule indicates the input parameters while the second “`params`” rule that follows the redirect operator (the arrow) indicates the output parameters. This means that functions in TOO language may return multiple values. If the action block is absent, the function is abstract.
   c. `act`: “act” stands for action. The condition operator (question-mark) indicates a conditional action. If present, then "expr" byproduct must result in a `boolean` thing. In such a case, the first action block is executed if this boolean evaluates to "true", and the second action block (if present) is executed otherwise. The optional iterative operator (asterisk) at the beginning indicates that the action should be executed indefinitely in a loop. In case the iterative operator appears together with the condition operator, the loop will break when reaching an empty/non-existing block (forming in this way the known while loop).
   d. `expr`: “expr” stands for expression. The “`expr`” syntax rule is composed of one or more sub-expressions. Only the first sub-expression may start with a “`Const`” token, and the “`call`” syntax rule cannot appear in the first sub-expression. The subscript operator (between the brackets) could only be used if the `(Const|Id|call)` evaluates

to an array (recall that we may declare an array using the array notation in the "`decl`" syntax rule). Last, the byproducts of this part, if exists, could be redirected by the redirect operator (the arrow) to a list of corresponding parameters. This is analogous to the := assignment operator in Go language.

e. `decl`: "decl" stands for declaration. The optional "`[]`" literal indicates an array (as opposed to a scalar). The first `Id` token before the colon indicates an alias name. The second `Id` token indicates the thing name. It might be a cascaded definition of minors (`Id+`). For example, in the following `foo` program:`foo {math pi}`, the declaration `math pi` refers to the minor `pi` which is declared in the `math` program: `math {pi:number}`. The `Const` token after the "`=`" literal is a string constant that denotes a JSON initializer. For example, `low:number=`113``, or `address=`{"city":"London", "street":"Downing", "number":10}``, where `address` is the following program: `address {city:string street:string number}`). The last Const token is a constant string that indicates an explicit path. For example, `math "/Stanford/Math"` means that thing `math` should be imported from "`/Stanford/Math`".

3. TOO discriminates between 4 types of things:

(a) minor vs. major. Referring to the "`thing`" syntax rule, the `Id` token denotes the major thing (also referred as the program), while the "`decl`" denotes a minor member.

(b) silent vs. non-silent. A thing is silent if it does not receive any signal (does not have a "when" syntax rule in the code), nor sends any signal (does not have an "`event`" syntax rule), recursively. Recursively means that in order to be silent, the minors must also be silent.

(c) abstract vs. non-abstract. An abstract thing may only contain abstract functions, with no "`when`" constructs, no "`event`" constructs, and no minors.

(d) local vs. remote.Remote thing is implemented by 3rd party and may run in a different location, on a different OS, developed in a different language, but it communicates with other things over the common network.

4. Non-silent things are executed as separate processes (Linux processes for example). Silent things cannot be run independently but must be embedded within other things. Non-silent things interact by communication via signals (the "`event`" and "`when`" syntax rules) and via functions that are activated by RPC-like mechanisms.

5. The things defined by the syntax rules are universal and intangible. When these things are instantiated during construction, they become tangible since they realize physical resources such as memory. instance could be unnamed ("_") or named ("Pod1").

6. TOO supports a single data-structure which is a set of minor instances referred to as members. Multiple data-structures are not allowed in a single program, nor nested data structures (e.g., `foo{bar{baz}}`). Each member could be a scalar or an array.

7. Arrays in TOO are implemented by maps. This means that each array element is named. the wildcard operator (star "*") can be used to obtain these named indices. For example, the "age" array in the following expression has two instances:

```
    age[“Bob”]=32, age[“Alice”]=34
```

Then the following expression:

```
     terminal Print(“name:”,*,“, age:”,age[*]),
```

Will print:

```
    name: Bob, age: 32
    name: Alice, age: 34
```

8. TOO sanitizes the code to maintain it universal. This means that all account-specific information is kept in a separate JSON file. For example, the code of following thermostat thing is universal:

```
thermostat {
    temperature:sensor
    switch

    temperature NotifiesLow () { switch TurnItOn () }
    temperature NotifiesHigh() { switch TurnItOff() }
}
```

While the JSON counterpart keeps the account-specific information. This JSON is referred to as the binding JSON and it specifies information regarding the instantiation of the minors. The following is an example for a binding JSON that could be associated with the thermostat thing.

```
------ ./OpenU/Project/thermostat/thermostat[_].json
{
  "path": "/OpenU/Project",
  "bindings": [{
    "id": "temperature:sensor",
    "instances":[{"id":"Como.T","attributes": {"initial": "", "auxiliary": ""}}]
  },{
    "id": "switch",
    "instances":[{"id": "Pod1", "attributes": {"initial": "", "auxiliary": ""}}]
  }]
}
```

This binding JSON means that the temperature:sensor is bound to instance "Como.T", and the switch is bound to instance "Pod1".

