DTI Guide: C3 Types

Everything within the C3 AI Suite is managed through or exists as a C3 Type. Types allow users to describe, process, and interact with data and analytics. Broadly speaking, C3 Types are like Java classes, and include '`fields`', '`methods`', and an inheritance structure. Once you understand the structure of C3 Types you will be able to write Types to load data, run analytics, build and deploy machine learning models, and configure application logic for your research project.

All C3 Types are defined in a .c3typ file, stored in '`src`' directory of a package. A .c3typ file can only define a single C3 Type. The .c3typ file name must match the name of the C3 Type being defined.

In this section we will describe the package to download for this tutorial then we will discuss the syntax, special keywords, fields, methods, and inheritance structure of C3 Types. Finally, we will review some examples.

lightbulbAD Package

In this tutorial we will use the `lightbulbAD` package. To download the source code for this package, please follow the instructions available here: Guide to download C3 lightbulbAD Package.

Syntax

To help familiarize yourself with the syntax for a C3 Type, let's look at how the '`SmartBulb`' Type is defined in the lightbulbAD package:

```
/*
 * Copyright 2009-2020 C3 (www.c3.ai). All Rights Reserved.
 * This material, including without limitation any software, is the confidential trade secret and proprietary
 * information of C3 and its licensors. Reproduction, use and/or distribution of this material in any form is
 * strictly prohibited except as set forth in a written license agreement with C3 and/or its authorized
distributors.
 * This material may be covered by one or more patents or pending patent applications.
 */

/**
 * A single light bulb capable of measuring various properties, including power consumption, light output, etc.
 */
entity type SmartBulb extends LightBulb mixes MetricEvaluatable, NLEvaluatable type key "SMRT_BLB" {

  /**
   * This bulb's historical measurements.
```

```
 */
bulbMeasurements: [SmartBulbMeasurementSeries](smartBulb)

 /**
  * This bulb's historical predictions.
  */
 @db(order='descending(timestamp)')
 bulbPredictions: [SmartBulbPrediction](smartBulb)

 /**
  * This bulb's latest prediction.
  */
 currentPrediction: SmartBulbPrediction stored calc "bulbPredictions[0]"

 /**
  * This bulb's historical events.
  */
 bulbEvents: [SmartBulbEvent](smartBulb)

 /**
  * The latitude of this bulb.
  */
 latitude: double

 /**
  * The longitude of this bulb.
  */
 longitude: double

 /**
  * The unit of measurement used for this bulb's light output measurements.
  */
 lumensUOM: Unit

 /**
  * The unit of measurement used for this bulb's power consumption measurements.
  */
 powerUOM: Unit

 /**
  * The unit of measurement used for this bulb's temperature measurements.
  */
 temperatureUOM: Unit

 /**
  * The unit of measurement used for this bulb's voltage measurements.
  */
 voltageUOM: Unit

 /**
  * A SmartBulb is associated to a {@link Fixture} through a SmartBulbToFixtureRelation.
  */
 @db(order='descending(start), descending(end)')
 fixtureHistory: [SmartBulbToFixtureRelation](from)

 /**
  * The current Fixture to which this bulb is attached.
  */
 currentFixture: Fixture stored calc 'fixtureHistory[0].(end == null).to'

 /**
  * Method to determine the expected lumens of a light bulb
  */
 expectedLumens: function(wattage: !decimal, bulbType: !string): double js server

 /**
  * Returns the life span of this smartBulb
  */
 lifeSpanInYears: function(bulbId: string): double js server

 /**
  * Returns the average life span of all smartBulbs.
  */
 averageLifeSpan: function(): double js server

 /**
  * Returns the id of the smart bulb with the shortest recorded life span to date.
```

```
     */
    shortestLifeSpanBulb: function(): string js server

    /**
     * Returns the id of the smart bulb with the longest recorded life span to date.
     */
    longestLifeSpanBulb: function(): string js server

}
```

At a glance, a .c3typ file has the following components:

- **Keywords** which define the name, inheritance structure, and properties of a C3 Type.
- **Fields** which define attributes or data elements on a C3 Type.
- **Methods** which define business logic on C3 Types.
- **Annotations** like '@db', which often precede fields or methods and further configure a C3 Type.
- **Constants** such as strings or integers.
- **Comments** which aren't parsed by C3 and instead provide a message to the developer.

