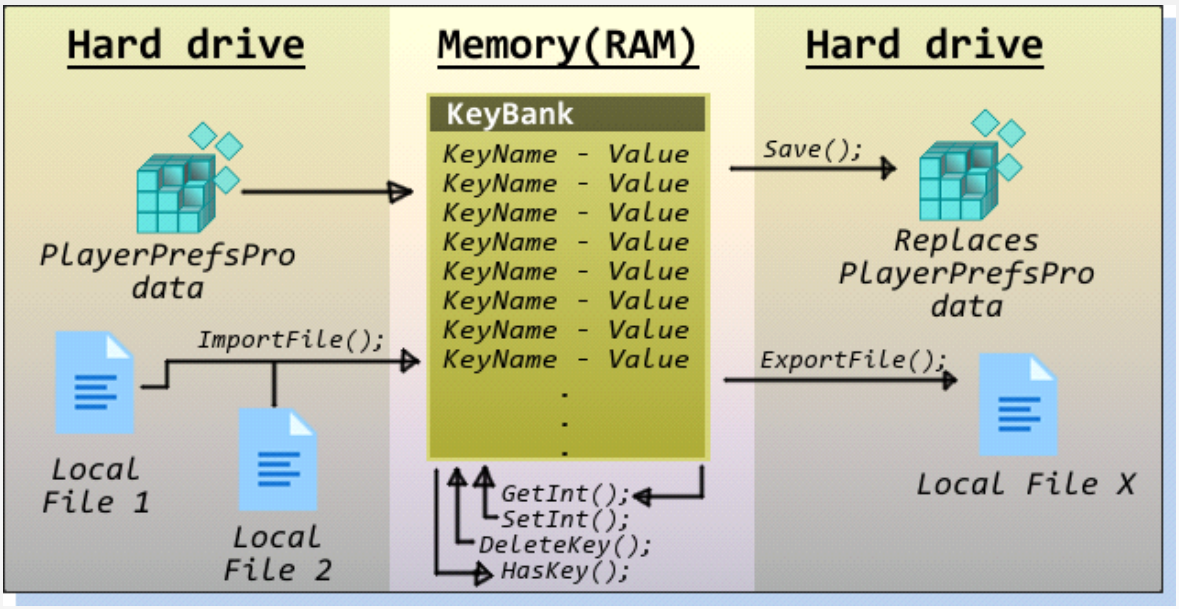
PlayerPrefsPro Documentation (V1.0)

A hassle-free PlayerPrefs' data manager & encryptor.

View in the Unity Asset Store

Table of contents

How does PlayerPrefsPro work?



PlayerPrefsPro works similarly to the Unity PlayerPrefs class, just by replacing the class name should be enough to get it working on your project. At its core, all values which need to be stored are associated with a key name, but this information is encrypted and cannot be modified externally, unlike with the default PlayerPrefs solution (because it was not meant to secure data).

All the key values will be stored in the hard drive as a single encrypted data block, meaning that, you cannot modify the key values individually once they have been saved by the application. This data block is decrypted by the game using a pair of seed numbers (Which you can customize later) and loaded into memory automatically once a key request has been made; From this point, all data is managed by the application itself in a structure named **KEY-BANK**, In order to output these values to the registry, the `Save();` call must be used at some point(such as a save game event in your game), otherwise, YOUR DATA WILL BE GONE on the next application run.

The keys stored in memory are only partially encrypted (In order to improve performance), however, they are compared against their checksum (which is stored in the bank too), so as soon as a key is called the comparison is performed, and if the values mismatch, the game will throw a customizable exception method (Cheat Engine won't work here).

You can alternatively create encrypted local files containing a list of specific keys, which can be then loaded back into the game. Registry storage can be used along with local files for better control over your data.

One main difference between the old system and this one, is how the `DeleteKey();` method works, since our values are stored as a single block, it is not possible to selectively delete key values from the registry, we instead delete the key from the key-bank, and the next time we do a Save() call, the new data block will be stored in the registry. This process is not done automatically after deleting a key for performance reasons, is more intensive to save all the data block that a single key.

Don’t worry if all this sounds confusing, this is only an explanation of how our solution works, **you can skip straight to the code snippets section for examples of how to use out of the box.**

NOTE: Key-names can be used more than once if the type is different, for example `SetInt("key1",0);` and `SetString("key1","xyz");`, but is not recommended.

Which data types are supported?

