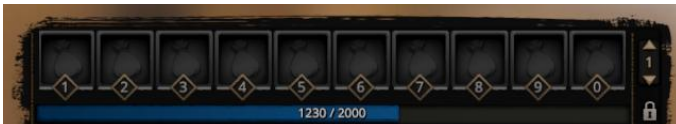


GETTING STARTED

1. First of all, make sure you have Unity UI Canvas in your scene. Drag the "**InventoryEngine**" prefab from the "[Assets/SoftKitty/InventoryEngine/Prefabs](#)" folder into your scene. It's a good practice to place it in your initial scene, as it won't be destroyed when switching between scenes. Alternatively, you can add it to every scene for convenience—no need to worry about duplication. The script will prevent multiple instances from existing simultaneously during gameplay.
2. Set up the "**ItemManager**" component on the "**InventoryEngine**" prefab according to your game design. For detailed guidance, please refer to the "[Item Manager Setting](#)" chapter.

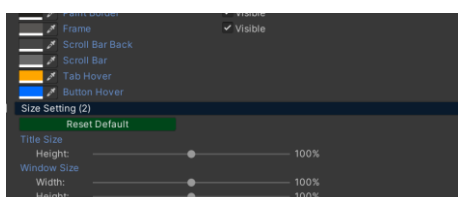


3. Drag the "**PlayerInventory**" prefab from the "[Assets/SoftKitty/InventoryEngine/Prefabs](#)" folder and place it as a child of your player game object. This data script manages the items and equipment your player carries.
4. (Optional) Drag the "**ActionBar**" prefab from the "[Assets/SoftKitty/InventoryEngine/Prefabs](#)" folder and place it under your UI Canvas transform. This prefab provides 10 shortcut slots with customizable key bindings for items/skills on each page, and you can set up as many pages as needed for players to switch between. The action bar also includes a progress bar at the bottom, which can be configured as a player XP bar to display experience and level, or as a player health bar. Alternatively, you can hide this bar using the "**UiStyle**" component.



5. You can now start testing some basic functions. In your own script, use the following calls:
 - a) `ItemManager.PlayerInventoryHolder.OpenWindow();` to open the player's inventory.
 - b) `ItemManager.PlayerEquipmentHolder.OpenWindow();` to open the player's equipment UI.

- c) [ItemManager.PlayerInventoryHolder.OpenForgeWindow\(bool enableCrafting=true, bool enableEnhancing=true, bool enableEnchanting=true, bool enableSocketing=true\)](#); to open the crafting/enhancing/enchanting/socketing UI.
 - d) [ItemManager.PlayerInventoryHolder.OpenWindowByName\("Skills", "Skills"\)](#); to open the skill list.
6. Now, let's add some more interesting functions. Drag the "**MerchantNPC**" prefab from the "[Assets/SoftKitty/InventoryEngine/Prefabs](#)" folder and place it under one of your NPCs. In your NPC control script, reference the "**InventoryHolder**" component on this prefab (let's name the variable "**Merchant**"). When the player interacts with this NPC, call [Merchant.OpenWindow\(\)](#); to open the store UI, allowing the player to trade with this NPC.
7. By now, you've probably noticed that the "**InventoryHolder**" component acts as a container for items—you're correct! You can add this component to any object in your game. When the player interacts with it, call [OpenWindow\(\)](#) to open the corresponding interface. The specific interface that appears depends on the "**Type**" setting of the "**InventoryHolder**". In steps 5 and 6, we tested most types of **InventoryHolder**. Now, let's test the last one. Drag the "**Crate**" prefab from the "[Assets/SoftKitty/InventoryEngine/Prefabs](#)" folder and place it under one of the crate or chest models in your game. When the player interacts with it, call [OpenWindow\(\)](#) to display the UI.
8. If your game includes loot packs that players can obtain from defeating monsters or completing quests, you can use the "**LootPack**" prefab found in the "[Assets/SoftKitty/InventoryEngine/Prefabs](#)" folder. When the loot pack drops, instantiate this prefab at the desired position. When the player interacts with the loot pack, call [GetComponent<LootPack>\(\).OpenPack\(\)](#) to open the interface and allow the player to collect their rewards.
9. Check [MainMenu.cs](#) in "[Assets/SoftKitty/InventoryEngine/ExampleScene/Scripts](#)" to learn how to setup callback for player uses, equips, or unequips an item. You can also learn how to save/load data, assign crafting tasks to NPCs, and manage NPC equipment and inventory.
10. Now that we've walked through all the pre-made interfaces, you can easily create your own by customizing the existing prefabs in the "[Assets/SoftKitty/InventoryEngine/Resources/InventoryEngine/UiWindows](#)" folder. Adjust the settings in the **UiStyle** component on each prefab to suit your needs. Alternatively, you can write your own script by inheriting from [ItemContainer.cs](#), [ContainerBase.cs](#), or [HiddenContainer.cs](#).



COMMON USE CASES AND INTEGRATION SCENARIOS

HOW TO ACCESS THE PLAYER'S INVENTORY AND EQUIPMENT COMPONENTS

The **InventoryHolder** component is responsible for managing all items for each unit.

Here's how you can access it in different scenarios:

1. Accessing the Player's Inventory and Equipment

Player's Inventory: Use **ItemManager.PlayerInventoryHolder** to get the player's inventory component.

Player's Equipment: Use **ItemManager.PlayerEquipmentHolder** to get the player's equipment component.

2. Accessing **InventoryHolder** for Other Units (e.g., **NPCs**, **Crates**, **Merchants**)

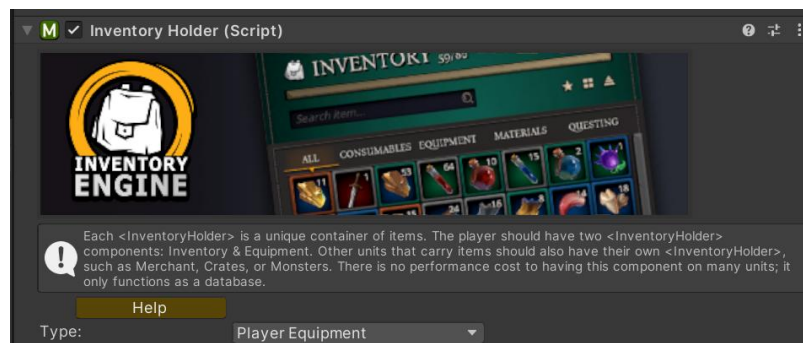
To retrieve the **InventoryHolder** component for other units, such as NPCs, crates, or merchants, use the following API call:

InventoryHolder.GetInventoryHolderByType(GameObject _gameObject, HolderType _type)

Parameters:

_gameObject: The target GameObject or its child transform that contains the **InventoryHolder** component.

_type: The type of **InventoryHolder** you're looking for.



Available types include:

Crate: Represents a crate or container.

Merchant: Represents a merchant or store inventory.

PlayerInventory: Represents the player's inventory.

PlayerEquipment: Represents the player's equipped items.

NpcInventory: Represents an NPC's inventory.

NpcEquipment: Represents an NPC's equipped items.

HOW TO RETRIEVE PLAYER'S ATTRIBUTES WITH EQUIPPED ITEMS

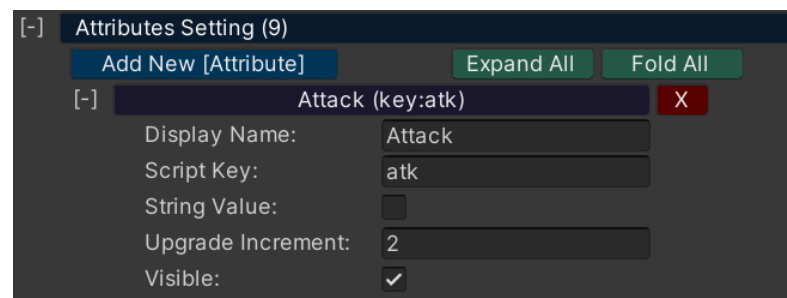
To get the total stats of the player, including bonuses from all equipped items, use the following method:

ItemManager.PlayerEquipmentHolder.GetAttributeValue(string _attributeKey)

Parameter:

_attributeKey: The string key of the attribute you want to retrieve. You can define these keys in the **InventoryEngine** prefab.