At a high level, the basic syntax to define C3 Types is as follows:

```
[remix, extendable] [entity] type TypeName extends, mixes AnotherType {
    /* comments */
    [field declaration]
    [method declaration]
}
```

Note: Everything within square brackets '[]' is optional.

Additional Resources

- C3 AI Developer Documentation:
  - https://developer.c3.ai/docs/7.12.25/topic/mda-types
  - https://developer.c3.ai/docs/7.12.25/topic/tutorial-c3-type-declaration
- C3.ai Academy videos:
  - The TypeSystem Module:
    - Types Overview
    - C3 Types
    - Type Definition

Keywords

A C3 Type definition is introduced with a series of keywords. These keywords tell the C3 AI Suite how to construct this Type, store it internally, and whether it inherits fields and methods from other already defined C3 Types.

- **type**: All .c3typ files have the keyword '`type`'. This keyword tells the C3 AI Suite that this file defines a C3 Type.
- **entity**: The keyword '`entity`' keyword indicates the C3 Type is a '`Persistable`' (i.e., needs to be stored in a database in the C3 AI Suite). Since a large majority of C3 Types are Persistable, '`entity`' is an important keyword to include in .c3typ files.
- **mixes**: Adding the keyword `` `mixes <AnotherType>` ``, a C3 Type inherits the properties (e.g., fields, methods) of `` `<AnotherType>` ''. A C3 Type can mix-in multiple Types.
- **remixes**: Adding the keyword`` `remix <AType>` `` allows a developer to modify an existing C3 Type (e.g., add new fields or methods, update existing fields or methods). Re-mixing is useful when you don't have access to the original .c3typ file for a particular C3 Type, but wish to edit that C3 Type.
- **extends**: Developers often add the keyword `` `extends <AnotherType>` `` to define a subclass of a particular C3 Type (e.g., SmartBulb extends Lightbulb). The extension Type (i.e., Smartbulb) inherits all the fields and methods of the original Type (i.e., Lightbulb). Additionally, data associated with the extension and original C3 Types are stored in the same database table. Therefore, you must specify a `` `type key` ``on the extension Type so the C3 AI Suite can distinguish data associated with the extension and original C3 Types. **Please note a C3 Type can only extend ONE other Type.**
- **extendable**: All extended Types (i.e., Lightbulb) MUST be marked with the keyword "`extendable`". Under the covers, the keyword "extendable" tells the C3 AI Suite to add a field (called '`key`') to the database table storing the extended (or base) C3 Type. This '`key`' field is used to distinguish data associated with different varieties of the extended C3 Type.
- **type key**: All extension Types (i.e., Smartbulb) MUST BE marked with the keyword '`type key <VALUE>`' (e.g., "type key SMRT_BLB").
- **schema name**: A set of keywords indicating the name of the database table used to store data for a Persistable C3 Type. Developers specify a schema name to customize database table names.

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/topic/mda-types
  - https://developer.c3.ai/docs/7.12.25/topic/mda-type-composability
- C3.ai Academy Videos
  - The TypeSystem Module:
    - Persistable Types
    - Type Inheritance

C3 Types

There are many categories of C3 Types.

We will cover the most commonly used categories in this tutorial:

- Primitive Types
- Persistable Types
- Generic (Parametric) Types
- Abstract Types

Primitive Types

Like many programming languages, the C3 AI Suite has primitives. Primitives define the units (or data types) of fields on a C3 Type (e.g., "int", "double", "float"). The C3 AI Suite includes a number of primitives, listed below for reference. Adding new primitives will require support from the C3 AI engineering team, however, most DTI researchers should be able to progress their research projects with the C3 AI Suite's existing primitives.

- binary
- boolean
- byte
- char
- datetime
- decimal
- double
- float
- int
- json
- long int
- string

C3 Also supports certain composites of primitive types. For example, the C3 Type `Mapp` is a map between values and keys. We can create a map directly in python in two ways:

1. Let C3 Figure out the type for us: `c3.MappObj(value=python_obj)` The new `Mapp` object can be accessed through the `.value` property of the `MappObj`
2. Tell C3 exactly which types are supported with `c3.Mapp(c3.MappType.fromString("map<string,any>"), {'a': 5, 'b': [1,2,3]})`

Similarly, C3 has the `Arry` type which supports plain arrays. We can specify them in the same ways.