The following data types have its own custom named methods in PlayerPrefsPro:

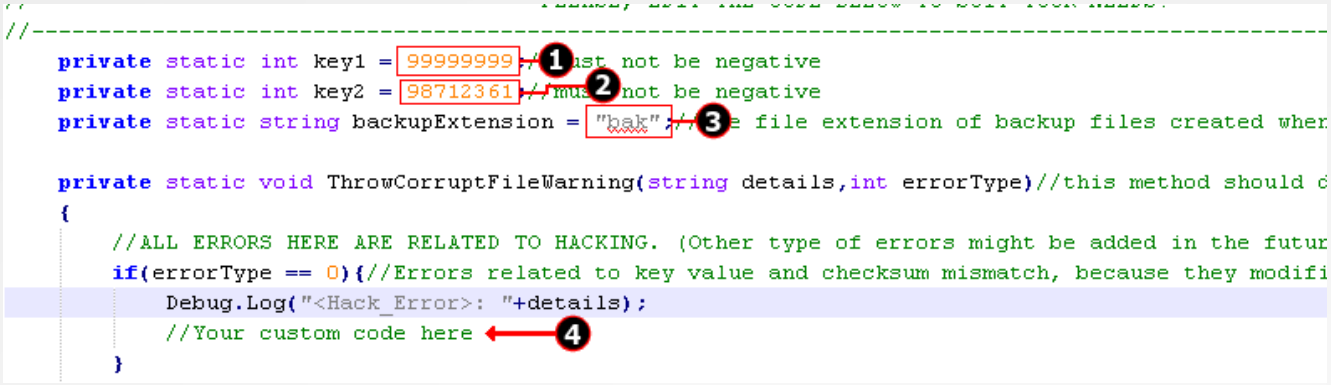
Int, int[], Int[,], String, String[], String[,], Float, Float[], Float[,], Bool, Bool[], Bool[,], DateTime, DateTime[], Vector2, Vector2[], Vector3, Vector3[], Vector4, Vector4[], Color, Color[].

More types could be added in the future if requested by enough asset users, however you can technically add your own types by following the asset logic, but it might not always work and requires certain tinkering. More about this in the **How to** section.

Setting Up PlayerPrefsPro

To install the PlayerPrefsPro class, simply put the PlayerPrefsPro.cs file anywhere in your project. It comes with some example scenes and documents, but you can perfectly delete them to keep the project clean.

By default, the only thing you will need to modify is the exception handling lines [4], there you will write what behavior you want your game to execute when a hack is detected. We recommend simply loading a new scene with a text "corrupted data". Remember that only hackers will be affected by this, so you don't really need to offer them a solution to revert damages, it's their own fault for cheating. You can supply a button to wipe all corrupt data in such case. See the **How to reboot save data if it becomes corrupted** section.



You can then call the following line: `SceneManager.LoadScene("your scene");`, or do an `Application.Quit();` to shutdown the game without giving reasons. Alternatively, you can just instantiate a message box or call your custom game notification system, is up to you, experiment to what works best for your game. To try to break up the game simply enter into the registry and modify anything in the weird named key values, you will see mostly random symbols with no particular meaning. Then, the next time your game uses a `Get<type>();` method, `Save();` or anything related to reading data in the key-bank, it should throw the exception.

The other settings which can be **optionally** changed are the key seeds; These keys will determine what symbol range to use for data encryption, the main purpose of this step is to make all games that use this save system to have a different encryption process, reducing the chances for a "general" hack tool to work with your game. The Chances that this will happen are very low, but just to be careful, you can replace both key1[**1**] and key2[**2**] values to a number in the following range: 90.000.000 – 99.999.999. Going out of this range can cause sometimes for exceptions to occur, we recommend staying in it. NOTE: key1 and key2 should be different values, just in case.

Lastly, you can edit the backup file extension [3]; In case you will export save files, PlayerPrefsPro can make backups of the file to avoid possible write errors, for example, if the application crashes while saving or the device gets shutdown while data is being written, then the next time the file will be detected as corrupted, the game will not throw an exception unless the backup file is corrupted too; if it is not, it will just replace the corrupted file and will load the last working save file. **Do not include "." in the extension name, only the file type.**

Migrating from PlayerPrefs to PlayerPrefsPro

All PlayerPrefs methods are covered by PlayerPrefsPro. Just replace "PlayerPrefs." with "PlayerPrefsPro." and you are done! Now the keys will be encrypted in the registry. Please note that PlayerPrefsPro will not use internally the same key names as PlayerPrefs does, in consequence, the save data used by PlayerPrefs will not migrate to PlayerPrefsPro; This means that PlayerPrefsPro can only recognize key names stored by itself, but it cannot read key names stored using the default

PlayerPrefs. This is generally not a problem unless you are planning to upgrade an already released game, in that case old save files will not be compatible (unless you migrate the data yourself with custom methods).

