INSTANCE SCRIPT INTERFACE

View online: https://www.construct.net/en/make-games/manuals/construct-3/scripting/scripting-reference/object-interfaces/iinstance

---

The `IInstance` script interface represents a single instance of an object type. Instances that appear in the layout have a IWorldInstance interface instead, but it derives from `IInstance`, so these methods and properties are available for any type of instance.

Many objects return a more specific class deriving from `IInstance` or `IWorldInstance` to add APIs specific to the plugin. See the Plugin interfaces reference for more information.

Getting an IInstance

Instances are typically accessed through IObjectClass methods like `getFirstInstance()`. For example, `runtime.objects.Sprite.getFirstInstance()` will return the first instance of the Sprite object type.

Instance events

The following events can be listened for on any instance using the `addEventListener` method. See instance event for standard event properties. Note many more kinds of addon-specific events can be fired. See the documentation on each addon's script interfaces for more information.

"destroy"

Fired when the instance is destroyed. After this event, all references to the instance are now invalid, so any remaining references to the instance should be removed or cleared to `null` in this event. Accessing an instance after it is destroyed will throw exceptions or return invalid data. The event object also has an `isEndingLayout` property to indicate if the object is being destroyed because it's the end of a layout, or destroyed for other reasons.

Instance APIs

---

addEventListener(type, func, capture)

removeEventListener(type, func, capture)

Add or remove an event handler for a particular type of event fired by an addon's script interface. An event object is passed as a parameter to the handler function. See instance event for standard event object properties. For information on which events are fired by specific addons and which additional event object properties are available, see the documentation on each addon's script interfaces.

dispatchEvent(e)

Dispatch an event, firing any handler functions that have been added for the event type. You can use `new C3.Event(eventName, isCancellable)` to create an event object that can be dispatched (e.g. `new C3.Event("click", true)`), and add any extra properties relevant to your event to that object. This can also be used by the addon SDK to cause your addon to fire an event in the script interface, e.g.:

```
const e = new C3.Event("click", true);
this.GetScriptInterface().dispatchEvent(e);
```

runtime

A reference back to the IRuntime interface. (This is particularly useful when subclassing instances, since in a custom class's methods you can always refer to the runtime with `this.runtime`.)

objectType

The IObjectType interface for this instance's object type.

plugin

The IPlugin interface (or derivative) for this instance's plugin.

instVars

If the object has any instance variables, they can be accessed by named properties under this property. For example if an object has an instance variable named *health*, it can be set and retrieved using `instance.instVars.health`. Note if the object has no instance variables, the instance won't have an `instVars` property at all.

> *In some cases, instance variables may have names that aren't valid JavaScript identifiers. In this case you can use the string property syntax, e.g.* `instance.instVars["health"]`.

> *You don't have to use instance variables to add custom properties to instances. In JavaScript you can simply assign new properties to existing objects, or use instance subclassing to use your own custom class with your own properties and methods.*

behaviors

If the object has any behaviors, they can be accessed by named properties under this property. For example if an object has a behavior named *Bullet*, it can be accessed using `instance.behaviors.Bullet`. Each behavior has its own properties and methods, which can be found in the Behavior interfaces reference section. Note if the object has no behaviors, the instance won't have a `behaviors` property at all.

> *In some cases, behaviors may have names that aren't valid JavaScript identifiers. In this case you can use the string property syntax, e.g.* `instance.behaviors["8Direction"]`.

---

uid

The unique ID of this instance, as a number. Note instances can be looked up by their UID using the runtime `getInstanceByUid()` method.

---

iid

The index ID (IID) of this instance, as a number. See instances for more details.

---

templateName

Read-only string of the name of the template used to create this instance, or an empty string if no template was used.

---

destroy()

Destroy the instance, removing it and allowing any associated memory to be released.

> *Do not make any further calls or access any properties after the* `destroy()` *call. The instance is no longer valid and any attempts to use it may throw exceptions.*

> *Destroying large numbers of instances with repeated calls to* `destroy()` *can be inefficient. The IRuntime method* `destroyMultiple()` *can perform this significantly more efficiently.*

---

getOtherContainerInstances()

Return an array of `IInstance` (or derivatives) representing other instances in the same container as this instance. This excludes the instance the method is called on.

***otherContainerInstances()**

Iterates over `IInstance` (or derivatives) representing other instances in the same container as this instance. This excludes the instance the method is called on.

---

dt

Return delta-time according to the object's own timescale. See Delta-time and framerate independence for more information.

---

timeScale

restoreTimeScale()

The `timeScale` property sets or gets the current instance-specific time scale, e.g. 1.0 for normal speed, 2.0 for twice as fast, etc. Note that once set, the instance uses its own time scale instead of runtime time scale, e.g. allowing an instance to keep moving when the runtime time scale is 0. Calling the `restoreTimeScale()` method will then switch it back to following the runtime time scale.

---

signal(tag)

Triggers *On signal* for this instance, and resolves any promise returned by `waitForSignal()`. Any events waiting for a signal with the given tag will resume when all picked instances are signalled.

---

waitForSignal(tag)

Returns a Promise that resolves when the given tag is signalled for this instance. It may be signalled by either the script API or an event sheet.

---

hasTag(tag)

Return a boolean indicating if the instance has the specified tag (case insensitive). This method only supports checking a single tag, and is more efficient for that purpose than `hasTags()`.

---

hasTags(...tags)

Pass multiple string arguments to check if the instance has all the specified tags (case insensitive), returning a boolean.

---

setAllTags(tags)

Pass in any iterable of strings, such as an array or Set, to set the instance's current tags (replacing all existing tags).

---

getAllTags()

Return a Set with all the tags an instance has.

---

callCustomAction(name, ...params)

Calling `inst.callCustomAction(name, ...params)` is a shorthand for calling `inst.objectType.callCustomAction(name, [inst], ...params)`. This is intended as a convenience when the custom action will only be run with a single instance picked. Refer to the IObjectClass `callCustomAction()` method for more details.

> *It is more efficient to use the IObjectClass `callCustomAction()` method to call a custom action once with multiple instances picked, than to use the IInstance `callCustomAction()` method repeatedly.*