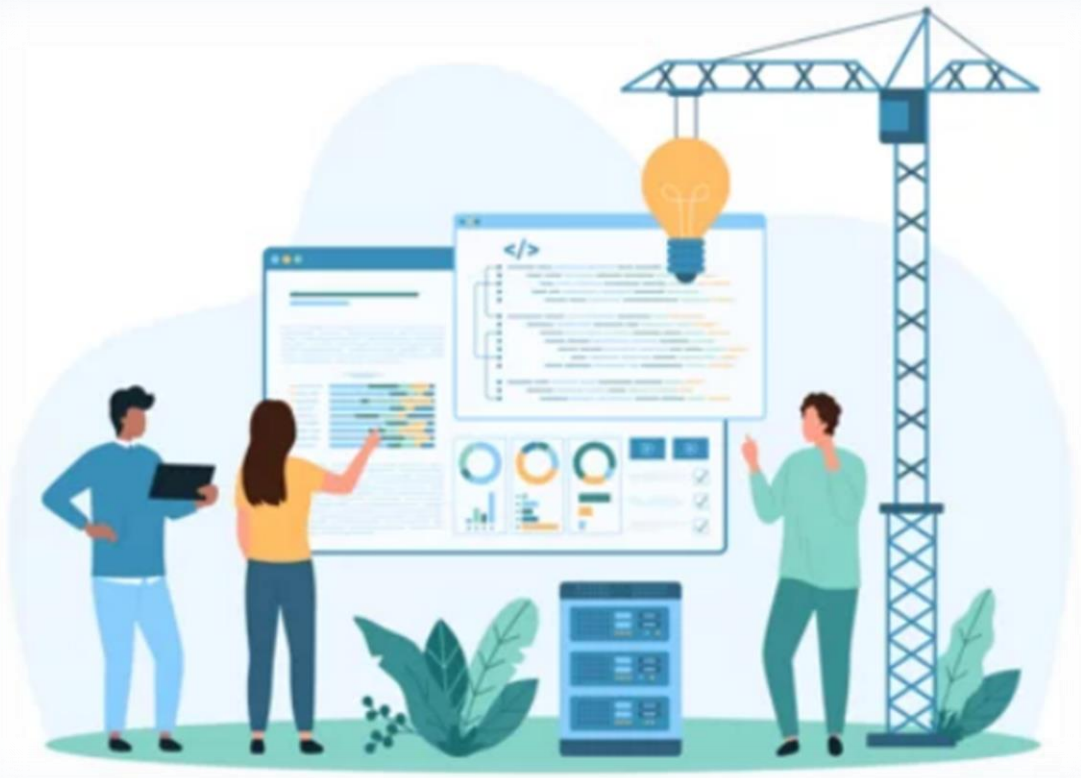
Foundry TypeScript Cheat Sheet

TABLE OF CONTENTS



Foundry TypeScript Cheat Sheet

Developers and Data Analysts are encouraged to use this Foundry TypeScript Cheat Sheet as an everyday reference guide for coding transformations in 1CDP.

IMPORT

Imports allow you to bring code, functions, classes, or modules from another file or library into the current code file.

ONTOLOGY API

```
import {X} from "@foundry/ontology-api";
```

Based on what you wish to import, 'X' in the above code snippet may be:

- Objects
- ObjectSet
- ObjectType

***Note**: This applies only to Ontology objects imported into your Code Repository for access.*

FUNCTIONS API

```
import {X} from "@foundry/functions-api";
```

Based on what you wish to import, 'X' in the above code snippet may be:	
Function	A code-based logic that takes input parameters and returns an output, integrated with the Ontology to interact with objects/object sets and used across dashboards and apps.
OntologyEditFunction	A function that creates, modifies, or deletes Ontology objects. Decorated with @OntologyEditFunction() and have an explicit void return type.
Integer	A data type representing whole numbers without a fractional component, used in various functions and data models.
Long	A data type representing a 64-bit integer, used for larger whole numbers exceeding the range of standard integers.
Float	A data type representing a single-precision 32-bit floating point number, used for decimal values requiring less precision.
Double	A data type representing a double-precision 64-bit floating point number, used for decimal values requiring more precision.
LocalDate	A data type representing a date without a time-zone in the ISO-8601 calendar system, used for date-related properties and functions.
Timestamp	A data type representing a point in time, typically including both date and time components, used for time-related properties and functions.
User	An entity representing an individual with access to the platform, often used in functions and permissions configurations.

Continued on the following page…

Foundry TypeScript Cheat Sheet

Developers and Data Analysts are encouraged to use this Foundry TypeScript Cheat Sheet as an everyday reference guide for coding transformations in 1CDP.