For example, in the screenshot below, the *Attack* attribute has a key of **"atk"**. You can use this key to retrieve the total *Attack* value contributed by all equipped items.



Example Usage:

To retrieve the total Attack value:

float totalAttack = ItemManager.PlayerEquipmentHolder.GetAttributeValue("atk");

This will return the combined value of the *Attack* attribute from all items the player has equipped.

For NPCs

The process is the same for NPCs. Replace ***ItemManager.PlayerEquipmentHolder*** with the NPC's corresponding ***InventoryHolder*** component:

float npcAttack = npcInventoryHolder.GetAttributeValue("atk");

With this method, you can calculate any attribute value for both players and NPCs based on their equipped items.

HOW TO GET, ADD, REMOVE, OR USE ITEMS FROM AN INVENTORYHOLDER

➤ **Get the Number of Items:**

Retrieve the number of a specific item in the inventory:

int itemCount = <InventoryHolder>().GetItemNumber(int _uid);

_uid: The **unique ID** of the item.

➤ **Add Items:**

Add items to an inventory:

<InventoryHolder>().AddItem(Item _item, int _number);

_item: The item you want to add.

_number: The quantity of the item to add.

If you need to create a new item from the database, use:

```
Item myNewItem = new Item(uid);
```

You can modify the item you created as needed (e.g., upgrade level, enchantments), and then add it to the inventory:

```
<InventoryHolder>().AddItem(newItem, 1);
```

➤ **Remove Items:**

Remove a specific number of items from the inventory:

```
<InventoryHolder>().RemoveItem(int _uid, int _number, int _index = -1);
```

_uid: The **unique ID** of the item to remove.

_number: The quantity of the item to remove. If the number exceeds the quantity available, it will be capped at the actual number.

_index: (Optional) If specified, removes the item from the slot at this index in the inventory interface. Use -1 to ignore the index.

➤ **Use an Item:**

Use an item from the inventory:

```
<InventoryHolder>().UseItem(int _uid, int _number, int _index = -1);
```

_uid: The **unique ID** of the item to use.

_number: The quantity of the item to use. If the number exceeds the available quantity, it will be capped at the actual number.

_index: (Optional) If specified, uses the item from the slot at this index in the inventory interface. Use -1 to ignore the index.

After using an item, the *ItemUseCallback* will be triggered. Refer to the **Callbacks** section for more details.

Managing Currencies:

➤ **Retrieve the current amount of a specific currency:**

```
int currencyAmount = <InventoryHolder>().GetCurrency(int _type);
```

_type: The **ID** of the currency.

➤ **Add or Remove Currency:**

Modify the currency amount (positive to add, negative to remove):

```
<InventoryHolder>().AddCurrency(int _type, int _add);
```

_type: The **ID** of the currency.

_add: The amount to change (positive for adding, negative for removing).

➤ **Set Currency to a Fixed Value:**

Directly set the currency to a specific value:

```
<InventoryHolder>().SetCurrency(int _type, int _value);
```

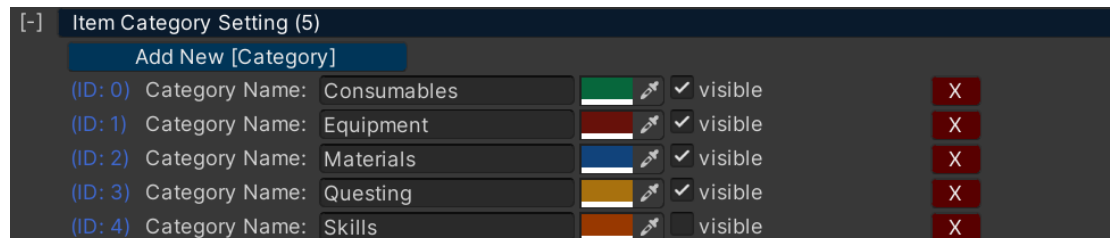
_type: The **ID** of the currency.

_value: The value to set for the currency.

ITEM MANAGER SETTING

The **ItemManager** component on the “**InventoryEngine**” prefab contains all the settings for the entire system. It’s organized into several categories, which we’ll introduce one by one:

ITEM CATEGORY SETTING

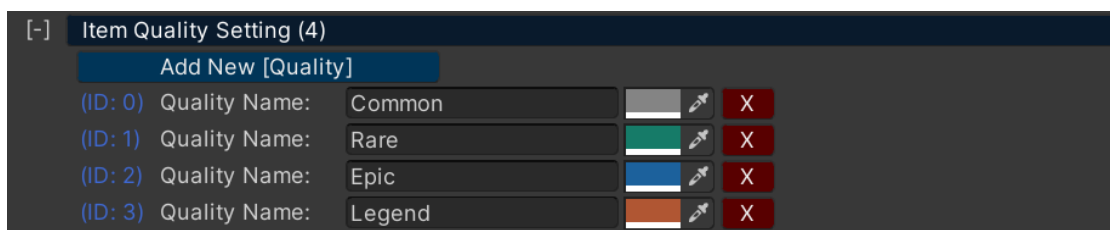


[-] Item Category Setting (5)				
Add New [Category]				
(ID: 0)	Category Name:	Consumables	<input checked="" type="checkbox"/> visible	X
(ID: 1)	Category Name:	Equipment	<input checked="" type="checkbox"/> visible	X
(ID: 2)	Category Name:	Materials	<input checked="" type="checkbox"/> visible	X
(ID: 3)	Category Name:	Questing	<input checked="" type="checkbox"/> visible	X
(ID: 4)	Category Name:	Skills	<input type="checkbox"/> visible	X

You can create as many categories for your items as you like (though it’s recommended to keep it to fewer than 5 visible categories due to UI space limitations).

- **Category Name:** This name will be displayed as tab cards at the top of the inventory UI.
- **Category Color:** The background color for items in this category.
- **Visible:** Determines whether the category is visible to the player or hidden. For example, if you want skills to be managed or assigned to the **ActionBar** but not displayed in the player’s inventory UI, you can set this category to NOT “**visible**.” Standard inventory UIs—like crate, store, loot, and inventory screens—won’t show items in this category. However, interfaces using scripts that inherit from **HiddenContainer.cs** will only display items in categories set to NOT “**visible**,” such as the “Skills” prefab in the ["Assets/SoftKitty/InventoryEngine/Resources/InventoryEngine/UiWindows"](#) folder.

ITEM QUALITY SETTING

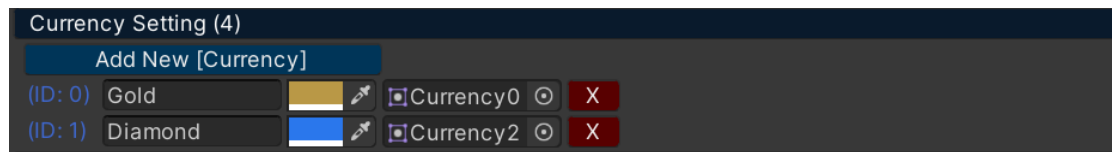


[-] Item Quality Setting (4)				
Add New [Quality]				
(ID: 0)	Quality Name:	Common		X
(ID: 1)	Quality Name:	Rare		X
(ID: 2)	Quality Name:	Epic		X
(ID: 3)	Quality Name:	Legend		X

You can create as many quality levels as needed. Each quality will have an index displayed as blue text at the front. When your script accesses the quality attribute of an item, the value will correspond to the index of the quality levels you set here.

- **Quality Name:** This name will be shown in the detailed information panel of items.
- **Quality Color:** The frame color for items of this quality.

CURRENCY SETTING



You can create as many currency types as needed. Each currency will have an index displayed as blue text at the front. When your script accesses the currency value of an **InventoryHolder** component, you'll need to call `<InventoryHolder>().GetCurrency(int _type)` using the index number as the parameter.

