応用テキスト

A05

継承とリソースによる管理

ver.1.0.0

バージョン4.2.1をベースに作成し、4.3.0にも対応しています。

Godotにたくさん用意されているノードは、Nodeクラスを継承してできています。例えば公式リファレンスでSprite2Dのページを見ると
https://docs.godotengine.org/ja/4.x/classes/class_sprite2d.html

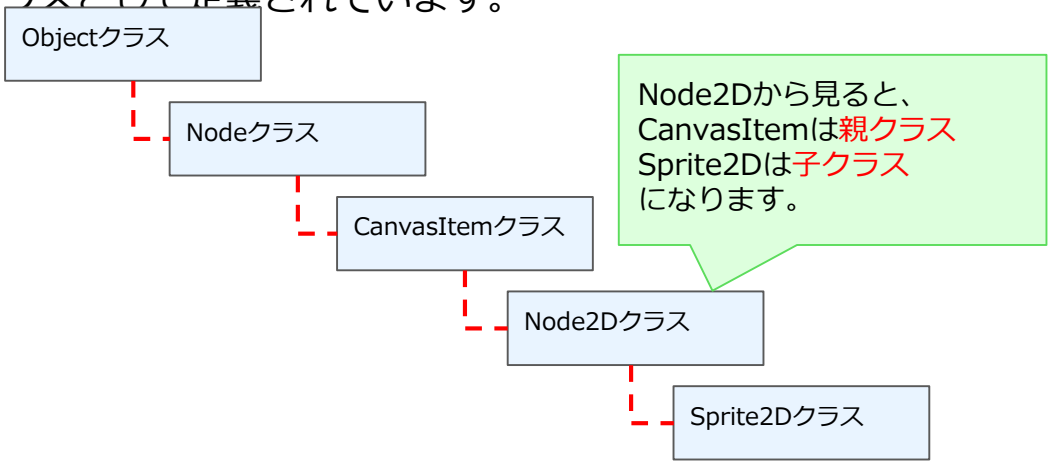
このように記載があります。

Sprite2D

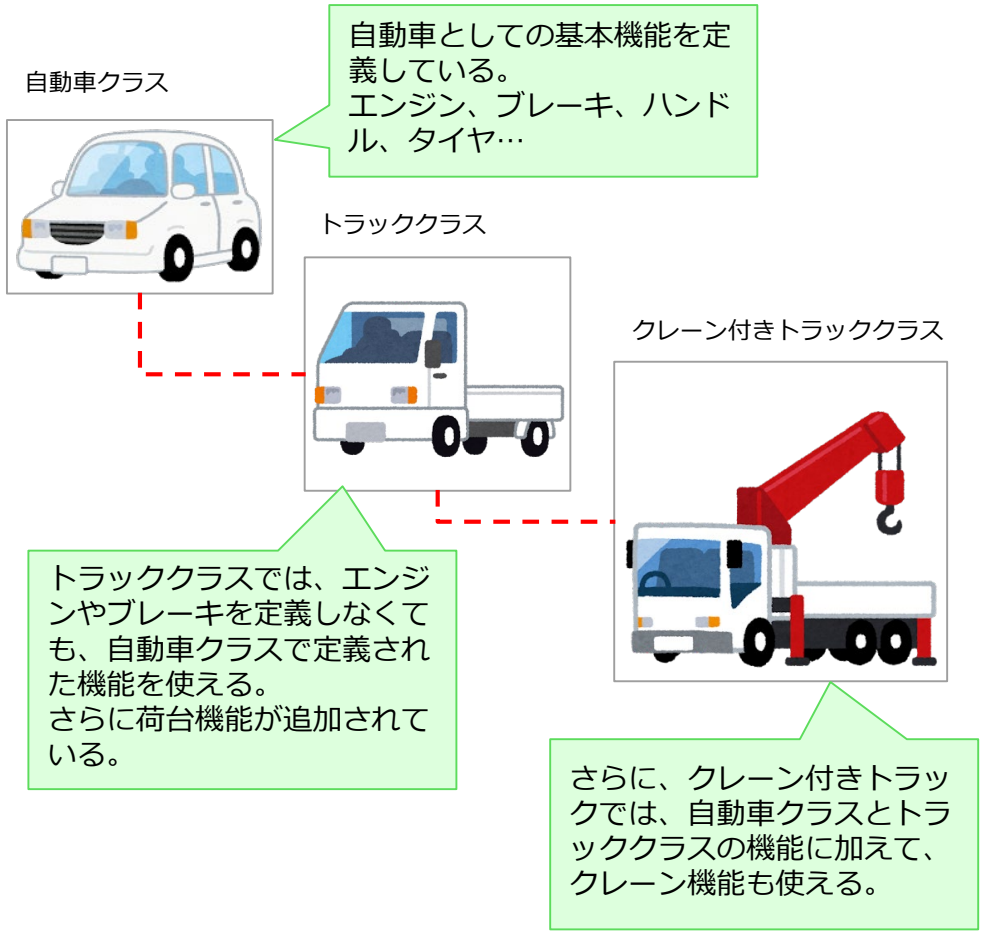
Inherits: Node2D < CanvasItem < Node < Object

Inherits（インヘリッツ）と書かれているのは**継承**（Inherit：インヘリト）のことで、Sprite2Dは、Node2Dを継承し、Node2DはCanvasItemを継承し…と、たどっていけます。
すべてのクラスはObjectクラスがベースです。