The information regarding the instantiation of the major thing(s) is found in another file, located one step up in the directory tree. This is another binding JSON named “bindings.json” and it  conforms to a similar schema.

For example,

```
------ ./OpenU/Project/bindings.json
{
  "path": "/OpenU/Project",
  "bindings": [{
      "id": "thermostat",
      "instances": [{"id": "_", "attributes": {"initial": "", "auxiliary": ""}}]
  }]
}
```

The following is the binding JSON schema (specified in Javascript):

```
export interface Bundle {
    path:       string
    bindings:   BoundThing[]
}

export interface BoundThing {
    id:         string
    instances:  Instance[]
}

export interface Instance {
    id:         string
    attributes: Attributes
}

export interface Attributes {
    initial:    string
    auxiliary:  string
}
```

9. Dependencies are resolved implicitly by proximity. Referring to the thermostat example, it depends on its two minors: sensor and the switch. Assuming that the thermostat is developed at directory `/Stanford/Class` and the sensor thing is found in both `/TOO/Sensor` and `/Stanford/Sensor`, then the compiler will automatically take the one implemented in `/Stanford/Sensor` since it's closer to `/Stanford/Class`. Nevertheless, the “`decl`” syntax rule

allows the developer to specify an explicit directory (in the last "`Const`" token) which overrides the implicit directory calculated by proximity. For example, the following directs the compiler to look for the switch code in an explicit directory "`/TOO/Core`":
`switch "/TOO/Core"`

10. Initialization in TOO is versatile.
TOO has a predefined `Initial` function that is called during construction. This function takes a JSON string and unmarshals this string to initialize the thing recursively (when the function is called, the minors of this thing are already constructed). The developer may decide that:

(a) Initialization is totally in the responsibility of the user, in which case the developer leaves this default code for Initial as is, or

(b) Initialization is partially in the responsibility of the user, allowing the developer to set some default values and permitting the user to override the defaults, in which case the developer inserts the initialization of the default values before the call to the unmarshal function, or

(c) Initialization is totally in the responsibility of the developer (such as in the case of initialization of the mathematical constant pi), in which case the developer completely removes the call to unmarshal and insert the initialization code (in the pi example, it will be `pi.Assign(3.1415)`).

11. Interfaces are similar to Go language interfaces. An interface is an abstract thing (see definition above). A thing is said to conform to the interface if it implements all the interface functions. The implemented function should match in name and parameters.

12. The NotifiesUp is a special event. Any non-silent thing has this event even if this event is not explicitly declared (using the "`event`" syntax rule). This event is triggered when the thing is up and running. The developer can use this rule for construction purposes that are not covered by the `Initial` function using a constant initializer. For example, referring to the thermostat above,

```
thermostat {
    switch
    temperature:sensor
    thermostat NotifiesUp() { switch TurnItOff() }
}
```

The developer can also trigger actions when its minors are up and running. For example, referring to the thermostat above, the thermostat makes sure that the switch is off when the switch comes up.

```
thermostat {
    terminal
    switch
    temperature:sensor
    switch NotifiesUp() {
        terminal Print(“Our switch is up and running”)
    }
}
```

13. The following are built-in functions that are included in every scalar thing. The developer may override them. For the clarification of the comments below, assume a “thermostat” thing with an instance named “Boiler”.

```
    Id() -> (string) // returns the instance name (thermostat Id() = “Boiler”)
    Cat() -> (string) // returns the category of the thing (thermostat Cat() =
“thermostat”)
    AsString() -> (string) // returns the JSON representation
    Marshal() -> (json:string)
    Unmarshal(json:string) -> (ok:boolean)
    Location() -> (location) // returns the location of this thing and zero
location if the thing does not have a location.
    Stop() -> () // stop the thing instance from running
    Exist() -> (boolean) // always returns true if the reference thing is a scalar
(thermostat Exist()). If the reference thing is an array, the function returns
true if the indexed element exists in the array (referring to the “age” example
below, age[“Washington”] Exist() will return false while age[“Bob”] Exist() will
return true).
    NotExist() -> (boolean) // the inverse of Exist
```

