



# About

absent-variablebanks is a completely free and open-source extension for my package [absent-variables](#) designed specifically for [Unity](#). If you want to contribute, please go to [Contributing](#) section. I'd be really happy to see you helping me with this!

In this documentation, you will learn how to use, modify and extend the system.

You can start learning by reading [Installing](#).



# About

absent-variablebanks is a completely free and open-source extension for my package [absent-variables](#) designed specifically for [Unity](#). If you want to contribute, please go to [Contributing](#) section. I'd be really happy to see you helping me with this!

In this documentation, you will learn how to use, modify and extend the system.

You can start learning by reading [Installing](#).



# Installing

In this section of the documentation, you will learn how to install this tool and get it ready to use.

If you've already installed the package, you can skip this section and go to [Mechanism](#).

## Prerequisites

First of all, this tool depends on some of my other tools. No, no, no, no! Don't run away! They are so lightweight and easy to setup.

<https://github.com/b1lodHand/absent-variables.git>

<https://github.com/b1lodHand/absent-attributes.git>

<https://github.com/b1lodHand/absent-utilities.git>

All you need to do is copying the URLs above and pasting them into you package manager.



Image

You can open the Package Manager via the: **'Window/Package Manager'** menu on the toolbar of Unity itself. Also, you can find the documentation of these tools in the same links above.

## Installing the Package

So, you've installed all the things necessary to run this tool. Now, it is the time of installing the tool itself.

This step is pretty much straightforward:

1. Go to: <https://github.com/b1lodHand/absent-variablebanks/releases>
2. Find the the best release for you (I'd recommend the latest).

3. Download the ".unitypackage" in that release by clicking and open that file with Unity.

4. **You have it!**

## What's Next?

This section is ended. Go to [Mechanism](#) to continue.



# Contributing

You can contribute via forking or branching the [repository](#) of this package. Please be clear with your commit descriptions.

Or you can just point out the issues of the package via the [issues](#) tab of github instead of contributing it directly.

I appreciate any contribution from small to big, I thank you for all you've done!

Have a nice day!



# Mechanism

In this section of the documentation, you will learn how this tool works in general. Let's give it a go!

## What are VariableBanks?

**VariableBanks** are simple [scriptable objects](#). There are only two properties important. '**Guid**', and '**ForExternalUse**' properties. I will be covering them in detail in a second.

## How to use VariableBanks?

### Cloning

---

So, because of the intended way of using scriptable objects, the system itself clones some of the banks created (we will be covering which ones are included in that *some*) when the **splash screen** appears. This cloning process is handled by **VariableBanksCloningHandler** class.

This cloning process is **async** when using **Addressables**.

*[!INFORMATION] VariableBanks with **ForExternalUse** property set to true **won't get cloned and won't get shown on the list:** '**VariableBankDatabase.GetBankNameList()**'.*

You can use '**VariableBanksCloningHandler.AddOnCloningCompleteCallbackOrInvoke(...)**' to get notified when the cloning process is ended, or just get notified instantly if the banks are already cloned.

### Referencing (Runtime)

---

Here, I must get into Unity's asset management procedure for clarity. When you build a game, these happen:

1. All of the files that has a **direct reference** on a game object contained in a scene gets packed with the game itself.
2. All of the files inside the '**Resources**' folder gets packed in a single, big bundle.
3. If you're using **Addressables**, any of the assets referenced in the addressables window gets packed with its asset group (you can see asset group in Addressables window).

So, to avoid **asset duplication** you must use only of these way of referencing banks for each bank. You can use Resources some of the banks, while using Addressables for some other and etc. But you should be aware of **asset duplication** as I said. You should write your own logic to getthat working, obviously.

## VariableBankReference

For this, there is a class named: '**VariableBankReference**'. It looks like a direct reference when serialized in the editor but the only thing it holds is the **Guid** of the selected variable bank. Because of this, **it does not cause duplication on build**.

