Peppermint Data Binding

Peppermint Data Binding is a lightweight data binding framework for Unity. It provides a simple and easy way for Unity games to utilize data binding.

Features

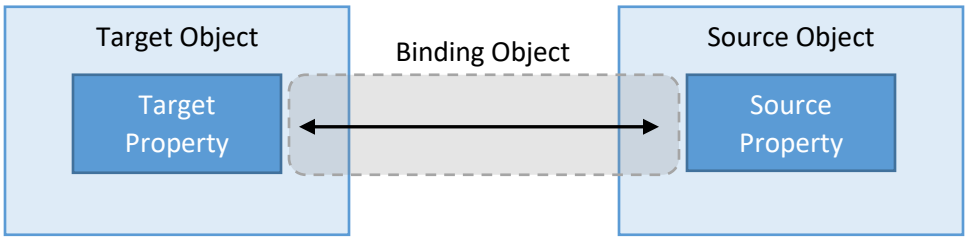
- Support OneWay, TwoWay and OneWayToSource binding modes.
- Support data conversion.
- Support binding to nested properties.
- Support binding to collections.
- Support dynamic binding.
- Support collection view.
- Support command.
- Support uGUI and NGUI.
- Optimized for performance.
- Support JIT/AOT compilation (iOS, Android, webGL).
- Editor tools to make data binding development easier.
- Model-View-ViewModel ready.
- Full source code included.

Data Binding Overview

Data binding is the process that establishes a connection between the application UI and business logic. If the binding has the correct settings and the data provides the proper notifications, then, when the data changes its value, the elements that are bound to the data reflect changes automatically.

Data Binding Concepts

Each binding always follows the model illustrated by the following figure.



As illustrated by the above figure, data binding is essentially the bridge between your binding target and your binding source. The figure demonstrates the following concepts:

- Typically, each binding has these four components: a binding target object, a target property, a binding source and source property. For example, if you want to bind the content of a UI.Text to the Name property of a Player object, your target object is the UI.Text, the target property is the text property, the source property is Name, and the source object is the Player object.
- The binding source object and binding target object can be any CLR object.
- Peppermint data binding only support properties. Most Unity components can be manipulated by their properties, which make them available for binding targets. If target object does not have proper property you need to create an adaptor to wrap the operation into property.

To enable your binding target properties to automatically reflect the dynamic changes of the binding source, your class needs to implement INotifyPropertyChanged interface.

Direction of the Data Flow

The data flow of a binding can go from the binding target to the binding source (for example, the source value changes when a user edits the value of a InputField) and/or from the binding source to the binding target (for example, your Text content gets updated with changes in the binding source) if the binding source provides the proper notifications. The framework currently supports the following different types of data flow:

OneWay

OneWay binding causes changes to the source property to automatically update the target property, but changes to the target property are not propagated back to the source property.

TwoWay

TwoWay binding causes changes to either the source property or the target property to automatically update the other.

OneWayToSource

OneWayToSource is the reverse of OneWay binding; it updates the source property when the target property changes.

Note that although the underlying Binding object supports two-way binding mode, the built-in two-way binders use a simplified approach to implement two-way data flow.

Data Conversion

If the source and target property are not the same type, the data binding will use the converter to do the type conversion. When you create the binding object, you can specify how to handle the type conversion. The framework support 3 different conversion modes:

- None
  Set to none if you want to disable the type conversion.
- Parameter

Use the specified data converter.

- Automatic
  Let the binding object automatically choose the best convertor when binding.

Usually, the binding object will use automatic conversion mode. If the source type can be converted to target type, the binding object will set the value to target directly. Otherwise it will find a converter in ValueConverterProvider by the source type and the target type.

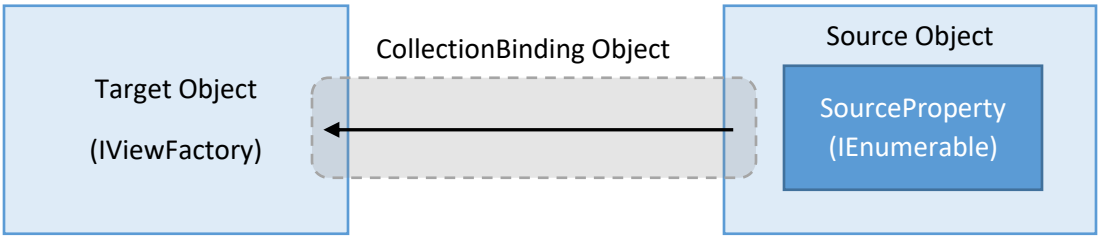
There are two types of converters: general converters and named converters. General converter is associated with unique type pair, which includes the source type and target type. The named converter is associated with a unique name.

Note that only one general converter can be associated with a type pair. If you need multiple converters which handle the same source type and target type, you can add them as named converters. E.g. the SpriteNameConverter is added as a named converter.

The framework has some built-in converters, which can handle most of the type conversion for you.

Collection Binding

In Peppermint data binding, binding to a collection object is also supported. For example, you can use scroll view to display a data collection. To establish a collection binding, you use the CollectionBinding object. The CollectionBinding has three components: a binding source, a source property, and the target object.

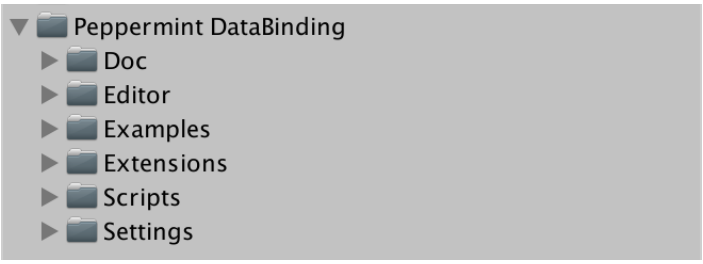


The above figure illustrated the model in collection binding. The CollectionBinding monitors the changes of source collection and manage the order of view objects. When an item is added, it creates a new view object from the target IViewFactory and binds the view to the item. When an item is removed, it releases its view object and unbinds it.

The source property can be any collection that implements the IEnumerable interface. However, to set up dynamic bindings so that insertions or deletions in the collection update the UI automatically, the collection must implement the INotifyCollectionChanged interface. This interface exposes an event that should be raised whenever the underlying collection changes.

Peppermint data binding provides the ObservableList<T>, which is a built-in implementation of INotifyCollectionChanged interface.

Folder Structure



- Doc
  Documentation for peppermint data binding.
- Editor
  Editor assets include icons, templates.
- Examples
  Include all tutorials and examples.
- Extensions
  Third-party add-on extensions.
- Scripts
  Data binding source code and editor source code.
- Settings
  Settings for edtior tools. It's created when you first launch the editor tool.

The examples folder contains assets and script files, these script files will be compiled into your game. To reduce the build size, you can remove this folder after completing the tutorials.

Class Overview

Binder

Binder is the MonoBehaviour where you setup the binding parameters in Unity. It creates the underlying binding objects internally. The framework contains some commonly used binders. Depending on their usage, there are categorized in

- Selector
  Selector is used to activate targets based on the source value.
- Setter
  Setter set the specified value to target property based on the source value.
- Getter
  Getter set the specified value to source property.
- Binder

General binder (one-way), bind target property to a binding source with specified source path.

- Two-way Binder
  Two-way binder, bind target property to binding source and listen for UI changes, if the UI changes, new value is set to source property.

You can easily create your own binder. Binder is just a MonoBehaviour, it creates the binding object with specified parameters and handles the adding and removing of created binding object with associated DataContext. See “JoystickBinderExample” for more information.

BindingManager

The core of data binding. It manages all source objects and registered DataContext objects.

DataContext

DataContext component is the place where the binding source object being specified.

DataContextRegister

Register data context to BindingManager with required source name.

CollectionView

CollectionView allows display the collection based on sort and filter, all without having to manipulate the underlying source collection itself.

Commands

Actions or operations that the user can perform through the UI are typically defined as commands. Commands is used in ButtonBinder.

Converter

The framework has 3 built-in value converters:

- DefaultConverter handles primitive type conversion.
- ImplicitConverter do the type conversion by calling the implicit operator. It supports all built-in implicit operators, e.g. Vector2/Vector3, Color/Color32, etc.
- SpriteNameConverter converts string to sprite.

You can create your own value converter, just implement the IValueConverter interface and add it to ValueConverterProvider. See “HtmlColorConverterExample” for more information.

Tutorial

Hello Data Binding

Here is a simple example to demonstrate how to use this framework. You can follow the instructions, and make yourself familiar with the data binding workflow. In this example, we want the UI display a simple message string.

Step A

First, we design the Model and determine which fields it should have. We create a new class which derived from BindableMonoBehaviour, and add a single string field “message”.

Then, we create a property “Message” to encapsulate this field. We must use this property to set its value.

```
public class HelloDataBinding : BindableMonoBehaviour
{
    private string message;

    public string Message
    {
        get { return message; }
        set { SetProperty(ref message, value, "Message"); }
    }
}
```

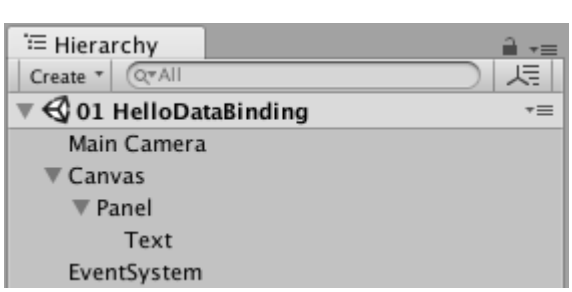
Next, we need to add this object to BindingManger, so the DataContext can find this source by its name. After we have done with this source, we need to remove it from BindingManager.

```
void Start()
{
    Message = "Hello, DataBinding!";

    // add current object as binding source
    BindingManager.Instance.AddSource(this, typeof(HelloDataBinding).Name);
}

void OnDestroy()
{
    // remove source
    BindingManager.Instance.RemoveSource(this);
}
```

We give the Message a literal string in Start method. That’s all the code we need to write.

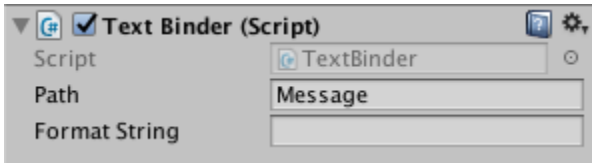
Step B

Next we need to create a UI for this model. You can design your UI in your own way, here we create the simplest one. We create a Panel and add a Text under the panel. Tweak the size and color, and now the UI is ready for data binding.

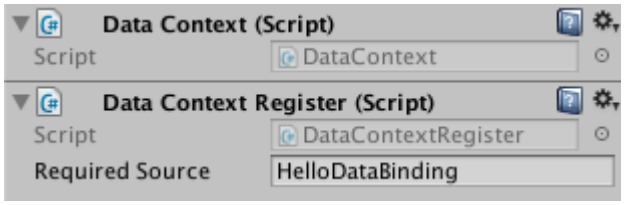


Step C

We need to add data binding components to make the UI support data binding. First, add a TextBinder to “Text” GameObject. The TextBinder will create a binding object, the target object is the TextBinder itself, the source path is “Message” and the source object will be the “HelloDataBinding”.

