

Introduction

Whether you are creating a traditional RPG, a roguelike, an MMO, or even a game with unique mechanics, Simple RPG Core can adapt to your needs. By providing a robust framework for managing attributes, statistics, levels, classes, and more, it allows you to focus on the creative aspects of game development, simplifying the implementation of the more monotonous parts with a scalable and maintainable 100% inspector-driven experience.

The possibilities are endless, unleash your creativity.

Vocabulary of Simple RPG Core

The package is developed around the concept of *entity*, so let's clarify what we mean by this term in the context of Simple RPG Core. In its most minimal version, an entity is a `GameObject` that has a set of statistics. Optionally, an entity can have attributes, can level up, and can have a class. Let's clarify what we mean by each mentioned term.

Statistics (Stat)

A statistic is a value that quantifies an aspect of the entity. The meaning of this aspect is solely due to the concept it refers to.

Examples

In an RPG, a statistic can be `physical damage`. The concept of physical damage refers the player to the amount of damage inflicted by physical attacks, whether with weapons or without. Other statistics can be `ability power`, `defense`, `speed`, `armor penetration`, `range`, etc.

Attributes

An attribute is a value that can influence the value of one or more statistics. The weight of its influence on the statistics can be variable.

Examples

In an RPG, attributes can be: `strength`, `dexterity`, `intelligence`, `constitution`, etc. Considering the previous example of statistics, `strength` could influence `physical damage`, `dexterity` would increase `speed`, `intelligence` would increase `ability power`, and `constitution` would increase `defense`.

Experience and Level

The entity can gain experience and level up. This functionality is used by the class to express how attributes and statistics grow with levels, for that particular class.

Class

The class is associated with a set of statistics and optionally a set of attributes. The class describes how statistics and attributes vary with levels.

Examples

In RPGs most common classes are: `warrior`, `rogue`, `mage`, `paladin`, and so on. These classes have different attribute values. For example, a warrior will have more `strength` and `constitution` than a mage. The `rogue` might have the highest `dexterity`, etc.

How is Simple RPG Core organized and how does it work?



Entity

A `GameObject` becomes an entity once the `EntityCore` and `EntityStats MonoBehaviours` (Mono) are added to it. `EntityCore` comes with a built-in `EntityLevel` (plain C# `class`) that manages the experience and the level of the entity.



Stat

A `Stat` is a class that derives from `ScriptableObject` (SO) and represents a statistic in the game. Each statistic has a name (the name given to the SO instance of the created `Stat`), and we can choose whether to provide it with a maximum and/or minimum value. Additionally, we can define how that statistic grows or is reduced, depending on certain `Attributes`.



StatSet

A `StatSet` is a class that derives from SO and defines a set of `Stats`.



EntityStats

`EntityStats` allows us to configure:

- the base statistics
- the flat modifiers
- the `StatToStat` modifiers
- the percentage modifiers We will see what these modifiers are in the section (TODO).

The base statistics can be *fixed*, or instead derive from a class if the entity has one assigned. If we use the fixed ones, we must also provide a `StatSet`, while if we use those of a class, the class's `StatSet` will be used. If the entity levels up and we want its statistics to grow with levels, we are forced to use a class, as the *fixed* statistics are immutable.



Class

Class derives from SO and represents a game class. Each class has a name, a **GrowthFormula** that defines how the base Max HP grows with levels, a **StatSet**, optionally an **AttributeSet**, and associates each **Stat** of the provided StatSet with a **GrowthFormula** that describes how the statistic varies with levels. Similarly, if an **AttributeSet** is provided, it will be possible to associate a **GrowthFormula** for each **Attribute** present in the set, to describe how the attributes vary with levels.



EntityClass

EntityClass derives from Mono and allows us to assign a **Class** to our entity.



Attribute

An **Attribute** is a class that derives from SO and represents an attribute in the game. Each attribute has a name and, like statistics, can have a maximum and minimum value.



AttributeSet

An **AttributeSet** is a class that derives from SO and defines a set of **Attributes**.



EntityAttributes

Optionally, we can add the Mono **EntityAttributes** to our entity if we want to give it attributes.

EntityAttributes allows us to specify how many attribute points to provide at each new level. These points can be spent on various attributes to increase their value. For **EntityAttributes** we can configure:

- the base attributes
- the flat modifiers
- the percentage modifiers Similarly to **EntityStats**, we can decide whether the base attributes are *fixed* or if they instead derive from the class associated with **EntityClass**.



Growth Formula

To express how **Stats**, **Attributes**, Max HP, and the experience required to level up vary at each level, we can use instances of **GrowthFormula**. This is a class that derives from SO and allows us to define a mathematical function, or a system of functions, that describe how a value changes as levels increase. We will see in more detail how to define a **GrowthFormula** in (TODO).

How is Simple RPG Core implemented?

The package is developed following the principles of SOAP (Scriptable Object Architecture Pattern), and has been inspired by the [GDC talk of Ryan Hipple](#)[↗]. In a nutshell, the main benefits provided by this architecture are:

- **encapsulation:** separation of game logic from data. Game logic code shouldn't mix with data. All data is nicely wrapped withing SO instances
- **game designers friendly:** game designers can make changes and balancements from the inspector without touching the code
- **greater reusability:** Each object is a `ScriptableObject` that can be reused by many components
- **greater testability:** being data separated from code, is easier to isolate and fix bugs. Moreover, SO events can be raised with ease at the press of a button from the inspector interface, easing and speeding up debugging even further.

Flexibility of Simple RPG Core

Although the package is specifically designed for RPG games or games with progression systems, its flexibility allows it to be used in almost any game. As it allows creating attributes like `strength`, `dexterity`, `agility`, etc., and statistics such as `physical attack`, `magic power`, `physical defense`, etc., in RPG, Roguelike, MMO games, etc., nothing prevents it from being used, for example, to implement a firearm. The attributes could be `weight`, `size`, `ergonomics`, etc., and the statistics `recoil`, `handling`, `stability`, `intimidation`, etc. Attributes can influence statistics. A heavier weapon could reduce `handling` but increase `stability`. A larger weapon could reduce `handling` but increase `intimidation`. A more ergonomic weapon could reduce `recoil` and increase `handling`. And so on... The weapon's levels, if present, influence the attributes and statistics, progressively improving them. Classes could represent weapon types (assault rifles, snipers, shotguns, etc.), and each class could have its own set of dedicated attributes and statistics. For example, shotguns could have, in addition to the aforementioned ones, the `barrel length` attribute that influences the `pellet spread` statistic.

Workflows

Mandatory and re-play fields

Fields marked with a red asterisk (*) are mandatory and must be filled out to ensure proper functionality of the framework.

Fields marked with an orange **R** are re-play fields. Any changes made to these fields during playtime will require a restart to ensure the changes take effect.

Some utilities

Almost every class provided by this package uses events or variables in the form of `ScriptableObject`. Therefore, let's quickly introduce these concepts so that we are clear about what we are talking about when we encounter them in the following paragraphs.

Game events as `ScriptableObjects`

The SOAP architecture allows us to implement the Observer pattern through scriptable objects. In the simplest case, with events without context, we can define various game events as `GameEvent` instances: a class that derives from `ScriptableObject`. For example, we can create an instance called `PlayerJumped` that represents the event "The player has jumped". This event will notify all listening systems when it occurs. Systems subscribe to this event using the `MonoBehaviour GameEventListener`. We can assign a `GameEvent` to this component, and it will handle the subscription and invoke a callback when the event is triggered. The callback is a [UnityEvent](#), so we can select a callback to invoke in response to our event directly from the inspector.

For more details, see the [Game Events section](#).