IMPORT

FUNCTIONS API (CONTINUED…)

```
import {X} from "@foundry/functions-api";
```

Based on what you wish to import, 'X' in the above code snippet may be:	
TwoDimensionalAggregation	A data structure used to aggregate data across two dimensions, often used in analytical contexts.
ThreeDimensionalAggregation	A data structure used to aggregate data across three dimensions, providing more complex analytical capabilities.
Attachment	A data type representing a single file or document that can be associated with an object, used for storing and retrieving files.
Attachments	A collection of Attachment objects allowing for multiple files or documents to be associated with an object.
MediaItem	A data type representing a piece of media, such as an image or video, used for storing and displaying media content.
Notification	A mechanism for alerting users about events or changes, often used in workflows and operational contexts.
BasicEmailNotificationContent	A data structure used to define the content of a basic email notification, including subject and body text.
BasicShortNotification	A data structure used to define the content of a short notification, typically for quick alerts or messages.
Filters	A functions API export used to combine multiple filters using and, or, and not.
UserFacingError	A type of error designed to be displayed to end-users, providing clear and actionable error messages.

APPLY DECORATORS

Decorators are functions applied to a declaration using the @ symbol, which allow you to modify or add functionality to classes, methods, properties, or parameters.

@OntologyEditFunction(): A type of function allowing you to create, modify, or delete objects within the Ontology.

@Function(): Code-based logic that takes input parameters and returns an output. Functions are natively integrated with the Ontology, meaning they take in objects/object sets and read property values for use across action types.

Foundry TypeScript Cheat Sheet

Developers and Data Analysts are encouraged to use this Foundry TypeScript Cheat Sheet as an everyday reference guide for coding transformations in 1CDP.

FILTER

Filters return an array or Object list that match the condition(s) provided.

BASIC FILTERS

Filter based on a single property

```
const aggregation = Objects.search().exampleDataAircraft()
                .filter(aircraft => aircraft.arrivalCity.exactMatch('NYC'))
```

ADVANCED FILTERS

Filter based on multiple properties, using "Or" /"And" statements to achieve a more robust filter

```
const filteredFlights = Objects.search()
            .flights()
            .filter(flight => Filters.or(
                Filters.and(flight.destination.exactMatch("SFO"),
flight.passengerCount.gt(100)),
                Filters.and(flight.destination.exactMatch("LAX"),
flight.passengerCount.gt(300))
            ));
```

FILTER OPTIONS

The following filter options can be used based on your needs:

Matching Filter
exactMatch()

**Supported by all types*

Combining Filters
- Filters.or()
- Filters.and()
- Filters.not()

**Filters must be imported from @foundry/functions-api*
**Multiple filters are separated by a comma within the ()*

Array Filter
contains()

String Filters
- phrase()
- phrasePrefix()
- prefixOnLastToken()
- matchAnyToken()
- fuzzyMatchAnyToken()
- matchAllTokens()
- fuzzMatchAllTokens()

Geohash Filters
- withinDistanceOf()
- withinPolygon()
- withinBoundingBox()

Numbers, Dates, and Timestamp Filters
range()
- lt() – Less than
- lte() – Less than or equal to
- gt() – Greater than
- gte() – Greater than or equal to

Linking Filter
.isPresent()

**Checks if there is an active link*

GeoShape Filters
- withinBoundingBox()
- intersectsBoundingBox()
- doesNotIntersectBoundingBox()
- withinPolygon()
- insersectsPolygon()
- doesNotIntersectPolygon()

Boolean Filters
- isTrue()
- isFalse()

Foundry TypeScript Cheat Sheet

Developers and Data Analysts are encouraged to use this Foundry TypeScript Cheat Sheet as an everyday reference guide for coding transformations in 1CDP.

APPLY OPERATIONS

Operations are fundamental code building blocks used to manipulate variables, perform calculations, and compare values.