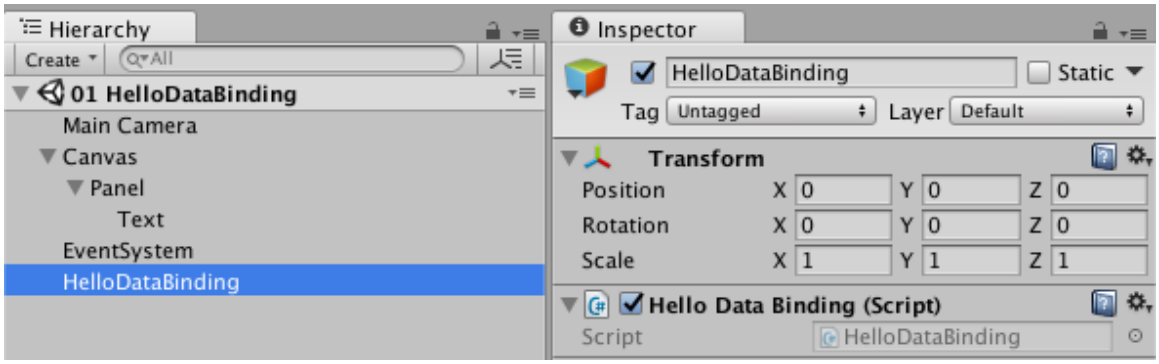


Then add DataContext and DataContextRegister to Panel node, and set required source to “HelloDataBinding”. The required source must match the name we specified in AddSource method, otherwise the DataContext could not be bound with the source.

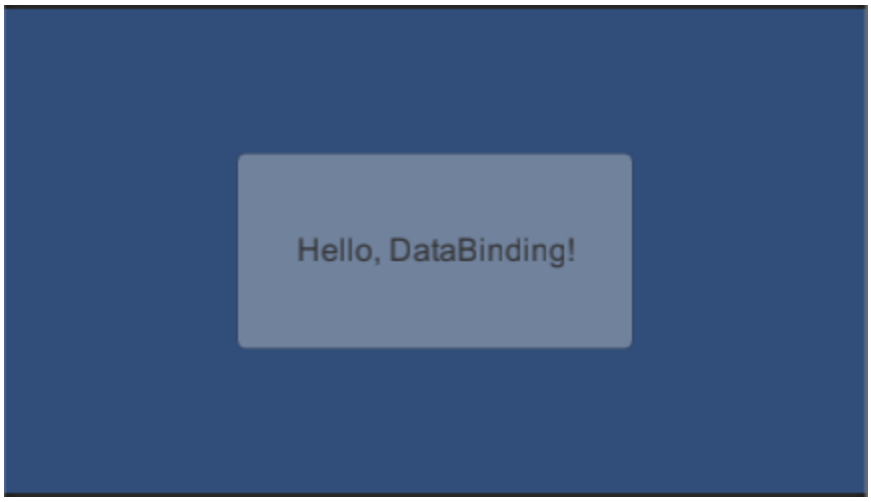


Step D

The last step, we need to add a HelloDataBinding object to the scene. Create a new GameObject, rename it to HelloDataBinding or any name you like, and add a HelloDataBinding component to it.



OK, let’s hit the play button. If everything is setup correctly, you will see the text changed to what we set in the code. Congratulations, you have completed the first tutorial.



The first tutorial is quite simple, it demonstrates the basic workflow of data binding.

1. In step A, you write the code to define the class, which is used as the binding source.
2. In step B, you create the UI, which is the view of your model. No data binding is involved in this step.
3. In step C, you add the data binding components to the UI.
4. In step D, you add the binding source object.

How it works

When you hit the play button, the following things will happen in their Start method.

- The HelloDataBinding added itself to the BindingManager.
- The TextBinder find a DataContext on its ancestor, create a binding object and add it to the DataContext.
- The DataContextRegister add the DataContext object to BindingManager.

Next the BindingManager checks if the required source has been added. If so, the BindingManager binds this source to the target DataContext. Then the DataContext binds the source to all binding objects. Finally, the binding object registers the event in the source, and update the target property with the source value.

When you stop the game, the following things will happen in their OnDestroy method.

- The HelloDataBinding removes itself from the BindingManager.
- The TextBinder removes the binding object from its DataContext.
- The DataContextRegister removes the DataContext from BindingManager.

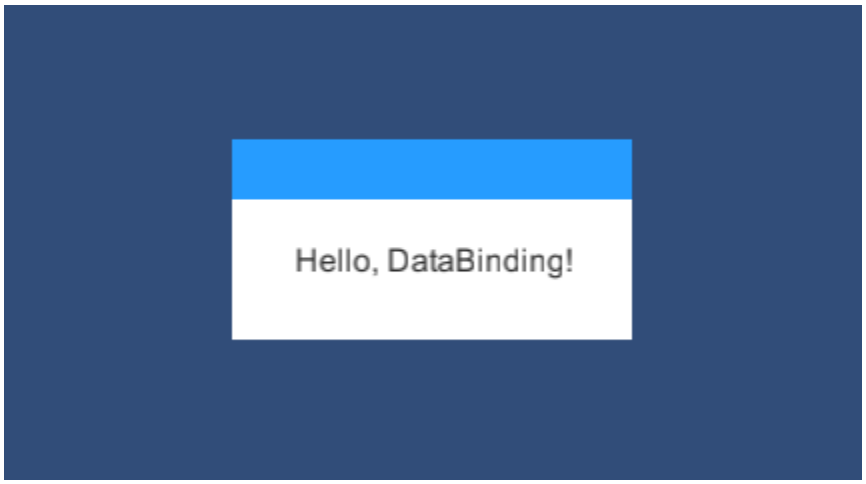
When the source/DataContext is removed from BindingManager, it will unbind the source from its associated DataContext. Then the DataContext itself will unbind the source for all binding objects. When the binding object is removed from DataContext, DataContext will check if the binding object is bound, if so it will unbind the binding object.

This is a brief introduction to the internal work of data binding, for more information you can check the source code.

Data binding vs no data binding

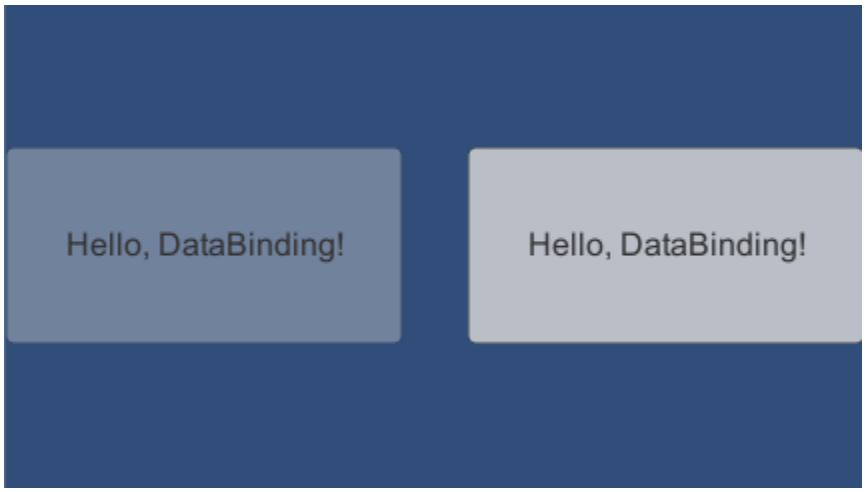
Because the view does not dependent on any Model/ViewModel, the UI can run without the HelloDataBinding source. We can verify it by disabling the HelloDataBinding GameObject. Hit the play button, the game will run without any error. The Model/ViewModel is unaware of the view, so the Model/ViewModel can run without any View. We can disable the Panel object, hit the play button again, and there is no error. That's the key feature of MVVM design pattern, a clean separation between application logic and the UI will make an application easier to test, maintain, and evolve.

For example, we want to make a better UI for this model. Once the UI is complete, all we need to do is step C, add data binding components. After that, your new UI is ready to replace the old view.



Without data binding, we need add a UI.Text public field in code, and set the reference in Unity (Or some indirect way to find the UI.Text). We use this field to update the text property of UI.Text. If we create a new UI, we must replace the reference with new one. It is okay for a simple UI, but it is time-consuming process for a complex UI.

If we want to extend the model to support two UI, both need to display the same Message. Without data binding, we need to modify the code to change the UI.Text field into two UI.Text fields or an array, then change the code where update the text and setup UI.Text reference in Unity. With data binding, we do not need to change any code. You can try it yourself, just duplicate the Panel, and adjust the position of the new Panel. Hit play button, it works with no problem.



The data binding only concerned about the source path, if the path name matches, any object can be a binding source. A view can be bind to a different type, the type does not need to inherit from the specified class or interface, just need compatible properties.

Hello Collection Binding

The first tutorial introduces the basic of peppermint data binding. In the second tutorial, we will demonstrate how to setup the collection binding. It will also shows how to handle button clicks.

In this tutorial, we need a scroll view, it will display a list of items. The UI of the item contains a name text and a button, if you click the button, it will remove this item from list. We also need two buttons, one button to remove all items in the list, and the other to add a new item to the list.

Step A

In this tutorial, we need create two classes. A simple Item class that represents the item in the list. The HelloCollectionBinding class is the view model class which contains an item list and methods for manipulating the list. Here we define the Item class as a nested class within the HelloCollectionBinding class.

```
public class Item : BindableObject
{
    private string name;
    private ICommand removeCommand;

    public string Name
    {
        get { return name; }
        set { SetProperty(ref name, value, "Name"); }
    }

    public ICommand RemoveCommand
    {
        get { return removeCommand; }
```

```
        set { SetProperty(ref removeCommand, value, "RemoveCommand"); }
    }
}
```

The Item class only has two properties, a Name property and a RemoveCommand property. The type of Name property is string, which is the name of the item. The type of RemoveCommand property is ICommand, we'll explain the ICommand later.

Next we need create a collection of items, as mentioned before the collection type must implement the INotifyCollectionChanged interface, so we use the built-in ObservableList to define the item list. The following code shows the list field and its property.

```
private ObservableList<Item> itemList;

public ObservableList<Item> ItemList
{
    get { return itemList; }
    set { SetProperty(ref itemList, value, "ItemList"); }
}
```

HelloCollectionBinding class need provide methods to manipulate the list: RemoveItem method removes the specified item from the list, AddNewItem method creates a new item object and add it to the list.

```
public void RemoveItem(Item item)
{
    itemList.Remove(item);
}

private void AddNewItem()
{
    // create new item
    var newItem = new Item();
    newItem.Name = string.Format("Item {0:D02}", nextIndex++);

    // set command
    newItem.RemoveCommand = new DelegateCommand(() => RemoveItem(newItem));

    // add to list
    itemList.Add(newItem);
}
```

In the Start method, we create the item list and add some items to the list. At the end of the Start method, we add current class instance as a binding source. In the OnDestroy method, we remove current instance from BindingManager.

```
void Start()
{
    itemList = new ObservableList<Item>();

    // create items
    for (int i = 0; i < initCount; i++)
    {
        AddNewItem();
    }
```

```
    // add current object as binding source
    BindingManager.Instance.AddSource(this, typeof(HelloCollectionBinding).Name);
}

void OnDestroy()
{
    // remove source
    BindingManager.Instance.RemoveSource(this);
}
```

To handle the button click, we add two commands, one for clearing the item list and the other for adding a new item to the list. The definition of ICommand property is the same as normal bindable property, the only difference is the field type.

```
private ICommand clearCommand;
private ICommand createCommand;

public ICommand ClearCommand
{
    get { return clearCommand; }
    set { SetProperty(ref clearCommand, value, "ClearCommand"); }
}

public ICommand CreateCommand
{
    get { return createCommand; }
    set { SetProperty(ref createCommand, value, "CreateCommand"); }
}
```

In data binding, actions or operations that the user can perform through the UI are typically defined as commands. Usually, command will be invoked as a result of button click. The ICommand interface defines an Execute method, which encapsulates the operation itself, and a CanExecute method, which indicates whether the command can be invoked at a particular time.

```
public interface ICommand
{
    // Return true if command can be executed, otherwise return false
    bool CanExecute();

    // Execute command
    void Execute();
}
```

You do not need to implement the ICommand interface, the peppermint data binding has a built-in implementation called DelegateCommand. The DelegateCommand class encapsulates two delegates that each reference a method implemented within you class.