このように、Godotで使うノードやその他のものはすべてクラスとして定義されています。



クラスと継承を利用する利点は、子クラスはそれより上位の親クラスで定義されたプロパティやメソッドを利用できることです。

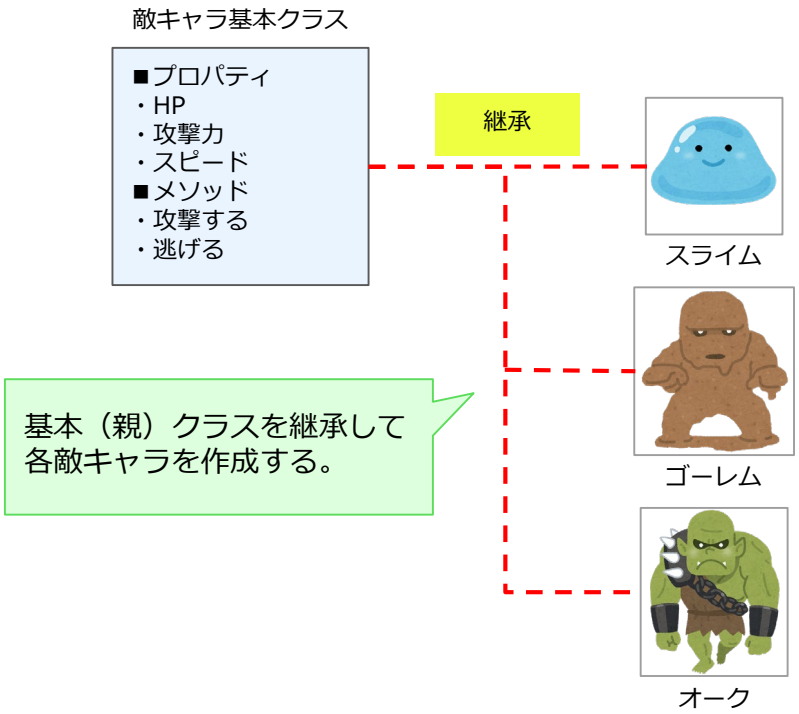


クラスと継承は、Godot標準に用意されているものだけではなく、ユーザーが独自にクラスを作成して、継承関係を持たせることも可能です。

スクリプトの継承

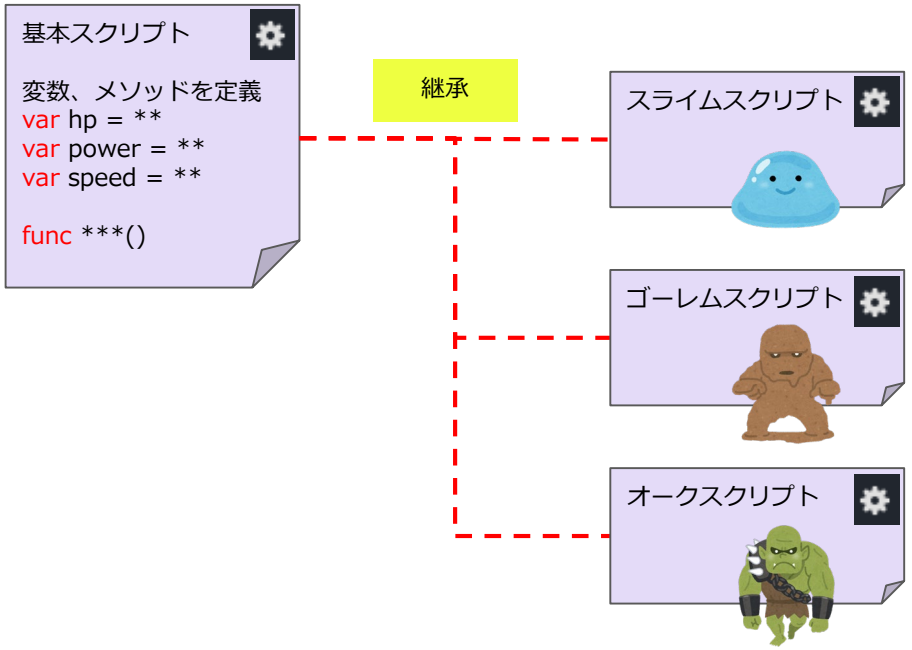
ゲーム内に登場する敵キャラを例に考えます。敵キャラには共通するノードやプロパティ、メソッドがあるはずです。

それを敵キャラ個別に定義すると無駄が多くなりますので、ベースとなる敵キャラ基本クラスをつくって、各敵キャラはそれを継承するようにしておくと効率良く管理できます。



具体的には、まず基本クラスとなる基本スクリプトを作成し、敵キャラ共通で使用するプロパティやメソッドを定義しておきます。

その後、スクリプトを継承して、各敵キャラシーンに割り当てます。



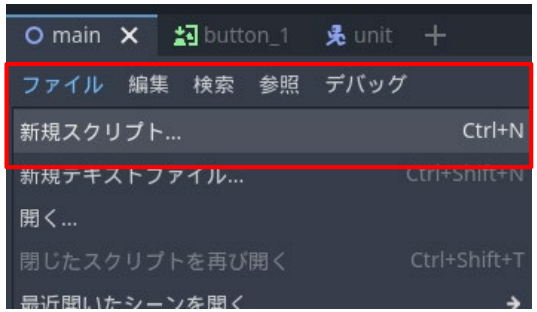
Godotではスクリプトはクラスとして扱われるため、特にクラスを意識する必要はありません。スクリプトにクラス名（class_name）を記載しておくと他のスクリプトから簡単に参照できるようになります。

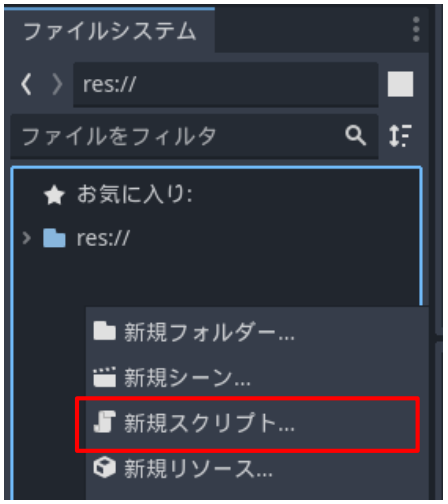
```
1   extends CharacterBody2D
2   class_name EnemyBase
3
```

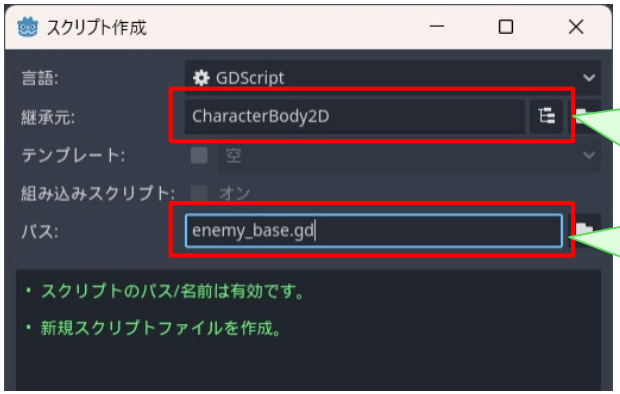
■基本スクリプトの作成

例として、ゲームに登場する敵キャラを効率よく管理できるように、敵キャラ基本スクリプトをつくります。
継承元のスクリプトになるため「**親スクリプト**」とよびます。

スクリプトは、Scriptのファイルメニューや、ファイルシステムドック上で右クリックなどからでも作成できます。







継承元は敵キャラシーンのノードにしておくため、ここではCharacterBody2Dにしました。

スクリプトファイル名として「enemy_base.gd」にしておきました。

これで継承元となる親スクリプトができました。
ここでは例として、このようなプロパティとメソッドを定義しておきます。

```
1   extends CharacterBody2D
2   class_name EnemyBase
3
4   @export var hp: int = 100
5   @export var power: int = 10
6   @export var speed: float = 100.0
7
8   func test(str):
9       print(str)
```