1. Let C3 Figure out the type for us: `c3.ArryObj(value=python_list)` The new `Arry` object can be accessed through the `.value` property of the `ArryObj`
2. Tell C3 exactly which types are supported with `c3.Arry(c3.ArryType.fromString('[int]'), [1,2,3,4])`

In both cases, these primitive types aren't persisted. They can be used anywhere a C3 Type specifies it must have an `Obj` type. (Especially useful for `Dyn BatchJobs` or `DynMapReduce` jobs for contexts).

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/type/PrimitiveType
- C3.ai Academy Videos
  - The TypeSystem Module:
    - Primitive Fields

Persistable Types

The most common C3 Type you will define is a '`Persistable`' Type. Persistable Types store data in a C3 AI Suite database. By default, all Persistable Types are stored in Postgres though they can also be stored in file systems (e.g., Azure Blob) or key-value stores (e.g., Cassandra), by adding an annotation to your .c3typ file (discussed below).

When defining a Persistable type, you MUST add the following two key terms to your .c3typ file:

1. **entity:** The keyword '`entity`' keyword tells the C3 AI Suite this is a 'Persistable' Type.
2. **schema name:** The schema names is the name of the database table where the C3 Type's data are stored. When defining a new Persistable Type, you need to add the keywords '`schema name`' followed by your chosen table name, in your .c3typ file. (e.g., in the C3 Type LightBulb, we have '`schema name "LGHT_BLB"`').
   Note: Extended C3 Types DO NOT NEED a scheme name and Schema names CANNOT exceed 30 characters.

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/topic/mda-types
  - https://developer.c3.ai/docs/7.12.25/topic/tutorial-persistable-types
- C3.ai Academy Videos

        - The TypeSystem Module:
            - Persistable Types

Generic (Parametric) Types

Like a Java or C++ Class, C3 Types can be parameterized (or genericized). In fact, the C3 AI Suite uses the exact same syntax as Java and C++ to define a Type's parameters (i.e., angle brackets '<>'). When defining a Generic (Parametric) Type, your Type name will be followed by angle brackets and a comma-separated list of parameters (usually other C3 Types). For example:

```
type TheType<V,U> {
        fieldA: V
        fieldB: [U]
}
```

Then, when using your Parametric Type in other places, you must specify the arguments for each parameter in the angle brackets. As example, you may define a new C3 Type with the following field:

```
type NewType {
        newField: TheType<TypeA, TypeB>
}
```

If your C3 Types will be heavily re-used by other developers you should consider using Parametric (Generic) Types.

Additional Resources

- Developer Documentation
    - https://developer.c3.ai/docs/7.12.25/topic/mda-types
    - https://developer.c3.ai/docs/7.12.25/type/Generic
- C3.ai Academy Videos
    - The TypeSystem Module:
        - Generic Types

Fields

As discussed above, a C3 Type mainly consists of two components: fields and methods.

Fields are attributes or properties of a C3 Type. To define a field on C3 Type, use the following syntax: **field name,** followed by a colon **':',** followed by a **ValueType** (e.g., "lumens: int"). By convention, field names are camelCase.

A C3 Type can have many different kinds of fields. Here are the four most commonly used fields:

- Primitive Fields
- Reference Fields
- Collection Fields
- Calculated Fields

Primitive Fields

Like many other programming languages the C3 AI Suite has primitive fields (e.g., int, boolean, byte, double, datetime, string, longstring). For example:

```
doubleField: double
intField: int
```

Primitive fields are stored in a particular C3 Type's database table.

Reference Fields

Reference fields point (or refer) to other C3 Types. Reference fields link (or join) two C3 Types together. Under the covers, the Reference field stores a pointer (i.e., foreign key reference or ID) to record of another C3 Type. To define a Reference field, use the following syntax: **field name,** followed by colon **':'**, followed by **Type Name** (e.g., "building: Building"). For example:

```
refField: AnotherType
```

Collection Fields

Collection fields contain a list of values or records. There are four categories of collections fields:

- **array**: denoted by `` `[MyType]` `` (contains an array of references to '`MyType`').
- **map**: denoted by `` `map<int, MyType>` `` (left argument is the key which can be any primitive type, right argument can be any C3 Type).