In the Start method, we create two DelegateCommand objects, and specify the delegate for Execute method. The canExecute delegate is optional, if not specified the command is always executable.

```
// create commands
clearCommand = new DelegateCommand(() => itemList.Clear());
createCommand = new DelegateCommand(AddNewItem);
```

You can use lambda expression to specify execute and canExecute delegate. Use lambda expression is recommended, because both execute and canExecute delegate has no parameters. If the method you specified need parameters, you can use closure to capture extra parameters when you create the DelegateCommand.

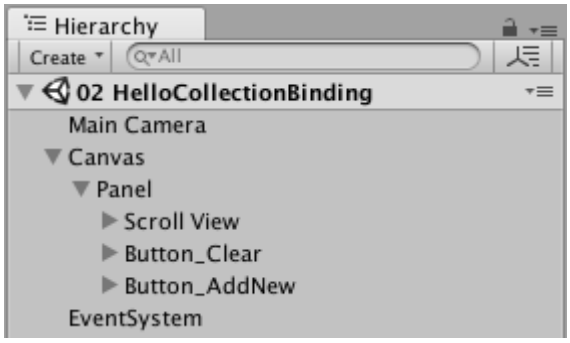
```
// create new item
var newItem = new Item();
...
// set command
newItem.RemoveCommand = new DelegateCommand(() => RemoveItem(newItem));
```

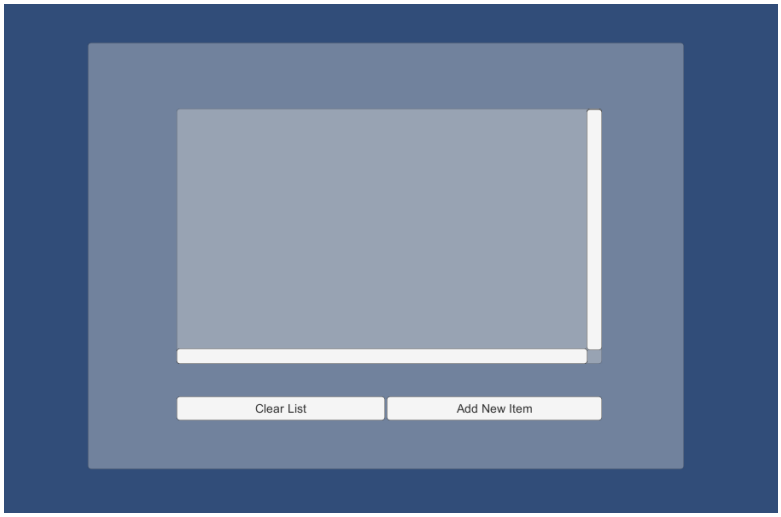
The above code from AddNewItem method shows how to use lambda expression to specify delegate with parameters. The RemoveCommand of item is relayed to method RemoveItem, notice that the RemoveItem method need a parameter to specify the target item to be removed. The lambda expression captures the newItem variable and when the execute delegate is invoked the captured variable newItem is passed to RemoveItem method.

That's how we define the HelloCollectionBinding class, you can find the full source code in example folder. To use collection binding, you need define the collection with ObservableList and make the item class bindable. You can create a new item object, remove it from the list and update the item property as usual, no special code is needed. The collection binding will handle the view updates for you. The ICommand property is used by ButtonBinder which relay a button click to the execute delegate.

Step B

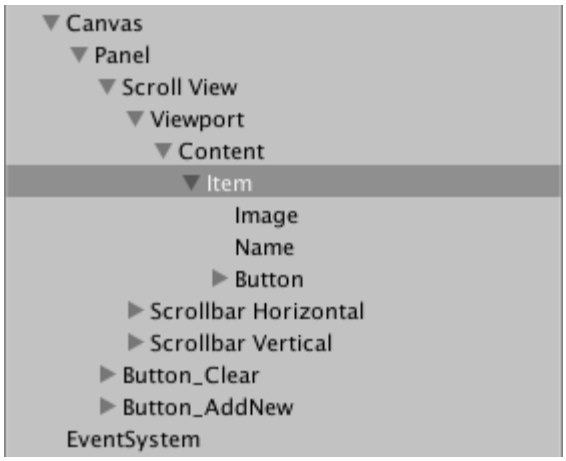
Let’s make the UI for this tutorial. Create a new scene, like the first tutorial, we create a new panel. Inside the panel, add a scroll view and two buttons. Adjust the position and the size, rename the left button to “Button_Clear” and set its text to “Clear List”, and rename the right button to “Button_AddNew” and set its text to “Add New Item”.



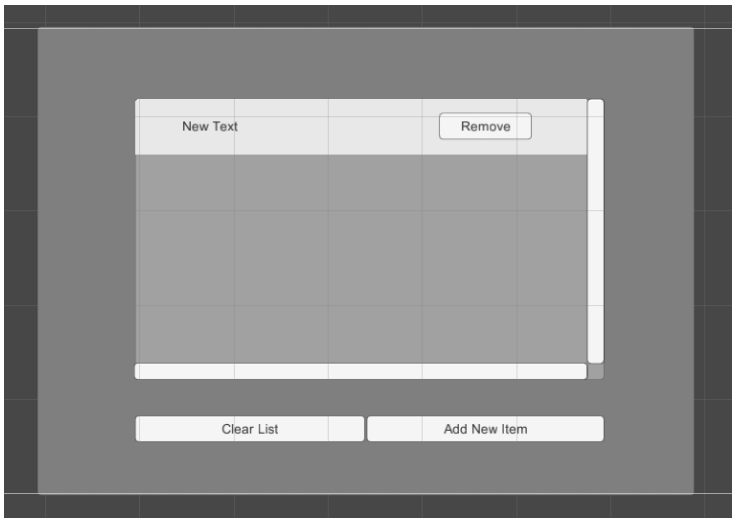


Next, we need create a view template for the item. View template is a Unity GameObject which define the looks of the item. In this tutorial, we create a GameObject under Content node of Scroll View. The following is the detail instructions:

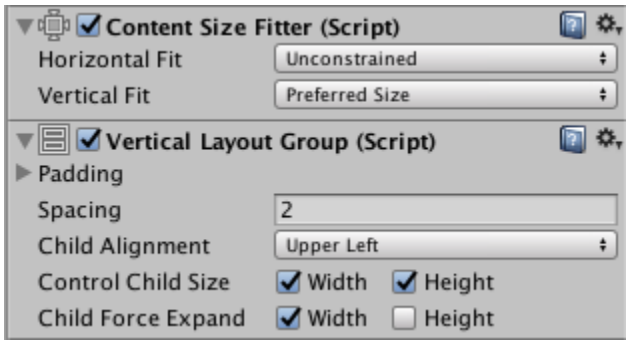
1. Create an empty new GO under Content transform of scroll view, rename it to “Item”.
2. Add a layout element component to the Item, set preferred height to 60.
3. Under Item node, add a child image, stretch the image to full size. This is the background of Item view.
4. Add a Text, renamed it to “Name” and adjust the position.
5. Add a button, change text to “Remove”.



After creating the view template, your UI will look like this

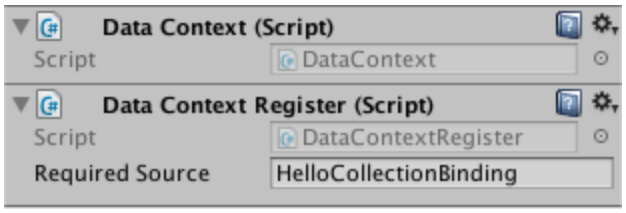


Next, we setup the content node, add a content size filter to Content node, set vertical fit to preferred size. Add a vertical layout group to Content node, set spacing to 2, and set child force expand to width. Now the scroll view is a vertical scroll view whose vertical size is controlled by its child nodes.

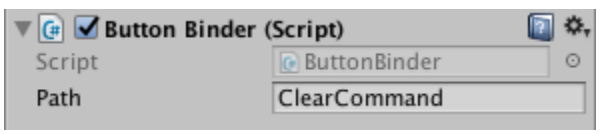


Step C

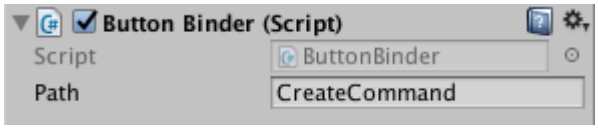
Like the first tutorial, we need to add data binding components to the UI. Add DataContext and DataContextRegister to Panel node, set required source to “HelloCollectionBinding”.



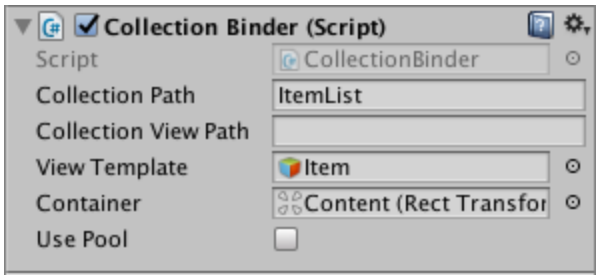
Add a ButtonBinder to Button_Clear, set path to “ClearCommand”.



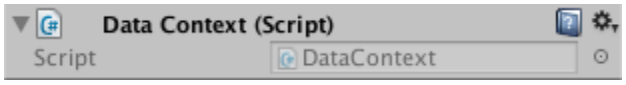
Add a ButtonBinder to Button_AddNew, set path to “CreateCommand”.



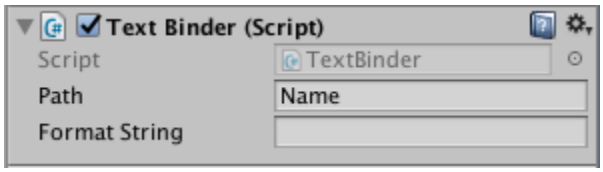
In Scroll View node, add a CollectionBinder component. Set collection path to “ItemList”, which is the source path of CollectionBinding object. Drag Item to view template, which is the UI template for created Item object. Drag Content node to Container, which is the parent node of all created item views.



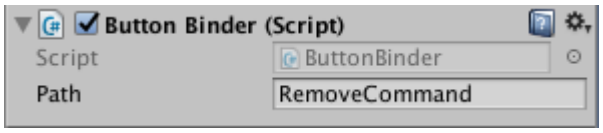
Next, we set up data binding components for Item. Add a DataContext to Item node, which must be on the Item node. The view template need a DataContext component to support data binding.



The Item class contains two properties, the Name property and RemoveCommand property. Find the Name node under Item node, add a TextBinder component, and set path to “Name”.



And add a ButtonBinder to Button node, set path to “RemoveCommand”.