Some quick examples:

```
PlayerPrefs.SetInt("myKey",10);
PlayerPrefs.HasKey("myKey");
```

```
PlayerPrefsPro.SetInt("myKey",10);
PlayerPrefsPro.HasKey("myKey");
```

There are a lot of new methods which are unique to PlayerPrefs, you can check the **Methods list** section for more information.

Managing local files and handling errors

PlayerPrefsPro allows you to output save files into a custom directory inside the applicationDataPath. You can include specific key-names into as many files as you need. The file extension should be included in the file name to be created, **but it must not be the same as the backup extension** defined in the setup step. For example, profile.bak is not a valid file name if your backup extension is "bak".

```
PlayerPrefsPro.ExportFile(string fileName, string[] keysToExport, bool makeBackup);
```

This is the method used to export files. The field "fileName" will hold your target file name, it is allowed to overwrite any file called the same way, so writing errors should never arise from it. Your file name can contain a subdirectory along with it, for example "Profiles/player1.sav" will create a folder in the applicationDataPath directory called "Profiles", and inside it, the “player1.sav” file will be created. The second field, is an array of key names; The method will try to find keys with that name and include them in the save file. If a key-name doesn't exist, it will simply be ignored. The third field is used to create a backup of the save file, if it is true, a file named "fileName" will be created in the same directory, but its extension will be the backup extension set up before.

The following example shows how to export three key values.

```
PlayerPrefsPro.SetInt("key1",10);
PlayerPrefsPro.SetString("key2","Hi there dude.");
PlayerPrefsPro.SetBool("key3",true);
string[] keysToExport = new string[]{"key1","key2","key3"};
PlayerPrefsPro.ExportFile("save.dat",keysToExport, true);
```

Please note that we are never calling `PlayerPrefsPro.Save();` That method would put the three keys into the registry, but at the moment, they only exist in the memory. This is good if you don't want to store values in registry, but only on local files, like most games do. Now, if you intend to load a save file, all you have to do is call the following method:

`int ImportFile(string fileName,bool checkForBackup);`

This method will return one of the following error codes when called:

0: *no error*

1: *file was not found (If backup check = false).*

2: *file was corrupted or hacked (If backup check = false).*

3: *file was corrupted or hacked and the backup was not found.*

4: *file was corrupted or hacked and the backup was corrupted too.*

While you can call the method without checking the returned error, it is not recommended to do so.

An example of a good practice is to first check if the file exists before even trying to load it.

```
if(PlayerPrefsPro.DoLocalFileExist("user.sav",true)){//“true” is to check for backup file too.
	int error = PlayerPrefsPro.ImportFile("user.sav",true);
	if(error != 0){
		//Here you show a message in your game about the file being corrupted.
		//Then you give some options to the player as start a new game.
	}
}
```

A note about performance

Writing a big volume of data into the hard drive can cause noticeable lag spikes. Since this asset allows to save types such as arrays, it is somewhat easy to overlook the fact of keeping data small, especially if you try to save, for example, a 500 by 500 2D array of integers, this can cause a very large save file to be created, and the game might even freeze while writing this data. In cases when this is totally needed, we recommend using a special method which will provide enough time to execute a statement before and after the method, this can be used for example, to show a "Saving..." sign and then destroying it after the data has been written.

To solve this problem can be done by using the following method:

`PlayerPrefsPro.instance.WaitForSave();`

This is a pseudo-coroutine, it will have a certain delay, but won't stop the game from freezing while loading, this cannot be used to show animated loading bars or things alike, this is an emergency fix in case your game needs it.

So, a typical use case would be:

```
IEnumerator SaveGame(){
	mySign.SetActive(true);
	yield return PlayerPrefsPro.instance.WaitForSave();
	mySign.SetActive(false);
}
```

Here "mySign" is an object which shows a saving text, it is enabled before calling the save and then is disabled when it finishes. The game will freeze during this time, depends on how much data needs to be saved.

Similarly, it can delay to write and read local files, in such cases we use the same principle, but with methods such as `WaitForExportFile();` or `WaitForImportFile();`.

You can force the load of registry encrypted values by calling the method `WaitForForceLoadPrefs();` More about this in the code snippets section.

