TYPESCRIPT IN CONSTRUCT

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

There are a few details specific to using TypeScript in Construct. This section outlines what you need to know.

TypeScript in the Construct editor

If you are writing TypeScript code in Construct itself, you should only have TypeScript files (.ts) in your project. Construct will automatically compile these to JavaScript (.js) files for you behind the scenes, so you don't need to worry about having .js files in your project. The only times you'll see the JavaScript output of your TypeScript code is when debugging or after exporting your project.

If you do have both a .ts and a .js file with the same name in your project, Construct gives priority to the .js file. In other words it will ignore your .ts file and just use the contents of the .js file.

Using an external editor

If you use an external editor to write TypeScript code, you most likely want to keep the .ts files external, and only import the .js output files to your Construct project. This is because your external code editor will use its own version of TypeScript. Using two different versions of TypeScript is likely to cause awkward compatibility problems, so the best workflow is to let your external editor compile TypeScript to JavaScript, and then just use the .js files in Construct so it uses them as-is. You can set your code editor to automatically compile TypeScript to JavaScript when saving a file, and set Construct to auto-reload the script files on preview, so then you can have an easy workflow where saving a TypeScript file will automatically reflect the changes in the next Construct preview.

Using an external editor also allows you to use a newer or different version of TypeScript to the one built in to Construct. Construct's built-in TypeScript version is logged to the browser console the first time a code editor is opened.

Suggested workflows

In short the two suggested workflows for TypeScript coding are:

1. Code TypeScript in the Construct editor: only have .ts files in your Construct project, and no .js files.
2. Code TypeScript using an external editor: don't have .ts files in your Construct project, only .js files.

TypeScript definition files

You can import .d.ts files - aka TypeScript definition files - to your Construct project. This will allow TypeScript to use the type definitions in those files. As per the design of TypeScript, definition files are not compiled to JavaScript and have no effect on how code is run - they only define types to be used by TypeScript's code validator.

Converting a project to TypeScript

You can switch an existing project using JavaScript code to TypeScript by right-clicking the *Scripts* folder in the Project Bar and choosing TypeScript ▶ Switch project to TypeScript. This will automatically switch all JavaScript code in event sheets to use the TypeScript language instead, and rename all .js project files to .ts. Note however it does not change any code, as it is not possible to automatically add type annotations. Therefore you will still need to go through your code and add type annotations until there are no more TypeScript errors (see the next section for advice on that). However this option should still save you some time if you want to switch your project's language.

Similarly a project using TypeScript can be switched back to JavaScript. Again you'll need to then remove type annotations for it to be valid JavaScript code.

Converting a folder project to TypeScript

Construct has special handling for switching a folder-based project to TypeScript. This applies when there are both .js and .ts files in the project folder, but only .js files in the Construct project (i.e. our recommended workflow for using an external editor). In this case choosing the *Switch project to TypeScript* option will instead replace .js files in the Construct project with the corresponding .ts files from the project folder. This provides a handy way to switch a project from using an external editor back to using Construct's built-in TypeScript support.

Adding types to existing JavaScript code

If you convert a JavaScript project to TypeScript, you'll probably see a lot of TypeScript errors. That's because in some places TypeScript requires type annotations, and leaving them out is marked as an error. You'll need to go through all the code adding type annotations until there are no more TypeScript errors reported. This section has some advice on how to do that. Note this is not an exhaustive list of all changes you'll need to make, but the common changes that you're likely to need to make are covered, as well as some advice specific to Construct.

As a starting example, the default *main.js* code file includes this function:

```
async function OnBeforeProjectStart(runtime)
{
	// ...
}
```

TypeScript will identify the `runtime` parameter as an error, because it does not have a required type annotation. Construct's runtime interface type is IRuntime. So the parameter must be marked as having the type `IRuntime`, as shown below.

```
async function OnBeforeProjectStart(runtime: IRuntime)
{
        // ...
}
```

In many cases TypeScript can automatically infer types from existing code (see Type inference) so a type annotation is not always needed in every place in the code. However there are several places like function parameters that will need type annotations to be added. Familiarity with Construct's built-in class names is useful as they are sometimes needed as types, such as `IRuntime` above - all the class names are included in the scripting reference section of the manual.