Be careful about the path string, there are case sensitive.

Step D

We create a new GameObject, renamed it to HelloCollectionBinding, and add HelloCollectionBinding component to it. We also need deactivate the Item node, because it is used as in place prefab.

If everything is setup correctly, hit the play button will run the tutorial. If any error is reported by the data binding, the log info will help you to find the error. Peppermint data binding will report common errors such as invalid source path, etc.

How it works

The CollectionBinding manages the lifetime of each item view and the order of item view. If you click the Add New Item button, a new item object is added to the list. The CollectionBinding will create a new view GameObject from specified view template, and add it to the container node as last child. Remove the item object from the list will cause CollectionBinding to remove its associated view GameObject. CollectionBinding also handles the DataContext binding and unbinding of each item. In each item view, the UI.Text get its content from the Name property, and the button bind to its RemoveCommand property.

These two tutorials demonstrate the basic concepts of peppermint data binding framework. The peppermint data binding includes samples and demos, you can learn more with these examples.

MVVM Overview

When you develop you game UI with data binding, MVVM will be the nature pattern to use. Peppermint data binding framework was designed to make it easy to build game UI using the MVVM pattern. The MVVM pattern is a big topic, in this document we just briefly introduce the basic concept of this pattern. If you need more information you can search it on the internet.

The Model-View-ViewModel (MVVM) pattern helps you to cleanly separate the business and presentation logic of your application from its user interface (UI).

In the MVVM pattern, the view encapsulates the UI and any UI logic, the view model encapsulates presentation logic and state, and the model encapsulates business logic and data. The view interacts with the view model through data binding, commands, and change notification events. The view model queries, observes, and coordinates updates to the model, converting, validating, and aggregating data as necessary for display in the view.

View

The view's responsibility is to define the structure and appearance of what the user sees on the screen. Usually, this is the UI you created in Unity.

ViewModel

The view model in the MVVM pattern encapsulates the presentation logic and data for the view. The view model implements properties and commands to which the view can data bind and notifies the view of any state changes through change notification events.

The view model is a non-visual class, it typically does not directly reference the view. It implements properties and commands to which the view can data bind. It notifies the view of any state changes via change notification events via the INotifyPropertyChanged and INotifyCollectionChanged interfaces.

Model

The model in the MVVM pattern encapsulates business logic and data. They are responsible for managing the application's data and for ensuring its consistency and validity by encapsulating the required business rules and data validation logic.

The model classes do not directly reference the view or view model classes and have no dependency on how they are implemented. The model classes typically provide property change notification events through the INotifyPropertyChanged interface. This allows them to be easily data bound in the view.

If the model classes do not implement the INotifyPropertyChanged interface. In those cases, the view model may need to wrap the model objects and expose the required properties to the view. The values for these properties will be provided directly by the model objects. The view model will implement the required interfaces for the properties it exposes so that the view can easily data bind to them.

MVVM Pattern Notes

Peppermint data binding framework make MVVM pattern available for Unity. It enables a developer-designer workflow. When the UI is not tightly coupled to the code, it is easy for designers to modify the view.

MVVM is a set of guidelines, not rules. Sometimes you may prefer to manipulate view component in your view model class, because it is easier or more efficient to manipulate the view instead of using data binding. The framework provides ComponentGetter which use data binding to set the component reference to your view model, you can use it to get any component reference in the view. You should notice that this will introduces a dependency between the view model and the view.

Editor Support

Peppermint data binding includes some useful editor utilities, which can make data binding development more easily. You can find these tools in menu Tools/Data Binding.

Code Builder

The Code Builder editor window contains 3 code builders, you can click the toolbar button to select a different code builder.

Bindable Property Code Builder

To make a class ready for data binding, it must implement the INotifyPropertyChanged interface. Implementing the INotifyPropertyChanged interface on many classes can be repetitive and error-prone because of the need to specify the property name in event argument. Peppermint data binding provides 3 bindable base classes from which you can derive your classes that is ready for data binding.

A derived class can raise the property change event in the set accessor by calling the SetProperty method. The SetProperty method checks whether the backing field is different from the value being set. If it is different, the backing field is updated and the PropertyChanged event is raised.

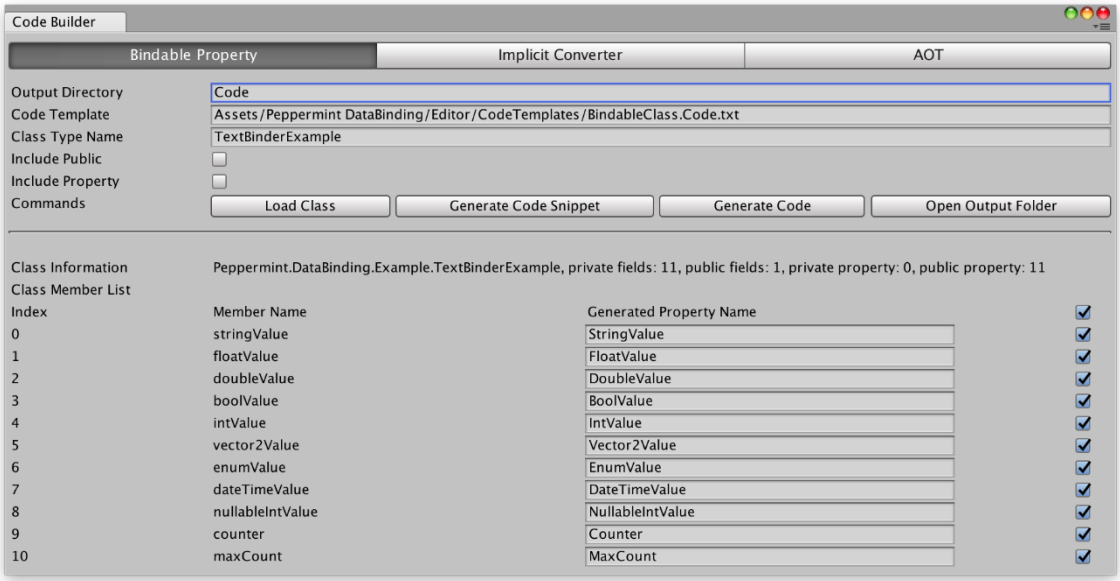
```
public string StringValue
{
    get { return stringValue; }
    set { SetProperty(ref stringValue, value, "StringValue"); }
```

```
}
```

The above code example shows a standard bindable property. If the class has many bindable properties, write these properties can be repetitive and error-prone. The Bindable Property Builder is a tool that help you from doing the tedious work. It is a code generator that generates code snippet for all bindable properties.

Quick guide

1. First you should define all fields in your bindable class.
2. Open the Code Builder Editor and click the Bindable Property tab, input the class name in Class Type Name text field, and click the Load Class button. If the type is found, it will display a list of members.
3. The default member list contains all private fields, if you want to include public fields, enable the Include Public toggle. The name of generated property can be customized, the default name just capitalize the first letter. If the member is not for binding, you can uncheck the toggle to exclude it from the list. To select or deselect all members, just click the first toggle.
4 Click the “Generate Code Snippet” button, the code builder will generate the code snippet which contains only the bindable properties. The result is copied to system clipboard/pasteboard.
5 Click the “Generate Code” button, the code builder will generate a C# source file to the specified output directory, the file is named after the Class Type Name. Click the Open Output Folder button will open the output folder in explorer/finder.



Bindable property builder parameters

- Output Directory
  The output directory for generated source code. Default is Project/Code.
- Code Template

Template file for generated C# source.

- Class Type Name
  The name of class. You can enter name or full name of your class, nested class is also supported.
- Include Public
  Indicate if public members are included in member list.
- Include Property
  Indicate if properties are included in member list.

Bindable property usually wraps a backing field, but make a bindable property that wrap another backing property is also supported. The following example defines a private property stringValue, and a bindable property StringValue.

```
private string stringValue { get; set; }

public string StringValue
{
    get { return stringValue; }
    set { SetProperty(stringValue, value, x => stringValue = x, "StringValue"); }
}
```

Starting from Version 1.2.0, Peppermint data binding added support for CallerMemberName. CallerMemberName is an attribute introduced in C# 5.0, which allows you to obtain the method or property name of the caller to the method. If the target runtime is set to .Net 4.6, you can avoid specifying the property name parameter when you call the SetProperty method. Code snippet generated by code builder will also omit the property name string. Caution, if you switch the runtime back to .Net 3.5, the code compilation will fail.

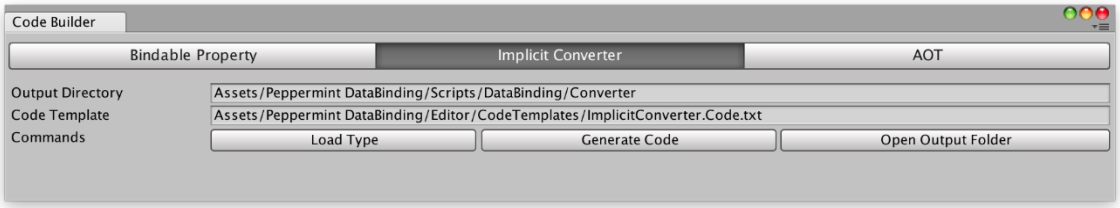
Implicit Converter Code Builder

Peppermint data binding only registers implicit converters with predefined types, so the runtime and the AOT code builder only need to handle these types. The builder will check all implicit operators in Unity.Engine assembly and your game assembly, and list all found types which declared the operators. Note that, only two-way implicit operators are included.

Quick guide

If this is the first time you open the builder, the default settings will be created. The output directory will be set to the directory of “ImplicitConverter.cs”.

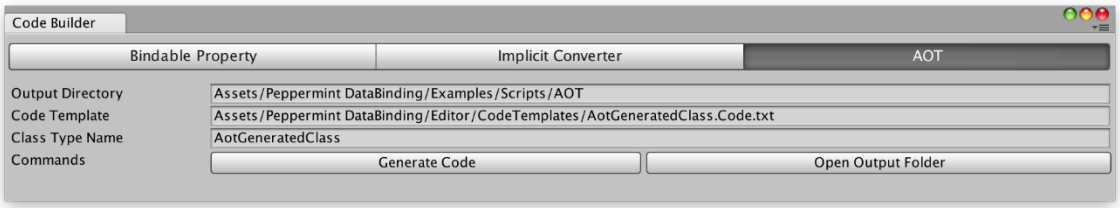
First click the “Load Type” button, the list of found types will be displayed, then click the “Generate Code” button, it will owerwrite the “ImplicitConverter.Code.cs” file. This code file contains only one method “GetTypeNames”, which returns the generated type name array.



If your game assembly defines implicit operators and you want to use them in data binding, then you should use this builder, otherwise the default "ImplicitConverter.Code.cs" file is ready for use.

AOT Code Builder

Peppermint data binding supports Mono and IL2CPP scripting backend, and supports JIT and AOT compilation. The underlying code use reflection to manipulate objects, e.g. get the property info, get/set the property value. Update object using the reflection call is slow, so the Peppermint data binding optimizes the code for property get and set by converting the reflection call to delegate. Calling delegate is very fast and it use less memory compare to the reflection call, it works well in JIT compilation, but need extra work for AOT compilation. To support AOT compilation, you need add extra type registration code. The AOT Code builder will generate the registration code for you automatically.