- **Currency Name:** This is the display name of the currency.
- **Currency Color:** The currency value text will be shown in this color.
- **Currency Sprite:** The sprite will be displayed as the icon for this currency.
- **Exchange Rate:** The Exchange Rate system allows you to define conversion rates between different types of currencies in your game. Here's how it works:
Defining Exchange Rates: You only need to set the exchange rates for directly related currencies.

For example:

1 Gold = 10 Silver

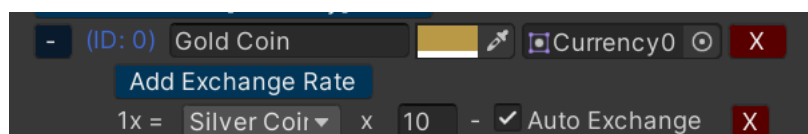
1 Silver = 10 Copper

You don't need to set 1 Gold = 100 Copper, as the system will automatically calculate the exchange rates for indirect conversions.

Auto Exchange: When the Auto Exchange option is enabled, any currency the player acquires will automatically convert into higher-level currency when possible.

For example:

If the player has 15 Silver, the system will automatically convert it into 1 Gold and 5 Silver.



ATTRIBUTES SETTING

Attributes Setting (10)

Add New [Attribute] Expand All Fold All

[-] Attack (key:atk) X

Display Name: Attack

Script Key: atk

String Value:

Upgrade Increment: 2

Visible: ☒

Attributes represent the stats of your characters, such as attack power, defense, running speed, maximum health points, etc. You can create as many attributes as needed, with each attribute being an *integer*, *float*, or *string* value.

- **Display Name:** The name displayed in the interface for this attribute.
- **Script Key:** A short key used to access the value of this attribute in your script. For example, if the script key for the attack power attribute is “atk” and you want to find out the player’s total attack power, you can call `<InventoryHolder>().GetAttributeValue(“atk”).`
- **String Value:** Check this if the attribute's value is a *string*. For example, you might have an attribute called “Creator,” and when a player crafts a weapon, you can set this attribute to the player’s name by calling `Item.UpdateAttribute(“Creator”, “Player Name”).`
- **Upgrade Increment:** Items can be upgraded through the “[Enhancement](#)” system, and each upgrade level increases the item's attributes by the amount specified here.
- **Visible:** Determines whether the attribute is visible in the interface or hidden for backend use.

CRAFTING SETTING

Crafting Setting (12)

☒ Enable Crafting

Material Category: Materials

Blueprint Tag: Blueprint

Crafting Time: 0.5 Sec

Players can **CRAFT** items using other items as materials. They must also have the corresponding blueprint in their inventory as hidden items. For detailed setup instructions, refer to the [Craft Materials](#) section.

- **Enable Crafting:** If you don’t plan to include a crafting system in your game, uncheck this to disable the feature.
- **Material Category:** Items used as crafting materials must belong to this category.
- **Blueprint Tag:** Items used as blueprints must have this tag, refer to the [Tags](#) setting section.
- **Crafting Time:** When crafting, a progress bar will fill from 1-100%. This setting determines how long the process will take until the item is created.

ENCHANTMENT SETTING

Enchantment Setting (12)

☒ Enable Enchanting

Enchanting Category: Equipment

Random number of enchantments player can get: (1~3)

Enchanting Success Rate: 30

Required Currency: Diamond 100

Required Item as Material: Enchantment x 1

Enchanting Time: 0.5 Sec

Enchantments List:

Add New [Enchantment] Expand All Fold All

[-] Knight (UID:0) X

Display Name: Knight

Attributes: (2) +

> hp : 5 X

> def : 3 X

Players can **ENCHANT** items to grant them additional attributes, which can be either positive or negative. Enchanting requires a specific type of currency and a particular item (such as a Magic Scroll).

- **Enable Enchanting:** Uncheck this if you don't plan to include an enchanting system in your game to disable the feature.
- **Enchanting Category:** Only items in this category can be enchanted.
- **Enchanting Success Rate:** The percentage chance of success when a player enchants an item.
- **Required Currency:** Set the currency type and the cost required to enchant an item.
- **Required Item as Material:** Select the material item and quantity needed for enchanting.
- **Enchanting Time:** When enchanting, a progress bar will fill from 1-100%. This setting determines how long the process will take to complete the enchantment.
- **Enchantment List:** Create a list of enchantments, each of which can have multiple attributes.

ENHANCEMENT SETTING

Enhancement Setting

☒ Enable Enhancing

Enhancing Category: Equipment

Maxium Enhancing Level: 15

Enhancing Success Curve:

☒ Destroy item when fail (Level> 3)

Required Currency: Diamond 50

Required Item as Material:

Power Stone x 2

Ancient Rune x 1

Enhancing Time: 0.5 Sec

Player can upgrade an item with the **ENHANCEMENT** system. The enhancing will cost one type of currency and two specific items. Each upgrade level will make the attributes of this items increased by the amount set by [Upgrade Increment](#) property in attributes setting.

- **Enable Enhancing:** If you plan to not include Enhancing system in your game, uncheck this to disable this feature.
- **Enhancing Category:** Only items belong to this category can be enhanced.
- **Maximum Enhancing Level:** The maximum level an item can be enhanced.
- **Enhancing Success Curve:** Success rate in percentage when player enhance an item. The rate will change by the curve from level.0~ **maximum enhancing Level**.
- **Destroy item when fail (Level>x):** When check this option, the item will be destroyed when fail by upgrading to level x+ .
- **Enable Glowing Effect:** Check this option to make the icon glow based on the item's enhancement level.
- **Glow Intensity Curve:** The icon's glow will follow this curve, ranging from level 0 to the **maximum enhancement level**.
- **Required Currency:** Set the currency type and cost when player enhance an item.
- **Required item as material:** Select the material item and number of cost when player enhance an item.
- **Enhancing Time:** When enhancing, there will be a progress bar move from 1-100%, when it's done, the item will be enhanced, this is how long the process will take.

SOCKETING SETTING

[-] Socketing Setting

☒ Enable Socketing

Socketed Items (Receiver) Category:

Equipment

Socketing Items (Plug-in) Category:

Materials

Socketing Items (Plug-in) Tag:

Socketing

☒ Random Skocketing Slots Number For New Item.

Minimal Socketing Slots Number: 1

Maxmium Socketing Slots Number: 4

☒ Lock Skocketing Slots Number By Default.

Random Chance to Lock Socketing Slots: 25

Unlock Socketing Slot Cost:

Gold Coin

 50

☒ Allow Remove Skocketing Items.

Remove Socketing Item Cost:

Gold Coin

 100

☐ Destroy Socketed Item When Remove

The **SOCKETING** system allows players to insert (socket) items of specified categories and tags into another item to boost its attributes. Follow the configuration options below to tailor the system to your game's needs.

- **Enable Socketing:** Toggle this option to enable or disable the socketing system in your game. If unchecked, the feature will not be available.
- **Socketed Items (Receiver) Category:** Defines the category of items that can have socketing slots (e.g., *weapons*, *armor*). Only items in this category can receive socketing items.
- **Socketing Items (Plug-in) Category:** Specifies the category of items that can be inserted (socketed) into socketed items (e.g., *gems*, *runes*).
- **Socketing Items (Plug-in) Tag:** Filters socketing items further by requiring them to have a specific tag (e.g., "*FireGem*" or "*MagicRune*"). Leave blank if no tag filtering is needed.
- **Random Socketing Slots Number For New Item:** Enable this to randomize the number of socketing slots when a new item is created using *new Item(int _uid)*.
- **Minimal Socketing Slots Number:** Sets the minimum number of socketing slots for newly created items when randomization is enabled.
- **Maximum Socketing Slots Number:** Sets the maximum number of socketing slots for newly created items when randomization is enabled.
- **Lock Socketing Slots Number By Default:** Enable this to lock socketing slots by default when a new item is created.
- **Random Chance to Lock Socketing Slots:** Specify the probability (as a percentage) of socketing slots being locked by default for newly created items.
- **Unlock Socketing Slot Cost:** The cost for the player to unlock a locked socketing slot. This cost can represent in-game currency or another resource.
- **Allow Remove Socketing Items:** Enable this to allow players to remove socketed items from their slots.
- **Remove Socketing Item Cost:** The cost for the player to remove a socketed item.
- **Destroy Socketed Item When Remove:** Enable this option to destroy the socketed item when it is removed. Disable it if you want the item to return to the player's inventory instead.