Int and Long Vars

Another common use of `ScriptableObject` in the SOAP architecture is to define variables. The main advantage of these variables in the form of SO is that they can be easily shared between various objects that may decide to share the same value. A common example is the player's game score. There could be a game manager that adds or removes points from this variable, while the UI HUD uses it to display its value on the screen. This way, we can keep the game manager and UI completely decoupled, passing shared values (like variables) through the inspector.

Int and Long Refs

`IntRef` and `LongRef` allow choosing whether to use a native value (`int` or `long`) or an `IntVar/LongVar`. As mentioned in the previous paragraph, `IntVar` and `LongVar` have the advantage of being shareable between different components/game objects, while native values are more immediate to use and require less setup (no need to instantiate an `IntVar/LongVar` and assign it in the inspector).

Thanks to a custom property drawer, it will be possible, from the inspector, to check a checkbox named `Use constant` to use a native value instead of a `Ref`, and vice versa.

`IntRef` and `LongRef` are widely used in the package's `MonoBehaviour`.

Game events

The package also supports game events with up to 4 context parameters. They are generics, but in Unity, it is not possible to instantiate classes that derive from `ScriptableObject` if they are generics with unspecified type parameters. To use them, we must explicitly declare classes that derive from the generic `GameEvent` and fix the type parameters with concrete types. To simplify the definition of new event types, with specific types as context parameters, the package provides `GameEventGenerator`. These generators, which derive from `SO`, allow generating the concrete classes of `GameEvent`. We will see these generators in more detail in the section (TODO). Some game events are already defined and made available by the package (see the [Samples](#) page).

Growth Formulas

Relative path: `Growth Formula` As already mentioned in [Introduction](#), `GrowthFormula` allows defining how a certain value varies as levels increase. A `GrowthFormula` can be instantiated through the hierarchy context menu by going to `Simple RPG Core -> Growth Formula`. The package provides a custom property drawer for `GrowthFormula`.


Max level for the values

In the inspector of a `GrowthFormula`, we can pass an `IntVar` to define up to which level to grow the values.

Use constant at level one

If the checkbox named `Use constant value at level 1` is checked, the respective constant value will be used.

Growth expressions

The various values of the `GrowthFormula` are defined by a function where values, the y-axis, are expressed as a function of the levels, the x-axis. Such a function is defined as a composite function. Each segment of the function is represented by a string that specifies a mathematical expression for a range of levels. The string can be defined by using the [Unity ExpressionEvaluator](#)  syntax. On top of it, the following terms can be used:

- `LVL`: the level at each iteration
- `PRV`: the previous value of the `GrowthFormula` (value evaluated at the previous level)
- `SPRV`: the second previous value of the `GrowthFormula` (value evaluated 2 levels ago)
- `SUM`: the sum of the values of the `GrowthFormula` from level 1 up to the previous level

Let's see an example of how to define a `GrowthFormula` for defining the Physical Attack of a warrior class. First of all, let's create a new `GrowthFormula` instance and name it `Warrior Physical Attack GF`. In the inspector, it should look like this:



Open

* Max Level

(Lx)Max Level (Int Var)



Use Constant At Lvl 1

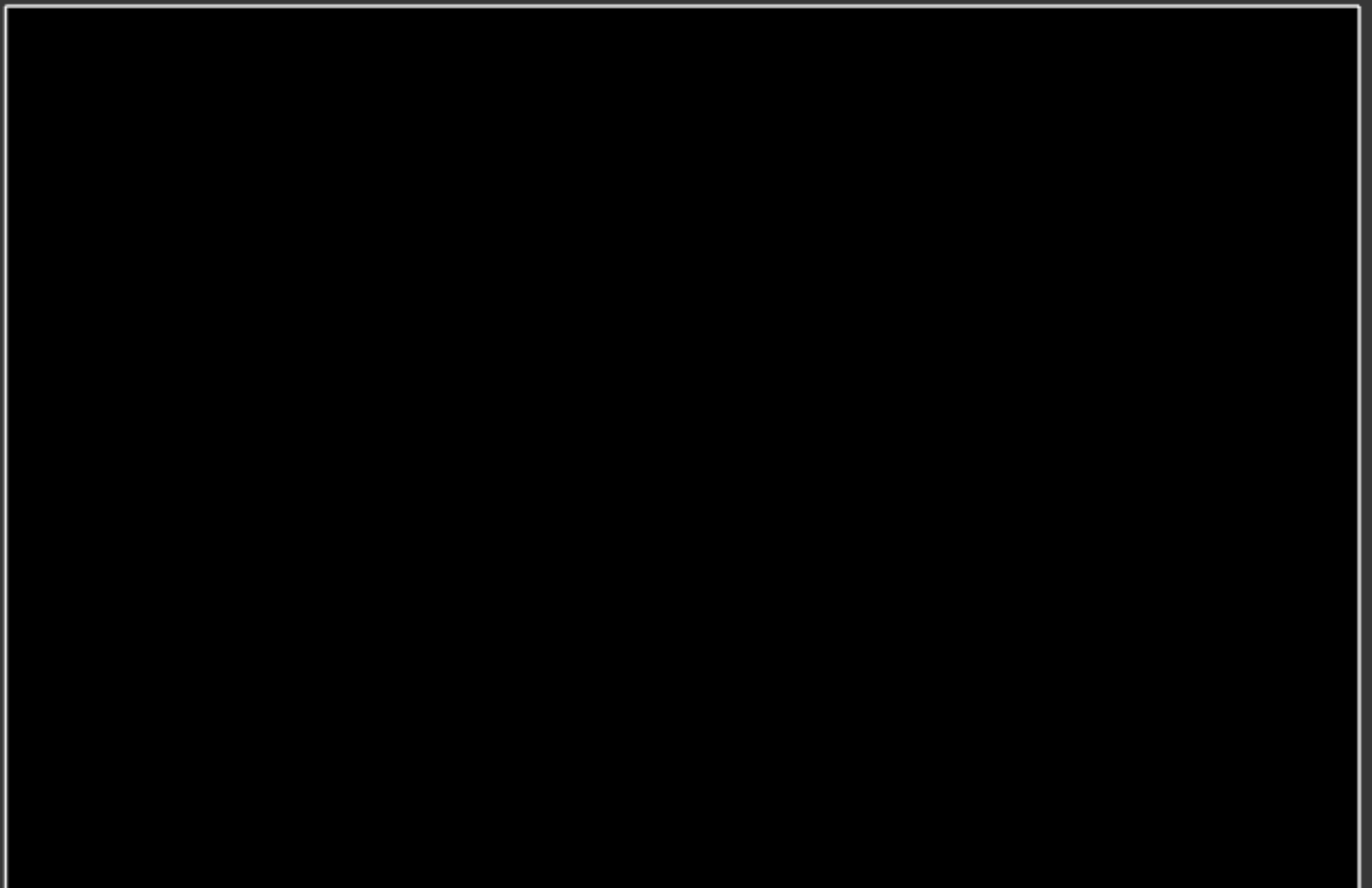


Constant At Lvl 1

0

Add new growth expression

Growth Values for each level - graph



1 4 7 10 13 16 19 22 25 28 30



Hold the mouse for a moment on top of the graph to see the value at a certain level

Growth Values for each level - table

lvl 1	0	lvl 16	0
lvl 2	0	lvl 17	0
lvl 3	0	lvl 18	0

lvl 4	0	lvl 19	0
lvl 5	0	lvl 20	0
lvl 6	0	lvl 21	0
lvl 7	0	lvl 22	0
lvl 8	0	lvl 23	0
lvl 9	0	lvl 24	0
lvl 10	0	lvl 25	0
lvl 11	0	lvl 26	0
lvl 12	0	lvl 27	0
lvl 13	0	lvl 28	0
lvl 14	0	lvl 29	0
lvl 15	0	lvl 30	0

The Max Level, a mandatory field, is set with an `IntVar` assigned by default. We can edit that variable to change the maximum level that will be computed for our growth formula.