- **set**: denoted by `set<MyType>`.
- **stream**: denoted by `stream<MyType>`. (This is a read-once type of collection).

Collection fields can also be used to model one-to-many and many-to-many relationships between two C3 Types**.** To do so, we use the **Foreign Key Collection Notation,** which specifies the foreign key by which two C3 Types (i.e., ThisType and AnotherType) are joined.

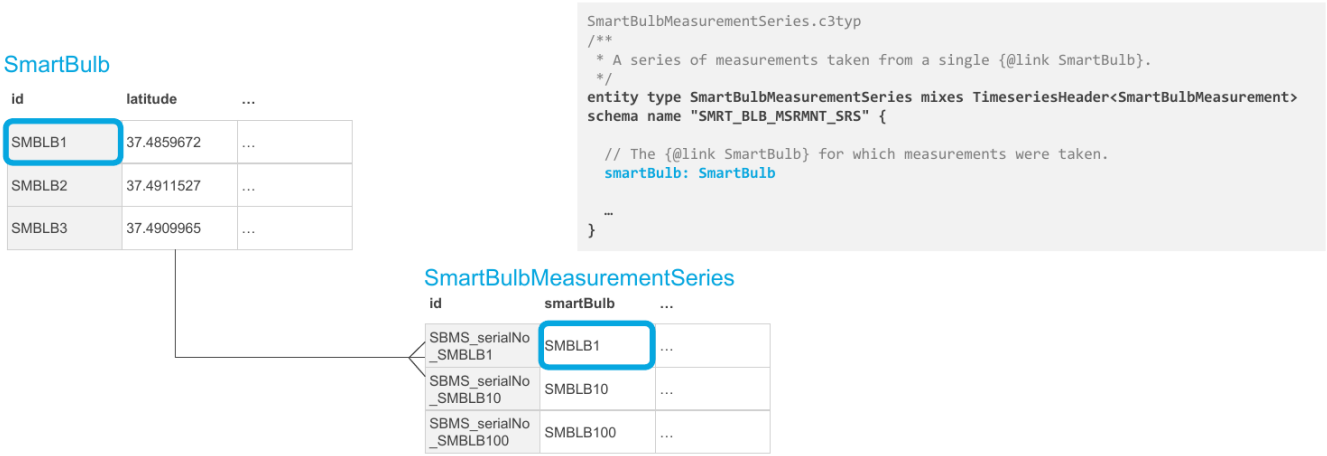
To define this annotation use the following syntax: `"fieldName : `[AnotherType]` (fkey, key)"` where:

- 'fkey' is a field on `AnotherType' to use as the foreign key
- 'key' is an optional field on ThisType, whose records should match those of the 'fkey' field on 'AnotherType' (defaults to `id` field of ThisType, if not specified).
- In other words, the Collection field will contain pointers to all records in AnotherType, where ThisType.key == AnotherType.fkey.

For example:

`fieldName: [AnotherType] (fkey[, key])`

Shown in the example code and diagram below, the field `bulbMeasurements`, contains a list of all the `SmartBulbMeasurementSeries` records, where `SmartBulb.id == SmartBulbMeasurements.smartBulb` (e.g., a list of various measurements relevant to a SmartBulb, like temperature or power).



Please see the 'Foreign Key Collection and Schema Names" section in the following C3 AI Develop Documentation for more details: https://developer.c3.ai/docs/7.12.25/topic/mda-fields

Calculated Fields

Calculated fields are derived from other fields on a C3 Type. Calculated fields typically included either JavaScript code or C3 AI ExpressionEngineFunctions and are specified as follows:

`fieldName: FieldType [stored] calc "field_formula"`

There are two types of calculated fields:

1. **Simple calculated fields**: The values of these fields are calculated at runtime and not stored. To define a simple calculated field use the following syntax: keyword **`calc`,** followed by an expression formula.
2. **Stored calculated fields**: The values of these fields are stored in the C3 Type's database table. To define a stored calculated field use the following syntax: keywords **`stored calc`**, followed by an expression formula (e.g, in the SmartBulb example above, '`stored calc fixtureHistory[0].(end == null).to`').