Example Use Cases

- A player can socket a "*Fire Gem*" (tagged as "*FireGem*" in the plug-in category) into a "*Sword*" (receiver category: weapon) to boost its attack power.
- The system can randomly generate a sword with *2-5 socketing slots*, some of which may be locked. The player can pay an in-game currency (e.g., *gold*) to unlock these slots and insert additional gems.
- Players can also pay to remove socketed gems, with the option to either destroy or return the gem to the inventory.

ITEM SETTING

[-] **Great Axe (UID:33)** X

Display Name: Great Axe

Description: Great sword can destroy anything.

Category: Equipment

Quality: Epic

Icon: Axe_4

Maximum Stack: 1

Price: 1200 Gold

Weight: 3

Drop Rates: 3 %

Useable: ☒

Consumable: ☐

Tradeable: ☒

Deletable: ☒

Visible: ☒

Attributes: (2) +

- > atk : 40 X
- > crit : 15 X

Use Actions: (2) +

- > attck_axe X
- > equip X

Tags: (2) +

- > MainHand X
- > TwoHanded X

Craft Materials: (3) +

- > Iron Ore x 2 X
- > Silver Ore x 5 X
- > Bear's Tooth x 3 X

- **Display Name:** The name of the item.
- **Description:** The text displayed in the detailed information panel. Keep it as short as possible.
- **Category:** The category to which this item belongs.
- **Quality:** The quality level of this item.
- **Icon:** Select the item's icon texture. The icon should be a transparent texture. A variety of icons are available in the "[Assets/SoftKitty/InventoryEngine/Textures](#)" folder.
- **Maximum Stack:** The maximum number of items that can stack in a single slot.
- **Price:** The trading price and its associated currency type.
- **Weight:** The weight of this item. The inventory interface includes a weight bar. You can create custom logic to slow down the player's movement or prevent them from moving when they are over-encumbered. Use `<InventoryHolder>().GetWeight()` to retrieve the current weight value of a character.



- **Drop Rate:** When this item is in a loot pack's item pool, the drop rate determines the percentage chance of it dropping.
- **Usable:** Toggle whether this item can be used via right-click or a key press on an **action slot**.
- **Consumable:** Toggle whether this item will be consumed upon use.
- **Tradable:** Toggle whether this item can be traded between the player and merchants.
- **Deletable:** Toggle whether this item can be deleted by the player.
- **Visible:** If unchecked, the item is treated as a Hidden Item, which is useful for special items like **"Skills."**
- **Attributes:** The attributes of this item. Access the attributes using [<InventoryHolder>\(\).GetAttributeValue\(key\).](#)
- **Use Actions:** When this item is used, the commands in this list are sent to all registered callbacks. Register callbacks using [ItemManager.PlayerInventoryHolder.RegisterItemUseCallback\(\).](#)
- **Tags:** Tags are useful for providing additional definitions to items. For example, for equipment, tags can define the slot it belongs to (Head, Torso, Legs, etc.), whether a weapon is two-handed or one-handed, or if an item is a crafting **blueprint**.
- **Craft Materials:** Specify the items required as materials when [crafting](#) this item.
- **Socketing Tags:** This item will not accept socketing items with any tag in this list.

USEFUL MODULES

ItemDragManager:

This module is used to drag items around.

- [ItemDragManager.PlayDeleteAnimation\(InventoryItem _source, Vector3 _pos, int _overrideNum = 0\)](#)
Plays the delete animation for an item icon, causing the icon to fall to the bottom of the screen.
- [ItemDragManager.StartDragging\(ItemIcon _source, RectTransform _rect, int _overrideNum = 0\)](#)
Starts dragging an item. After the drag-and-drop action, the [_source](#) will be called by **EndDrag()**.

CraftingProgress:

This module is used to monitor if an **"InventoryHolder"** is executing a crafting job and to display a progress bar showing the crafting progress. To use the module, simply drag the

“[Prefabs/Ui/CraftingProgress.prefab](#)” to your UI canvas and assign the “**InventoryHolder**” component of your NPC or crafting table to the “**Holder**” slot in the “**CraftingProgress**” component.



HoverInformation:

This module shows detailed information about an item.

- [HoverInformation.SetCompareHolder\(InventoryHolder holder\)](#)
Sets an **InventoryHolder** so that the hovering item's attributes can be compared with the equipment of this **InventoryHolder**.
- [HoverInformation.ShowHoverInfo\(ItemIcon source, Item item, int num, RectTransform anchor, float priceMultiplier\)](#)
Shows detailed information for the provided item icon. The information panel will align with the _anchor RectTransform.

HintManager:

This module displays short hint text in a small floating window, useful when the mouse hovers over buttons or icons. To use this module, simply add the **<HintText>** component to any UI element and fill in the hint text.

DynamicMsg:

This module displays popup messages and large flashy icons of items when players acquire new items.

- [DynamicMsg.PopMsg\(string text\)](#)
Displays a message with the provided text.
- [DynamicMsg.PopItem\(Item item, int number = 1\)](#)
Shows a flashy big icon of an item.

NumberInput:

This module prompts the player to input numbers. For example, when a player tries to split an item stack, this module can be used to ask how many items they want to split.

- [NumberInput.GetNumber\(int startValue, int maxiumValue, RectTransform rect, NumberCallback callback\)](#)
Displays a panel prompting the player to input a number. After the player confirms, the _callback is called with the resulting number.

SoundManager:

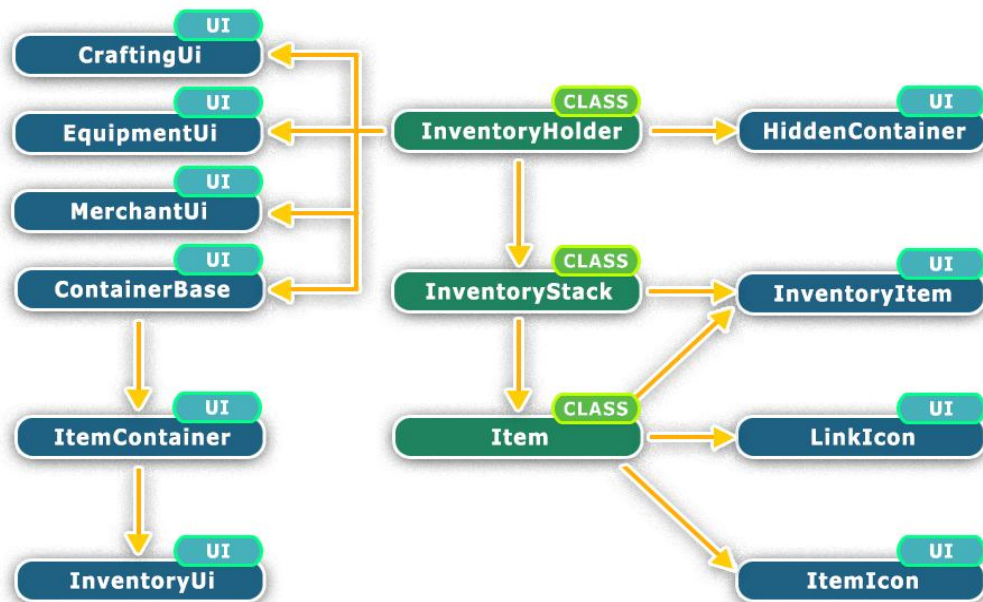
This module plays a sound from the Resources folder. Place your sound clips in “*Resources/Sound/*.”

- *SoundManager.Play2D(string soundName, float volume = 1F)*
Plays a 2D sound with the provided audio clip name.
- *SoundManager.Play3D(string soundName, Vector3 position, float volume = 1F)*
Plays a 3D sound at the specified position using the provided audio clip name.