⚠ **WARNING**

When modifying the value of a variable referenced in growth formulas, such as Max Level, the growth formulas are not directly updated unless you select them in the inspector. To update all growth formulas simultaneously after changing the maximum level, a command is available in the menu: `Tools > SOAP RPG Framework > Validate All Growth Formulas`.

Validation occurs automatically during script compilation, upon entering play mode, and when instantiating a prefab. This is achieved through the `OnValidate` callback, which ensures that formulas are updated accordingly.

The `Use constant value at level 1` checkbox lets us decide whether to use a constant value at level 1 or not. If checked, the `Constant Value` field will be enabled, and we can set a value for it. In this case, we set it to 10.

The `Add new growth expression` button lets us add a growth expression for a certain range of levels of our choice. If we press it, we will see the following:

After adding these growth expressions, the `GrowthFormula` for the `Warrior Physical Attack GF` should look like this:



Open

* Max Level ☐

Use Constant At Lvl 1 ☒

Constant At Lvl 1

From level 2 to level 10

From Level

Growth Expression

Remove

From level 11 to level 11

From Level

Growth Expression

Remove

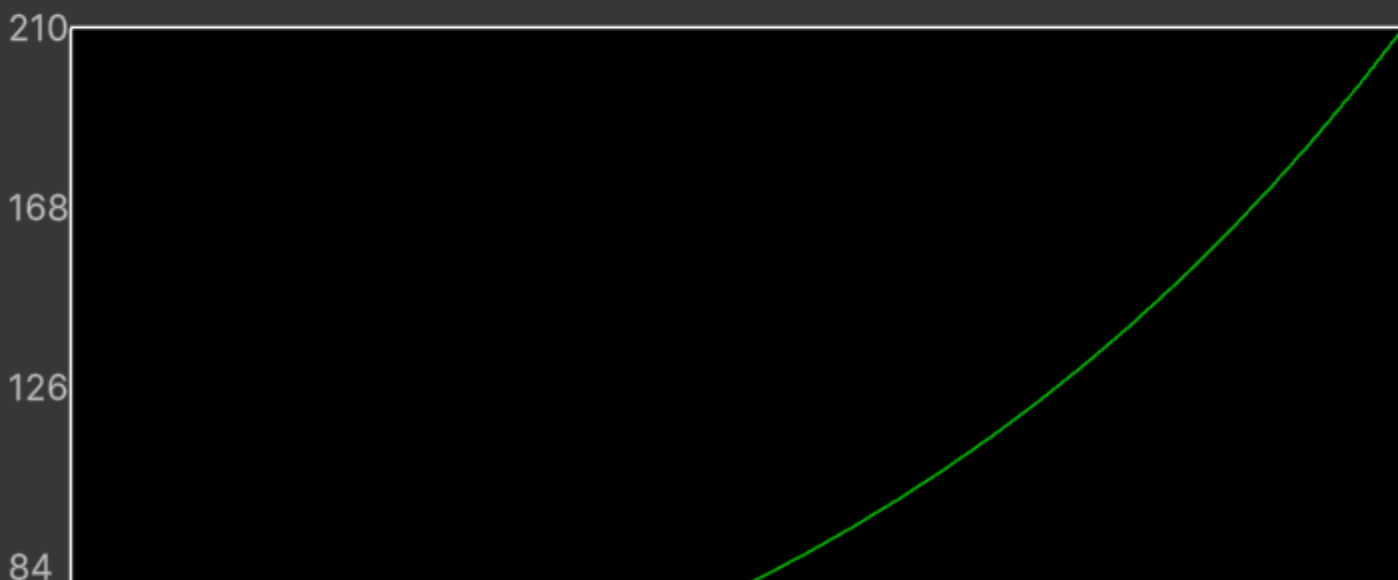
From level 12 to level 30

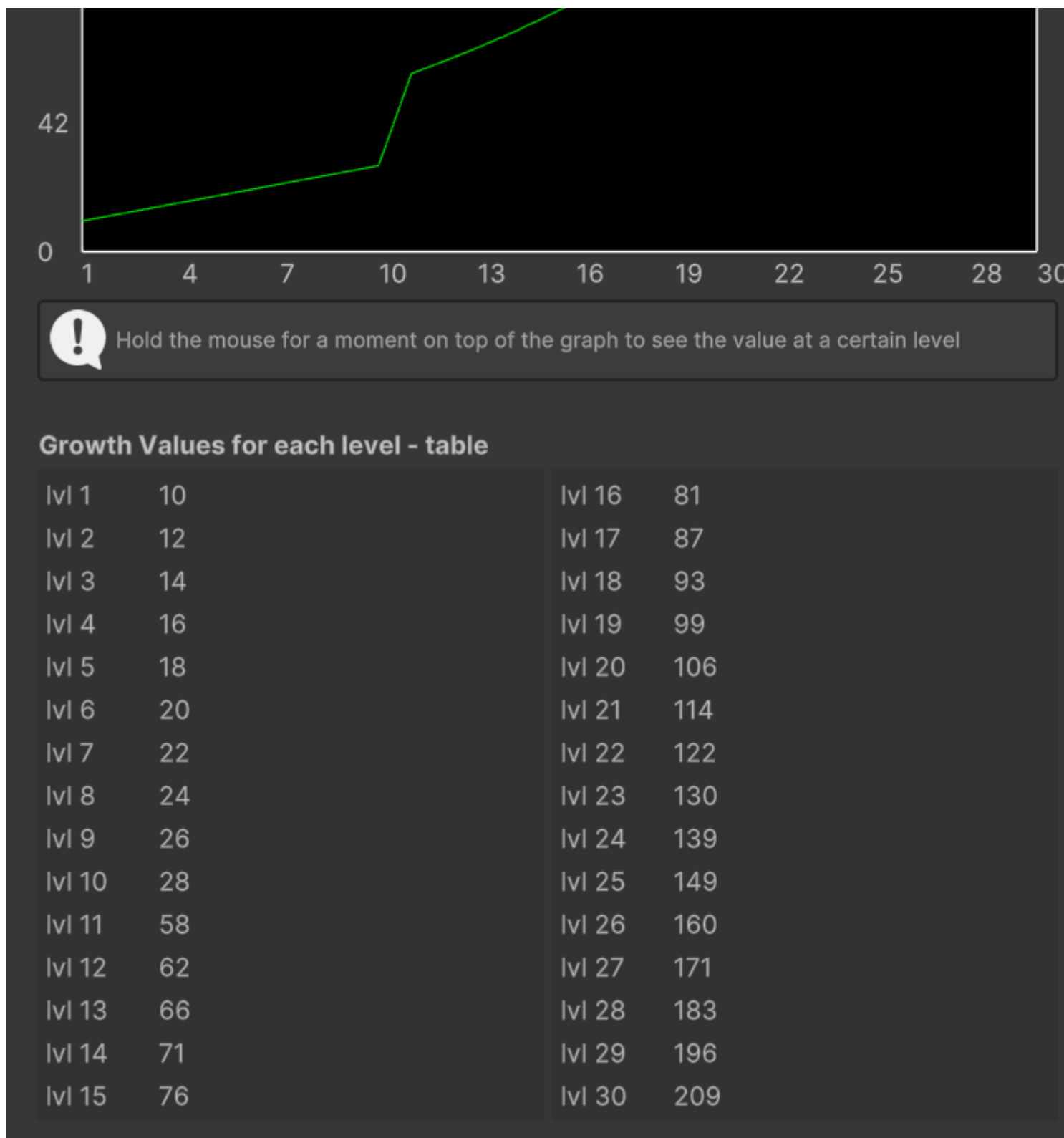
From Level

Growth Expression

Remove

Add new growth expression

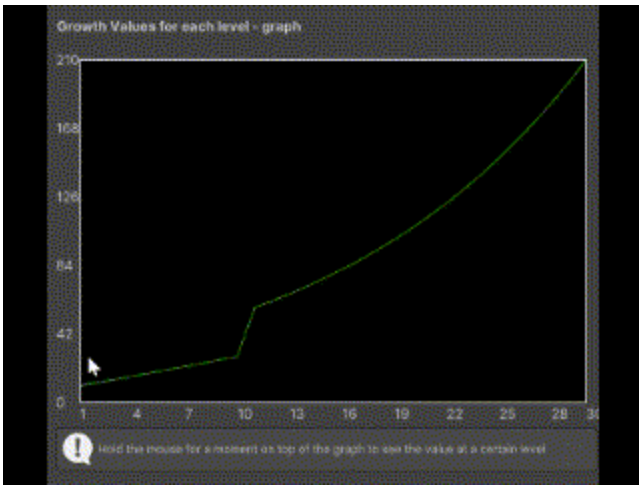
Growth Values for each level - graph



With this setup, the **GrowthFormula** will correctly calculate the Physical Attack values for the warrior class based on the specified rules.