Quick guide

1. You need create a stub class, the class name and namespace can be anything, but it must be a public partial class. You can find the stub class "AotGeneratedClass" defined for the example project.
2. Open the AOT Code Builder editor windows, specify the "Output Directory". For the example project, we specify the parent folder of "AotGeneratedClass" as the output directory, so the generated code file will be written to this folder.
3. Input "AotGeneratedClass" to the "Class Type Name" text field, which is the stub class you created in step 1.
4. Click the "Generate AOT Code" button, it will generate the code for the partial class, and output a new code file "AotGeneratedClass.Code.cs".
5. If the output directory is a subdirectory of assets folder, Unity will automatically include this script file. Otherwise you need copy the generated file to you project. Now the data binding is ready for AOT compilation.

Let's open the generated code, and see the code that the builder created for you.

The generated class contains two methods, the "private void RegisterProperties()" and the "private void RegisterImplicitOperators()". These method are private, it only used to make the complier to generate proper execution code for the runtime.

These methods contains a series of function calls

```
AotUtility.RegisterProperty<object, object>();
AotUtility.RegisterProperty<object, bool>();
AotUtility.RegisterProperty<object, double>();
…
```

AotUtility.RegisterProperty<TObject, TValue> is a generic method, add a call to this method will make the delegate creation available for any bindable property with the specified type. For example, if we add a call to AotUtility.RegisterProperty<object, bool>(), all properties whose type is bool will use the fast delegate call, if this method is not presented, the call is still available, but it will use the reflection to set/get property value.

You do not need to register all bindable property types. You only need to register the value types, such as primitive types, enum types and struct types. All reference types are handled by type object, e.g. ICommand, AnimatorTrigger, ObservableList<T>, etc.

Notice that the method call is generated by scan the code assembly. It will include all properties which call the SetProperty method and properties referenced by the NotifyPropertyChanged method. It works well in most cases, but if the property is bound directly (without call SetProperty method or not referenced by any NotifyPropertyChanged method), it will not be recognized by the code scanning. You can associate the property with BindableProperty attribute to let the builder knows it's a binding property. Or you can add the type registration code manually.

The method RegisterImplicitOperators is used to register implicit converters. It registers all available implicit operators, the available types are generated by the implicit operator builder.

Don't warry if you miss any type registration, the framework will log an error in the console when the specified fast delegate can not to be created, which means you missed a type registration for the specified type. The data binding still works, it will fall back to reflection call.

Note that, the AOT code is only for AOT compilation, it is not needed for JIT platform, such as the Editor and Android Mono. The generated code will remain valid if no new value type is added for data binding, so you don't need to generate the AOT code every time the scripts are updated.

AOT Code builder parameters

- Output Directory
  The output directory for generated source code. Default is Project/Code.
- Code Template
  Template file for generated C# source.
- Class Type Name

The name of target class. The class must be a partial class.

Code Check Tool

To detect source changes, your class needs to provide proper property changed notifications. If your class is derived from bindable class, you should always use the property to update the value. A common mistake is to set the backing field directly, so the event is not raised. Another mistake is that the property name you specified is not correct, the following code example shows these errors.

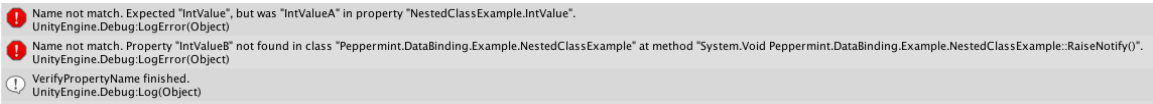
```
public int IntValue
{
    get { return intValue; }
    set { SetProperty(ref intValue, value, "IntValueA"); }
}

public void RaiseNotify()
{
    NotifyPropertyChanged("IntValueB");
}
```

This first error is the SetProperty method, the last parameter is an invalid property name. The correct value is “IntValue”, but the actual value is “IntValueA”. The second error is the NotifyPropertyChanged method, the parameter “IntValueB” is not a valid property name, and the correct value is “IntValue”. These errors may occur when you rename the property, but the property name string is not renamed.

These errors can be detected by runtime, but it has a big impact on performance. Instead, an editor tool is provided for detection these errors. The Code Check Tool will do a static analysis of the assembly, and verify all SetProperty and NotifyPropertyChanged method invocations.

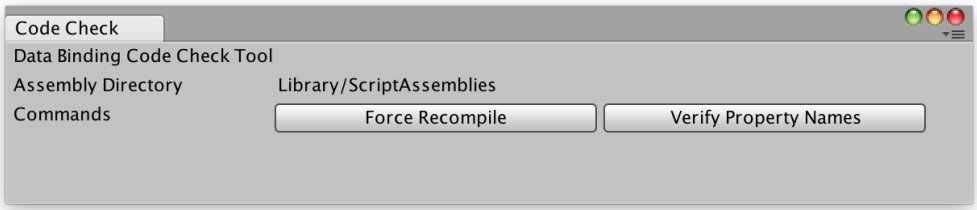
Quick guide

Open the Code Check Tool, Click the “Verify Property Names” button. The tool will load and verify all script assemblies, and log the error into console. The error reported by the above code example is shown here.



Code Check Tool parameters

- Force Recompile
  Force Unity to recompile the code and generate the assembly file.
- Verify Property Names
  Call external tool to do the verification. Detected error will be redirected to Unity console.



Note that the tool checks for errors by static analysis, it only supports literal string as property name, and does not support variable.

Data Binding Graph

Data binding graph is a viewer which displays data binding components within a transform node. To create a graph, you need to select a transform node from the hierarchy window, and then click the Create Graph in toolbar.

The generated graph only contains 3 types of components: Binder, DataContext and DataContextRegister, and their numbers are displayed in the right corner of the toolbar. The component is displayed as a colored node: DataContext is blue, DataContextRegister is green and Binder is yellow. The reference between components is displayed as a curve. The graph is generated by searching the transform hierarchy using DFS, so the horizontal position represent the depth in hierarchy, and the vertical position represent the order.

The graph window shows the outline of data binding components in selected view, which make it clear and easy to locate the component. Click any node in the graph will select the component in inspector. Hold middle mouse button in the blank area to drag the graph.

The source path of each binder is displayed in Binder node and the collected source path list is also displayed in DataContext node. This information should be useful, for example, you can verify if the view is compatible with your binding source, or to check if there is any unbound properties, etc.

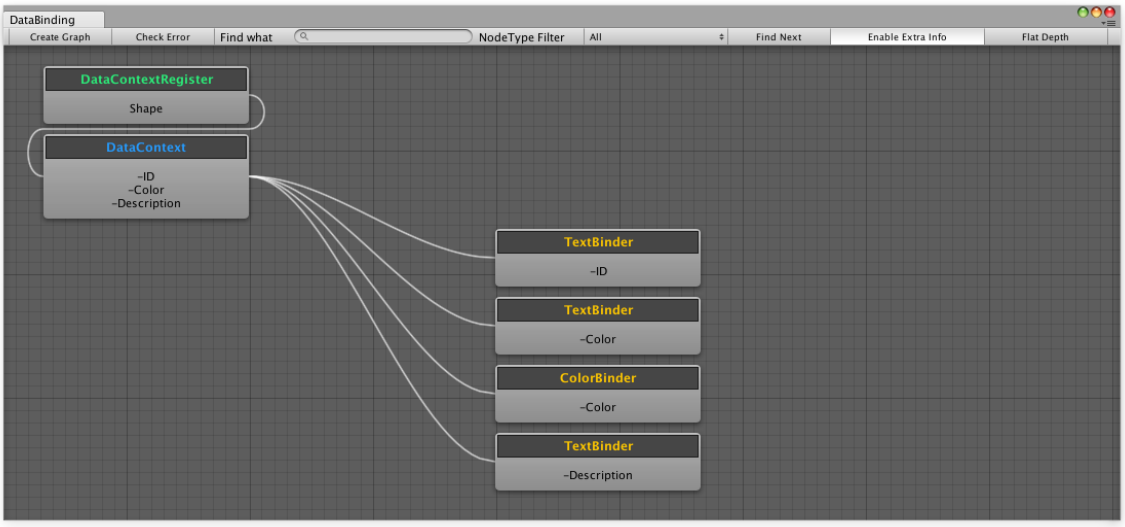
You can search the node in the graph. Input the text in “Find what” text field, and choose the node type you are interested in. Click the “Find Next” button, if the text is found within any node, the node is selected and the view will move so as to center on this node.

Data Binding Graph toolbar

- Create Graph
  Create a graph for the selected node.
- Check Error
  Check if there are unused DataContext components. The unused DataContext is a DataContext that has no child binder.
- Find what
  Input the text you want to find.
- Node Type Filter
  Filter the search with specified node type.
- Find Next

Search for the next node.

- Enable Extra Info
  Collect extra info when creating the graph.
- Flat Depth
  Flatten graph node depth.
- Enable Drag
  Enable Node dragging.

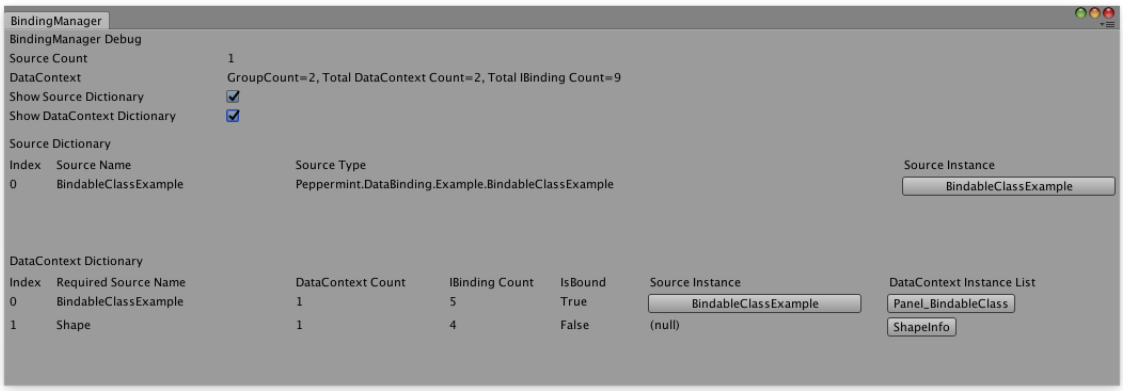


BindingManager Debug

BindingManager Debug is an editor window to show the runtime status of the BindingManager. It only works when the game is playing.

The source section shows the source name, source type and source instance. If the source is UnityEngine.Object, a button with its name is shown, you can click the button to select this source, otherwise it is shown as a text label.

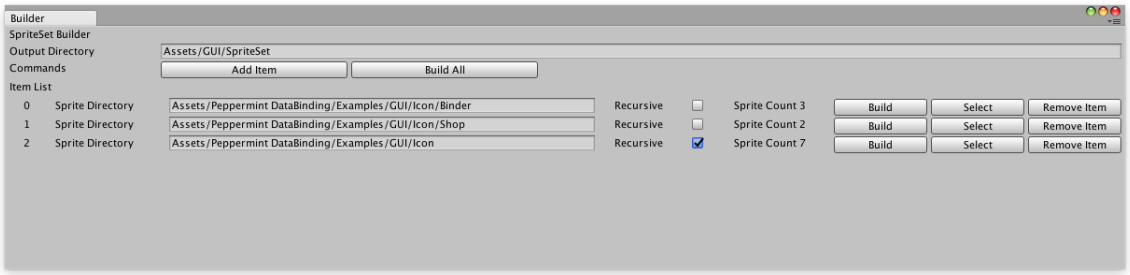
The data context section shows data context group information. Each group contains a required source name, data context count, binding instance count, isBound and the data context instance list. If this group is bound, the binding source instance is also shown, otherwise is shown as null. The data context instance list includes all registered data context objects. Like the source instance, if it's shown as a button, you can click it to select this data context.



