# DEFINING ACTIONS, CONDITIONS AND EXPRESSIONS

**View online:** https://www.construct.net/en/make-games/manuals/addon-sdk/guide/defining-aces

The set of actions, conditions and expressions which are available in your addon is defined in **aces.json**. The term "ACE" is used to refer to an action, condition or expression in general. ACEs are grouped by category. The overall structure of the file format is as follows.

```
{
	"category1": {
		"conditions": [
			condition1,
			condition2,
			...
		],
		"actions": [
			action1,
			action2,
			...
		],
		"expressions": [
			expression1,
			expression2,
			...
		]
	},
	"category2": {
		"conditions": [
			...
		],
		"actions": [
			...
		],
		"expressions": [
			...
		]
	}
}
```

Note that Construct 2 uses numbers for IDs, whereas Construct 3 uses strings. These strings are then also used to identify related language strings in the language file.

### JSON schema

The addon SDK provides a JSON schema to help you write aces.json files, as it provides autocomplete and validation in compatible editors. Construct will ignore a `"$schema"` property at the top level instead of interpreting it as a category to help make it easy to use the schema.

## Never delete ACES after release

*Once you have released your addon, **never delete any actions, conditions or expressions from it**. This will corrupt everyone's projects that use your addon, because Construct will no longer be able to find the deleted action, condition or expression in your addon. Instead mark the features deprecated so they are hidden.*

## Categories

Each category key is the category ID. This is not displayed in the editor; the string to display is looked up in the language file.

For behaviors only, a default category of an empty string may be used. This category will use the behavior name. Other categories may still be used, in which case Construct 3 will append the category name after the behavior name, e.g. "MyBehaviorName: My category".

## Common properties of ACE definitions

Each entry in the `"conditions"`, `"actions"` and `"expressions"` arrays is a JSON object which defines a single condition, action or expression. An example minimal condition definition for the System *Every Tick* condition is shown below.

```
{
        "id": "every-tick",
        "scriptName": "EveryTick"
}
```

The `id` and `scriptName` are the only required properties for conditions and actions. Expressions require `id`, `expressionName` and `returnType`. All other properties are optional.

The definitions for conditions, actions and expressions all share a few common properties. These are detailed below. Then the properties specific to each kind is documented after that.

---

**id**

A string specifying a unique ID for the ACE. This is used in the language file. By convention this is lowercase with dashes for separators, e.g. "my-condition".

---

**c2id**

If you are porting a Construct 2 addon to Construct 3, put the corresponding numerical ID that the Construct 2 addon used here. This allows Construct 3 to import Construct 2 projects using your addon.

**scriptName / expressionName**

The name of the function in the runtime script for this ACE. Note for expressions, use `expressionName` instead, which also defines the name typed by the user in expressions.

**isDeprecated**

Set to true to deprecate the ACE. This hides it in the editor, but allows existing projects to continue using it.

**highlight**

Set to true to highlight the ACE in the condition/action/expression picker dialogs. This should only be used for the most regularly used ACEs, to help users pick them out from the list easily.

**params**

An array of parameter definitions. See the section below on parameters. This can be omitted if the ACE does not use any parameters.

## Condition definitions

Condition definitions can also use the following properties.

**isTrigger**

Specifies a trigger condition. This appears with an arrow in the event sheet. Instead of being evaluated every tick, triggers only run when they are explicity triggered by a runtime call.

**isFakeTrigger**

Specifies a fake trigger. This appears identical to a trigger in the event sheet, but is actually evaluated every tick. This is useful for conditions which are true for a single tick, such as for APIs which must poll a value every tick.

**isStatic**

Normally, the condition runtime method is executed once per picked instance. If the condition is marked static, the runtime method is executed once only, on the object type class. This means the runtime method must also implement the instance picking entirely itself, including respecting negation and OR blocks.

**isLooping**

Display an icon in the event sheet to indicate the condition loops. The condition method should use ILoopingConditionContext to implement its loop.

**isInvertible**

Allow the condition to be inverted in the event sheet. Set to `false` to disable invert.

**isCompatibleWithTriggers**

Allow the condition to be used in the same branch as a trigger. Set to `false` if the condition does not make sense when used in a trigger, such as the *Trigger once* condition.

## Action definitions

Action definitions can also use the following properties.

**isAsync**

Set to `true` to mark the action as asynchronous. Make the action method an `async` function, and the system *Wait for previous actions to complete* action will be able to wait for the action as well.

## Expression definitions

Expressions work slightly differently to conditions and actions: they must specify a `returnType`, and instead of using a `scriptName` they specify an `expressionName` which doubles as both what is typed for the expression as well as the runtime script function name.

**returnType**

One of `"number"`, `"string"`, `"any"`. The runtime function must return the corresponding type, and `"any"` must still return either a number or a string.