StatsUi.cs:

This module displays the total attributes of an *InventoryHolder*. Attach this script to a UI panel, assign the necessary UI elements, and call *Init(InventoryHolder equipHolder, Attribute[] _baseAttributes)* to display the stats. *_baseAttributes* represents the player's attributes without any equipment.

SCRIPTING REFERENCE



Quick Links:

[InventoryHolder](#) | [InventoryStack](#) | [Item](#) | [InventoryIcon](#) | [InventoryItem](#) | [LinkIcon](#)

InventoryHolder.cs

- `public List<InventoryStack> Stacks`
The list of InventoryStack objects. All normal items are stored here.
- `public List<InventoryStack> HiddenStacks`
The list of InventoryStack objects where all hidden items are stored.
- `public CurrencySet Currency`
The class to managing the currencies.
- `public List<int> CurrencyValue`
The list of the currency values.
- `public int InventorySize = 10;`
The size of the Stacks. Items will be rejected if the inventory is full.
- `public float MaxiumCarryWeight = 1000F;`
The maximum carry weight for this *InventoryHolder*.
- `public float SellPriceMultiplier = 1F;`
This multiplier is applied when selling items. For example, a merchant NPC with a SellPriceMultiplier of 1.2 will sell an item priced at 1000 for 1200.
- `public float BuyPriceMultiplier = 1F;`
This multiplier is applied when buying items. For example, a merchant NPC with a BuyPriceMultiplier of 0.8 will buy an item priced at 1000 for 800.
- `public static InventoryHolder GetInventoryHolderByType(GameObject _gameObject, HolderType _type)`
Works similar with GetComponent, but only return the *InventoryHolder* Component of specified *HolderType* of the providing GameObject.
- `public UIWindow OpenWindow()`
Opens the appropriate interface based on the type of this *InventoryHolder*.
- `public UIWindow OpenForgeWindow(bool _enableCrafting=true, bool _enableEnhancing = true, bool _enableEnchanting=true, bool _enableSocketing=true)`
Opens the crafting/enhancing/enchanting/socketing window for this *InventoryHolder*, using the items in Stacks as materials. You can enable/disable different modules by the args.
- `public UIWindow OpenWindowByName(string _prefabName, string _displayName)`
Opens a window prefab inheriting from *UIWindow.cs* and sets its title.
- `public float GetAttributeValue(string _attributeKey)`
Retrieves the total value of an attribute by its [ScriptKey](#) from all equipped items.
- `public float GetAttributeValueByTag(string _attributeKey, List<string> _tags)`
Retrieves the total value of an attribute by its [ScriptKey](#), counting only equipment with matching [tags](#).
- `public int GetItemNumber(int _uid)`
Retrieves the number of items with a specific UID.

- `public int GetItemNumberWithHighestUpgradeLevel(int _uid)`
Retrieves the number of items with a specific UID, but only counts the items with the highest upgrade level if there are multiple items with the same UID.
- `public int GetHighestUpgradeLevel(int _uid)`
Retrieves the highest upgrade level of the item with a specific UID.
- `public int GetAvailableSpace(int _uid, int _maximumNumber = 999)`
Returns how many more items with the specified UID can be stacked in this **InventoryHolder**. Set a cap number to improve performance.
- `public int GetCurrency(int _type)`
Retrieves the currency value by its index number.
- `public void AddCurrency(int _type, int _add)`
Adds to the currency value by its index number. The _add value can be negative.
- `public void SetCurrency(int _type, int _value)`
Overrides the currency value by its index number.
- `public string GetSaveDataJsonString()`
Returns a JSON string of all the item data in this **InventoryHolder**.
- `public string Save()`
Save the data of this **InventoryHolder** to the **SavePath** (public variable) .
- `public void Load()`
Loads item data into this InventoryHolder with the "Save Data Path" set in the inspector.
- `public void Load(string _data)`
Loads item data into this InventoryHolder from a JSON string.
- `public void Load(string _path)`
Loads item data into this InventoryHolder by a full path of the save file.
- `public void ItemChanged()`
Whenever you manually change the item data of an **InventoryHolder**, be sure to call **ItemChanged()** to inform other scripts to update linked data.
- `public InventoryHolder GetSnapshot()`
Captures a snapshot of the current data state of this **InventoryHolder**. You can revert to this snapshot later if needed.
- `public void RevertSnapshot(InventoryHolder _snapshot)`
Reverts all item data to the provided snapshot.
- `public void RegisterItemUseCallback(ItemUseCallback _callback)`
Registers a callback that is triggered when an item from this **InventoryHolder** is used. Remember to call **UnRegisterItemUseCallback()** when the game object of your script is destroyed.
- `public void UnRegisterItemUseCallback(ItemUseCallback _callback)`
Unregisters the callback. It's best to call this in **OnDestroy()** when the callback receiver is destroyed.

- `public void ClearAllItemUseCallback()`
Clears all registered callbacks for item use.
- `public void RegisterItemChangeCallback(ItemChangeCallback _callback)`
Registers a callback that is triggered when items in this **InventoryHolder** are changed.
Remember to call **UnRegisterItemChangeCallback()** when the game object of your script is destroyed.
- `public void UnRegisterItemChangeCallback (ItemChangeCallback _callback)`
Unregisters the callback. It's best to call this in **OnDestroy()** when the callback receiver is destroyed.
- `public void ClearAllItemChangeCallback ()`
Clears all registered callbacks for item changes.
- `public void RegisterItemDropCallback(ItemDropCallback _callback)`
Registers a callback that is triggered when items in this **InventoryHolder** has been dropped by dragging out of the window.
Remember to call **UnRegisterItemDropCallback ()** when the game object of your script is destroyed.
- `public void UnRegisterItemDropCallback (ItemDropCallback _callback)`
Unregisters the callback. It's best to call this in **OnDestroy()** when the callback receiver is destroyed.
- `public void ClearAllItemDropCallback ()`
Clears all registered callbacks for item drop.
- `public void ClearInventoryItems()`
Clears all normal items in this **InventoryHolder**.
- `public void ClearHiddenItems ()`
Clears all hidden items in this **InventoryHolder**.
- `public float GetWeight ()`
Retrieves the total weight of all items in this **InventoryHolder** (hidden items are not counted).
- `public InventoryStack AddItem(Item _item, int _number = 1)`
Adds a specified number of items. Returns any items that could not be received (if any).
- `public InventoryStack AddHiddenItem(Item _item, int _number = 1)`
Adds a specified number of hidden items. Returns any items that could not be received (if any).
- `public void MoveItem(int sourceIndex, int _targetIndex)`
Moves an item from one index to another.
- `public InventoryStack Split(int sourceIndex, int _number)`
Splits the stack and returns the split stack.
- `public void DeleteItem(int _uid, int _upgradeLevel, List<int> _enchancements, List<int> _sockets)`
Deletes an item with a specific UID, upgrade level, sockets and enchantments.

- `public int GetItemIndex (int _uid, int _upgradeLevel, List<int> _enchantments, List<int> _sockets)`
Returns the slot index of an item with a specific UID, upgrade level, socketing and enchantments.
- `public Item GetItemByTag(List<string> _tags, bool _allMatch=true)`
Returns the item matches the specific tag list, set _allMatch to false if you require the item matches any tag in the list, set _allMatch to true if require the item matches all tags in the list.
- `public Item FindItem(int _uid, int _upgradeLevel, List<int> _enchantments, List<int> _sockets)`
Returns an item with a specific UID, upgrade level, sockets and enchantments.
- `public int RemoveItem(int _uid, int _number, int _index=-1)`
Removes a specified number of items with a specific UID and index. Set _index to -1 if you don't know the index. Returns the total number of items removed.
- `public void UseItem(int _uid, int _number, int _index = -1)`
Uses a specified number of items with a specific UID and index. Set _index to -1 if you don't know the index.
- `public static void EquipItem(InventoryHolder _inventoryHolder, InventoryHolder _equipmentHolder, int _itemId, int _inventorySlotIndex)`
Equip an item from _inventoryHolder to _equipmentHolder by specific item uid and slot index of inventory.
- `public void OpenWindow ()`
Opens the appropriate interface based on the type of this **InventoryHolder**.
- `public UiWindow OpenForgeWindow(bool _enableCrafting=true, bool _enableEnhancing = true, bool _enableEnchanting=true, bool _enableSocketing=true, float _craftingTimeMultiplier=1F, string _name="Forge")`
Opens the crafting window for this **InventoryHolder**, using the items in Stacks as materials.
- `public UiWindow OpenWindowByName(string _prefabName, string _displayName)`
Opens a window prefab inheriting from **UiWindow.cs** and sets its title.
- `public void CloseWindow ()`
Close the opened window associated with this **InventoryHolder**.

class InventoryStack

- `public InventoryStack(int _uid, int _number)`
Creates a new **InventoryStack** with the specified item **UID** and quantity.
- `public InventoryStack(Item _item, int _number)`
Creates a new **InventoryStack** with the specified **Item** and quantity.