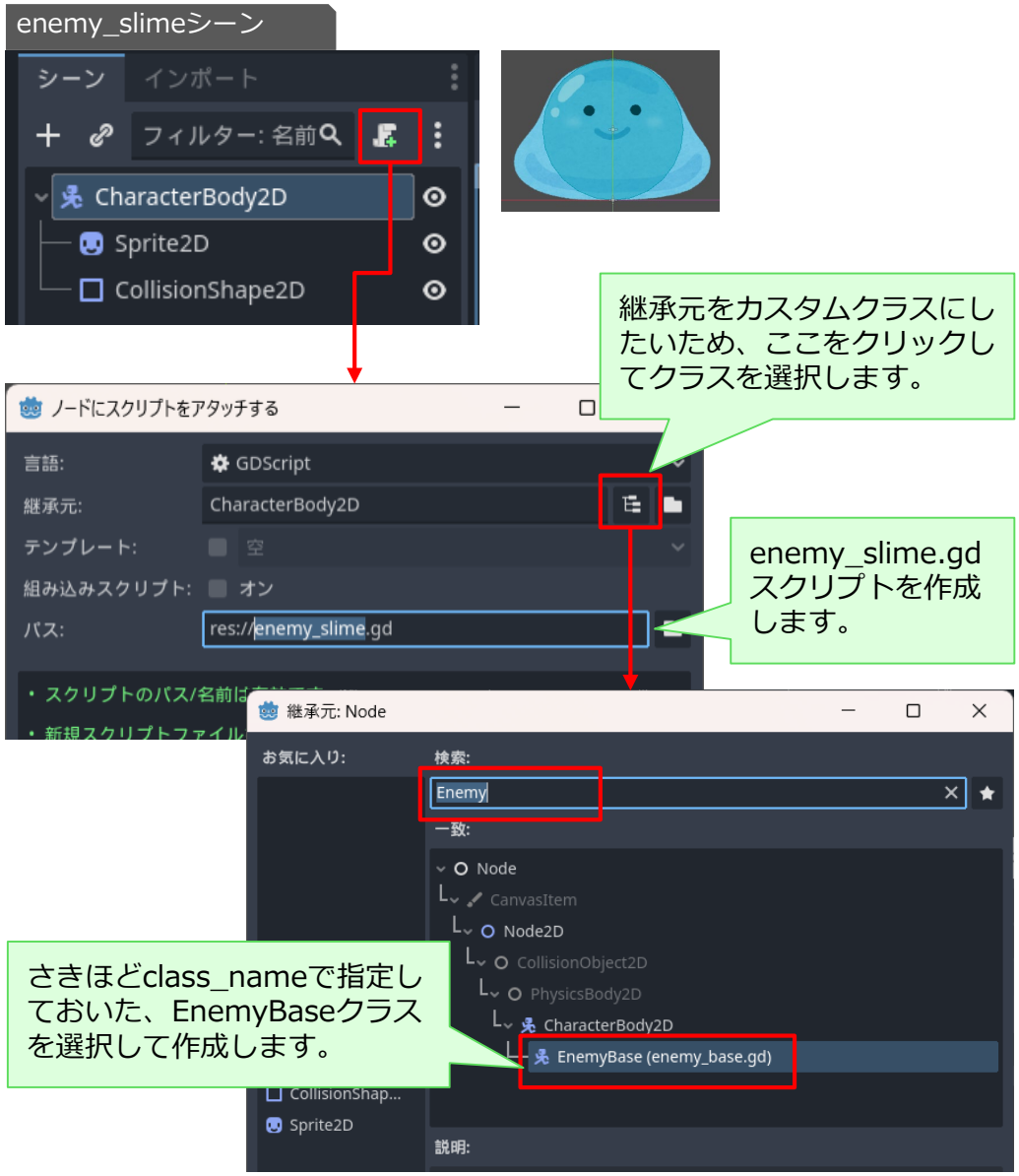
class_nameでクラス名を指定しておくと、後で分かりやすくなります。

@exportを付けているため、継承先ではインスペクターで値を指定できるようになります。

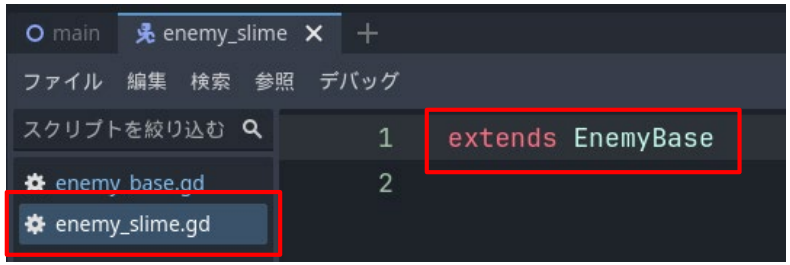
親スクリプトに書かれた変数や関数は、子クラスからプロパティ、メソッドとして利用できるようになります。

親スクリプトを継承

個別の敵キャラシーンを作成します。まずは通常通りノードを構成して保存します。その後、スクリプトをアタッチしますがこの時に継承元として親スクリプトを指定します。



このように、継承されていることが確認できます。



試しに、親スクリプトとなるEnemyBaseクラスのtest()メソッドを呼び出して確認します。

enemy_base.gd

```
8 func test(str):
9     print(str)
```

enemy_slime.gd

```
1 extends EnemyBase
2
3 func _ready():
4     test("テスト")
```

実行結果

```
テスト
```

親スクリプトで、@export指定しておいたので、インスペクターで変数の値を設定できます。

オーバーライドとシーンの継承

■オーバーライド

親スクリプトを継承したスクリプトで、親スクリプトと同じ名前の関数定義をして、親スクリプトのメソッドを上書きすることを「オーバーライド」と言います。

例えば、以下のような場合は、子スクリプトに書かれた処理で上書きされるため、実行結果も子スクリプトのメソッドのものになります。

親スクリプト

```
1   extends CharacterBody2D
2   class_name EnemyBase
3
4   func _ready():
5       print("親スクリプト")
```

子スクリプト

```
1   extends EnemyBase
2
3   func _ready():
4       print("子スクリプト")
```

実行結果

```
子スクリプト
```

完全に処理を上書きするのではなく、処理を追加したい場合は、親スクリプトのメソッドを呼び出すこともできます。

子スクリプト

```
1   extends EnemyBase
2
3   func _ready():
4       super()
5       print("子スクリプト")
```

実行結果

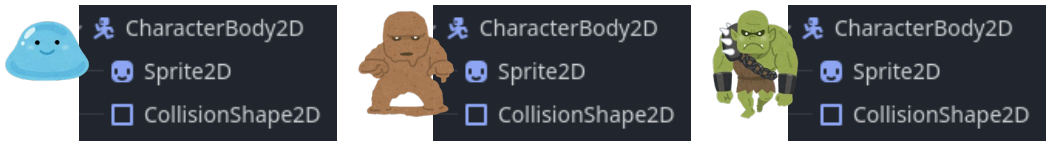
```
親スクリプト
子スクリプト
```

super() は、実行されたタイミングで親スクリプトの同名のメソッドを実行し、その後、super() 以下の処理を続けて実行します。

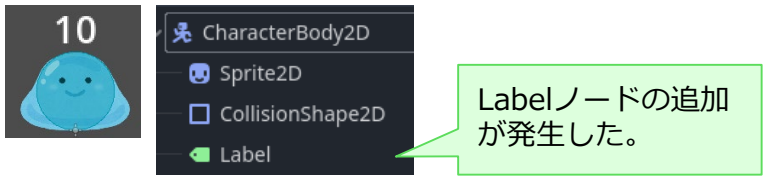
■シーンの継承

ここまではスクリプトを継承して、効率よくシーンを管理する方法を説明しましたが、シーン自体も継承できます。

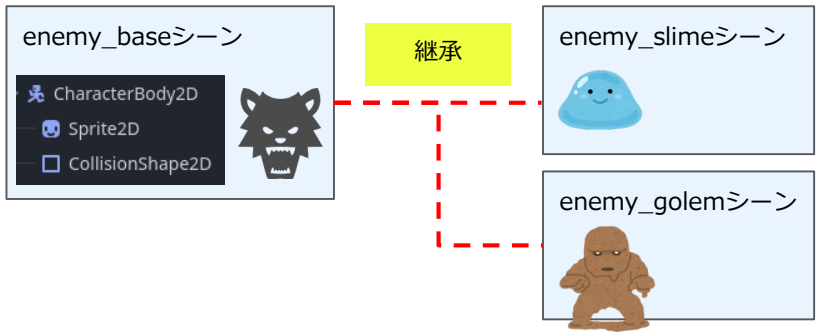
例えば、敵キャラクターのシーンを構成するノードについても、このようにそれぞれ同じノード構成になっている場合がほとんどです。