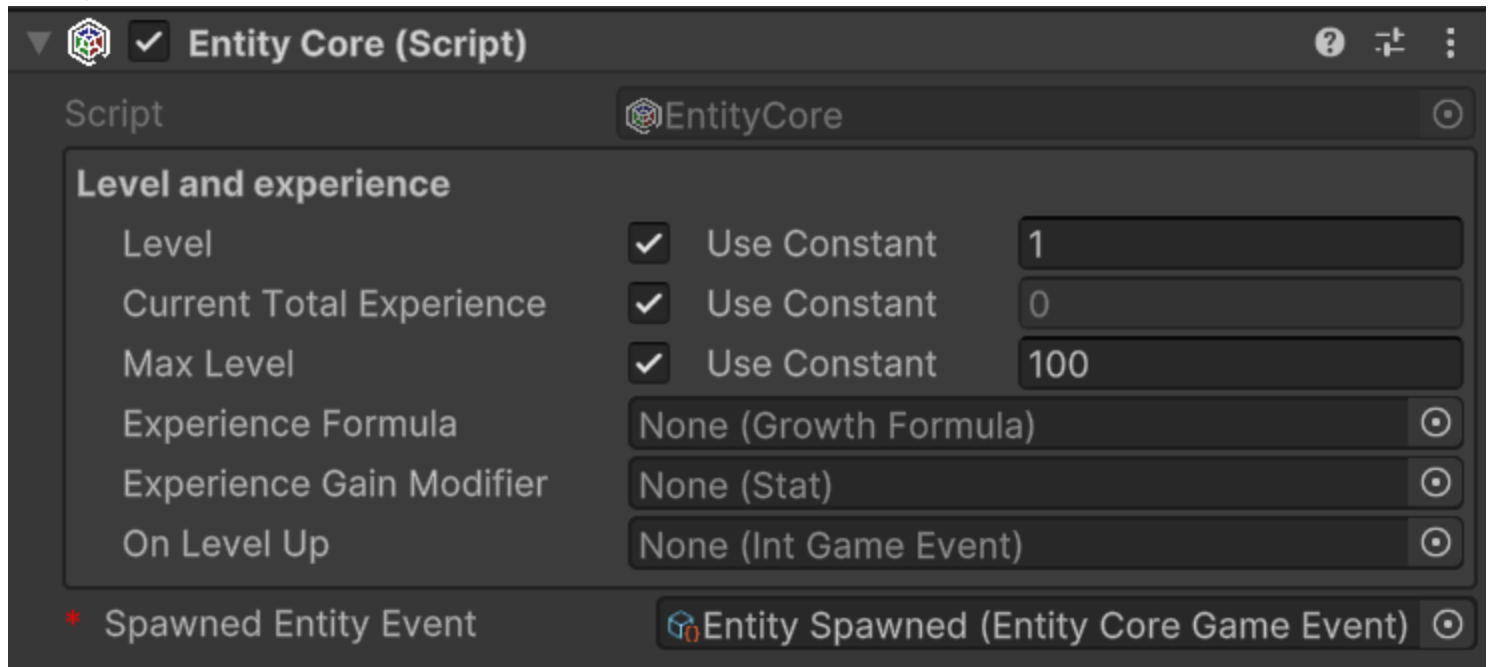
Interactive Chart

If you hold your mouse for a moment onto the chart, a label will show up, showing the exact value of the growth formula at the pointed level:



Make a `GameObject` an entity

To make a `GameObject` an entity, we need to add the `MonoBehaviour EntityCore` to it. Select your object from the hierarchy and click, in the inspector, on "Add component". Then search for and select `EntityCore`.



From the inspector, we can configure several values. Let's analyze them one by one.

Level: defines the entity's level. By changing its value, we can assign a different level to the entity directly from the inspector. This can be useful for testing purposes. You'll notice the `Use Constant` checkbox. If checked, you can pass an `IntVar` instead of using a constant.

Current Total Experience: Represents the total experience possessed by the entity.

⚠ WARNING

If you've passed a `LongRef` for the current total experience, the value contained in this variable should not be modified manually. If `Use constant` is checked instead, the value is readonly.

`Max Level`: The maximum level the entity can reach

`Experience Formula`: `GrowthFormula` that describes how the total experience required to reach the next level grows at each level.

`On Level Up`: `IntGameEvent` that should be raised when the entity levels up.

`Spawned Entity Event`: `EntityCoreGameEvent` that should be raised when this entity's `Start()` method is executed.

You may notice that a game event is already assigned to `Spawned Entity Event`. This is because an instance of that game event has been explicitly assigned directly in the inspector of the `EntityCore` script. This choice was made since in most cases the same event instance will always be used for entity spawning. This means you don't have to reassign this event every time you create a new entity in Unity. As we'll see later, this default assignment mechanism has been used for other components as well.

Creating Simple RPG Core assets

All the instances of the various assets that derive from `ScriptableObject`s can be created in the following ways:

- Context menu: Right click on the hierarchy > Create > Simple RPG Core
- Top bar: Assets > Create > Simple RPG Core
- Hotkeys: By pressing the respective keyboard shortcut while a folder or an element of the hierarchy is currently selected

i NOTE

For Mac users the `Ctrl` key corresponds to the `Cmd` key.