- `public InventoryStack ()`
Creates an empty *InventoryStack*.
- `public InventoryStack Copy ()`
Returns a copy of this *InventoryStack*.
- `public InventoryStack Merge (InventoryStack _source)`
Merges another *InventoryStack* (`_source`) with this one if they contain the same item; otherwise, it swaps them. Returns the resulting stack. You should assign the result to the source.
- `public InventoryStack Split (int _number)`
Splits this *InventoryStack*, returning the split stack.
- `public void Set (InventoryStack _data)`
Overrides this *InventoryStack* with the provided data.
- `public void Delete ()`
Deletes the items in this *InventoryStack*.
- `public void OverrideNumber (int _number)`
Overrides the quantity of items in this *InventoryStack*.
- `public void AddNumber (int _number)`
Adds the specified number of items to this *InventoryStack*; the `_number` can be negative.
- `public bool Consume (int _number)`
Consumes the specified number of items from this *InventoryStack* and returns `true` if successful.
- `public float GetWeight ()`
Returns the total weight of the items in this *InventoryStack*.
- `public bool isTagMatchText (string _tag)`
Returns whether the items in this *InventoryStack* have the specified `tag`.
- `public bool isTagsMatchList (List<string> _tags, bool _allMatch = true)`
Returns whether the tags of this item match the specified `tag` list.
- `public bool isTagContainText (string _text, bool _caseSensitive = true)`
Returns whether the items in this *InventoryStack* have any tag contains the specified `text`.
- `public string GetTagContainText (string _text, bool _caseSensitive = true)`
Returns the tag of the item which contains the specified `text`.
- `public int GetItemId ()`
Returns the UID of the items in this *InventoryStack*. Returns `-1` if the stack is empty.
- `public int GetType (int _emptyResult=0)`
Returns the category ID of the items in this *InventoryStack*. Returns `_emptyResult` if the stack is empty.
- `public int GetUpgradeLevel ()`
Returns the upgrade level of the items in this *InventoryStack*.

- `public int GetAvailableSpace()`
Returns the available space in this stack. For example, if the maximum stack size is 99 and the current quantity is 10, the available space would be 89.
- `public bool isSameItem(int _uid, int _upgradeLevel, List<int> _enchancements, List<int> _sockets)`
| `public bool isSameItem(int _uid, int _upgradeLevel, List<int> _enchancements)`
| `public bool isSameItem(int _uid, int _upgradeLevel)`
| `public bool isSameItem(int _uid)`
Returns whether the items in this **InventoryStack** match the provided conditions.
- `public bool isEmpty()`
Returns whether this **InventoryStack** is empty.
- `public bool canBeMerged(int _uid)`
Returns whether the items in this **InventoryStack** can be merged with the provided item **UID**.

class Item

- **Properties**
- `public int uid;`
The unique id of this Item, in most scenarios you will need this uid to access an item.
- `public string name;`
The display name of this Item.
- `public string nameWithAffixing;`
Retrieve the item display name with prefixes and suffixes.
- `public string description;`
The description text of this Item.
- `public int type;`
The category index of this item. You can manage the categories on the "**InventoryEngine**" prefab.
- `public Texture2D icon;`
The icon texture of this item.
- `public int quality;`
The quality index of this item. You can manage the quality levels on the "**InventoryEngine**" prefab.
- `public bool tradeable;`
Whether this item is tradable with NPC merchant.
- `public bool deletable;`
Whether this item is deletable when drag out from inventory.
- `public bool useable;`
Whether this item is useable when right click or triggered from hot bar.
- `public bool consumable;`

Whether this item will be consumed when use.

➤ `public bool visible;`

Whether this item is visible on the interface. If an item is not visible, it will be put into HiddenStacks of InventoryHolder component.

➤ `public int price;`

The base price of this item when trading.

➤ `public int currency;`

The currency index of the trading price.

➤ `public int maxiumStack;`

The maximum number this item can be stack.

➤ `public int upgradeLevel;`

The enchancement level of this item.

➤ `public float weight;`

The weight value of this item.

➤ `public int dropRates;`

The drop rate of this item (0~100)%

➤ `public bool favorite;`

Whether this item is in player's favorite list.

➤ `public List<Attribute> attributes;`

The attributes list of this item,

➤ `public List<int> enchantments;`

The enchantments list of this item.

➤ `public List<Vector2> craftMaterials;`

The materials list required to craft this item.

➤ `public List<string> actions;`

The action string list of this item.

➤ `public List<string> tags;`

The tag string list of this item.

➤ `public int socketingSlots;`

The socketing slots number, if Socketing module is enabled, player will be able to socketing other items with specified tag and category into this item to boost its attributes..

➤ `public List<int> socketedItems;`

The list of socketed items, if Socketing module is enabled, player will be able to socketing other items with specified tag and category into this item to boost its attributes.

When the value of the list is -2, means this socket is locked, if it is -1 means this socket is empty, any number larger or equal to 0 represents the uid of this item.

➤ `public List<string> socketingTag;`

This item will only receive socketing items with any tag in this list.

Methods:

- `public Item (bool _initializeEnchantments = false, bool _initializeSocketingSlots = false)`

Create an Item by the UID of the database on the "**InventoryEngine**" prefab.

When **_initializeEnchantments** is set to true, depending on your **Enchantments** settings on the "**InventoryEngine**" prefab the enchantments of this item could be randomized.

When **_initializeSocketingSlots** is set to true, depending on your **Socketing** settings on the "**InventoryEngine**" prefab the socketing slots of this item could be randomized.

For example: *myNewItem= new Item(23, true,true); //23 is the uid of the item*

- `public Item Copy()`

Get an instance of this item data.

- `public float GetAttributeFloat (string _key)`

Returns the float value of the attribute with the provided [ScriptKey](#). Returns 0 if no such attribute is found

- `public int GetAttributeInt (string _key)`

Returns the int value of the attribute with the provided [ScriptKey](#). Returns 0 if no such attribute is found.

- `public string GetAttributeString (string _key)`

Returns the string value of the attribute with the provided [ScriptKey](#). Returns "None" if no such attribute is found.

- `public void UpdateAttribute (string _key, float _value)`
| `public void UpdateAttribute (string _key, int _value)`
| `public void UpdateAttribute (string _key, string _value)`

Updates the value of the attribute with the provided [ScriptKey](#).

- `public void RemoveAttribute (string _key)`

Removes an attribute with the provided [ScriptKey](#).

- `public void ClearAttribute ()`

Removes all attributes.

- `public void ResetAttributes()`

Resets all attributes to their original settings.

- `public bool isTagMatchText (string _tag)`

Returns whether the items have the matching **tag**. Tags such as "#1", "#2", "#3" are ignored.

- `public bool isTagsMatchList (List<string> _tags, bool _allMatch = true)`

Returns whether the tags of this item match the specified [tag](#) list.

- `public bool isTagContainText (string _text, bool _caseSensitive = true)`

Returns whether the items in this **InventoryStack** have any tag contains the specified [text](#).

- `public string GetTagContainText (string _text, bool _caseSensitive = true)`

Returns the tag of the item which contains the specified [text](#).