ゲームの仕様変更により、下図のようにHPの数値表示を行うように変わったとすると、それまでにつくった全ての敵シーンで同じような変更をしなければなりません。



そこでシーンについても、ベースとなるシーンを作成しておき、それを継承したシーンにすることで、効率よく管理できるようになります。

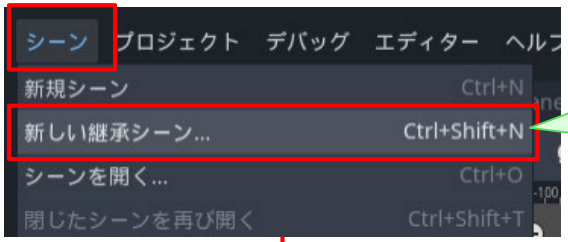


継承シーンの作成

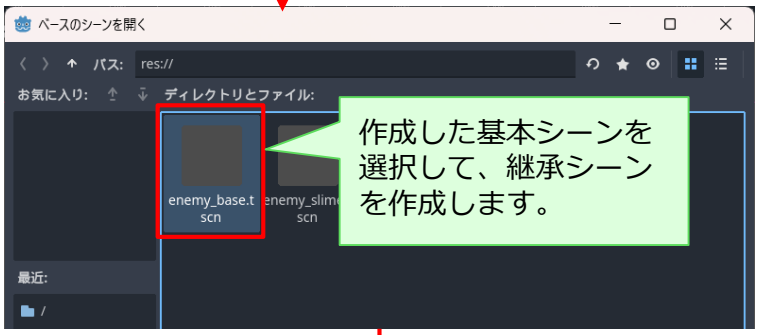
■継承シーン作成の基礎

まずはベースとなる敵キャラの基本シーンを作成します。

ここではSprite2DやCollisionShape2Dは特に何も設定していません。



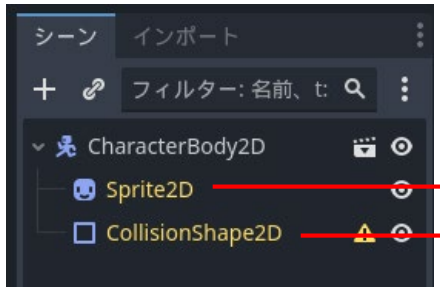
[シーン]メニューから[新しい継承シーン...]を選択します。

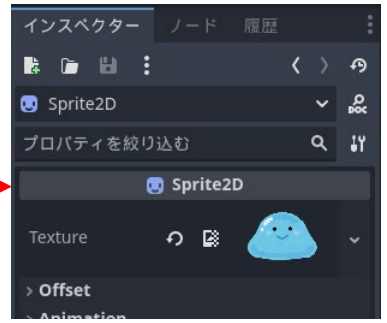


作成した基本シーンを選択して、継承シーンを作成します。

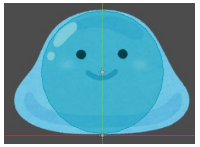
継承シーンができました。ルートノード以外は黄色文字で表示されます。

作成した継承シーンは個別に編集可能です。





継承シーン内のノードは通常のノードのように編集でき、編集しても基本シーンへは影響ありません。



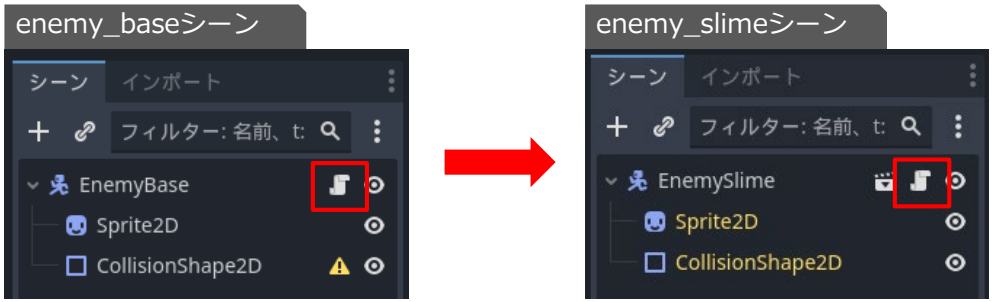
逆に、基本シーンに加えた変更は、継承シーンにも影響を与えますが、継承シーンで変更したものはそのままです。

Labelノードの追加が発生した。

継承シーンにも同じ変更が受け継がれる。

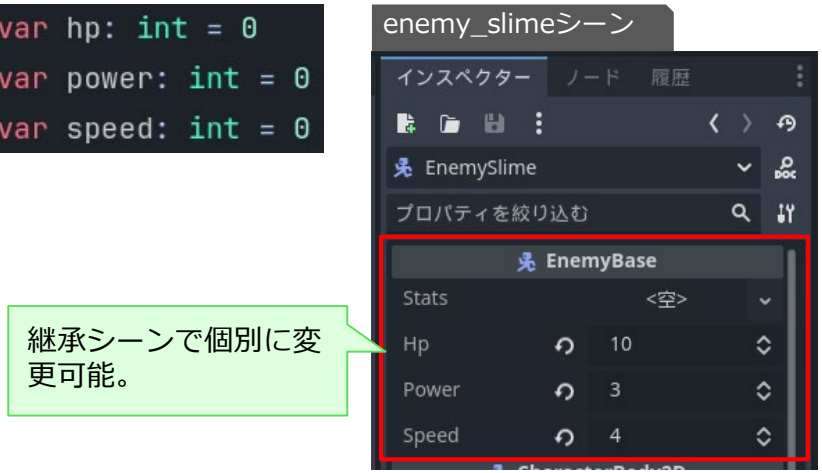
■継承シーンのスクリプト

シーンのスクリプトについては、基本シーンにあらかじめスクリプトをアタッチしておいた場合は、そのまま継承されます。



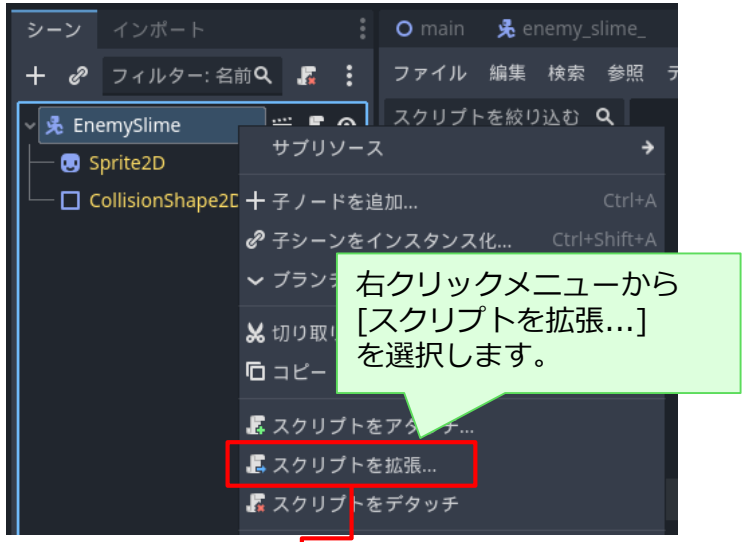
ただしこのままだと継承シーンで作成した敵キャラはすべてパラメータなども同じになってしまいます。変更したいパラメータは@exportをつけておけば、継承シーンごとに変更が可能です。

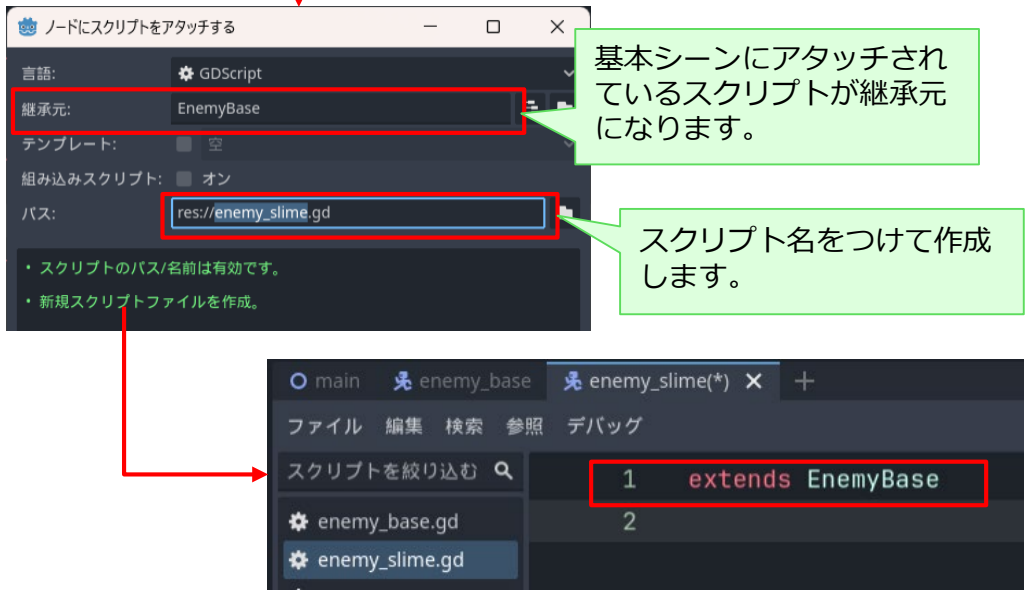
```
@export var hp: int = 0
@export var power: int = 0
@export var speed: int = 0
```



