

SUBCLASSING INSTANCES

View online: <https://www.construct.net/en/make-games/manuals/construct-3/scripting/guides/subclassing-instances>

Construct uses the `IInstance` and `IWorldInstance` interfaces to provide access to instances. Many plugins provide derivatives of these to provide plugin-specific APIs. For example the Text plugin adds a `text` property with an `ITextInstance` class that derives from `IWorldInstance`. The [Plugin interfaces](#) section covers these.

However in your projects it is often desirable to have a further customised class to represent instances. For example all your Sprite instances will provide an `ISpriteInstance` interface, but this is still a fairly generic class to represent many different aspects of your project, such as both the player and enemies. *Subclassing* allows you to use your own custom classes like `PlayerInstance` or `EnemyInstance` to represent different objects in your game. Then any time you ask Construct for instances, such as with `getAllInstances()`, you'll get references to your custom classes instead of a generic `ISpriteInstance` or `IWorldInstance` class.

To use subclassing, follow the steps provided here. The [Spell Caster code scripting example](#) also demonstrates how to use this, with a custom `GoblinInstance` class to represent the enemy goblins in the game. A [TypeScript version](#) is also available. The code samples here use JavaScript, but subclassing works similarly in TypeScript with the necessary type annotations - see [TypeScript in Construct](#).

Step 1: create your class

First write a class that extends from the class normally used by the instance. To help make this easy, Construct creates a special kind of class for every kind of instance in the project in the `InstanceType` namespace. For example when extending an object named *Player*, extend from `globalThis.InstanceType.Player`.

Make sure your object names are valid JavaScript identifiers. Construct has more permissive rules around naming objects, such as allowing them to start with a number, but JavaScript names do not allow that. Invalid names may be adjusted by Construct, and the easiest approach is to just make sure all your object names are also valid in JavaScript.

In this example we'll extend a Sprite instance for the object named *Player*, so the class extends from `globalThis.InstanceType.Player`.

```
class PlayerInstance extends globalThis.InstanceType.Player
{
    constructor()
    {
```

```

        super();
    }
}

```

Often it is sensible to organize code by using a separate script file for the class.

Step 2: set the instance class

Next, use the `ObjectClass.setInstanceClass()` method to set your custom class. This must be done before any instances in the project are created, to make sure they all use the right class. Therefore this must be called in `runOnStartup`, which runs before the runtime has finished loading, so no instances exist yet.

```

runOnStartup(async runtime =>
{
    runtime.objects.Player.setInstanceClass(PlayerInstance);
}

```

Step 3: customise the class

Now you can add custom properties and methods to your class. For example the Player class may need to use an ammo counter, and a *shoot* method to fire one of their bullets. You can write these as you would with a normal JavaScript class.

```

class PlayerInstance extends globalThis.InstanceType.Player
{
    constructor()
    {
        super();

        // Start with 5 bullets
        this.ammo = 5;
    }

    shoot()
    {
        // Decrement ammo count
        this.ammo--;

        // create a bullet instance, etc.
    }
}

```

Note that here `ammo` is not an instance variable or anything else associated with Construct's event system: it's just a normal JavaScript object property.

When using TypeScript, you'll need to use class fields to declare the property type. See [TypeScript in Construct](#) for more details.

Since `IInstance` has a `runtime` property, within your class you can always use `this.runtime` to refer to the [runtime script interface](#).

You may also wish to make use of [private properties and methods](#) to ensure some details remain internal to your class.

Over time as the Construct engine is improved, there are likely to be more properties and methods added to the base classes. Your derived class's properties and methods override any in base classes, so could potentially hide new APIs added in future. To avoid this causing problems, try to use as specific names as possible that only apply to your project, and avoid generic terms used elsewhere by Construct. If you want to be completely safe, use a different naming scheme for your own additions, such as beginning every property with an underscore. Also, name clashes will never occur with private properties or methods (starting with `#`), so it's a good idea to use those where you can.

Step 4: use your custom features

Now whenever you retrieve instances of the player from the existing APIs, you'll get `PlayerInstance` classes instead of the default based on `ISpriteInstance`. Then you can read your custom properties and call custom methods.

```
// Assume called in "beforelayoutstart" event
function OnBeforeLayoutStart(runtime)
{
    // Get player instance from Construct
    const playerInstance = runtime.objects.Player.getFirstInstance();

    // Example uses of custom class
    console.log("Ammo = " + playerInstance.ammo);
    playerInstance.shoot();
}
```

Conclusion

Subclassing is straightforward to set up, and lets you use custom classes for different objects in your project. This can make your code a lot clearer, and helps you to use the full power of JavaScript/TypeScript classes with instances in Construct's runtime.