**isVariadicParameters**

If `true`, Construct 3 will allow the user to enter any number of parameters beyond those defined. In other words the parameters (if any) listed in `"params"` are required, but this flag enables adding further `"any"` type parameters beyond the end.

## Parameter definitions

ACEs can all define which parameters they use with the `"params"` property. This property should be set to an array of parameter definition objects. Below shows an example for the System *Compare two values* condition.

```
{
        "id": "compare-two-values",
        "scriptName": "Compare",
        "params": [
                {       "id": "first-value",    "type": "any" },
                {       "id": "comparison",             "type": "cmp" },
                {       "id": "second-value",   "type": "any" }
        ]
}
```

Note that expressions can only use `"number"`, `"string"` or `"any"` parameter types.

### id

A string with a unique identifier for this parameter. This is used to refer to the parameter in the language file.

### c2id

In some circumstances, it is necessary to specify which Construct 2 parameter ID a parameter corresponds to. However normally it can be inferred by the parameter index.

### type

The parameter type. Expressions can only use `"number"`, `"string"` or `"any"`. However conditions and actions have the following options available:

- `"number"` — a number parameter
- `"string"` — a string parameter
- `"any"` — either a number or a string
- `"boolean"` — a boolean parameter, displayed as a checkbox
- `"combo"` — a dropdown list. Items must be specified with the `"items"` property.
- `"combo-grouped"` — a dropdown list with grouped items (using optgroup elements). Item groups must be specified with the `"itemGroups"` property.
- `"cmp"` — a dropdown list with comparison options like "equal to", "less than" etc.

- `"object"` — an object picker. The types of plugin to show can be filtered using an optional `"allowedPluginIds"` property.
- `"objectname"` — a string parameter which is interpreted as an object name
- `"projectfile"` — a dropdown list from which any project file in the project can be chosen. The parameter value at runtime is a relative path to fetch the project file from. The `"filter"` option can also be specified to filter the list by a file extension, e.g. `".txt"` to only list .txt files.
- `"layer"` — a string parameter which is interpreted as a layer name
- `"layout"` — a dropdown list with every layout in the project
- `"keyb"` — a keyboard key picker
- `"instancevar"` — a dropdown list with the non-boolean instance variables the object has
- `"instancevarbool"` — a dropdown list with the boolean instance variables the object has
- `"eventvar"` — a dropdown list with non-boolean event variables in scope
- `"eventvarbool"` — a dropdown list with boolean event variables in scope
- `"animation"` — a string parameter which is interpreted as an animation name in the object
- `"objinstancevar"` — a dropdown list with non-boolean instance variables available in a prior `"object"` parameter. Only valid when preceded by an `"object"` parameter.

---

### initialValue

A string which is used as the initial expression for expression-based parameters. Note this is still a string for `"number"` type parameters. It can contain any valid expression for the parameter, such as "1 + 1". For `"boolean"` parameters, use a string of either `"true"` or `"false"`. For `"combo"` parameters, this is the initial item ID.

---

### items

Only valid with the `"combo"` type. Set to an array of item IDs available in the dropdown list. The actual displayed text for the items is defined in the language file.

> *If you remove a combo item after publishing your addon, existing projects using that combo item will revert to the default item.*

### itemGroups

Only valid with the `"combo-grouped"` type. Set to an array of item groups available in the dropdown list, with each group being represented by an object with properties `"id"` for the group ID, and `"items"` being an array of strings of item IDs in the group. The actual displayed text for the groups and items is defined in the language file.

```
// example "combo-grouped" parameter definition
{
	"id": "dinosaur",
	"type": "combo-grouped",
	"itemGroups": [{
		"id": "theropods",
		"items": ["tyrannosaurus", "velociraptor", "deinonychus"]
	}, {
		"id": "sauropods",
		"items": ["diplodocus", "saltasaurus", "apatosaurus"]
	}
}
```

### allowedPluginIds

Optional and only valid with the `"object"` type. Set to an array of plugin IDs allowed to be shown by the object picker. For example, use `["Sprite"]` to only allow the object parameter to select a Sprite.

### filter

Optional and only valid with the `"projectfile"` type. Set to a file extension including the dot to filter the list of provided project files to only those with the given file extension, e.g. `".txt"`.

### autocompleteId

Optional and only valid with the `"string"` type. Set to a globally unique ID and string constants with the same ID will offer autocomplete in the editor. This is useful for "tag" parameters. Note the ID must be unique to all other plugins and behaviors in Construct, so it is a good idea to include your plugin or behavior name in the string, e.g. "myplugin-tag".

## Language strings

The aces.json file does not include any strings displayed in the editor UI. These are all kept in a separate language file to facilitate translation. Therefore to finish adding ACEs, the relevant UI strings like the list name and description must be added to the language file. See The language file for more information.