SpriteSet Builder

SpriteSet is just a collection of sprites. You can create a new SpriteSet asset by open the "Assets/Create/Peppermint/Sprite Set" menu. You can drag any sprite you want to include to the "Sprite List". The name of the sprite will be used as a key for querying the sprite, so it must be unique. The SpriteSet is used in SpriteNameConverter, which query the sprite by the sprite name. To reduce the memory size, you can separate the sprites into different sets. For example, you can create one set for in game icons, one set for menu icons and one set for common/shared icons. A common way to separate the sprites is using directory, all sprites belonging to the same set should be in the same directory.

SpriteSet Builder is a tool which can build the SpriteSet from specified directory. You only need to config the build directories once, after the directories are updated, you and click the "Build All", all SpriteSet assets will be updated.



Quick guide

If this is the first time you open the SpriteSet builder, it will create a settings file in "…/Peppermint DataBinding/Settings".

First you need to specify the output directory for generated SpriteSet assets. The default directory is "Assets/GUI/SpriteSet".

Next you can add a config item to the list by click the "Add Item" button. In each item, you can specify the sprite directory, the builder will search all sprites in this directory. Enable "Recursive" will recursively search subdirectories for sprites.

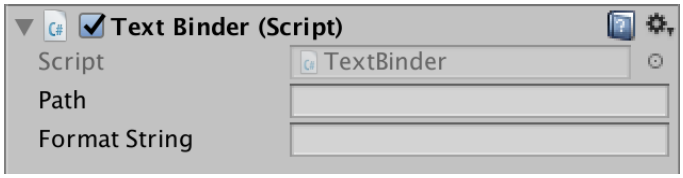
Click the “Build” button will write the generated SpriteSet to the output directory, the last directory name will be used as the name of generated SpriteSet asset. Click “Select” will open the generated SpriteSet asset. You can remove any item by click the “Remove Item” button.

After you update the sprites, you simply click the “Build All” button to make sure all SpriteSet assets are updated.

Built-in Binders

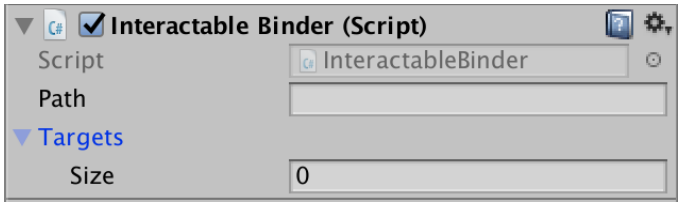
TextBinder

Bind the text property to a property of any type. If the format string is empty, the value's ToString method is called, otherwise the specified format string is used. The source type can be any type.



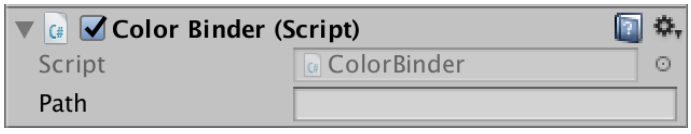
InteractableBinder

Bind Selectable targets to a Boolean property. The value control the interactable of specified targets.



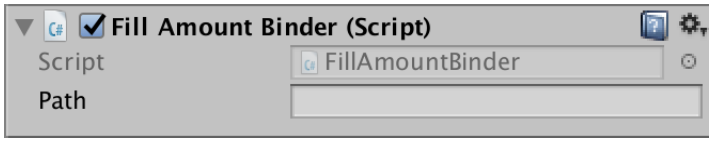
ColorBinder

Bind the color property of UI.Graphic to a color property. The source type must be Color.



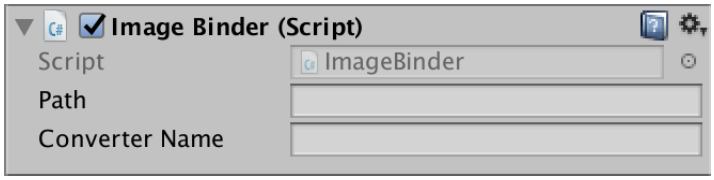
FillAmountBinder

Bind the fillAmount property of Image to a float property.



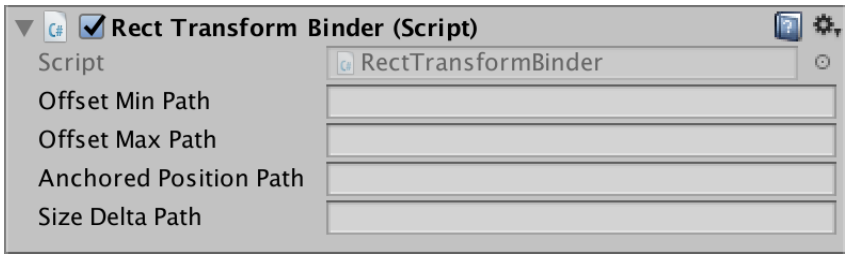
ImageBinder

Bind the sprite property of Image to a string property. ImageBinder must specify a converter, which convert a string to a sprite object. See SpriteNameConverter for more information.



RectTransformBinder

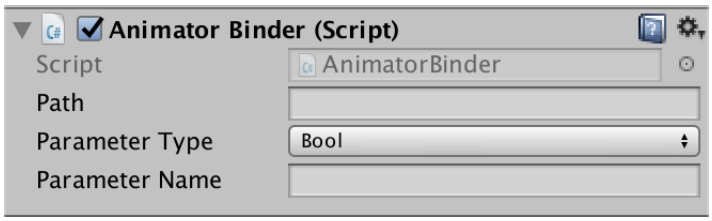
Bind the position properties of RectTransform to source properties. Specify the source path you want to bind, empty path will be ignored.



AnimatorBinder

Bind the specified parameter of Animator to a property. It supports bool, int, float and trigger parameter types. Parameter name is the associated parameter name in animator. If the parameter type is trigger, the source type is AnimatorTrigger.

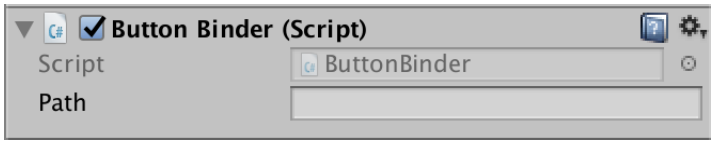
Animator will reset its parameters after deactivated, the AnimatorBinder will restore the parameter in OnEnable method. Trigger is also restore if the event is called within the max period.



ButtonBinder

Bind Button to an ICommand property. The source type must be ICommand. The Execute method will be called if Button is clicked. The interactable of Button is controlled by the CanExecute method.

The Execute method has no parameter, you can use closure to capture extra parameters when you create the DelegateCommand.



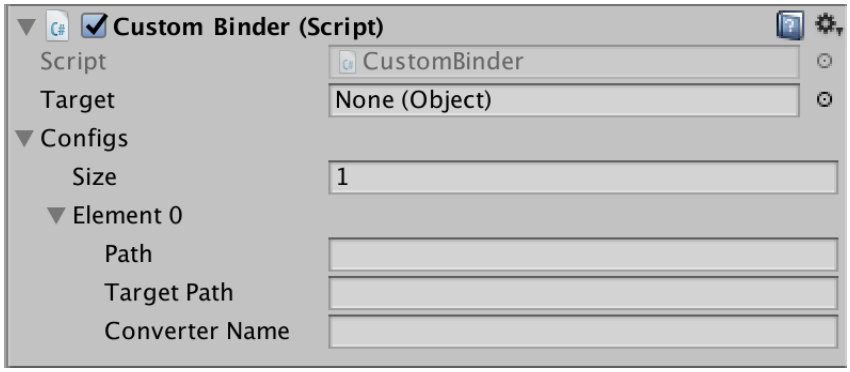
CustomBinder

CustomBinder can bind any property of Unity Object.

Set the target object first, you can set it by dragging a unity object to target field. Then specify the number of binding in configs. In each config, the path is the source path of the binding. The target path will be shown as popup menu, you can choose the target property from the popup. If the target is null, the target path will be shown as string field. The converter name is optional.

Create a binder for some rarely used property is impractical. CustomBinder is a configurable binder, it can bind any property of Unity.Object.

The CustomBinder just do simple binding which update the target property if source property changes. If the type does not match it will use the converter you specified or the default converter. If you need more than just value copying, you should create your own binder.

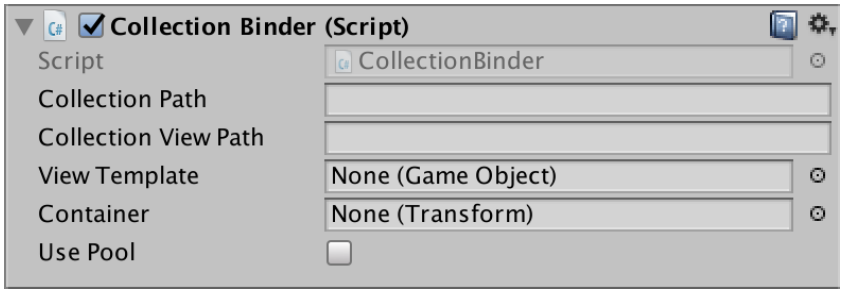


CollectionBinder

CollectionBinder handles collection binding.

View template is the UI template for collection item, it must contains a DataContext component to work with data binding. Collection path is the source path of collection binding. The container is the parent of all created view, CollectionBinding does not handle any layout calculation, so it can be used with any layout group. Collection view path is the source path of the created CollectionView, if you do not use it just leave it empty. Enable use pool can improve the performance, see ViewPoolManager for more information.

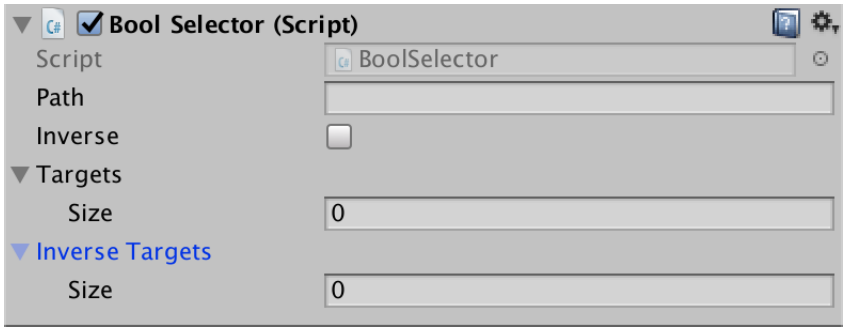
ObservableList<T> is the best collection type for collection binding. Although the source type can be any IEnumerable type, such as T[], List<T>, these types do not implement INotifyCollectionChanged interface, so the view will not get updated if new item is added or deleted. If you know the collection is not changed after binding, you can bind to these types.



BoolSelector

Activate targets by comparing the boolean value of the source property.

If the value is true, targets are activated, otherwise inverseTarget are activated. Set inverse to true will negate the value.



IntSelector

Activate targets by comparing the integer value of the source property.

Set the size of configs array first. In each config, specify the value to be tested and the targets to be activated. The test function decides how the test is performed. When the value changes, the

IntSelector will test the value with each value in configs, the passed targets will be activated, the failed targets will be deactivated.