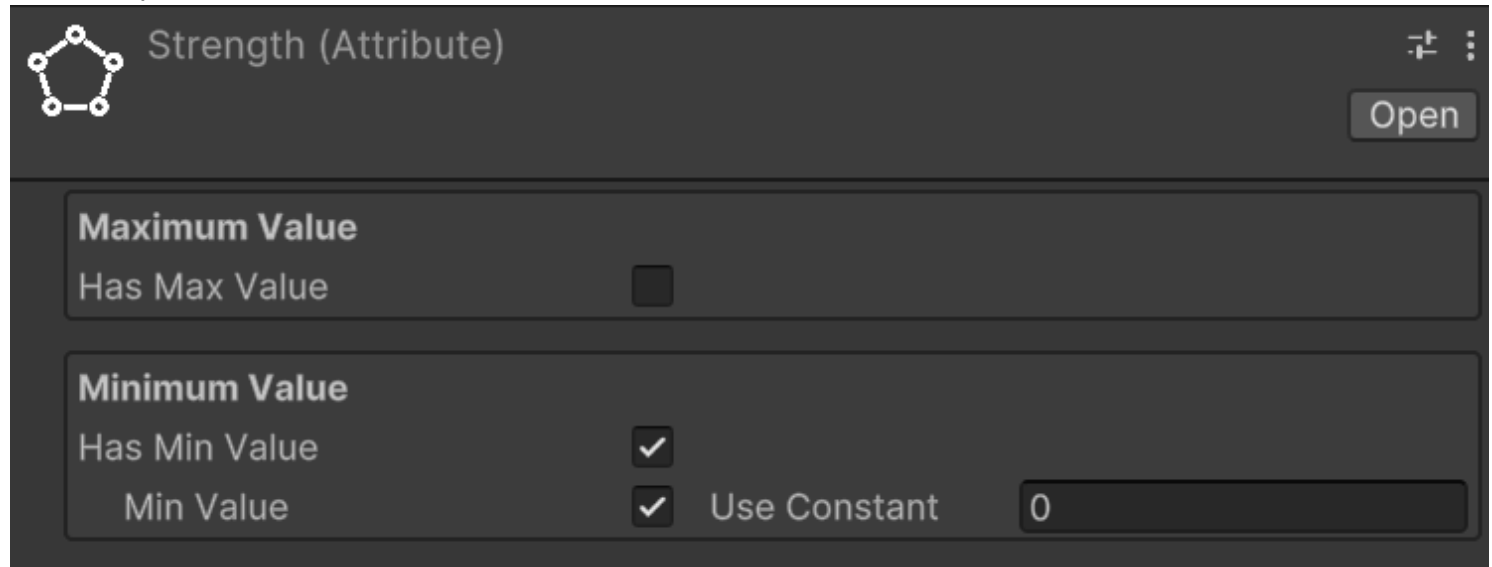
Create attributes

Keyboard shortcut: `Ctrl + Alt + A`

Relative path: `Attribute`

Once created a new attribute you can name it as you wish and you'll be able tweak some settings in the inspector. For example lets create a **Strength** attribute. Create an **Attributes** folder in your hierarchy, then press **A** and name the newly created attribute **Strength**.

In the inspector it should look like:



By checking **Has Max Value**, we will set a maximum value for the attribute. By default, there is no maximum value.

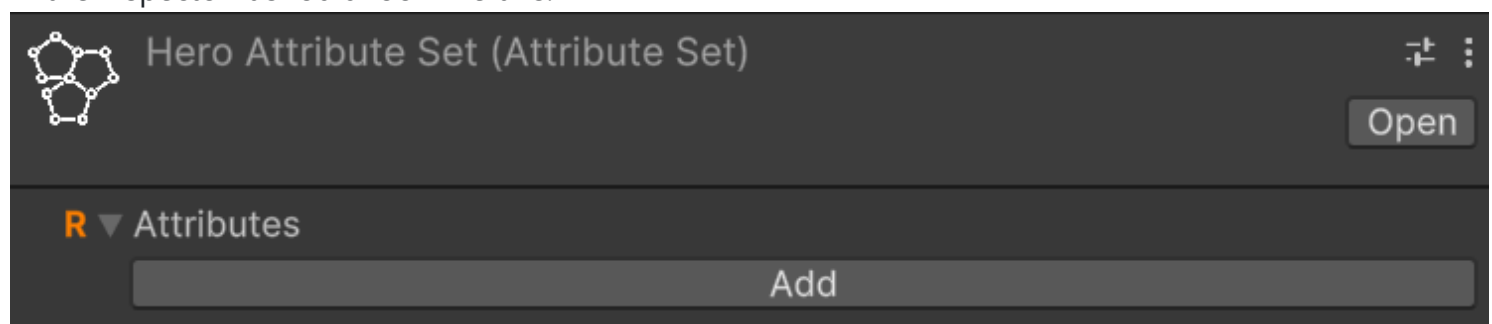
By checking **Has Min Value**, we will set a minimum value for the attribute. By default, the minimum value is zero.

Repeat the process for also the **Constitution**, **Intelligence**, and **Dexterity** attributes.

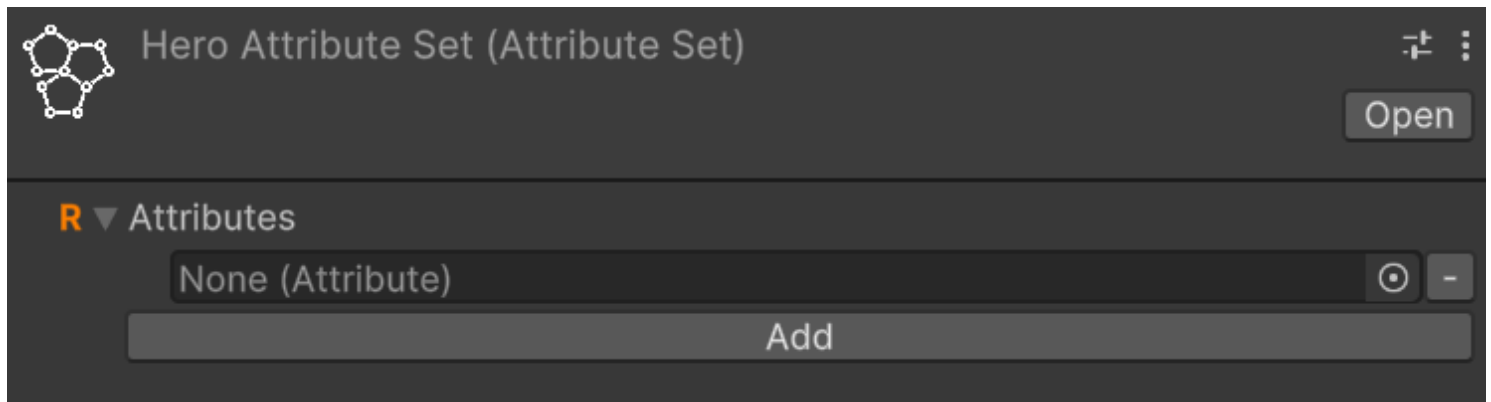
Create an attribute set

Relative path: **Attribute Set**

Now that we have some attributes let's create an **AttributeSet** named, for example, **Hero Attribute Set**. In the inspector it should look like this:



An attribute set without attributes isn't very useful, so let's add the previously created ones, one at a time. To do this, click on the **Add** button. Notice that an entry with **None (Attribute)** appears:

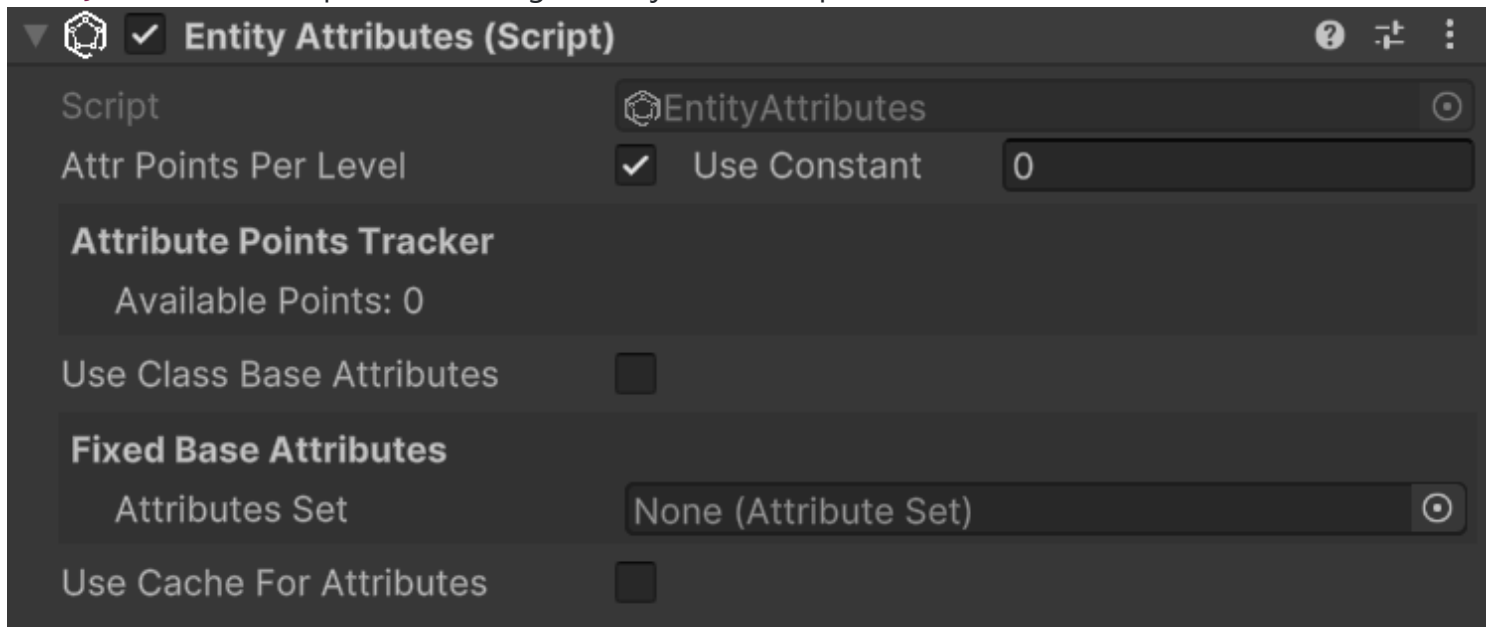


To assign an attribute to the entry, we can either drag & drop from the hierarchy or click on the small circle button on the right of the newly appeared entry. This mechanism is the same used for public variables or, more generally, for fields annotated with `SerializeField`, so it will be familiar to you. Let's add `Strength` using whichever method you prefer. Repeat the process of adding an attribute to the set for `Constitution`, `Intelligence`, and `Dexterity` as well.

If you want to remove an attribute from the set, you can click on the small `-` button on the right of the attribute you want to remove.

Add `EntityAttributes` to an entity

The next step is to assign the attribute set we created to an entity. To do this, let's add the `EntityAttributes` component to our game object. The inspector will look like this:



An entity has base points for attributes, which can be either fixed or derived from a class, a configurable amount of attribute points that can be arbitrarily assigned, and these points are granted at each level-up, along with flat and percentage modifiers for the attributes. Except for the modifiers, which can only be assigned via code, all other values can be configured from the inspector.

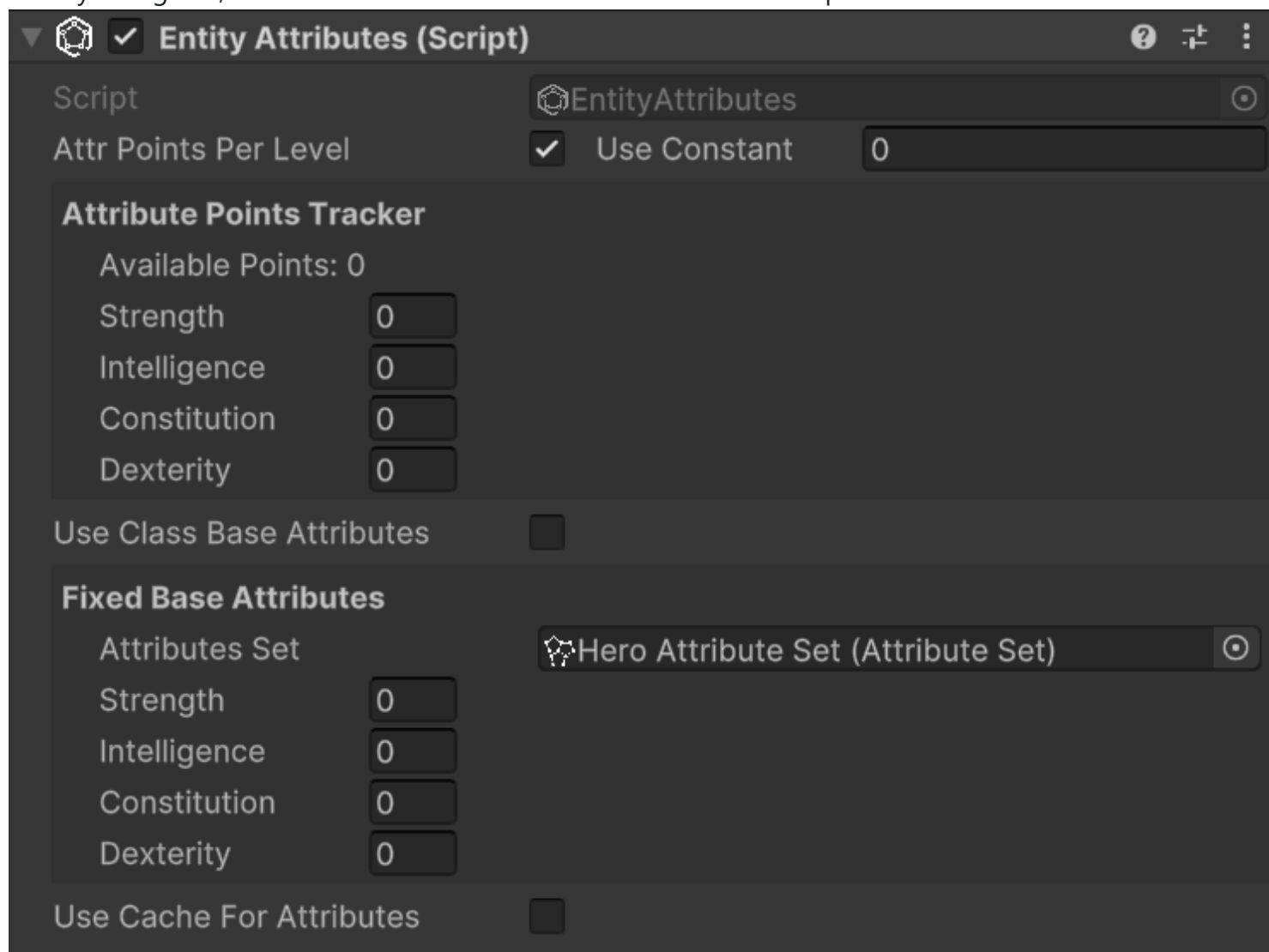
Attr Points Per Level defines how many arbitrarily spendable attribute points are provided at each level-up. They are assigned starting from level 2 on.

Attribute Points Tracker allows monitoring and assigning spendable points. **Available Points** defines how many unspent points are still available.

If you change the level of the entity you'll see that available points change accordingly. And as you spend them, **Available Points** will decrease.

Moreover, there is a checkbox labeled **Use Class Base Attributes**. For now, let's leave it unchecked since we haven't added a class yet. However, in this case, we need to manually assign an attribute set.

Therefore, let's set the **Attribute Set** field found under **Fixed Base Attributes** with the **Hero Attribute Set**. By doing this, we now have access to additional fields in the inspector:



We can assign values to the attributes of **Fixed Base Attributes** as we see fit.

Adding Modifiers

While base attributes are set in the inspector, modifiers can be added through code using these methods:

```
// Add flat bonus
entityAttributes.AddFlatModifier(attribute, value); // Adds fixed amount

// Add percentage bonus
entityAttributes.AddPercentageModifier(attribute, percentage);
```

The modifiers are applied in this order:

1. Base value
2. Spent attribute points
3. Flat modifiers
4. Percentage modifiers

For example, with:

- Base Strength: 10
- 2 spent points
- Flat modifier: +3
- 40% Strength increase

The final calculation would be:

1. Base (10) + Spent (+2) = 12
2. 12 + (Flat) + 3 = 15
3. 15 + (15 * 0.4) = 21

When adding modifiers through code, the attribute cache will automatically be invalidated to ensure the correct value is returned on the next access.