- `public void AddEnchantment(int _uid)`
Adds an enchantment by its **UID**.
- `public void RemoveEnchantment(int _uid)`
Removes an enchantment by its **UID**.
- `public void ReplaceEnchantment(List<int> _enchantments)`
Replaces the current enchantments with a list of **enchantment UIDs**.
- `public List<int> RandomEnchantment ()`
Adds a group of random enchantments based on the success rate in the Enchantment Setting.
Returns the result as a list of UIDs. Note that the result could be empty.
- `public void ResetEnchantment()`
Removes all enchantments.
- `public int Upgrade()`
Upgrades the item by one level.
- `public void ResetUpgrade()`
Resets the upgrade level to 0.
- `public void ResetAll()`
Resets all settings, including enchantments, attributes, and upgrade level.
- `public void ReplaceSocketing(List<int> _sockets)`
Override the socketed item list.
- `public Color GetQualityColor()`
Retrieves the color associated with the item's quality level.
- `public string GetQualityName()`
Retrieves the name of the item's quality level.
- `public Color GetTypeColor()`
Retrieves the color associated with the item's category.
- `public string GetTypeName()`
Retrieves the name of the item's category.

ItemIcon.cs / LinkItem.cs / InventoryItem.cs

These are interface scripts attached to each item slot.

Properties

- `public IconType Type;`
Reference: Only for showing information about an item.
Link: Shortcut to an item; it updates its information when the stats of the linked item change.
Item: Represents an item carried by an **InventoryHolder**.

- `public bool CanHover`
Determines if this item slot reacts to pointer hover events.
- `public bool CanBeLinked`
Indicates whether this item slot can be dragged to the shortcut slots
- `public bool Draggable` (For `InventoryItem.cs` and `LinkItem.cs` only)
Determines if this item slot is draggable.
- `public bool AutoLinkToHighestUpgradeLevel` (For `LinkItem.cs` only)
When set to true, the shortcut will automatically re-link to the highest upgrade level of the same item from the player's inventory.
- `public bool Splittable` (For `InventoryItem.cs` only)
Determines if this item slot is splittable.
- `public bool Deletable` (For `InventoryItem.cs` only)
Determines if this item slot is deletable.
- `public bool RecieveUntradeable` (For `InventoryItem.cs` only)
Indicates whether this item slot can receive untradeable items. For example, merchant slots should have this set to false.
- `public bool RecieveDragging` (For `InventoryItem.cs` only)
Indicates whether this item slot can receive dragged items.
- `public string LimitedByTag` (For `InventoryItem.cs` and `LinkItem.cs` only)
If this item slot only accepts items with a specific tag, specify that tag here.
- `public InventoryHolder LimitedByOwner` (For `InventoryItem.cs` and `LinkItem.cs` only)
If this item slot only accepts items from a specific owner, specify that owner here.

Methods:

- `public void RegisterClickCallback(int _id, OnItemClick _callback)`
Registers a callback for when this icon is clicked.
- `public void RegisterLinkedCallback(int _index, OnItemLinked _callback)`
(For `LinkItem.cs` only)
Registers a callback for when this icon is linked with an **Item**
- `public int GetNumber()`
Returns the number of items in the slot.
- `public int GetItemId()`
Returns the item ID. Returns -1 if the slot is empty.
- `public bool GetHover()`
Returns whether this item slot is hovered over by the mouse.
- `public void SetItemId(int _id)`
Overrides the item ID.

- `public void SetItemNumber(int _num)`
Sets the number of items in the slot.
- `public void SetUpgradeLevel(int _level)`
Sets the upgrade level of the item.
- `public void SetAppearance(Texture _icon, Color _backgroundColor , Color _frameColor ,
bool _numVisible=false, bool _upgradeVisible=false)
|public void SetAppearance(Item _item, bool _numVisible = false, bool _upgradeVisible
= false)`
Sets the icon and colors for the slot.
- `public void SetEmpty()`
Sets the slot to empty.
- `public void ToggleOutline(bool _visible)`
Toggles the outline effect of this slot.
- `public void SetFavorite(bool _favorite)`
Toggles whether this item is marked as a favorite.
- `public void SetVisible(bool _visible)`
Toggles this slot between fully visible or half-transparent.
- `public bool isTagMatchText (string _tag)`
Returns whether the item matches the provided **tag**.
- `public bool isTagsMatchList (List<string> _tags, bool _allMatch = true)`
Returns whether the tags of this item match the specified **tag** list.
- `public bool isTagContainText (string _text, bool _caseSensitive = true)`
Returns whether the item have any tag contains the specified **text**.
- `public string GetTagContainText (string _text, bool _caseSensitive = true)`
Returns the tag of the item which contains the specified **text**.
- `public InventoryStack GetStackData ()` (For InventoryItem.cs & LinkIcon.cs only)
Retrieves the **InventoryStack** data of this item.
- `public override InventoryHolder GetStackHolder ()` (For InventoryItem.cs & LinkIcon.cs only)
Retrieves the **InventoryHolder** of this slot.
- `public void SetHolder(InventoryHolder _holder)` (For InventoryItem.cs & LinkIcon.cs only)
Sets the **InventoryHolder** for this slot.
- `public override Item GetItem ()` (For InventoryItem.cs & LinkIcon.cs only)
Retrieves the **Item** in this slot.
- `public bool isNameMatch(string _name)` (For InventoryItem.cs only)
Returns whether the item name matches the provided string.
- `public bool isTypeMatch(int _type)` (For InventoryItem.cs only)
Returns whether the item category matches the provided category ID.
- `public bool isTradeable(bool _emptyResult=true)` (For InventoryItem.cs only)
Returns whether the item can be traded. Returns **_emptyResult** when the slot is empty.

- `public bool isUseable(bool _emptyResult = true)` (For `InventoryItem.cs` only)
Returns whether the item can be used. Returns `_emptyResult` when the slot is empty.
- `public bool isConsumable(bool _emptyResult = true)` (For `InventoryItem.cs` only)
Returns whether the item can be consumed. Returns `_emptyResult` when the slot is empty.
- `public bool isDeletable(bool _emptyResult = true)` (For `InventoryItem.cs` only)
Returns whether the item can be deleted. Returns `_emptyResult` when the slot is empty.
- `public void Split(int _result)` (For `InventoryItem.cs` only)
Splits the item in this slot.
- `public void LinkItem(InventoryHolder _holder, Item _item, int _num, bool _empty)` (For `LinkIcon.cs` only)
Links this icon with the provided `InventoryHolder` and item.
- `public void SwapLink(LinkIcon _source)` (For `LinkIcon.cs` only)
Swaps the linked item with another `LinkIcon`.
- `public void Split()` (For `InventoryItem.cs` only)
Pop up the split interface to split this item into two stacks.
- `public void MarkFav ()` (For `InventoryItem.cs` only)
Mark this item as Favorite item.
- `public void Drop ()` (For `InventoryItem.cs` only)
Drop this item.

WindowsManager.cs

This script manages all UI windows.

Properties

- `public static bool anyWindowExists;`
Returns true if there is any UI window exists.
- `public static UiWindow ActiveWindow;`
Returns the UI window in front of all other windows.

Methods:

- `public static void CloseAllWindows()`
Close all opened windows.
- `public static UiWindow GetWindow(string _prefabName, InventoryHolder _holder, bool _bringToTop=true)`
Open an window by the UI prefab name and the `InventoryHolder` component, you can find the prefabs in [SoftKitty/InventoryEngine/Resources/InventoryEngine/UiWindows/](#)
- `public static bool isWindowOpen(InventoryHolder _key)`
Check if the window of an `InventoryHolder` is opened.

➤ `public static UIWindow getOpenedWindow(InventoryHolder _key)`

Returns the window class by an *InventoryHolder* as the key.

CALLBACKS

<INVENTORYHOLDER> *ItemChangeCallback(Dictionary<Item,int> _changedItems)*

this callback is triggered whenever items within an “*InventoryHolder*” are changed. It’s particularly useful for monitoring changes in a player’s *inventory* or *equipment*.