継承シーンではSprite2Dなどの見た目のみを変更して、敵キャラ固有の機能がない場合はこの方法で良いでしょう。

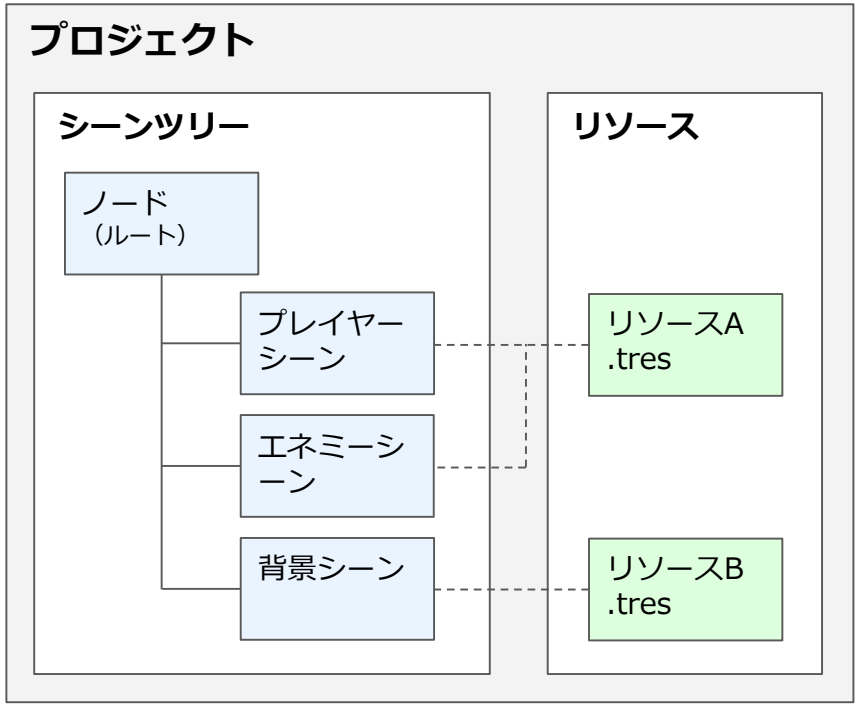
継承した敵キャラごとに、なんらか特別な動作や機能が必要な場合は、スクリプトの継承を使います。



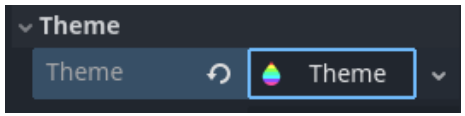


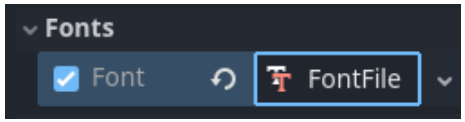
スクリプト単体で継承した場合と同様の結果になります。

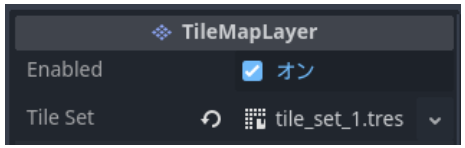
Godotのゲームプロジェクトを構成するメインとなる部品はシーンとノードです。リソースはシーンやノードで利用される様々な再利用可能なデータのようなものです。



例えばボタンの見た目を設定するテーマファイルや、フォントファイル、タイルセットなどが、よく使われるリソースです。
保存するとファイル拡張子が「.tres」となっています。



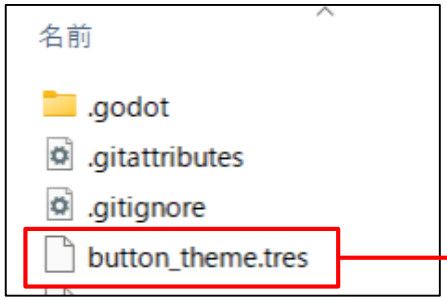


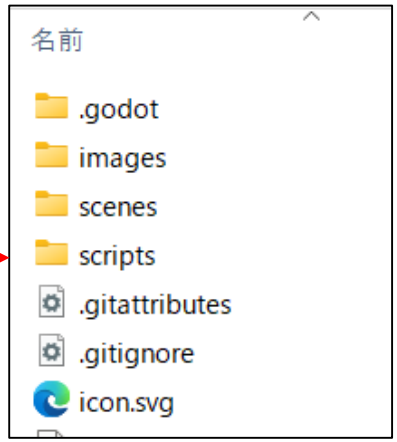


他にも、画像ファイルなどもリソースとして扱われます。

保存したリソースファイルは単独のデータファイルとして扱えます。例えばこのようなボタンをデザインしてテーマファイルを保存しました。

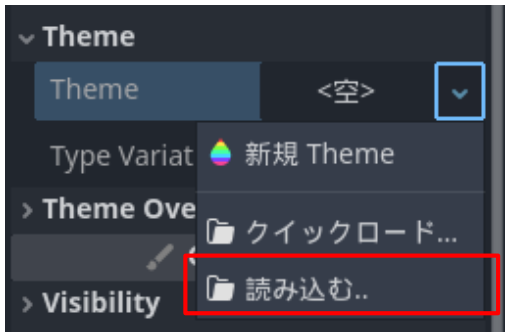
Windowsのエクスプローラ上でプロジェクトの中を見ると、このようになっています。





他のプロジェクトでも使う場合はリソースファイルをコピーします。

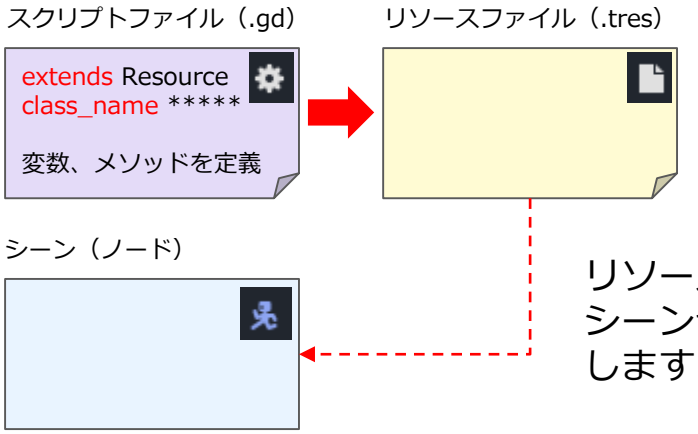
リソースファイルを読み込むと、異なるプロジェクトでも同じリソースを使うことができます。