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/topic/mda-types
  - https://developer.c3.ai/docs/7.12.25/topic/mda-fields
  - https://developer.c3.ai/docs/7.12.25/topic/tutorial-types-of-fields
  - https://developer.c3.ai/docs/7.12.25/type/Field
  - https://developer.c3.ai/docs/7.12.25/type/PrimitiveType
- C3.ai Academy Videos
  - The TypeSystem Module:
    - Types of Fields
    - Primitive Fields
    - Reference Fields
    - Collection Fields
    - (Stored) Calculated Fields

Methods

At a high-level, methods are pieces of code that take in arguments or parameters and return new values. To define a method on a C3 Type, use the following syntax, **method name** followed by a colon '**:**' followed by a **function signature**, (e.g., function() or function(wattage: !decimal, bulbType: !string)), followed by a colon '**:**', followed by a **return parameter**, along with an **implementation language** and **execution location**.

Function Signatures

Here's the syntax for a method's function signature:

```
function(parName1: [!] parType1, parName2: [!] parType2):
```

We see the keyword **'function'** followed by a comma-separated list of input parameters. To define an input parameter, use the following syntax: **argument name** followed by a colon **':',** followed by **argument type** (e.g., bulbID: string). Optionally, you can also add an exclamation point '!' before the argument type to indicate the argument is required.

The function itself is NOT implemented in the .c3typ file, rather in a separate .js (JavaScript), .py (Python) or .r (R) files stored in the same directory as the .c3typ file.

Return Parameter

Following the function signature is a return parameter. Return parameters must be a C3 Type.

Implementation Language/Execution Location

Finally, let's discuss the implementation language and execution location.

Language Specifications

Methods can be implemented in JavaScript, Python, or R:

- **py**: Use the 'py' keyword to indicate this is a Python method.
- **js**: Use the 'js' keyword to indicate this is a JavaScript method.
- **r:** Use the 'r' keyword to indicate this is an R method.

Currently, these are the only three programming languages natively supported by the C3 AI Suite to define methods.

Execution Location

Methods can be executed in three locations:

- **server**: Indicates the method will run on the C3 AI Suite by a worker node. Server should be used for methods requiring high processing power (e.g., methods that fetch or create large amounts of data).
- **client**: Indicates the method will run on the client's browser.
- **all:** Indicates the method will run wherever it is called from. If the call comes from the browser, the function will execute in the browser. If the call comes from a server, the execution will happen in the server.

Please note, R and Python methods can only be executed on the **server**. Javascript methods can be executed in either the **server** or **client.**

Additionally, for Python, we need to specify a valid '`ActionRuntime`' for python function to be executed in. ActionRuntimes are defined in C3 AI Suite and new ActionRuntimes can be defined in a package and are essentially conda environments. For example, '`server`' will use the '`py-server`' ActionRuntime.

Function Definitions

As mentioned above, the function itself is defined in a different file from the .c3typ file. The name of this file should be the same as the .c3typ file where the method is defined (e.g., SmartBulb.js and SmartBulb.c3typ), but have a different extension, per the function's implementation language (e.g., 'py' for Python, 'js' for JavaScript, 'r' for R). The function names in this file must exactly match the method names in the .c3typ file. For example, we define a C3 Type as such:

```
entity type NewType {
        field: double
        funcA: function(num: !int) : float js server
    funcB: function(num: !int, name: string) : double py server
}
```

This C3 Type will be defined in a file called 'NewType.c3typ'. We also need two other files, 'NewType.js' and 'NewType.py', in the same directory as 'NewType.c3typ'. Here's a peek inside:

'NewType.js'

```
function funcA(num) {
    ...
}
```

'NewType.py'

```
function funcB(num, name='default') {
    ...
}
```

Additional Resources

- Developer Documentation
    - https://developer.c3.ai/docs/7.12.25/topic/mda-fields
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-home
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/defining-methods
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-implementing-methods
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-required-parameters
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-overloading-methods
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-overriding-methods
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-method-types
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-method-error-handling
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-testing-methods
    - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/methods-debugging-methods
- C3.ai Academy Videos
    - C3 Fundamentals **'Methods' Module**:
        - Methods Overview
        - Declaring Methods
        - Required Parameters
        - Implementing Methods
        - Testing Methods
        - Debugging Methods

Inheritance

Like a Java or C++ Class, a C3 Type can inherit fields and methods from other C3 Types. The keywords`**mixes**` and `**extends**` signal inheritance. These keywords follow the new C3 Type's name in your .c3typ file.