You can use one of these classes to select a bank, and then in runtime, you can get its '**Bank**' property to find the cloned bank **runtime** with that guid (after ensuring the cloning process is completed successfully).

When a bank gets cloned, the **Guid remains the same on the cloned one**. That's way the system above works.

Before trying to get the '**Bank**' property of a reference, ensure that the cloning process has ended successfully. If you're in editor, use '**reference.TargetGuid**' to get the original bank's guid.

If you're using direct references and handle the cloning process independently, you can **avoid using VariableBankReference class**. Ensuring there are no asset duplications.

## VariableBank.GetInstance(...);

Or as an alternative way, you can use '**VariableBank.GetInstance(...)**' method to get a cloned bank with a specific Guid **runtime**.

You cannot use this method in editor. It only works runtime. Also make sure that the cloning process has ended successfully before calling it.

## Referencing (In Editor)

---

Everything is much simpler when you're in editor. You can just call:

```
VariableBankDatabase.GetBankIfExists(string targetGuid);
```

and you have the bank.

Remember, the system uses Guids instead of direct references to avoid any **asset duplications**. So be aware of this while writing editor code.

## How To use Manipulators (Comparers & Setters)

**Comparers** and **Setters** are your best friend when you want to manipulate any variable stored in a VariableBank **runtime**.

They are simply classes which contain a string for the **Guid** of a bank, a string for the target variable name, an enum for processing type and a value which represents the new value.

## Fixed Manipulators

**Fixed Comparers** and the **Fixed Setters** are a little different than normal ones. They don't have a bank selector in editor, instead you give them the target bank's Guid manually with the '**SetFixedBank(...)**' function. This is an example of it:

```
private void OnValidate()
{
    if (Application.isPlaying) return; // Not needed. Just for extra error handling.

    m_comparers.ForEach(comparer => comparer.SetFixedBank(m_reference.TargetGuid));
    m_setters.ForEach(setter => setter.SetFixedBank(m_reference.TargetGuid));
}
```

The code above sets the bank Guids of manipulators of a component every time a change in the inspector occurs. This is the most practical way of using fixed manipulators with components.

If you don't work with lists, you can also use '**OnEnable()**' instead of using '**OnValidate()**'

You can also create your own manipulators via deriving from the **BaseVariableComparer** or **BaseVariableSetter** classes. You can find an example [here](#)..

In this example, I used **direct references**. But I mark all **BlackboardBanks** as **ForExternalUse** banks automatically on creation, so this does not cause any duplications.

## What's Next?

This section is ended. Go to [Mechanism](#) to continue.







# Mechanism

In this section of the documentation, you will learn how this tool works in general. Let's give it a go!

## What are VariableBanks?

**VariableBanks** are simple [scriptable objects](#). There are only two properties important. '**Guid**', and '**ForExternalUse**' properties. I will be covering them in detail in a second.

## How to use VariableBanks?

### Cloning

---

So, because of the intended way of using scriptable objects, the system itself clones some of the banks created (we will be covering which ones are included in that *some*) when the **splash screen** appears. This cloning process is handled by **VariableBanksCloningHandler** class.

This cloning process is **async** when using **Addressables**.

*[!INFORMATION] VariableBanks with **ForExternalUse** property set to true **won't get cloned and won't get shown on the list:** '**VariableBankDatabase.GetBankNameList()**'.*

You can use '**VariableBanksCloningHandler.AddOnCloningCompleteCallbackOrInvoke(...)**' to get notified when the cloning process is ended, or just get notified instantly if the banks are already cloned.

### Referencing (Runtime)

---

Here, I must get into Unity's asset management procedure for clarity. When you build a game, these happen:

1. All of the files that has a **direct reference** on a game object contained in a scene gets packed with the game itself.
2. All of the files inside the '**Resources**' folder gets packed in a single, big bundle.
3. If you're using **Addressables**, any of the assets referenced in the addressables window gets packed with its asset group (you can see asset group in Addressables window).