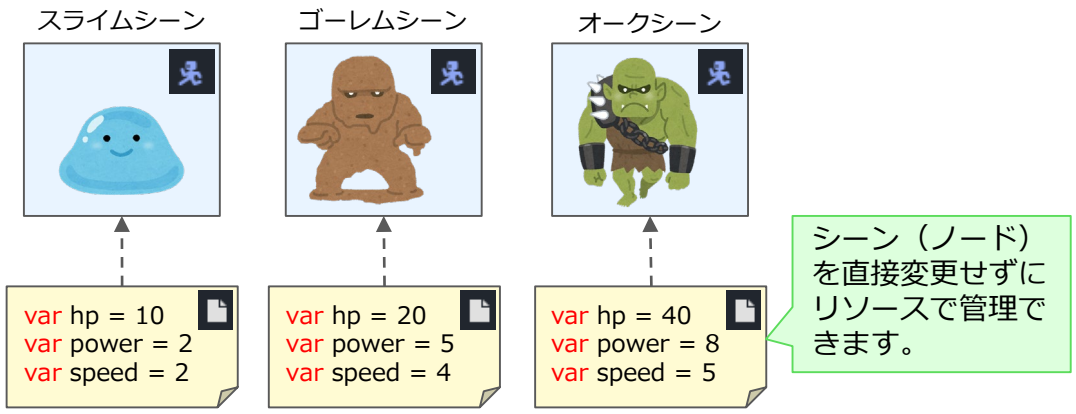
カスタムリソース（1）

カスタムリソースは、標準のリソース（マテリアル、テクスチャ、オーディオなど）とは別に、独自のデータ構造を持ったオリジナルのリソースのことです。

カスタムリソースを使うには、まずスクリプトファイルに変数やメソッドを定義し、それを元にリソースファイルを作成します。



例えば敵ユニットのパラメータをカスタムリソースとして抜き出しておくと、ノードを直接変更することなく設定値を管理することができます。

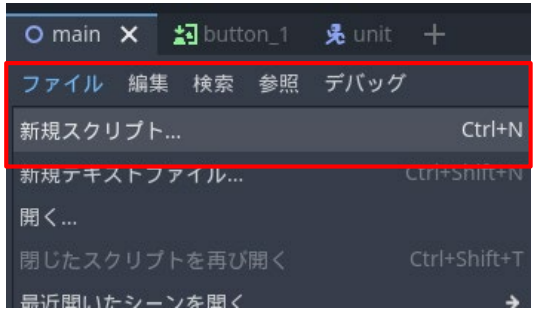


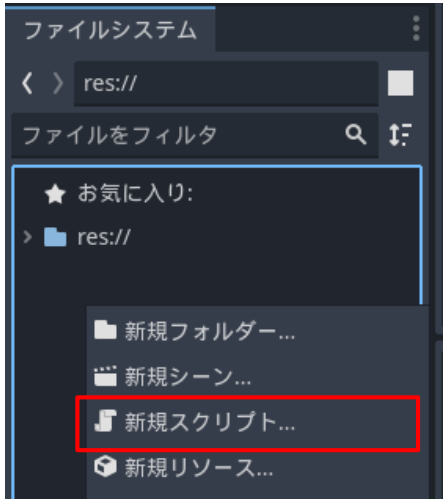
■カスタムリソースの作成

例として、ゲームに登場する敵ユニットのパラメータを管理できるようなものをつくります。

①スクリプトの新規作成

スクリプトは、Scriptのファイルメニューや、ファイルシステムドック上で右クリックなどからでも作成できます。

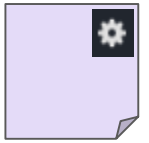




継承元を「Resource」にしておくこと。

スクリプトファイル名として「enemy_stats.gd」にしておきました。

これでResourceを継承したスクリプトファイルになります。



Next Page

カスタムリソース（2）

②パラメータの設定

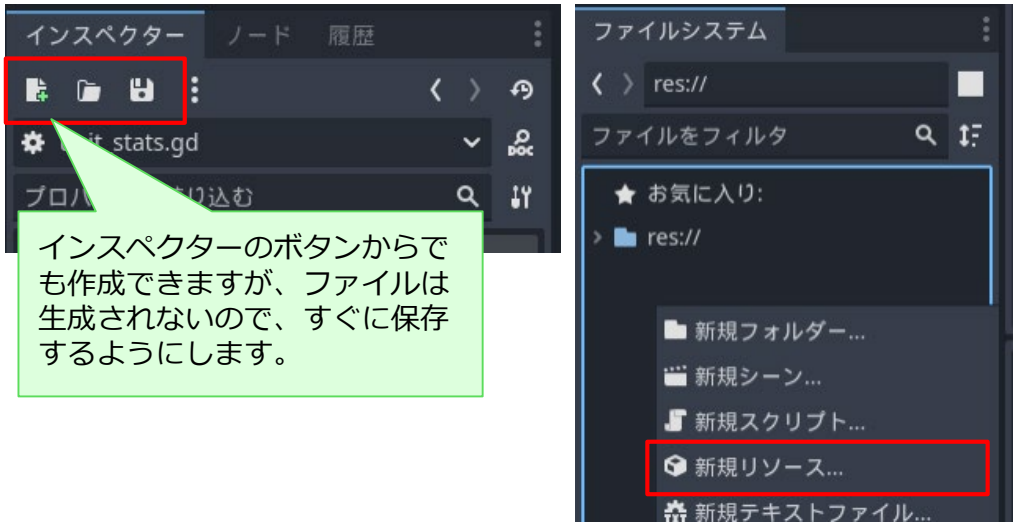
スクリプトにリソースの元となる変数やメソッドを書き込みます。ここでは例として敵ユニットのパラメータとなる変数を設定します。

```
1   extends Resource
2   class_name EnemyStats
3
4   @export var hp: int = 20
5   @export var power: int = 5
6   @export var speed: int = 4
```

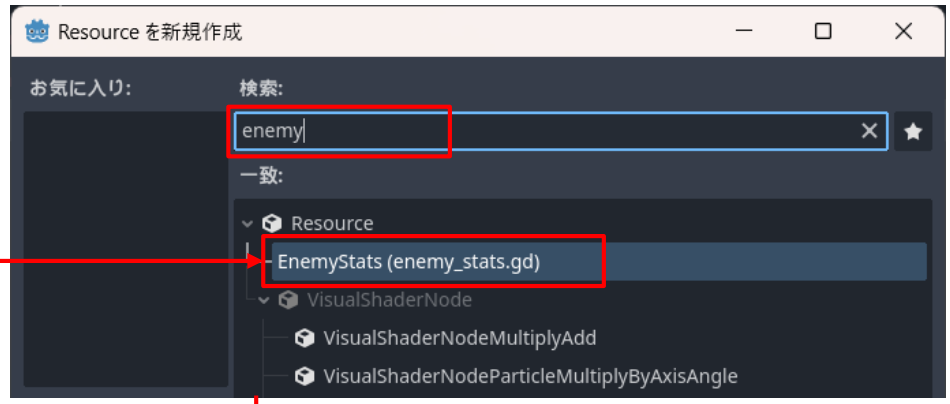
2行目ではクラス名を指定しています。これは次の工程で、このスクリプトを元にリソースを作成する際の呼び出し名になります。