The callback returns a *Dictionary* where each entry represents an item that has been modified. The *key* is the item, and the *value* indicates the quantity changed. For example:

If the player drinks a potion, the *Dictionary* *key* will be the potion item, and the *value* will be *-1*.

If the player loots a sword, the *key* will be the sword item, and the *value* will be *1*.

TO REGISTER THIS CALLBACK:

First, define the callback function in your script:

```
public void OnEquipmentItemChange(Dictionary<Item,int> _changedItems)  
{  
.....//Your logic here  
}
```

In the *Start* method of your script, register the callback:

```
ItemManager.PlayerEquipmentHolder.RegisterItemChangeCallback(OnEquipmentItemChange);
```

Remember to **UNREGISTER** the callback in *OnDestroy* to avoid memory leaks when the *GameObject* is destroyed:

```
ItemManager.PlayerEquipmentHolder.UnRegisterItemChangeCallback(OnEquipmentItemChange);
```

Here is a full **EXAMPLE** of how you could use this callback to change player's equipment model:

```
private void Start()
{
    ItemManager.PlayerEquipmentHolder.RegisterItemChangeCallback(OnEquipmentItemChange); //Register callback for player's equipment
}

2 个引用
public void OnEquipmentItemChange(Dictionary<Item, int> _changedItems)
{
    foreach (var _item in _changedItems.Keys) {
        if (_changedItems[_item] > 0) //Equip
        {
            //Your logic to read the _item information, and change the equipment model
        }
        else //Unequip
        {
            //Your logic to read the _item information, and change the equipment model
        }
    }
}

@Unity 消息 | 0 个引用
private void OnDestroy() // Don't forget to call UnRegisterItemChangeCallback() when the gameObject going to be destroyed.
{
    ItemManager.PlayerEquipmentHolder.UnRegisterItemChangeCallback(OnEquipmentItemChange);
}
```

<INVENTORYHOLDER> *ItemDropCallback (Item _droppedItem,int _number)*

this callback is triggered whenever items within an “*InventoryHolder*” are dropped by dragging out of the window. It's particularly useful for instantiate item prefabs in the game world when player drop an item.

The callback returns an *Item* that has been dropped and its amount. The *_droppedItem* is the *Item*, and the *_number* indicates the amount of this item has been dropped. For example:

If the player drops 10 potion, the *_droppedItem* will be the potion item, and the *_number* will be 10.

TO REGISTER THIS CALLBACK:

First, define the callback function in your script:

```
public void OnItemDrop(Item _droppedItem,int _number)
{
    // Your logic here
}
```

In the *Start* method of your script, register the callback:

```
ItemManager.PlayerInventoryHolder.RegisterItemDropCallback(OnItemDrop);
```

Remember to **UNREGISTER** the callback in *OnDestroy* to avoid memory leaks when the *GameObject* is destroyed:

ItemManager.PlayerInventoryHolder.UnRegisterItemDropCallback(OnItemDrop);

<INVENTORYHOLDER> *ItemUseCallback(string _action, int _id, int _index)*

This callback is triggered whenever the player *uses* an *item*, which could be a consumable, equipment, or skill.

The callback provides three pieces of information:

- *_action*: The "Use Action" string defined in the *ItemManager*.
- *_id*: The unique ID of the item.
- *_index*: The slot index of the item within its container (e.g., inventory).

EXAMPLE: In the demo scene setup, if the player drinks a "Health Potion (L)" located in the 4th slot of the inventory:

- *_action* will be "AddHp_200"
- *_id* will be 1
- *_index* will be 4

TO REGISTER THIS CALLBACK:

First, define the callback function in your script:

```
public void OnItemUse(string _action, int _id, int _index)  
{  
    .....// Your logic here  
}
```

In the *Start* method of your script, register the callback:

ItemManager.PlayerInventoryHolder.RegisterItemUseCallback(OnItemUse)

Remember to **UNREGISTER** the callback in *OnDestroy* to avoid memory leaks when the *GameObject* is destroyed:

ItemManager.PlayerInventoryHolder.UnRegisterItemUseCallback(OnItemUse);

Here is a full **EXAMPLE** of how you could use this callback to add player's hp when player drink potion:

```

@Unity 消息 | 0 个引用
private void Start()
{
    ItemManager.PlayerInventoryHolder.RegisterItemUseCallback(OnItemUse); //Register callback to trigger when player's item being use.
}

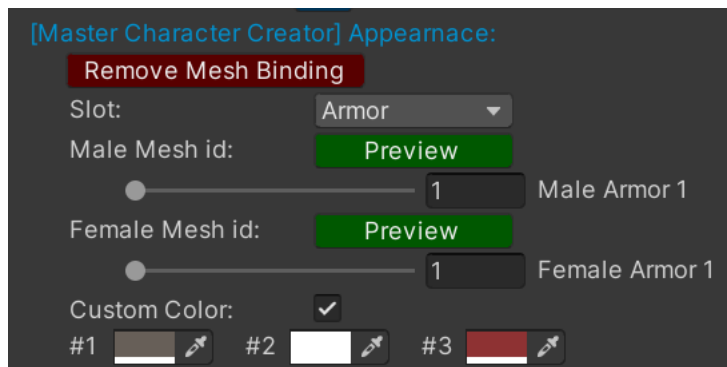
2 个引用
public void OnItemUse(string _action, int _id, int _index) //When player using an item, this callback will be called.
{
    string[] _args = _action.Split("_"); //For example, the _action of the potion will split into "AddHp" and "200"
    if (_args.Length>2)
    {
        int _value = int.Parse(_args[1]); //Convert "200" into int number.
        switch (_args[0]) { //Check the action name and excute corresponding logic
            case "AddHp":
                Player.AddHealth(_value); //Your logic to add hp for player.
                break;
            case "AddSp":
                Player.AddStamina(_value); //Your logic to add sp for player.
                break;
        }
    }
}

@Unity 消息 | 0 个引用
private void OnDestroy() // Don't forget to call UnRegisterItemUseCallback() when the gameobject going to be destroyed.
{
    ItemManager.PlayerInventoryHolder.UnRegisterItemUseCallback(OnItemUse);
}

```

WORKING WITH [MASTER CHARACTER CREATOR]

If you have Master Character Creator in your project, Master Inventory Engine will automatically recognize and connect to it. In the Item Setting of the InventoryEngine prefab, you'll find a button labeled Create Mesh Binding. Click this button for your equipment item to access the following settings:



- Slot: Choose the equipment slot for this item.
- Male Mesh ID: Specify the mesh ID for this item when equipped on a male character. Click the Preview button to see the mesh.
- Female Mesh ID: Specify the mesh ID for this item when equipped on a female character. Click the Preview button to see the mesh.
- Custom Color: Toggle this checkbox to customize the colors of the mesh, allowing you to have multiple equipment items with different colors using the same mesh.

You can access the appearance data through `Item.equipAppearance`. Below is an example script that demonstrates how to update the player's model when the player equips an item:

```
private void Start() {
```

```

ItemManager.PlayerEquipmentHolder.RegisterItemChangeCallback(OnEquipmentItemChange);
}

public void OnEquipmentItemChange(Dictionary<Item, int> _changedItems) {
    foreach (var _item in _changedItems.Keys) {
        if (_changedItems[_item] > 0)
        {
            Player.Equip(_item.equipAppearance); //Equip
        } else {
            Player.Unequip(_item.equipAppearance.Type); //Unequip
        }
    }
}
}

```

SEAMLESS MOUSE CONTROL BETWEEN GAMEPLAY AND UI

In some games, the player uses mouse movement to control the camera's rotation, with the cursor hidden during gameplay. This setup can be challenging when players need to interact with the UI, such as an inventory or character screen, since the cursor remains unavailable.

To address this, the system provides a **boolean** value: ***WindowsManager.anyWindowExists***. This variable returns **true** if any UI window is currently open. Developers can use this value to detect when a UI window is active, allowing them to temporarily unbind the mouse from controlling the camera and make the cursor visible. When ***WindowsManager.anyWindowExists*** is **false**, they can re-hide the cursor and return the mouse control back to the camera.

This setup ensures seamless transitions between gameplay and UI interactions, enhancing the overall player experience.