STRING OPERATIONS

- **charAt(index):** Returns the character at the specified index.
- **charCodeAt(index):** Returns the Unicode of the character at the specified index.
- **concat(...strings):** Combines the text of two or more strings and returns a new string.
- **endsWith(searchString, length?):** Determines whether a string ends with the characters of a specified string.
- **includes(searchString, position?):** Determines whether one string may be found within another string.
- **indexOf(searchValue, fromIndex?):** Returns the index within the calling string object of the first occurrence of the specified value.
- **lastIndexOf(searchValue, fromIndex?):** Returns the index within the calling string object of the last occurrence of the specified value.
- **match(regexp):** Retrieves the matches when matching a string against a regular expression.
- **matchAll(regexp):** Returns an iterator of all results matching a string against a regular expression, including capturing groups.
- **replace(searchValue, replaceValue):** Returns a new string with some or all matches of a pattern replaced by a replacement.
- **replaceAll(searchValue, replaceValue):** Returns a new string with all matches of a pattern replaced by a replacement.
- **search(regexp):** Executes a search for a match between a regular expression and this string object.
- **slice(beginIndex, endIndex?):** Extracts a section of a string and returns it as a new string.
- **split(separator, limit?):** Splits a string object into an array of strings by separating the string into substrings.
- **startsWith(searchString, position?):** Determines whether a string begins with the characters of a specified string.
- **substring(indexStart, indexEnd?):** Returns the part of the string between the start and end indexes, or to the end of the string.
- **toLocaleLowerCase(locales?):** The characters within a string are converted to lower case while respecting the current locale.
- **toLocaleUpperCase(locales?):** The characters within a string are converted to upper case while respecting the current locale.
- **toLowerCase():** Returns the calling string value converted to lower case.
- **toString():** Returns a string representing the specified object.
- **toUpperCase():** Returns the calling string value converted to uppercase.
- **trim():** Removes whitespace from both ends of a string.
- **trimEnd():** Removes whitespace from the end of a string.
- **trimStart():** Removes whitespace from the beginning of a string.

MATHEMATICAL OPERATIONS

- **Math.ceil(x):** Returns the smallest integer greater than or equal to a number.
- **Math.cos(x):** Returns the cosine of a number.
- **Math.exp(x):** Returns e raised to the power of a number.
- **Math.floor(x):** Returns the largest integer less than or equal to a number.
- **Math.max(...values):** Returns the largest of zero or more numbers.
- **Math.min(...values):** Returns the smallest of zero or more numbers.
- **Math.pow(base, exponent):** Returns base to the exponent power.
- **Math.random():** Returns a pseudo-random number between 0 and 1.
- **Math.round(x):** Returns the value of a number rounded to the nearest integer.
- **Math.sign(x):** Returns the sign of a number, indicating whether it is positive, negative, or zero.
- **Math.sqrt(x):** Returns the square root of a number.
- **Math.trunc(x):** Returns the integer part of a number by removing any fractional digits.

Foundry TypeScript Cheat Sheet

Developers and Data Analysts are encouraged to use this Foundry TypeScript Cheat Sheet as an everyday reference guide for coding transformations in 1CDP.

MAPS (DICTIONARIES)

Maps are a data structure allowing you to store key-value pairs, like dictionaries in other programming languages.

- **clear():** Removes all elements from the map object.
- **delete(key):** Removes the specified element from a map object by key.
- **entries():** Returns a new iterator object that contains an array of [key, value] for each element in the map object.
- **forEach(callback, thisArg?):** Executes a provided function once for each key/value pair in the map object.
- **get(key):** Returns the value associated with the specified key, or undefined if the key does not exist.
- **has(key):** Returns a boolean indicating whether an element with the specified key exists in the map object.
- **keys():** Returns a new iterator object that contains the keys for each element in the map object.
- **set(key, value):** Adds or updates an element with a specified key and value to the map object.
- **values():** Returns a new iterator object that contains the values for each element in the map object.
- **size:** Returns the number of elements in a map object.

SET METHODS

Sets are a data structure allowing you to store a collection of distinct elements of the same type.

- **add(value):** Adds a new element with the given value to the set.
- **clear():** Removes all elements from the set.
- **delete(value):** Removes the specified element from the set.
- **entries():** Returns a new Iterator object containing an array of [value, value] for each element in the set.
- **forEach(callback, thisArg?):** Executes a provided function once for each value in the set.
- **has(value):** Returns a boolean indicating whether an element with the specified value exists in the set.
- **keys():** Returns a new iterator object with the values for each element in the set.
- **values():** Returns a new iterator object with the values for each element in the set.

ARRAY/OBJECT SET METHODS

Arrays are a data structure allowing you to store a collection of elements of the same type.

- **Filter:** Creates a new array with all elements that pass the test implemented by the provided function, often used to filter objects based on specific criteria.
- **forEach:** Executes a provided function once for each array element, commonly used to iterate over arrays and perform operations on each element.
- **Includes:** Determines whether an array includes a certain value among its entries, returning true or false as appropriate.
- **Join:** Creates and returns a new string by concatenating all the elements in an array, separated by commas or a specified separator string.
- **Keys:** Returns a new array iterator object that contains the keys for each index in the array.
- **Map:** Creates a new array populated with the results of calling a provided function on every element in the calling array. It can also refer to FunctionsMap for mapping keys to values.
- **Pop:** Removes the last element from an array and returns that element. This method changes the length of the array.
- **Push:** Adds one or more elements to the end of an array and returns the new length of the array.
- **Reduce:** Executes a reducer function (that you provide) on each element of the array, resulting in a single output value.
- **toLocaleString:** Returns a string with a language-sensitive representation of this number. It can be used to format numbers or dates according to locale-specific conventions.
- **Values:** Returns a new array iterator object that contains the values for each index in the array.