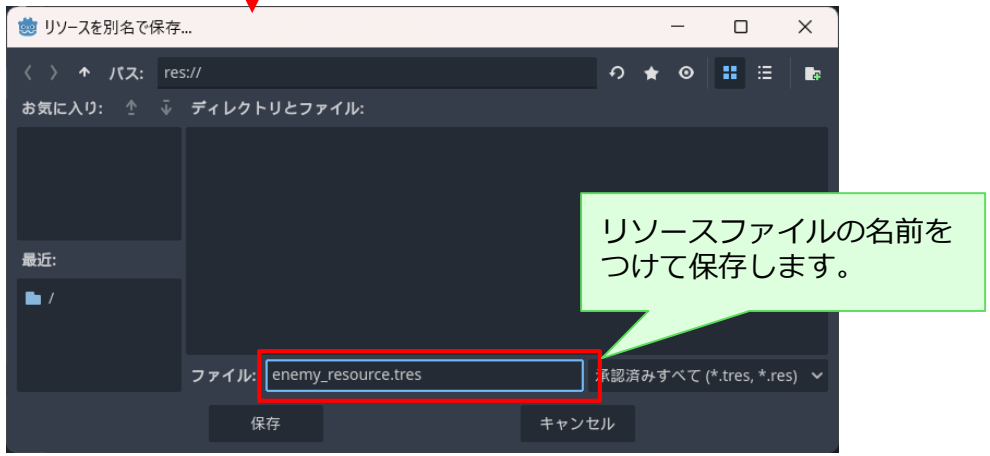
③リソースの作成

作成したスクリプトファイルを元にリソースファイルを作成します。



新規作成画面が開いたら、さきほどクラス名に設定した名称を入れて検索をすると見つけられます。





これでリソースファイルができました。

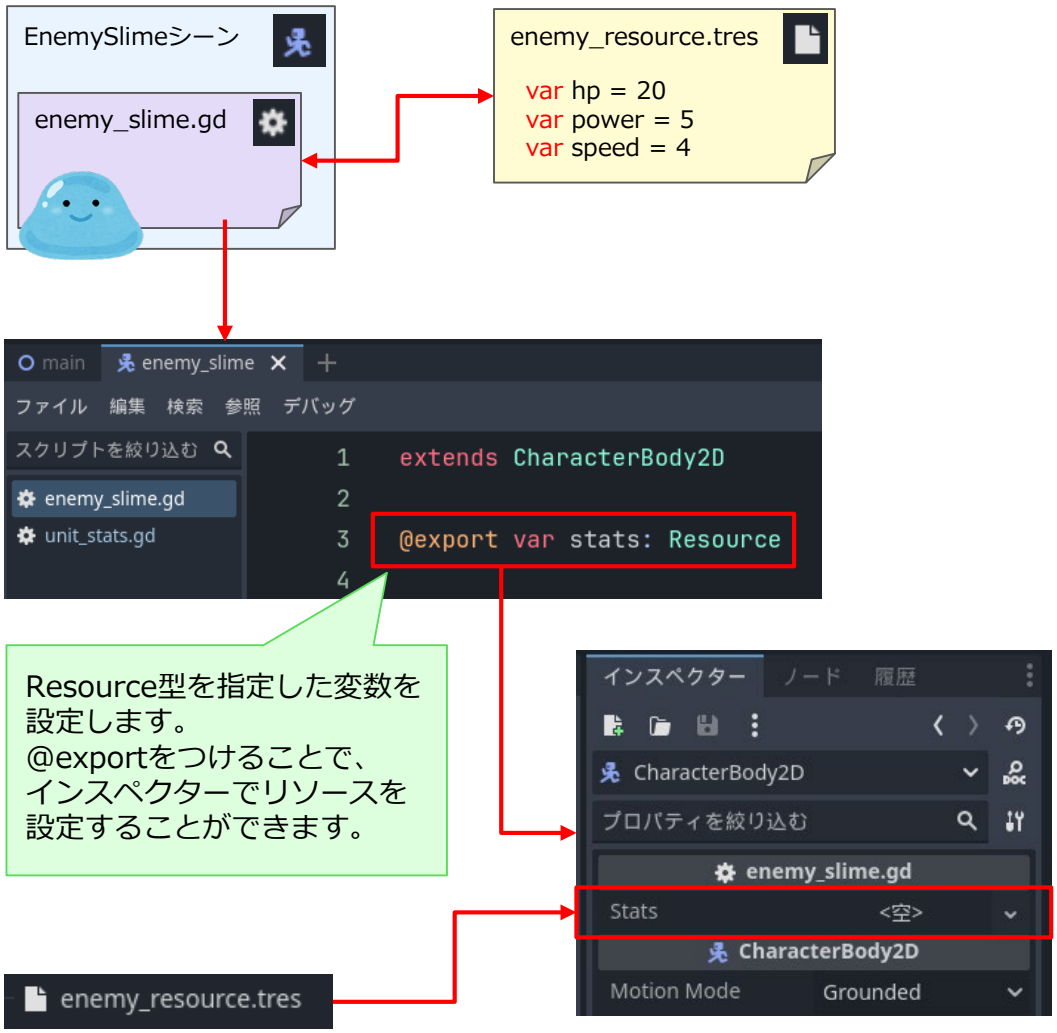
enemy_resource.tres

```
var hp = 20
var power = 5
var speed = 4
```

Next Page

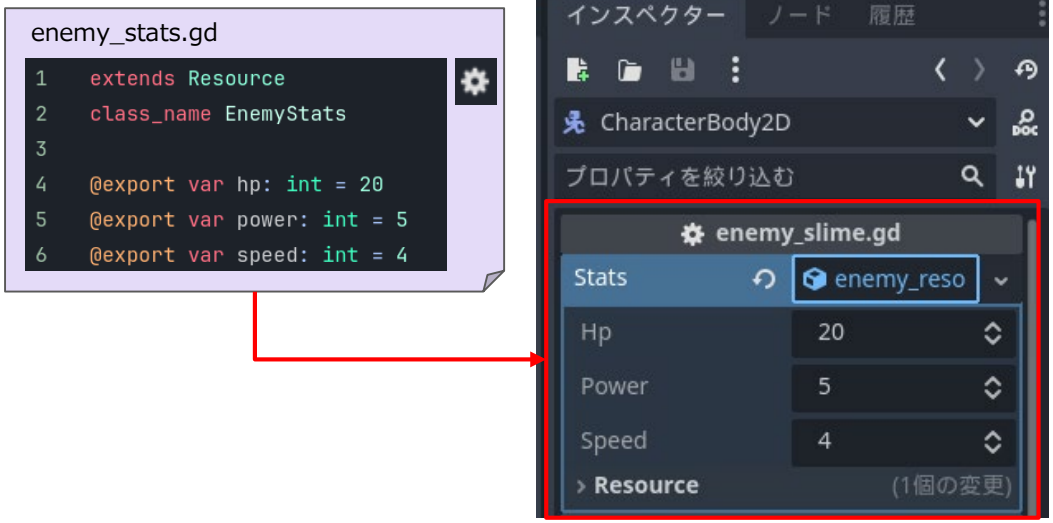
カスタムリソース（3）

④カスタムリソースの設定
作成したカスタムリソースをシーンに結びつけます。カスタムリソースを使用するシーン（ノード）のスクリプトファイルで設定します。



作成したカスタムリソースをファイルシステムから、インスペクターにドラッグして設定します。

これで、はじめに作成したスクリプトファイルで設定した変数を、ノードのインスペクターで編集可能になります。




```gdscript
1  extends CharacterBody2D
2
3  @export var stats: Resource
4
5  func _ready():
6      print("HP:", stats.hp)
```

カスタムリソースのパラメータには、リソース用に設定した変数を通じてアクセスします。

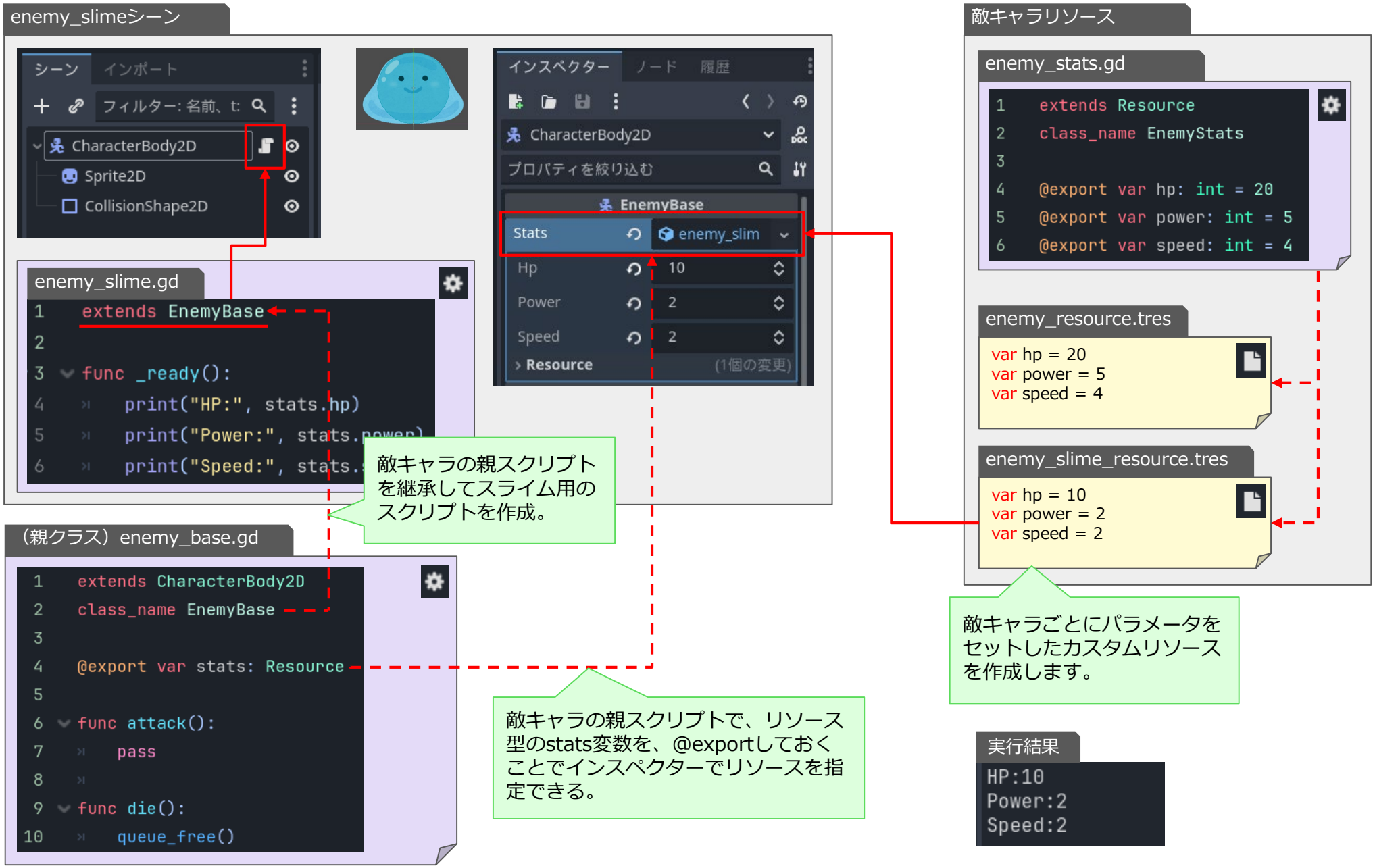
実行結果

```
HP:20
```

手順としては少し複雑になりますので、シンプルなゲームをつくる場合には不要ですが、こんな方法もあることは覚えておくと良いでしょう。

継承スクリプトとカスタムリソース

継承スクリプトとカスタムリソースを組み合わせる管理方法の全体像です。



コンポジション

スクリプトやシーンの継承のしくみを使って、シーンで共通する要素を1つにまとめて管理し、効率よく開発を進められますが、異なるアプローチとして「Composition（コンポジション）」という手法があります。