Using PlayerPrefs along PlayerPrefsPro and local files

It is perfectly possible to use Unity's PlayerPrefs along with PlayerPrefsPro and even load in several local files. When any information is loaded into the game using PlayerPrefsPro, it will INCLUDE the keys available in that data block, but it will not delete any already existing key. If a key already exists it will simply be replaced with the most recently loaded data block value. Unity´s PlayerPrefs doesn't have to cope with this since all keys are managed individually, not by block. For example, assume you store "key1" and "key2" using Unity's PlayerPrefs, if you call `PlayerPrefsPro.HasKey("key1");` it will return false! because PlayerPrefsPro can only detect its own created keys! however, it's perfectly fine to use both classes at the same time. One reason you might want to work with both systems is performance, since the smaller the PlayerPrefsPro data block is, the less time it will take to store/read/modify or decrypt the keys, remember it can only store all keys at once, not each individually.

One case where this can be applied could be, for example, storing configuration options such as key binding or any other settings, which could be edited anytime using Unity's PlayerPrefs, and, for data such as level progression, scores or achievements use PlayerPrefsPro just to prevent it from being edited. You can of course store all data using PlayerPrefsPro alone, but this would not be optimal, especially if there are huge volumes of data to be loaded or saved.

If you have too much information, but don't want to store it all at once in the key-bank to improve performance, you can output specific information (such as level data) to individual files, and only when you need to load the level, call that file to retrieve the keys needed. Remember that when calling

`PlayerPrefsPro.Save();` it will store ALL KEYS in the key-bank at that moment in the registry, performance wise is always better to store the keys you need in local files.

```
//1-We save the settings separately (Can be edited anytime at registry)
PlayerPrefs.SetInt("master volume",100);
PlayerPrefs.SetString("keybinds","82734234724");
PlayerPrefs.SetFloat("auto aim",0.1f);
PlayerPrefs.Save();
//2-We save the user progress and stats into the registry (Cannot be edited by users)
PlayerPrefsPro.SetInt("level",3);
PlayerPrefsPro.SetFloat("cash",14.52f);
PlayerPrefsPro.SetIntArray("Ammo",new int[]{45,12,0,56,200,1,2,0});
PlayerPrefsPro.Save();//NOTE that this save is different, only for PlayerPrefsPro
keys.
//3-We store individual level information
string level1Data = GetLevelData(1);//assume this method convert a level to a string.
string level2Data = GetLevelData(2);
PlayerPrefsPro.SetString("level_1",level1Data);
PlayerPrefsPro.SetString("level_2",level2Data);
string[] levelData = new string[]{"level_1","level_2"};//list of keys to store
PlayerPrefsPro.ExportFile("Levels/Level1.dat",levelData, true);
```

Please note that we didn't `Save();` again at the end, this is because we don't need to save those keys in the registry, we can use them if we need them by loading the "Levels/Level1.dat" file. In the case where you only want to save specific keys in the registry, you can call `PlayerPrefsPro.Save(keyList);` and as the argument you pass the key-names you want to include, any other key won't get written into the registry but will remain existing in the key-bank.

Deleting keys efficiently

Sometimes you could end up with a bunch of keys in the key-bank that most probably won't be used again, in such cases several actions can be taken. You can manually delete the unwanted keys, delete all keys but a certain exceptions, or clear the entirety of the key-bank and repopulate it again with the files you will use at the time.

The following methods will try to delete the keys in the key-bank if they exist (But not from registry).

- To delete a key simply call `PlayerPrefsPro.DeleteKey("myKey");`
- To delete all values from the key-bank (run-time), call `PlayerPrefsPro.CleanBank();`
- To delete all but a list of keys, call `PlayerPrefsPro.CleanBankExcludingKeys(keyList);` where “keyList” is a **string[]** containing the key names to keep.

The following methods will delete values from memory AND from the registry.

- To delete all PlayerPrefsPro keys call `PlayerPrefsPro.DeleteAllProKeys();`. This deletes registry keys as well.
- To delete all Unity's PlayerPrefs and PlayerPrefsPro keys call `PlayerPrefsPro.DeleteAll();`. This affects registry keys as well.

HOW TO: (Code snippets)

How to do the most basic saving system

Registry based saving system:

```
//step 1: we put our keys inside the key-bank.
PlayerPrefsPro.SetString("playerName","Neo");
PlayerPrefsPro.SetInt("hp",9000);
PlayerPrefsPro.SetColor("hairColor",Color.white);
//step 2: we write all keys into the registry.
PlayerPrefsPro.Save();
```

How to do a basic load of the registry data

Data in the registry is automatically loaded as needed, for example, upon calling a `GetInt();` method, the data block in the registry will be decrypted. This data should be loaded only once, but if you need to force load the registry at a given moment you can use the `ForceLoadPrefs();` method.

```
//Example 1: regular use and not so big data block.
PlayerPrefsPro.GetString("playerName");
```

That's it! when you called `GetString();` all the data was loaded into the game, but! this will only happen once. Please note that if "playerName" key doesn't exist, it will simply return an empty string, but it cannot cause errors. All methods will return the default value if the key doesn't exist.