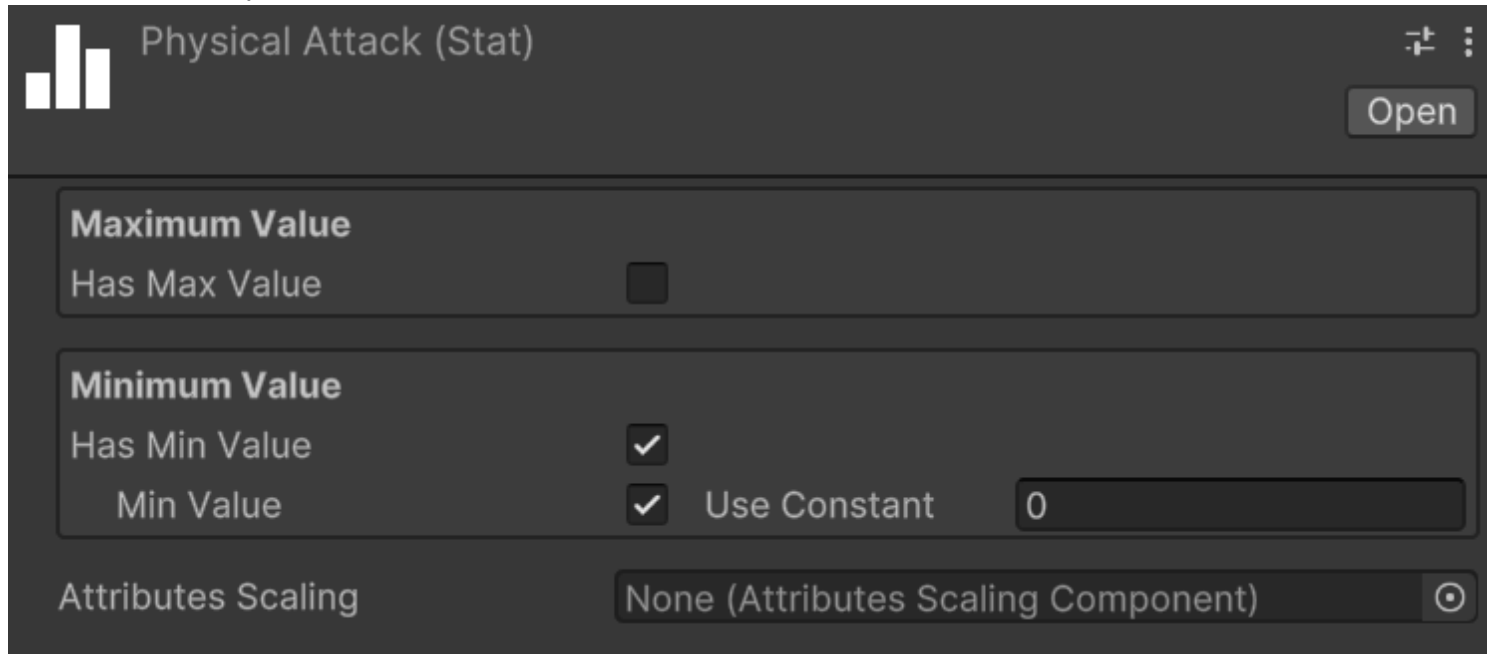
Create stats

Keyboard shortcut: **Ctrl + Alt + S**

Relative path: **Stat**

As with attributes, you can create stats as you wish and assign them the names you prefer. Let's create the **Physical Attack** stat together. Create a new **Stats** folder, select it and press **S**. Name it **Physical**

Attack. In the inspector, it should look like this:



The image shows a UI inspector for a stat named "Physical Attack (Stat)". It has a title bar with a bar chart icon and an "Open" button. The main area is divided into three sections: "Maximum Value" with a "Has Max Value" checkbox (unchecked), "Minimum Value" with a "Has Min Value" checkbox (checked) and a "Min Value" section containing a "Use Constant" checkbox (checked) and a text field with the value "0", and "Attributes Scaling" with a dropdown menu set to "None (Attributes Scaling Component)".

As with attributes, you can assign both a maximum and a minimum value to a stat.

Repeat the process for the **Magical Power**, **Defense**, and **Critical Chance** stats.


Unlike attributes, however, stats include **Attributes Scaling**.

Create an Attribute Scaling Component for Stats

*Relative path: **Scaling** -> **Attribute Scaling Component***

Let's create a new **Attribute Scaling Component** to use with the strength stat we created earlier. Create a new folder named, for example, **Attribute Scalings for Stats**, and inside it, create an attribute scaling component called **Physical Attack Strength Scaling**.

Assign the previously created **Hero Attribute Set** to the **Set** field. You will see the attributes of the set appear. Here, you can assign scaling values using **double**. For example, set the scaling of **Strength** to **1.0**. This component defines a 100% scaling on the value of **Strength**.

Physical Attack ASC (Attributes Scaling Component)

Open

Script


AttributesScalingComponent

* Attribute Set

Hero Attribute Set (Attribute Set)

Scaling Attribute Values

Strength	1
Intelligence	0
Constitution	0
Dexterity	0

 Use double values. For example: 0.8 means 80%, 1.6 means 160%.

Now, assign this scaling component to the **Physical Attack** stat to ensure it scales with the **Strength** attribute.

Create a stat set

Relative path: **Stat Set**

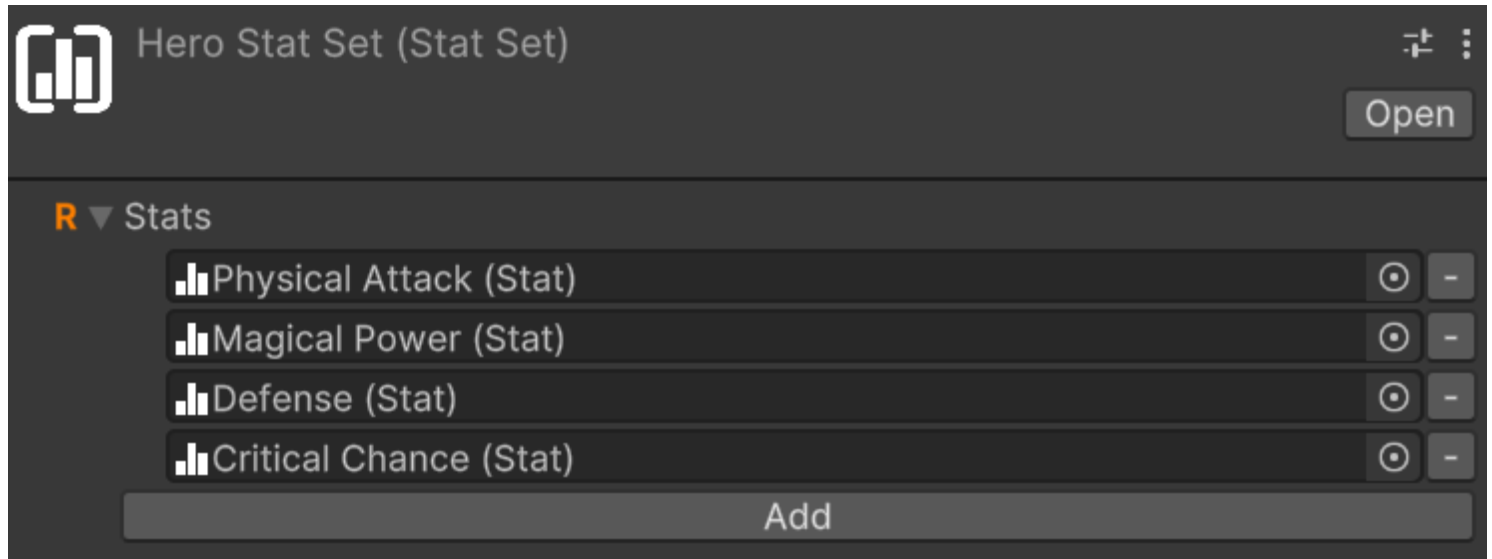
Now that we have some stats, let's create a **StatSet** named, for example, **Hero Stat Set**.