Foundry TypeScript Cheat Sheet

Developers and Data Analysts are encouraged to use this Foundry TypeScript Cheat Sheet as an everyday reference guide for coding transformations in 1CDP.

AGGREGATE

Aggregates are functions that combine multiple values into specified summary statistics grouped by common values.

```
public async aircraftAggregationExample():
Promise<TwoDimensionalAggregation<string>> {
        const aggregation = Objects.search().exampleDataAircraft()
                .filter(aircraft => aircraft.arrivalCity.exactMatch('NYC'))
                .groupBy(aircraft => aircraft.departureCity.topValues())
                .count();

        return aggregation;
```

LINK

Links are the relationship between two Object types, which can be 1 to Many or Many to Many

CREATE LINK

```
// 1:1 Link
employee.department.set(department)

// 1:Many/Many:Many Link
employee.projects.add(project)
```

REMOVE LINK

```
// Remove Links
employee.department.clear()
```

LOOP

Loops are a sequence of instructions that repeat until a condition is met.

```
// A For Loop that sets every promotionFlag property to 'Yes'
employees.forEach(e => {
            e.promotionFlag = 'Yes')
        })
```

CHECK LOGS

Logs are a record containing information about the system and program's execution, which are helpful for debugging.

```
console.log(n)
```

***Note:** log displays whatever the 'n' variable is*

EDIT OBJECTS

Edit Objects are functions that can be used to edit Ontology Objects through TypeScript.

CREATE NEW OBJECT

```
// Creating new Object with Unique PK ID
    const newEmployee = Objects.create().employee(Uuid.random())
```

DELETE OBJECT

```
// Deleting an existing Object
    let oldEmployee = Objects.search().employee().filter(e =>
e.id.exactMatch('100000')).all()[0];

    oldEmployee.delete()
```

SET PROPERTY VALUE

```
// Declare New Variable with Type String
    let newEmployeeFirst: string = 'Alex'

// Constant Variable using Object Search and Filter
    const employee = Objects.search().employee().filter(e =>
    e.id.exactMatch('100000')).all()[0]

// Set Property Value to new variable
    employee.firstName = newEmployeeFirst
```

Foundry TypeScript Cheat Sheet

This page differentiates and explains various elements of TypeScript syntax for your reference:

1. Import the Function Decorator and Integer, FunctionsMap types from @foundry/functions-api

2. Import Ontology Objects from @foundry/ontology-api

3. Decorator - can be either @OntologyEditFunction(), used when making edits to an Object/the Ontology, or @Function(), used for anything else

4. Declares whether a function is **Public** (able to be referenced outside the current TypeScript repository) or **Private** (not able to be referenced outside the repository, used if you want specific segments of code to be hidden)

5. Keyword used for **asynchronous** functions, which saves time and computation power for larger functions

6. Function name

7. Input variables - variable name is separated from return type by a colon and multiple variables are separated by a comma

8. Return type - use “Promise<Type>” for async and “void” for @OntologyEditFunctions()

9. Keyword (i.e., const, let, var) which has a special meaning and cannot be used as an identifier (variable names, function names)

10. Iterates the function over all objects in a set, one at a time

11. Get all **linked Objects** from an individual iterator Object

12. If block - a conditional statement that runs the code block if the specified condition is met

13. Else block – a conditional statement that runs the code block only when the if conditional is not met

14. Return value - the output of the function

```
1  import { Integer, FunctionsMap, Function } from "@foundry/functions-api"
2  import { RedRash, RashLinkedLocation, ObjectSet} from "@foundry/ontology-api"

3  @Function()
4  public 5 async 6 mapCounts(7 laws: ObjectSet<RedRash>): 8 Promise<FunctionsMap<string, Integer>> {
9      const map = new FunctionsMap<string, Integer>();
       await case.all().forEach(c => 10
11         c.rashLinkedLocation.all().forEach(
               key => {
                   let id = key.mapboxId!
12                 if (map.has(id)){
                       let count = map.get(id)! + 1
                       map.set(id, count)
                   }
13                 else {
                       map.set(id, 1)
                   }
               }
           )

       })
14     return map;
   }
```

Questions? Please contact your project lead