Now, let’s assume you want to load the registry, but you don't want to risk any possible lag spike during the game running, we recommend calling this method at game start:

```
//Example 2: Force load data block.
void Start(){
      PlayerPrefsPro.ForceLoadPrefs();
}
```

How to save only specific keys into the registry

The `Save();` method have an override which support a list of keys to save, this way only these keys get exported to registry if they exist. Similar to the regular `Save();` example, we do the following:

```
//step 1: we put our keys inside the key-bank.
PlayerPrefsPro.SetString("playerName","Neo");
PlayerPrefsPro.SetInt("hp",9000);
PlayerPrefsPro.SetColor("hairColor",Color.white);
//step 2: we write only the first two keys into the registry.
string[] toExport = new string[]{"playerName","hp"};//note that hairColor is not
included
PlayerPrefsPro.Save(toExport);
```

And done! only "playerName" and "hp" were stored in the registry, however, the “hairColor” key still exist at run-time. This is very useful for performance reasons, for example, if you are also using local files to import data into your bank, you might not want to save it into the registry, but instead, save it over the original file again when needed, speeding up the load times of your game.

How to export and import a file

In case you want to save data outside of the registry (Unity's PlayerPrefs location), you can use the following methods to selectively output the keys you need. You can create as many files and directories as you want.

Example: Creating a file.

Assume you have 4 keys in the key-bank, they are called key1, key2, key3 and key4. The type is not relevant, assuming the key names are unique.

Then, you create the list of keys you want to output. Place the information in a string array.

```
string[] keysToExport = new string[]{"key1","key2","key3","key4"};
```

```
PlayerPrefsPro.ExportFile("TestDir/test1.dat",keysToExport,  true);//true means, create a backup file too.
```

Now, if you wanted to Import the keys in the file later on, you simply have to call the following:

```
int  code  =  PlayerPrefsPro.ImportFile("TestDir/test1.dat",true);//true means, check for backup if the main file is broken or missing.
```

The variable "code" is the return got from the `ImportFile();` method, 0 means no error, other than that means corrupted file or something similar. Please see the **Managing local files and handling errors** section.

How to check if a key exist

```
bool found = PlayerPrefsPro.HasKey("Neo");
```

If the key exist, it will return true, if not, false.

In case you used the same key name twice with different types, you can check by type instead:

```
bool found = PlayerPrefsPro.HasKey("Neo", KeyEx.kString);
```

KeyEx is a public **enum** supplied by PlayerPrefsPro including the basic types. Other methods use the same principles of supplying key types, but to avoid doing this better use unique key names.

How to check if a file exist

```
bool found = PlayerPrefsPro.DoLocalFileExist("user.sav",true);//second argument is: check for backup if the main file is corrupted or missing.
```

If the file exist, it will return true, if not, false.

Please, avoid naming a file the same as another, even if extension is different the backup file will be the same for both, which doesn't make sense.

How to rename or clone a file.

Save files can be renamed and/or cloned once created, in case you need to do so, make sure you check first if the file actually exists in a given directory, otherwise an IO exception will be thrown and the game execution will stop. Proceed as the example below:

```
//Check if the origin file exists and target name doesn’t before cloning the file:

if(PlayerPrefsPro.DoLocalFileExist(“f1.sav”,true) && !PlayerPrefsPro.DoLocalFileExist(“f2.sav”,true)){
        PlayerPrefsPro.CloneSaveFile(“f1.sav”, “f2.sav”);
}

//Check if origin file exists and target name doesn’t before renaming the file:

if(PlayerPrefsPro.DoLocalFileExist(“f1.sav”,true) && !PlayerPrefsPro.DoLocalFileExist(“f2.sav”,true)){
        PlayerPrefsPro.RenameSaveFile(“f1.sav”, “f2.sav”);
}
```

How to configure the exception method

When the key-bank data becomes corrupted because of a memory hack attempt, the game will throw an exception. To handle this exception, we recommend two things, tell them the data is corrupted and stop game execution, or simply closing the app without explanation.

```
//Sending them to a scene with an error message. Only a part of the method is shown here.
if(errorType == 0){
	Debug.Log("<Hack_Error>: "+details);
	//Your custom code here
	SceneManager.LoadScene("myErrorScene");
}
```

You could use `Application.Quit();` instead.