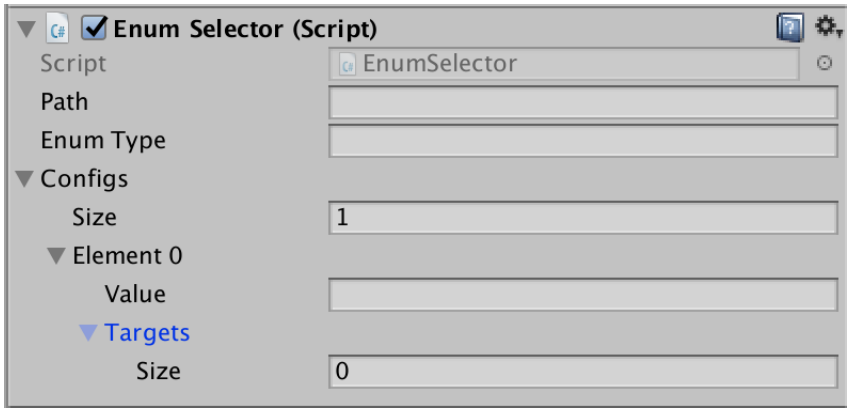
EnumSelector

Activate targets by comparing the string value of the source property.

EnumSelector is a custom selector for enum type. To setup this selector, you need to specify the enumType field first, which is the type name of the enum. If the type name is valid, the editor will use enum popup for the value field, otherwise it will use string field instead.

The enumType field supports type name, partial type name, and namespace qualified type name, e.g. "EnumA", "FooClass+EnumA", "Namespace.FooClass+EnumA". If the type name conflict, you can try partial type name, and namespace qualified type name respectively.

The editor will search all script assemblies and all “UnityEngine” assemblies.

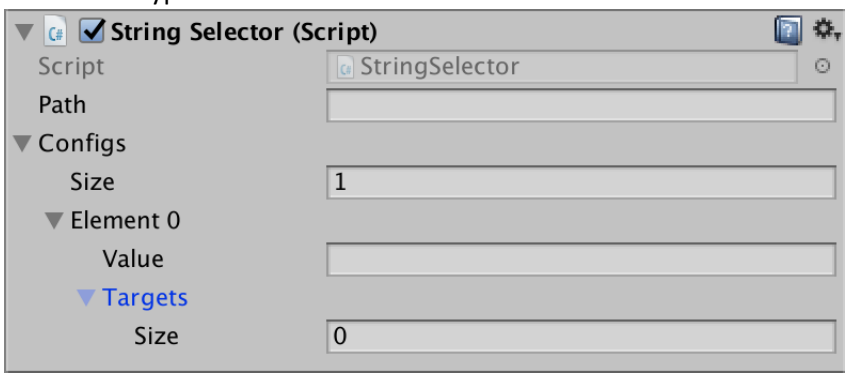


StringSelector

Activate targets by comparing the string value of the source property.

The configuration is similar to IntSelector and the testing function is string equality. StringSelector is a general selector, the source type can be any type, and the default converter will handle the convention for you.

You can bind to Enum type, just set the value to the name of enum constant. You can also bind to Boolean type and set value to True or False.



ColorSetter

ColorSetter set the color of UI.Graphic object by comparing the string value of the source property.

The configuration is similar to StringSelector. If the value matches the given config, the color from the matched config will be used, otherwise the default color is used.

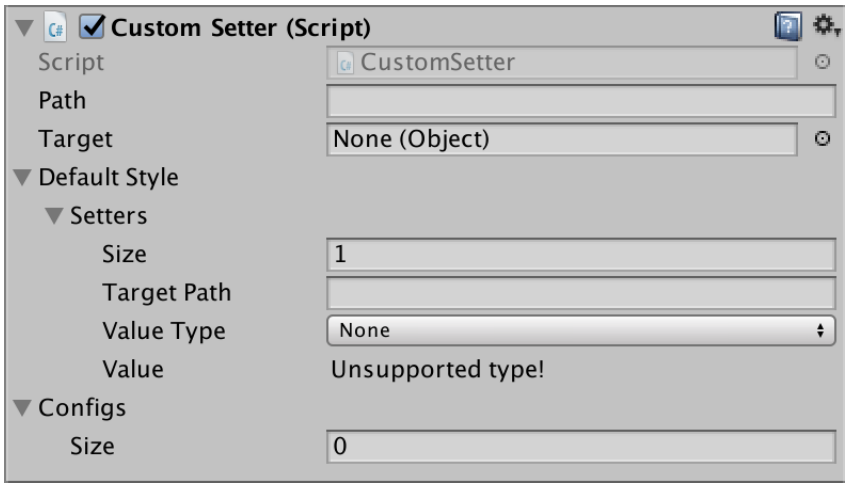
CustomSetter

CustomSetter can be used to set any property of Unity Object.

The configuration is similar to StringSelector. If the value matches the given config, the style is applied to target object.

You should specify the target first. The style contains a group of StyleSetter, and in each StyleSetter you can choose the target path from popup menu. The value type is automatically selected based on the property type, you only need to specify the value.

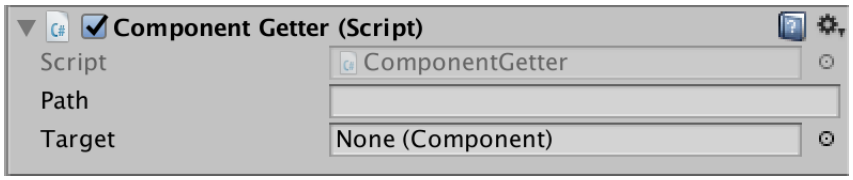
If the target is null, the target path will be shown as string field and the value type must be manually setup.



ComponentGetter

Gets component from the view.

ComponentGetter creates a one-way to source binding. Once bound the source gets the value from the target, and the source value is set to null if unbound. You can set the target by dragging a component to target field.



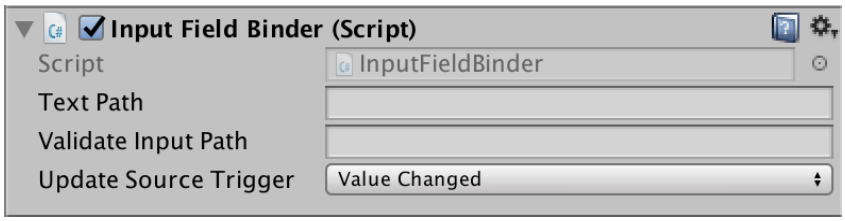
ToggleBinder

Binds Toggle to a boolean property. ToggleBinder is a two-way binder, it get the value from the source, and update the source if Toggle's value changes.

InputFieldBinder

Binds InputField to a string property. InputFieldBinder is a two-way binder, it get the text value from the source (string type), and update the source if InputField's value changes.

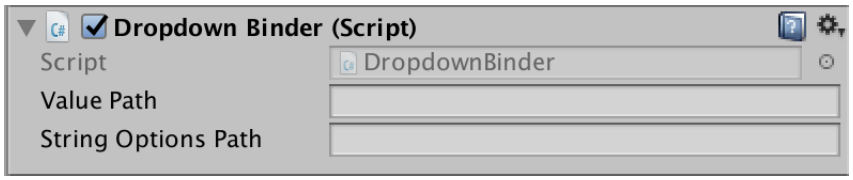
Text path is the source path of the text, and the update source trigger decides when to update the source. If you need to validate input characters, specify validate input path, otherwise leave it empty. The source type of validate input is a delegate, which must be compatible with the InputField.OnValidateInput delegate.



DropdownBinder

Bind Dropdown to an int property. DropdownBinder is a two-way binder, it get the value from the source (int type), and update the source if Dropdown's value changes.

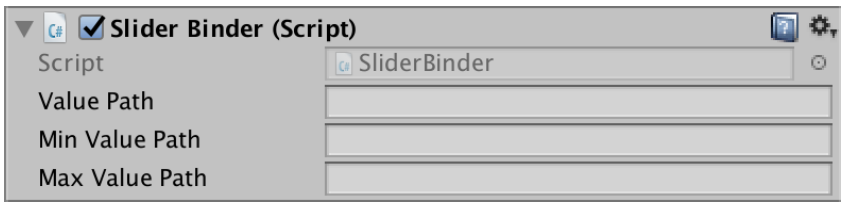
The value path must specified, it's the index of currently selected option. String options path is optional, if specified, a list of string will be used as options.



SliderBinder

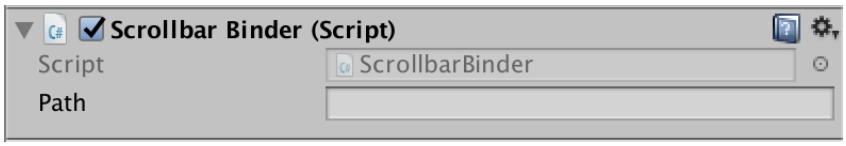
Bind Slider to a float property. SliderBinder is a two-way binder, it get the value from the source (float type), and update the source if Slider's value changes.

You must specify value path, which is the source path of Slider's value. The min value path and max value path is optional. If specified, it control the min and max value of the Slider.



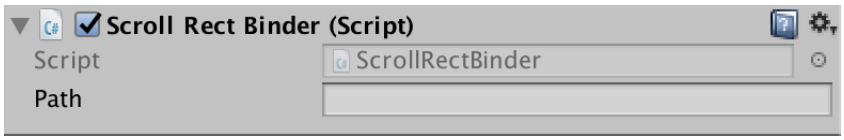
ScrollbarBinder

Bind Scrollbar to a float property. ScrollbarBinder is a two-way binder, it get the value from the source (float type), and update the source if Scrollbar's value changes.



ScrollRectBinder

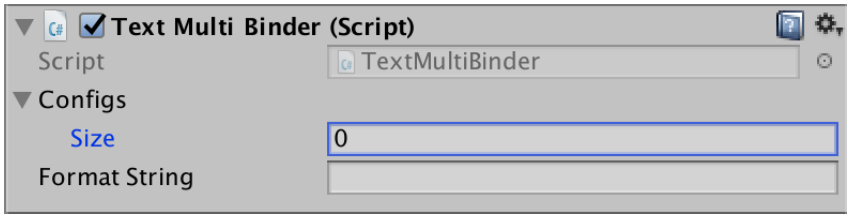
Bind ScrollRect to a Vector2 property. ScrollRectBinder is a two-way binder, it get the value from the source (Vector2 type), and update the source if ScrollRect's value changes.



TextMultiBinder

Bind the text property to multi properties.

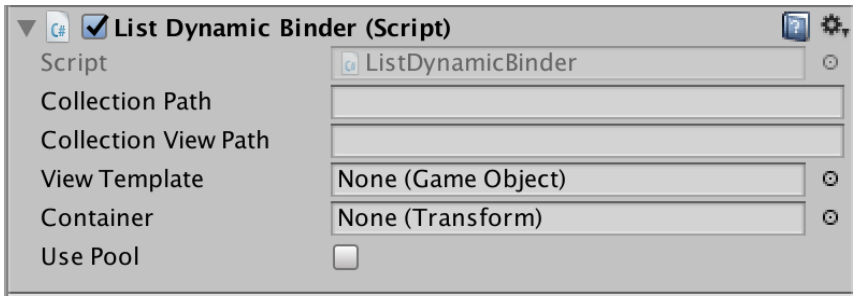
First set the size of configs, and then set the target path in each entry. If the format string is empty, string.Concat method is used to concatenate all string values, otherwise the specified format string is used. The source type can be any type.