A stat set without stats isn't very useful, so let's add the previously created ones, one at a time. To do this, click on the **Add** button. Notice that an entry with **None (Stat)** appears. To assign a stat to the entry, we can either drag & drop from the hierarchy or click on the small circle button on the right of the newly appeared entry. This mechanism is the same used for public variables or, more generally, for fields annotated with **SerializeField**, so it will be familiar to you.

Let's add **Physical Attack** using whichever method you prefer.

Repeat the process of adding a stat to the set for **Magical Power**, **Defense**, and **Critical Chance** as well.

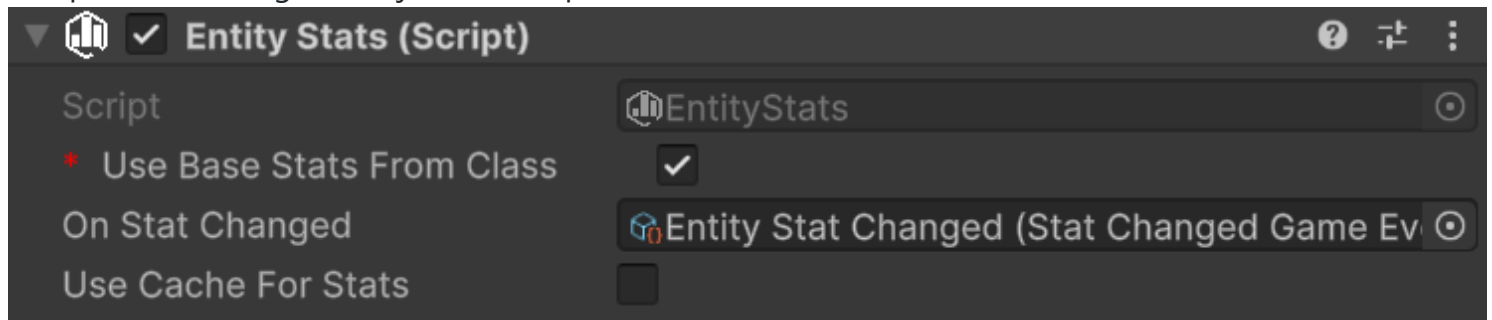
The stat set should look like:



If you want to remove a stat from the set, you can click on the small - button on the right of the stat you want to remove.

Add EntityStats to an Entity

The next step is to assign the stat set we created to an entity. To do this, let's add the **EntityStats** component to our game object. The inspector will look like this:

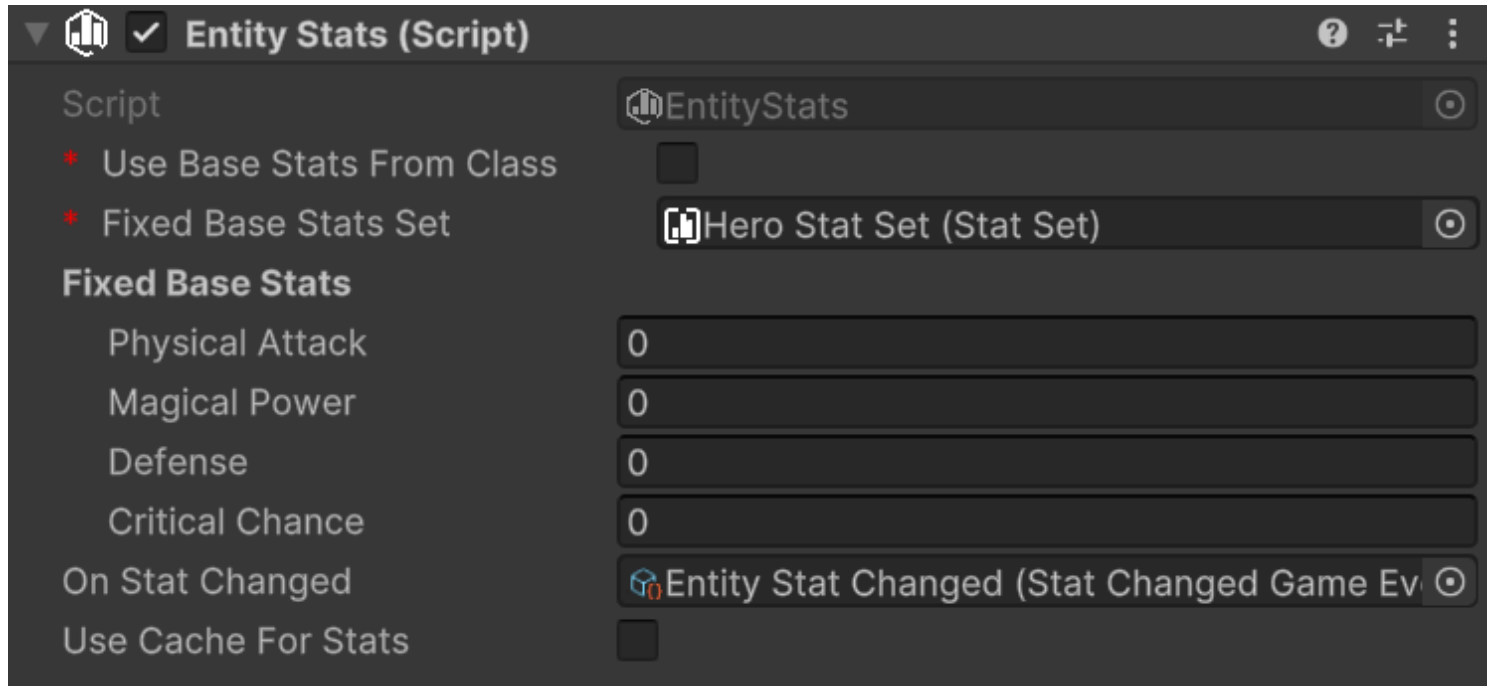


An entity has base stats that can be either fixed or derived from a class. Additionally, stats can be modified through flat modifiers, stat-to-stat modifiers, and percentage modifiers.

Use Class Base Stats checkbox determines whether the base stats should come from the entity's class (if one is available) or from fixed values defined in the inspector. For now, let's leave it unchecked since we haven't added a class yet.

With **Use Class Base Stats** unchecked, we need to manually assign a stat set. Set the **Stat Set** field under **Fixed Base Stats** with our **Hero Stat Set**. This will reveal additional fields in the inspector where

we can set the base values for each stat:



On Stat Changed event gets raised whenever any stat value changes due to modifiers. You can use this to update UI elements or trigger other game logic.

Use Cache enables caching of final stat values. This is useful for performance when you have many entities or complex stat calculations.

Adding Modifiers

While base stats are set in the inspector, modifiers can be added through code using these methods:

```
// Add flat bonus
entityStats.AddFlatModifier(stat, value); // Adds fixed amount

// Add stat-to-stat scaling
entityStats.AddStatToStatModifer(targetStat, sourceStat, percentage);

// Add percentage bonus
entityStats.AddPercentageModifier(stat, percentage);
```

The modifiers are applied in this order:

1. Base value
2. Flat modifiers
3. Stat-to-stat modifiers
4. Percentage modifiers

For example, with:

- Base Physical Attack: 100
- Flat modifier: +20
- 50% of Strength (value 40) as Physical Attack
- 25% Physical Attack increase

The final calculation would be:

1. Base (100) + Flat (+20) = 120
2. $120 + (40 * 0.5) = 140$
3. $140 + (140 * 0.25) = 175$

When adding modifiers through code, the `OnStatChanged` event will automatically be raised if the final value changes.