How to display a Loading/Saving dialogue

When the data block aimed to save or load is too large (Generally 2D arrays or extremely large strings), you will notice the game will freeze for some frames or even seconds, in these cases it is convenient to show a "loading" or "saving" dialog, this dialog however won't be animated, since file reading operations are aimed to use the full application processing power. Best practices in these cases consist of the following:

```
//example 1: loading the registry values at the first loading screen.
IEnumerator LoadRegistry(){
	mySign.SetActive(true);//we make the sign visible
	yield return PlayerPrefsPro.instance.WaitForForceLoadPrefs();
	mySign.SetActive(false);//we make the sign invisible
}
```

```
//Example 2: Saving values into the registry
IEnumerator SaveToRegistry(){
	mySign.SetActive(true);//we make the sign visible
	yield return PlayerPrefsPro.instance.WaitForSave();
	mySign.SetActive(false);//we make the sign invisible
}
```

```
//Example 3: Waiting for file to export
IEnumerator ExportKeysToFile(string[] KeyList){
	mySign.SetActive(true);//we make the sign visible
	yield return PlayerPrefsPro.instance.WaitForExportFile("file.sav",KeyList,true);
	mySign.SetActive(false);//we make the sign invisible
}
```

```
//Example 4: Waiting for file to Import. NOTE THE ERROR CODE.
IEnumerator ImportFile(){
	mySign.SetActive(true);//we make the sign visible
	yield return PlayerPrefsPro.instance.WaitForImportFile("myFile.sav",true);
	mySign.SetActive(false);//we make the sign invisible
	//here we check what error code the import process yielded:
	int error = PlayerPrefsPro.requestStatus;
	if(error !=  0){ Debug.Log("Some error occurred");}
}
```

Remember that 0 is “no error”, 1 is “missing file” and anything beyond that is a “hacked or corrupted file error”.

How to reboot save data if it becomes corrupted

In case someone hacks your game or the save file simply became corrupted for whatever reason, the backup file should be used as the recovery (In case of local files.), but if you did not create backup files then the only solution is to wipe clean all the related data.

```
//Clearing up all traces of data:
void WipeData(){
	PlayerPrefsPro.DeleteAllProKeys();
	//then, delete all your local files and its backups if they exist.
	PlayerPrefsPro.DeleteSaveFile("filename1.sav");
	PlayerPrefsPro.DeleteSaveFile("filename2.sav");
	PlayerPrefsPro.DeleteSaveFile("filename3.sav");
}
```

If you have too many files to wipe, simply add all file names to an array and iterate with a for loop, just as the following:

```
string[] fileList = new string[]{"file1","file2","file3","file4","file5","file6"};
for(int x = 0; x < fileList.Length; x++){
	PlayerPrefsPro.DeleteSaveFile(fileList[x]);
}
```

How to save keys sharing the same name

If you need to use a key name more than once for any reason (We don't recommend it though), you can do it as long as the type is not the same; For example, `SetInt("key1",1);` and `SetFloat("key1",2.0f);` will create two keys of the same name but different type, which is valid. When you call a function such as `HasKey("key1");` it will return true if any of the two keys exist. To overcome this, some methods have an override to specify the type you are looking for.

```
//Deleting a duplicated key of specific type
void Start(){
	PlayerPrefsPro.SetInt("key1",1);
	PlayerPrefsPro.SetFloat("key1",2.0f);
	PlayerPrefsPro.DeleteKey("key1", KeyEx.kFloat);//delete only the float type key.
}
```

KeyEx is a public enumerator which PlayerPrefsPro uses to identify key types internally. Please note that duplicated key names are not fully supported and might throw exceptions in certain cases, **we strongly recommend never using a key name twice, only if you really need to for whatever purpose.**

How to make save files not possible to share

In case you want a save file to only be usable by the user who created it, you will need to include some form of authentication data among the stored keys, for example, in the case of *Steam* games, you could include the *userID* among the file keys and then after loading the file doing a comparison with the current *userID* generated by the *Steam API*; In case they differ is up to you to take the necessary actions.

This will work with both registry stored keys and local files, since the data block cannot be modified by the users. So, the general idea is to find a way to identify the current user and then include that information in the save files and do the required comparisons. Bear in mind that this is a delicate approach, make sure the user can always have access to his authentication information, for example, is not a good idea to use *hardware based identifiers*, since the same user could try to run the game on another computer, making it fail in the comparison with the *device ID*. The best practice is to always use the *login e-mail* or *userID* of the platform as the identifier.

