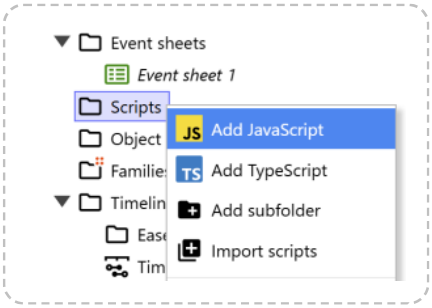
SCRIPT FILES

View online: https://www.construct.net/en/make-games/manuals/construct-3/scripting/using-scripting/script-files

For those more familiar with coding, script files provide a code editor to write a JavaScript or TypeScript file which works independently of an event sheet.

Adding script files

Script files can be added in the *Scripts* folder of the Project Bar. Existing JavaScript files (.js) and TypeScript files (.ts) can also be imported using the *Import scripts* option instead.



Once you add a script a code editor appears. The first script added will have some default code added to help you get started.

Execution of the main script

Construct loads only the *main script* just after the Construct engine scripts run on startup. This is before any loading screen appears. It is a good place to write any initialization code and imports/exports the entire project will use. Since the Construct engine has still not yet initialized at this point, there is no `runtime` variable (representing the runtime script interface) available at the top level. Instead Construct provides a special `runOnStartup` function that runs a callback once the runtime is ready, and provides the `runtime` variable as a parameter.

```
runOnStartup(async runtime =>
{
	// This code runs on startup with
	// the 'runtime' variable available
});
```

The callback can be an `async` function, meaning the `await` keyword can be used inside it, so this is a convenient place to run any asynchronous initialization while also making use of the runtime script interface.

Note that the callback runs **when the loading screen appears.** The project is not yet fully loaded and no objects exist yet. Therefore any code that attempts to interact with objects in your project

in that callback won't work. Instead you'll need to wait for the `"beforeprojectstart"` event, which fires just before the first layout starts running.

```
runOnStartup(async runtime =>
{
	// Wait for "beforeprojectstart" event
	runtime.addEventListener("beforeprojectstart", () => OnBeforeProjectStart(runtime));
});

async function OnBeforeProjectStart(runtime)
{
	// Now the project is loaded and objects exist.
	// You can interact with objects etc. here.
}
```

Note that the function for the `"beforeprojectstart"` event can be async, and Construct will wait for it to complete before the project starts running. This means you can also use `await` in case you need to do any asynchronous initialization there.

Continuing execution

The only code in the main script that is automatically executed is the top-level scope and any `runOnStartup` callbacks. Beyond that no code in your script will run any more, unless you add event listeners to run callbacks, as previously shown with `"beforeprojectstart"`. The main other event to listen for is the runtime `"tick"` event. Since this fires every tick it provides a good place to keep running code throughout your game. The JavaScript code example below demonstrates a typical way to use this.

```
runOnStartup(async runtime =>
{
	runtime.addEventListener("beforeprojectstart", () => OnBeforeProjectStart(runtime));
});

async function OnBeforeProjectStart(runtime)
{
	runtime.addEventListener("tick", () => Tick(runtime));
}

function Tick(runtime)
{
	// Code to run every tick.
	// Note 'runtime' is passed.
}
```

Using additional scripts

As noted previously, the only script Construct automatically loads and runs is the *main script*. This appears in bold in the Project Bar.

When you select script files, the Properties Bar shows a *Purpose* property for the script. The main script has the purpose set to *Main script*, and you can only have one main script in your project.

Since you can only have one main script, you must choose whether the main script is JavaScript or TypeScript. There is some support for mixing JavaScript and TypeScript code, but you may find it easier to choose one language to use throughout your project.

To use any other script files in your project, you must `import` them in the main script. This also lets you control the order they are loaded and run. These other script files should have the *Purpose* set to *'(none)'* (indicating Construct won't use it automatically) and also `export` the things it wants to be used by other scripts. See the Imports & exports example for a basic demonstration of using modules in Construct.

To learn more about imports and exports, refer to the MDN guide on JavaScript Modules.

Using external scripts

Sometimes you want to load a separate script file that is external and not loaded via an import. A good example of this are Web Workers - you may use something like `new Worker("myworker.js")`, where "myworker.js" must always be in a separate file.

You should place these script files in the **Files** folder of the Project Bar instead of the **Scripts** folder. The reason for this is Construct has special processing for everything in the Scripts folder, including moving the files to a different folder on export, or minifying all the scripts there, and these steps can cause the worker script file to stop working after export. On the other hand script files in the Files folder are not processed and are just copied as-is on export, so things like worker scripts will work consistently both in preview and export.

Integration with scripts in events

Construct loads all script files as modules. Unlike legacy "classic" mode scripts, modules have their own top-level scope. This means things like a top-level function declaration is *not* available in other script files.

Instead you can add imports to the script file with the purpose *Imports for events* which then become available for scripts in events. See the section *Using imports* in Scripts in event sheets for more details.

Another option is to write globals as explicit properties of `globalThis`, e.g. `globalThis.myFunction = function () { ... }` and call it via `globalThis.myFunction()`, but using modules is preferable.

Errors

Unlike scripts in event sheets, errors arising from the top level of script files are not automatically handled by Construct. If an unhandled exception is thrown, the browser will halt any further execution of script in that file. Typically this causes the rest of your code to stop working, and is considered a crash. See the section debugging scripts to find out how to deal with such issues.

Note one difference is exceptions or rejections in a `runOnStartup` callback are automatically handled by Construct. The error will be logged to the browser console and the runtime will continue to start up and run the game - but note if an error occurred it may not run as expected.