These two keywords signal different kinds of Type inheritance.

Mix-ins

Mix-ins are the most common type of inheritance and signaled by the '`mixes`' keyword. When a C3 Type 'mixes' another C3 Type, it inherits all the fields and methods of the original C3 Type. In general, mix-ins are used to incorporate non-persistable Types. The new C3 Type is stored in a new table and has its own schema.

```
type ParentType {
    fieldA: int
    funcA: function(): ReturnType
}

type ChildType mixes ParentType {
    fieldB: double
    funcB: function(): ReturnType
}
```

In this example, ChildType mixes ParentType, and inherits 'fieldA' and 'funcA' defined on ParentType.

Persistable Inheritance

Persistable inheritance is signaled by the '`extends`' keyword. When a C3 Type '`extends`' another C3 Type, it inherits all the original C3 Type's fields and methods. The extension and original C3 Types must be marked with '`entity`' keyword and share the same database table. To distinguish data from the original and extension C3 Types, the C3 AI Suite adds a '`type key`' field to this database table. Additionally, only C3 Types marked with the `` `extendable` `` keyword can be extended.

```
extendable entity type ParentType schema name "PRT_TYP" {
    fieldA: int
    funcA: function(): ReturnType
}

entity type ChildType extends ParentType type key "CHLD" {
    fieldB: double
    funcB: function(): ReturnType
}
```

In this example, ParentType is an extendable Persistable type stored in the table '`PRT_TYP`'. ChildType extends ParentType and is also stored in the table '`PRT_TYP`'. Records associated with ChildType have '`type key' == 'CHLD`' to distinguish them from ParentType's data.

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/topic/mda-type-composability
  - https://developer.c3.ai/docs/7.12.25/topic/mda-schema-name-and-type-key
  - https://developer.c3.ai/docs/7.12.25/topic/tutorial-type-inheritance
- C3.ai Academy Videos
  - C3 Fundamentals **'TypeSystem' Module**:
    - Type Inheritance
    - Schema Name and Type Key
    - Type Composability

Annotations

With annotations you can further configure a C3 Type. Annotations can appear before Type definitions, field declarations, and method declarations. As example, annotations can be used to specify the database technology used to store a C3 Type (e.g., Cassandra, Azure Blob), mark the ActionRuntime in which to run a particular method, deprecate a field or method, specify how time-series data should be treated, and specify how to trigger analytics or DFEs (DataFlow Events).

Here are a few examples of defining annotations:

```
  /**
   * This bulb's historical predictions.
   */
  @db(order='descending(timestamp)')
  bulbPredictions: [SmartBulbPrediction](smartBulb)
```

With the `` `@db(order = 'descending(timestamp)'` `` annotation, the bulbPrediction field's records are stored in descending order by timestamp.

```
@ts(treatment='avg')
lumens: double
```

With the `` `@ts(treatment = 'avg')` `` annotation, the value of the lumens field for a given SmartBulb is the average of all that Smartbulb's lumens data.

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/topic/mda-annotations
  - https://developer.c3.ai/docs/7.12.25/type/Annotations
- C3.ai Academy Videos
  - C3 Fundamentals **'TypeSystem' Module**:
    - Annotations
    - Db Annotations

Final Types

Types, methods, and fields can be made *final* using the '`final`' keyword. The fields and methods of a Final Type cannot be modified or overridden (via 'mixes' or 'exends').

```
type FinalType {
    final method: function(): string
}
```

Additional Resources

- Developer Documentation
    - https://developer.c3.ai/docs/7.12.25/topic/mda-types

Abstract Types

The C3 AI Suite also supports abstract types. To define an Abstract Type, add the keyword '`abstract`' before the Type definition. Abstract Types cannot be instantiated and must mix-in to another concrete Type. This provides a great way to standardize interfaces. Here's an example:

```
abstract type Interface {
        field1: int
    method1: function(double): int
}
```

Additional Resources

- Developer Documentation
    - https://developer.c3.ai/docs/7.12.25/topic/mda-types
- C3.ai Academy Videos
    - C3 Fundamentals **TypeSystem' Module**

Examples

To see more examples of C3 Types, check out the .c3typ files in the lightbulbAD package: Guide to download C3 lightbulbAD Package.