Instance types

When using TypeScript, Construct generates a special class representing an instance for every object type and family in the project. This includes type definitions for things like the instance variables, behaviors and effects specific to that object. These classes are all in the `InstanceType` namespace with the name of the object. For example `InstanceType.Player` is the type for an instance of the *Player* object type.

Construct's naming rules are more permissive than JavaScript/TypeScript's naming rules, so in some cases the generated class name may be different to the object type name. The best approach is to make sure all your object names are valid JavaScript identifiers, which generally means starting with an alphabetic letter.

Optional types

Many of Construct methods, such as `objectType.getFirstInstance()`, can return `null` (in this case, if no instances exist at all). This means the method's return type can optionally be `null`. TypeScript will show an error if you try to use something that could be `null`. An example of this is shown below.

```
const playerInst = runtime.objects.Player.getFirstInstance();
playerInst.x += 10; // Error: 'playerInst' is possibly 'null'
```

If you know for sure that there is always an instance of the object and so it will never return `null`, you can add an exclamation mark `!` after the expression to tell TypeScript you know it won't be `null`.

```
// Note '!' added to line below
const playerInst = runtime.objects.Player.getFirstInstance()!;
playerInst.x += 10; // OK
```

This is known as the non-null assertion operator.

Subclassing

If your project uses subclassing to customize the instance class for Construct objects, then you'll find some Construct APIs still return instances of the default type. For example instances of a Globin sprite object will be typed as the default *InstanceType.Goblin* instead of a custom *GoblinInstance* class, e.g.:

```
const inst = runtime.objects.Goblin.getFirstInstance()!;
// 'inst' is of type InstanceType.Goblin - so it won't have any of
// the properties or methods of the custom GoblinInstance class
```

To solve this, the methods available on `IObjectType` are in fact generic, so you can make them return the correct type. This means adding the `<Type>` generic syntax like so:

```
const inst = runtime.objects.Goblin.getFirstInstance<GoblinInstance>()!;
// 'inst' is now of type GoblinInstance and so can use the properties
// and methods of the custom class
```

Note that you may then have to import the module that defines `GoblinInstance`, as otherwise TypeScript doesn't know about its type.

Object literals

Sometimes it's useful to write an object literal, which the *Spell Caster Code* example does for sharing global variables from a module, similar to this:

```
const Globals = {
	score: 0,
	playerInstance: null
};
```

In this case, TypeScript will correctly infer the type of `score` as *number*, but it will infer the type of `playerInstance` as *null*. The type *null* means the variable can only ever have the value `null` and assigning anything else to it will be an error! Due to the syntax of object literals, which already use a colon, it's not always obvious at first how to add a specific type to this property. The solution is to use the generic-style syntax `<Type>` like so:

```
const Globals = {
	score: 0,
	playerInstance: <InstanceType.Player | null> null
};
```

Class properties

Commonly in JavaScript, class properties are added in the constructor.

```
class MyClass {
	constructor()
	{
		this.prop1 = "hello";
		this.prop2 = 123;
	}
}
```

TypeScript does not infer the class properties from the constructor, so it will show an error for `prop1` and `prop2`, e.g. *Property 'prop1' does not exist on type 'MyClass'*. Instead you must declare the class properties, and their types, at the class-level like so:

```
class MyClass {

	prop1: string;
	prop2: number;

	constructor()
	{
		this.prop1 = "hello";
		this.prop2 = 123;
	}
}
```

Note JavaScript does allow class property definitions in a similar way, with the feature known as class fields. Using this feature in JavaScript should make it easier to switch to TypeScript, as you can then just add type annotations to the existing class fields.

Imports

Typically when importing other JavaScript files in your project, you'd write a relative import for another .js file like so:

```
import Globals from "./globals.js";
```

How do you write the import for TypeScript? The answer is: exactly the same way! Even though the import ends with .js, TypeScript knows the file is really generated from the .ts file, and so everything just works. Don't try to change it, as otherwise it won't work after it's compiled to JavaScript.

Example

The Spell Caster TypeScript example demonstrates the Spell Caster JavaScript example but updated to use TypeScript. In the Example Browser you can find a number of examples that have both JavaScript and TypeScript variants, which allows you to compare them and identify what changes were made. You can also try practicing by taking a JavaScript example, using the

Switch project to TypeScript option, and adding type annotations; if you get stuck, check the TypeScript version to see what the solution is. Usually if a non-obvious change was needed for TypeScript, the project will contain a special comment explaining what was done.