ListDynamicBinder

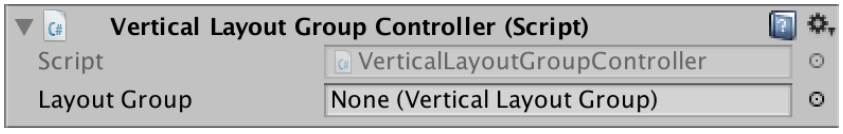
ListDynamicBinder is a collection binder which binds to IList object.

Unlike the CollectionBinder, it only creates and binds the visible items of the list. It requires a dynamic controller to calculate the visible items and handle the layout. It's very useful for scroll views with a large amount of items, e.g. leaderboard.



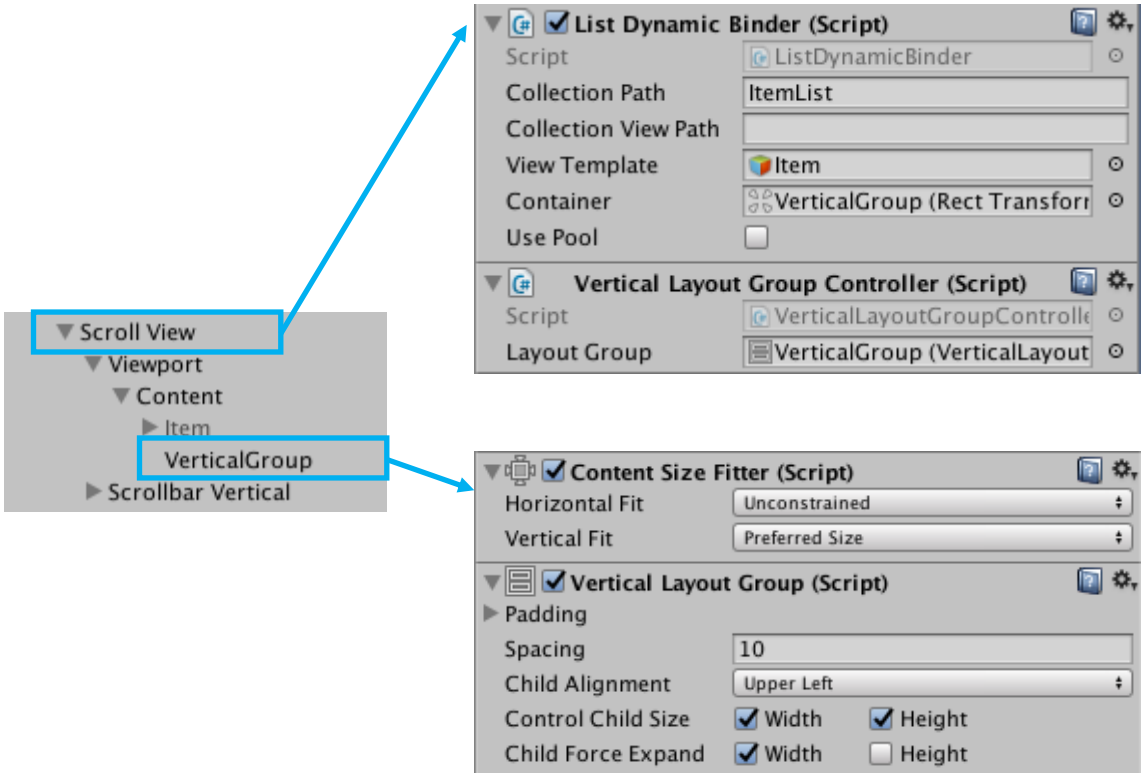
There are two built-in dynamic controllers:

- VerticalLayoutGroupController (For vertical scroll view)
- HorizontalLayoutGroupController (For horizontal scroll view)



Here are the steps for set up a vertical dynamic scroll view.

1. Create a new “Scroll View”, add a ListDynamicBinder and VerticalLayoutGroupController to this GameObject.
2. Add a child GameObject “VerticalGroup” under “Content”. Set the anchor to top-stretch and set the pivot to top-center. (For horizontal scroll view, set the anchor to stretch-left and set the pivot to middle-left)
3. Add a VerticalLayoutGroup to “VerticalGroup”, disable the “Height” of “Child Force Expand”.
4. Add a ContentSizeFitter to “VerticalGroup”, set “Vertical Fit” to “Preferred Size”.
5. On “Scroll View” node, drag the “VerticalGroup” to the “Layout Group” field of VerticalLayoutGroupController.
6. Set the parameters of ListDynamicBinder, most of the parameters are the same as CollectionBinder. The only difference is the “Container” field. You should set it to the “VerticalGroup” node, instead of “Content” node.
7. Add a LayoutElement to the view template prefab, enable the “Preferred Height” and set the value.



For horizontal dynamic scroll view, the steps are the same. Simply change the “Vertical” to “Horizontal”, and change the “Height” to “Width”.

If “Use Pool” of ListDynamicBinder is enabled, the binder will get the pool from ViewPoolManager. If “Use Pool” is disabled, the binder will create a local view pool instead. Most of the time, the pool contains only one view object, so you can disable “Use Pool” to enable the automatically managed local view pool.

How it works

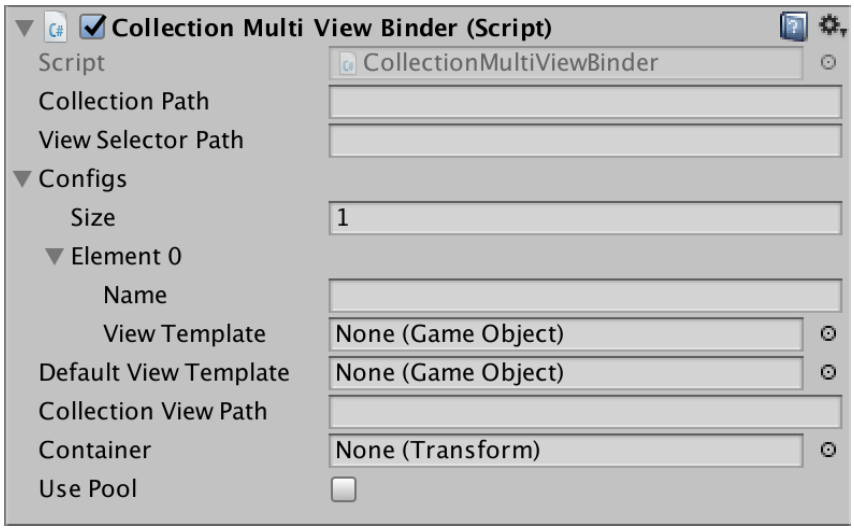
When the collection or the scroll position changes. The dynamic controller calculates the visible items based on the viewport size and the content position. The binder creates the visible items, and attach them to "VerticalGroup". The VerticalLayoutGroup component will do the actual layout calculation. Next, the dynamic controller will calculate and set the vertical position for the "VerticalGroup".

CollectionMultiViewBinder

CollectionMultiViewBinder is an extended version of CollectionBinder. It supports multiple view templates.

"View Selector Path" is the source path for the view selector. It's a delegate that will be called when the view is created. You can set different view templates in the configs. In each config, you must specify a view template, and set a name tag (optional). When the delegate is called, you can decide which view template to be used based on the item object. If the "View Selector Path" is empty, it will use default type name matching.

You can also specify "Default ViewTemplate", it will be used if there is no matching view template.



Extensions

Peppermint data binding can be easily integrated with third-party add-ons. Starting from version 1.3.0, peppermint data binding will include extension packages in “Peppermint DataBinding/Extensions” folder. You can import extension packages to enable third-party add-on support. Before importing the extension package, make sure the required third-party add-on is already imported.

NGUI

NGUI extension includes binders to support NGUI’s built-in controls, such as UILabelBinder, UIToggleBinder, etc. It also includes new collection binders for NGUI. Tested with NGUI 3.11.4.

Most control binders are similar to their uGUI counterpart, and they have the same parameters. You can check the document and examples to learn how to set these parameters.

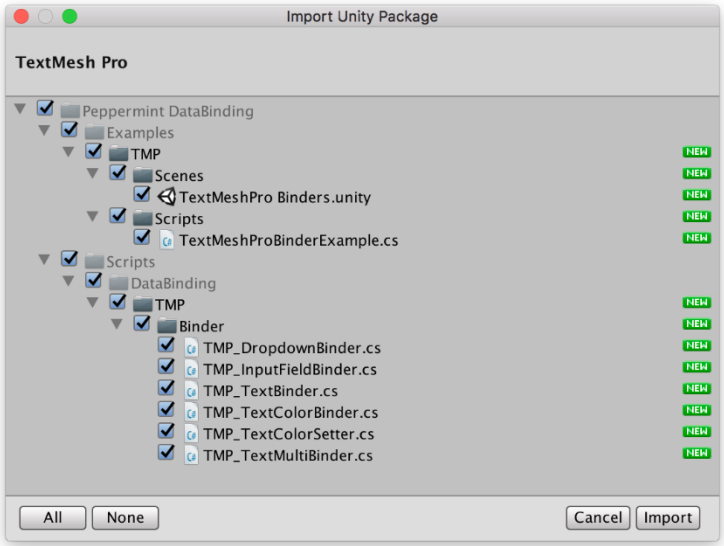
To set up collection binding in NGUI, you must use the new UICollectionBinder. Unlike the CollectionBinder used in uGUI, the UICollectionBinder requires an ICollectionController to update the view after collection changes. You can use the built-in UIScrollViewCollectionController, or create your own as needed.

To set up list dynamic binding, you still use the ListDynamicBinder, and then add the IDynamicController designed for NGUI. The UIGridDynamicController uses UIGrid to do the layout calculation, which supports horizontal and vertical arrangement.

In the examples folder, it contains new examples that demonstrates how to set up these new binders. Note that these examples reference NGUI’s built-in assets and will not work if the referenced assets are missing.

TextMesh Pro

TextMesh Pro extension includes binders to support TMP's built-in controls, such as TMP_TextBinder, TMP_InputFieldBinder, TMP_DropdownBinder, etc. Tested with TextMesh Pro 1.2.2 and Unity 5.5.0f3.



Upgrade Guide

To upgrade Peppermint data binding to its latest version, I recommend following these steps.

1. Back up your project first.
2. Load your project in Unity and create a new scene.
3. Delete the old "Peppermint DataBinding" folder (except the "Peppermint DataBinding/Settings" folder).
4. Import the latest package from asset store.

Setup assembly definition files

Starting from version 1.4.0, Peppermint Data Binding fully support assembly definition files. If you are using Unity 2017.3 or later version, you can setup assembly definition files to reduce compilation time. Here is a quick guide.

1. Add an assembly definition file "DataBinding.asmdef" in "Peppermint DataBinding/Scripts/DataBinding" folder and set "Platforms" to "Any Platform".
2. Add another assembly definition file "DataBinding.Editor.asmdef" in "Peppermint DataBinding/Scripts/Editor/DataBinding". Set "Platforms" to "Editor" only and add the assembly definition file you created in the first step to its "References" list.
3. Set up assembly definition file for your scripts folder and add the "DataBinding" to its References list.

If you need more information about assembly definition files, you could check Unity document for details.

Contact

For any questions about this framework, feel free to contact me at:

E-mail: peppermint-unity@hotmail.com