How to add a custom data type

If you need to save other data type which is not supported by default, you can try to add your own, but **this process is not recommended for beginners, you could potentially break the asset.**

To make the process simpler, you don't really add support for a new data type, instead, you find a way to make a string out of your type and then store is using `SetString();`, but you need a way to revert that string back to your datatype.

First, you need to know if the target data type is fully serializable, then, you need to experiment to see if the Json class within unity supports said data type. In the case of custom created data types, chances are they won't be supported by the Json class (they will return empty). In such cases, you have to find a way to convert your type to string. For example, the DateTime class is not serializable by Json, however, the DateTime class has a "ToString(args)" method to convert date to string, and at the same time, it has a "ParseExact(args)" method to convert a string back to DateTime, so all we do is:

***1-** Get any type you intend to use, and find a way or create a method to convert to **string** and read from **string** to build back your type.*

*2- Once the string data of your class is generated, you store it as a **string** type with the method `PlayerPrefsPro.SetString("mykey",stringContent);` Then, You get the value back with `GetString("myKey");`*

For example:

```
MyClass myObject = new MyClass();

string content = MyClass.ToString();//assuming ToString() method exist in this class.

PlayerPrefsPro.SetString("myClass",content);

//done! now to get it back you do:

string data = PlayerPrefsPro.GetString("myClass");

MyClass myObject = MyClass.FromString(data);//assuming FromString() method exist in this class.
```

You could benefit from finding an asset to convert to and from any type to string, but bear in mind that only serializable fields can be stored, for example, the GameObject class only serializes the object uniqueID, but it does not serialize the components of it! so, you can't convert a GameObject to string just as easily as you would with a number; Each data type has its challenges to be converted to and from string, hence, the reason why is always easier to work with basic data types. Think about it, unless you are creating a very complex game where you need to store the state of the current level (like corpses limb positions) most probably you will be more than fine with storing only basic data types. Remember that this solution was not designed to handle large amounts of data, but to encrypt sensitive information.

PlayerPrefsPro's Public Methods List

List of Set<Type> methods

The following methods create a key name in the key-bank and assigns a value to it.

```
void SetInt(string keyname,int content);

void SetString(string keyname,string content);

void SetFloat(string keyname,float content);

void SetBool(string keyname,bool content);

void SetDateTime(string keyname,DateTime content);

void SetVector2(string keyname,Vector2 content);

void SetVector3(string keyname,Vector3 content);
```

```
void SetVector4(string keyname,Vector4 content);
void SetColor(string keyname,Color content);
void SetIntArray(string keyname,int[] content);
void SetStringArray(string keyname,string[] content);
void SetFloatArray(string keyname,float[] content);
void SetBoolArray(string keyname,bool[] content);
void SetDateTimeArray(string keyname,DateTime[] content);
void SetVector2Array(string keyname,Vector2[] content);
void SetVector3Array(string keyname,Vector3[] content);
void SetVector4Array(string keyname,Vector4[] content);
void SetColorArray(string keyname,Color[] content);
void SetIntArray2D(string keyname,int[,] content);
void SetStringArray2D(string keyname,string[,] content);
void SetFloatArray2D(string keyname,float[,] content);
void SetBoolArray2D(string keyname,bool[,] content);
```

List of Get<Type> methods

The following methods retrieve the value of a key name in the key-bank. If the key doesn't exist, the default value is returned. Each method has an override to set a custom default value.

```
int GetInt(string keyname);
int GetInt(string keyname,int defaultValue);
string GetString(string keyname);
string GetString(string keyname,string defaultValue);
float GetFloat(string keyname);
float GetFloat(string keyname,float defaultValue);
DateTime GetDateTime(string keyname);
DateTime GetDateTime(string keyname,DateTime defaultValue);
Vector2 GetVector2(string keyname);
Vector2 GetVector2(string keyname,Vector2 defaultValue);
Vector3 GetVector3(string keyname);
Vector3 GetVector3(string keyname,Vector3 defaultValue);
```

```
Vector4 GetVector4(string keyname);
Vector4 GetVector4(string keyname,Vector4 defaultValue);
Color GetColor(string keyname);
Color GetColor(string keyname,Color defaultValue);
int[] GetIntArray(string keyname);
int[] GetIntArray(string keyname,int[] defaultValue);
string[] GetStringArray(string keyname);
string[] GetStringArray(string keyname,string[] defaultValue);
float[] GetFloatArray(string keyname);
float[] GetFloatArray(string keyname,float[] defaultValue);
bool[] GetBoolArray(string keyname);
bool[] GetBoolArray(string keyname,bool[] defaultValue);
DateTime[] GetDateTimeArray(string keyname);
DateTime[] GetDateTimeArray(string keyname,string[] defaultValue);
Vector2[] GetVector2Array(string keyname);
Vector2[] GetVector2Array(string keyname,Vector2[] defaultValue);
Vector3[] GetVector3Array(string keyname);
Vector3[] GetVector3Array(string keyname,Vector3[] defaultValue);
Vector4[] GetVector4Array(string keyname);
Vector4[] GetVector4Array(string keyname,Vector4[] defaultValue);
Color[] GetColorArray(string keyname);
Color[] GetColorArray(string keyname,Color[] defaultValue);
int[,] GetIntArray2D(string keyname);
int[,] GetIntArray2D(string keyname,int[,] defaultValue);
string[,] GetStringArray2D(string keyname);
string[,] GetStringArray2D(string keyname,string[,] defaultValue);
float[,] GetFloatArray2D(string keyname);
float[,] GetFloatArray2D(string keyname,float[,] defaultValue);
bool[,] GetBoolArray2D(string keyname);
bool[,] GetBoolArray2D(string keyname,bool[,] defaultValue);
```