So, to avoid **asset duplication** you must use only of these way of referencing banks for each bank. You can use Resources some of the banks, while using Addressables for some other and etc. But you should be aware of **asset duplication** as I said. You should write your own logic to getthat working, obviously.

## VariableBankReference

For this, there is a class named: '**VariableBankReference**'. It looks like a direct reference when serialized in the editor but the only thing it holds is the **Guid** of the selected variable bank. Because of this, **it does not cause duplication on build**.

You can use one of these classes to select a bank, and then in runtime, you can get its '**Bank**' property to find the cloned bank **runtime** with that guid (after ensuring the cloning process is completed successfully).

When a bank gets cloned, the **Guid remains the same on the cloned one**. That's way the system above works.

Before trying to get the '**Bank**' property of a reference, ensure that the cloning process has ended successfully. If you're in editor, use '**reference.TargetGuid**' to get the original bank's guid.

If you're using direct references and handle the cloning process independently, you can **avoid using VariableBankReference class**. Ensuring there are no asset duplications.

## VariableBank.GetInstance(...);

Or as an alternative way, you can use '**VariableBank.GetInstance(...)**' method to get a cloned bank with a specific Guid **runtime**.

You cannot use this method in editor. It only works runtime. Also make sure that the cloning process has ended successfully before calling it.

## Referencing (In Editor)

---

Everything is much simpler when you're in editor. You can just call:

```
VariableBankDatabase.GetBankIfExists(string targetGuid);
```

and you have the bank.

Remember, the system uses Guids instead of direct references to avoid any **asset duplications**. So be aware of this while writing editor code.

## How To use Manipulators (Comparers & Setters)

**Comparers** and **Setters** are your best friend when you want to manipulate any variable stored in a VariableBank **runtime**.

They are simply classes which contain a string for the **Guid** of a bank, a string for the target variable name, an enum for processing type and a value which represents the new value.

## Fixed Manipulators

**Fixed Comparers** and the **Fixed Setters** are a little different than normal ones. They don't have a bank selector in editor, instead you give them the target bank's Guid manually with the '**SetFixedBank(...)**' function. This is an example of it:

```
private void OnValidate()
{
    if (Application.isPlaying) return; // Not needed. Just for extra error handling.

    m_comparers.ForEach(comparer => comparer.SetFixedBank(m_reference.TargetGuid));
    m_setters.ForEach(setter => setter.SetFixedBank(m_reference.TargetGuid));
}
```

The code above sets the bank Guids of manipulators of a component every time a change in the inspector occurs. This is the most practical way of using fixed manipulators with components.

If you don't work with lists, you can also use '**OnEnable()**' instead of using '**OnValidate()**'

You can also create your own manipulators via deriving from the **BaseVariableComparer** or **BaseVariableSetter** classes. You can find an example [here](#)..

In this example, I used **direct references**. But I mark all **BlackboardBanks** as **ForExternalUse** banks automatically on creation, so this does not cause any duplications.

## What's Next?

This section is ended. Go to [Mechanism](#) to continue.





# Components

## Variable Bank Acquirer

**Variable Bank Acquirer** is a deprecated component in real. Using **VariableBankReference** class nested in another component instead of using this component directly is much easier to manage.

It does nearly the same thing with the **VariableBankReference**.

## What's Next?

**You've done it!** You've read all the way to the end, buddy. I really appreciate you. Know you know how this tool works in a detailed way.

There is nothing else you need to read. But if you want to know how you can publish any modifications you've made on the tool to the open-source community, I'd suggest you reading [Contributing](#) section.

And again, thank you for reading this all the way!

**Have a nice day!**



Roadmap

[Edit this page](#)



# Namespace `com.absence.savesystem.variablebanks`

## Classes

[ExtensionMethods](#)

[TempVariableBankData](#)

[VariableBankSerializationSurrogate](#)