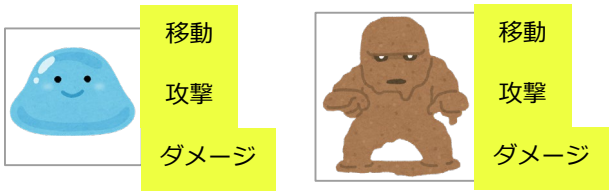
これはGodotに限定した手法ではありませんが、継承に比べるとシンプルで再利用性が高いため、よく使う機能はコンポーネントとして作成しておいて、他のプロジェクトでも再利用する使い方もできます。

利用例として、敵キャラに共通する機能を考えます。

移動　攻撃　ダメージ

継承の場合はまとめて1つのスクリプトに書いておいて継承するようなイメージですが、コンポジションの場合は、これらの機能をそれぞれ「**コンポーネント**」と呼ばれる部品にし、敵キャラに接続するイメージです。

移動　コンポーネント
攻撃　コンポーネント
ダメージ　コンポーネント

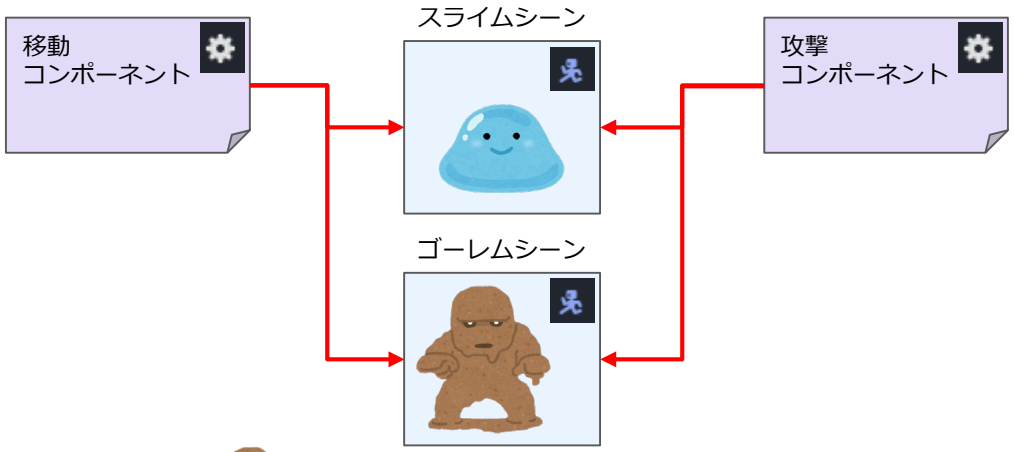


コンポーネントは単体で使うことができるので、必要なコンポーネントだけを組み合わせて使うことができます。

移動はしないけど、攻撃、ダメージの挙動はする場合。

継承の場合は、親スクリプトと子スクリプトが密接に関連しているのに対して、コンポジションを使う場合は、1つひとつが部品化されているため、柔軟に使いやすく、一度つくったコンポーネントを使いまわししやすいことも利点です。

コンポーネント自体はGodotの特別な機能ではなく、スクリプトやノードを使って作成します。

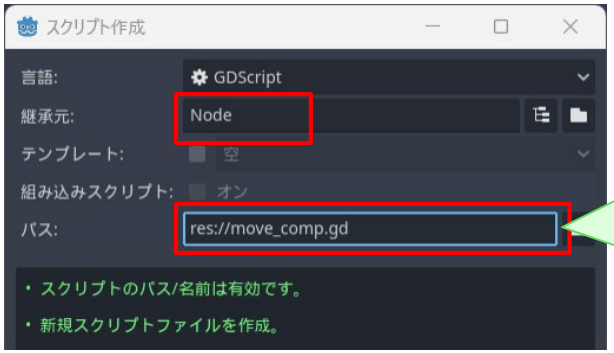



```gdscript
extends Node
class_name MoveComp

@export var character_body: CharacterBody2D
@export var speed: int = 200
@export var dir: float = -1.0

func _process(delta: float):
```

スクリプトファイルを作成して、シーン内のNodeノードにアタッチしています。
汎用部品をつくって使いまわしやすくなります。

コンポーネントの作成

例としてキャラクターの移動処理を行うコンポーネントを作成し、敵キャラシーンに作成したコンポーネントを割り当てることで、敵キャラクターが移動するようになります。



新規スクリプトを作成します。ここではパスにres://move_comp.gdとしています。継承元はNodeのままです。

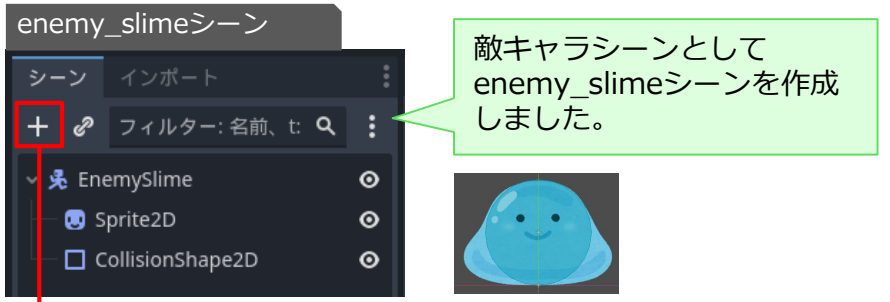
コードは以下のようにしています。@exportで設定値を変えられるようにしておきます。

move_comp.gd

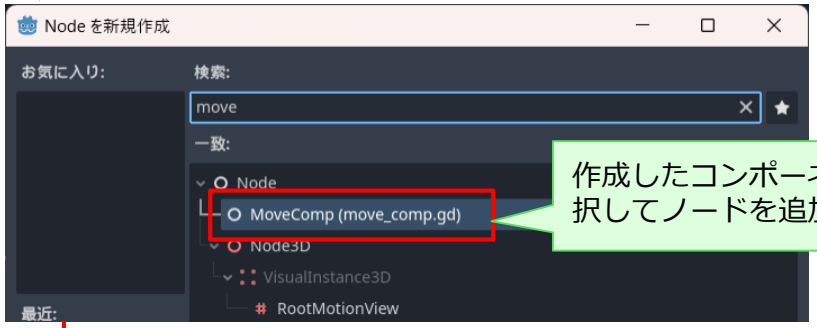
```
extends Node
class_name MoveComp

@export var character_body: CharacterBody2D
@export var speed: int = 200
@export var dir: float = -1.0

func _process(delta: float):
    character_body.velocity.x = speed * dir
    character_body.move_and_slide()
```

変数：character_bodyには、移動するキャラクターのノードを後から指定できるようにしておきます。

対象となるノードに対して、シンプルな移動処理のみ記載しています。このように必要な機能だけをもった部品をつくるのがコンポーネントの特徴です。



敵キャラシーンとしてenemy_slimeシーンを作成しました。



作成したコンポーネントを選択してノードを追加します。

コンポーネントは通常のノードを追加する操作と同じようにシーンに追加します。

インスペクターでCharacter Bodyにルートノードを割り当てれば設定完了です。