Other methods:

```
bool HasKey(string key);//If a key exists in PlayerPrefsPro with a given name, regardless of type.(There can exist multiple types of the same key name)

bool HasKey(string key,KeyEx kType);//If a key exists in PlayerPrefsPro with a given name and of a specific Type.

void DeleteKey(string key);//Deletes all the keys named “key” loaded at run-time. You need to call Save() afterwards (or at some point) to store the changes in preferences.

void DeleteKey(string key,KeyEx type);//Deletes all the keys named “key” of a specific type.

void DeleteAll();//Deletes all Unity's PlayerPrefs and PlayerPrefsPro keys from bank and registry.

void DeleteAllProKeys();//Deletes all the keys related to PlayerPrefsPro from key-bank and registry.

void CleanBank();//Similar to DeleteAll() but only deletes the keys loaded at run-time, it does not affect the registry values.

void CleanBankExcludingKeys(string[] keysToExclude);//Like CleanBank() but excludes a list of key names from being deleted.

void Save();//Saves information into the registry or PlayerPrefs location of the target platform.

void Save(string[] keysToExport);//Saves specific keys information into the registry. WARNING! All keys but these ones will be removed from the registry data block.

bool DoLocalFileExist(string fileName,bool checkForBackupToo);//Returns true if a file named “fileName” or its backup exist.

void ExportFile(string fileName,string[] keysToExport, bool makeBackup);//Exports all keys named keysToExport including all types.

void ExportFile(string fileName,string[] keysToExport, KeyEx[] types, bool makeBackup);//Exports all keys in the “keysToExport” array of a specific type (In case duplicated key names of different types exist) to a local path.

int ImportFile(string fileName,bool checkForBackup);//Imports a file from the local path if it exists. It will return an error code.

void DeleteSaveFile(string fileName);//Deletes a save file and its associated backup if it exists.

bool CloneSaveFile(string fileNameOrigin, string fileNameTarget);//Clones an existing save file into another named “fileNameTarget”.

bool RenameSaveFile(string sourceFile, string newName);//Renames a save file in the local path. Will return false if there was any error.

void ForceLoadPrefs();//Loads the registry information (PlayerPrefs) into memory.
```

Pseudo coroutines: (Allows a time frame to show static loading/saving messages.)

To use the below methods, use a "`yield return + MethodName();`" see the How to display a load dialogue section examples for more info.

```
IEnumerator WaitForForceLoadPrefs(); //Same as ForceLoadPrefs().

IEnumerator WaitForExportFile(string fileName,string[] keysToExport, bool makeBackup);//Same as ExportFile().

IEnumerator WaitForExportFile(string fileName,string[] keysToExport, KeyEx[] types, bool makeBackup);//Same as ExportFile().

IEnumerator WaitForSave(); //Same as Save().

IEnumerator WaitForSave(string[] keysToExport); //Same as Save().

IEnumerator WaitForImportFile(string fileName,bool checkForBackup);//Same as ImportFile().
```

Debugging:

```
void ShowKeyBankContent();//This will show the content of the key-bank in a pseudo-decrypted way, you will rarely need to use this method. It doesn't work very well with large amounts of data.

string[] GetKeyBankKeyNames();//This returns the names of all the keys in the key bank at the moment.
```

Contact / Support.

For any question about the asset please write an email to earrgames@gmail.com.

Thanks for using PlayerPrefsPro!

Sincerely, Emmanuel (earrgames).