14. The following are built-in functions that are included in every array thing. The developer may override them. For the clarification of the comments below, assume an “age” array of numbers (`[]age:number`) that has two entries: `age[“Bob”]=34` and `age[“Alice”]=35`. Note that all functions are applied to the entire array and not to an element of the array. For example: `age Length()`.

```
    Cat() -> (string) // returns the thing name (age.Cat() will return “age”)
    Length() -> (integer) // returns the number of elements in the array (2)
    AsString() -> (string) // returns the array JSON ({“Bob”:34,“Alice”:35})
    Marshal() -> (json:string) // same as AsString
    Unmarshal(json:string) -> (ok:boolean) // replaces the array with a JSON
representation of another array ({“Barbie:22,“Ken”:22}). Return true if this
replacement completed successfully
```

```
    Empty() -> () // equivalent to a zero Length()
    Delete(key:string) -> (ok:boolean) // removes the element indexes by “key”.
Returns true if the key was found in the array and was successfully deleted.
    Instances() -> (o:[]string) // returns a new array that contains the names of
the keys (o[“0”]=“Bob”,o[“1”]= “Alice”)
    Sort(ascending:boolean) -> () // order the array for the range loop
(Sort(true) and then age[*] will range: “Alice”, “Bob”)
    InverseOrder() -> () // reverse the order of the elements in the array
    IsOrdered(ascending:boolean) -> (ordered:boolean)
    RenameInstance(old:string, new:string) -> (ok:boolean)
    Stop() -> ()
```

15. Synchronous vs. Asynchronous calls.
Asynchronous function calls do not have any return values and hence the RPC does not block the call. Synchronous calls are blocking, waiting for the returned values (or timeout). A call to a function of a silent thing is never blocking. A call to a function of a silent thing will never use RPC since the silent thing is embedded locally in the binary of the running process, as opposed to a function of a non-silent thing that runs in a separate process.

16. The signals that may trigger a “`when`” construct could be initiated by (a) a specific source, or (b) a finite set of sources, or (c) any source (wildcarded).
Example for (a):
`temperature[“Room12”] NotifiesHigh() {...}`

Example for (b):
`temperature[x] NotifiesHigh() {...}`, where `temperature` is an array of finite number of instances

Example for (c):
`temperature[x] NotifiesHigh() {...}`, where `temperature` is an array containing a single instance named “+” or “*”

17. Aliases (cyclic aliases, alias basis)
Alias refers to the Id in the “`decl`” that preceded the colon (“:”) symbol. It is used as an alternate name for clarifying the code, or as a way to solve ambiguity. For example, in case a program is working with two “`number`” things, one could have the “`low:number`” alias name, and the other could have the “`high:number`” alias name. The Id+ on the right side of the colon (“:”) symbol represents the alias base. Aliasing an alias is allowed. In the following example, “`pi:number`” and then “`archimedes:pi`”, `archimedes` aliases `pi` that aliases `number`. The base at the root of the hierarchy is referred to as the basis. In this case, the basis will be “`number`”).

18. Scoping rules define the visibility of variables across code blocks. They are similar to other block-structured programming languages. For example, the following expression with the redirect operator: `a Plus(b) -> c`, will create an instance of a temporary number that will cease to exist at the end of the innermost block `{}`. Usage of "`c`" in the block will refer to this temporary "`c`", even if "`c`" is defined also in outer blocks.

19. Temporary things can only be created by the redirect operator and they cease to exist then the execution block ends. Minor things persist for the entire duration of the program's execution.

20. Inheritance rules are similar to Go language. This means that the developer can refer to the minors of a minor (recursively) as if they are the root minors. The following example will print the same thing twice. The first print demonstrates a "has a" relationship; the laptop has a computer (the same way that it has a terminal), and the computer has a CPU. The second print demonstrates a "is a" relationship; laptop is a computer and hence has a CPU. Both relationships are legitimate and exist at the same time.

```
computer {
    cpu
    io
}

laptop {
    computer
    terminal
    laptop NotifiesUp() {
        terminal Print(computer cpu) // laptop has a computer
        terminal Print(cpu) // laptop is a computer
    }
}
```

21. TOO uses the CamelCase convention for things (e.g., `mainSwitch`), and the PascalCase for functions and events (e.g., `TurnItOn`). The GUI of the TOO platform converts the CamelCase to lowercase cyan with middle dots (e.g., `main•switch`), and the PascalCase to lowercase magenta with middle dots (e.g., `turn•it•on`).

22. TOO uses "//" for comments at the end of the line.