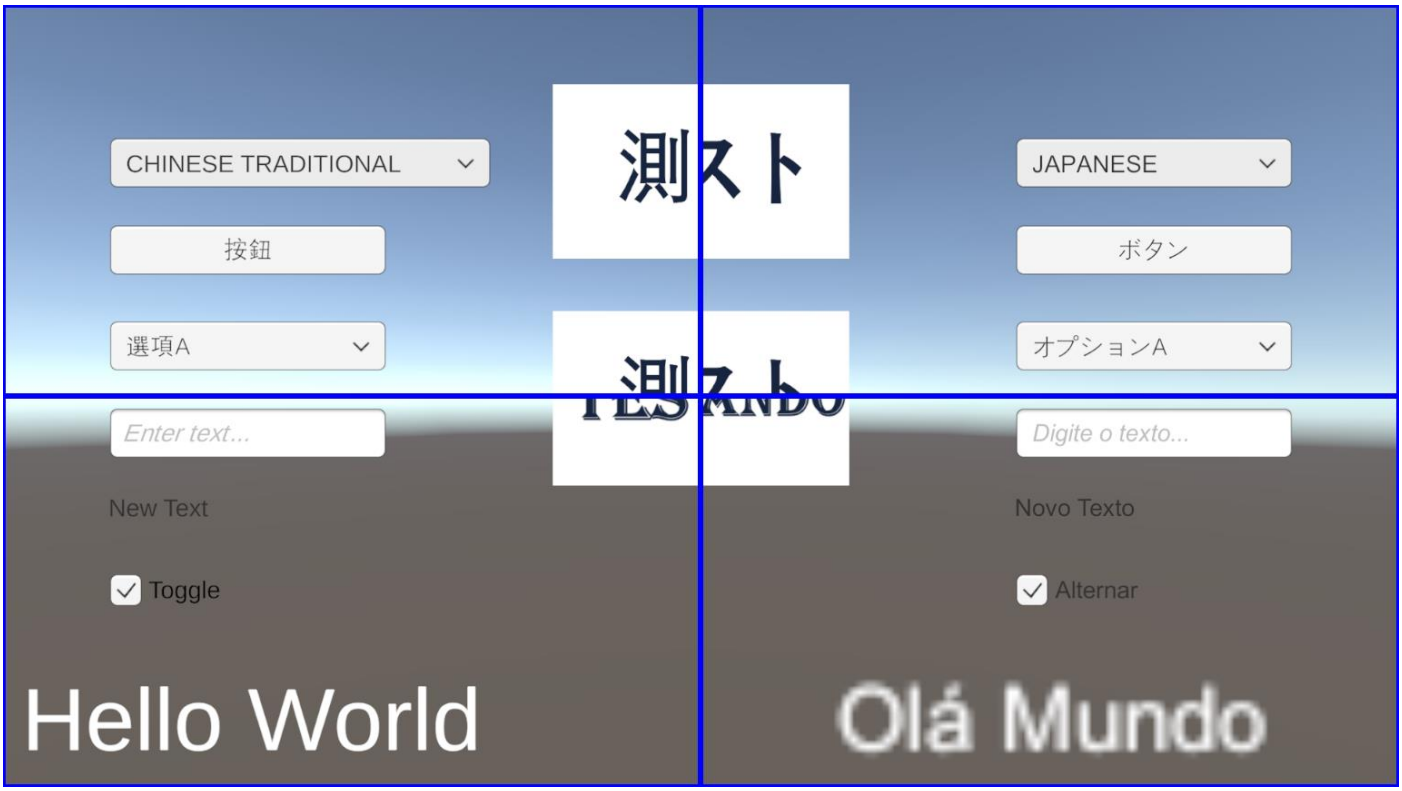

Language Tool

Documentation

_____

Creator: Lucas Gomes Cecchini

Online Name: AGAMENOM

Overview

This document will help you to use **Assets Language Tool** for **Unity**.

With it you can translate your game into several languages, automatic selection of languages (if you have a file available) and a simple text file to interpret, making it possible for anyone to translate the game into a new language.

It also has simple features to translate images and sound.

Instructions

You can get more information from the Playlist on YouTube:
https://youtube.com/playlist?list=PL5hnfx09yM4JkAyxrZWaFjhO3NMWxP_1F

Script Explanations

Language Manager Delegate

This script defines a static class responsible for broadcasting language change notifications throughout an application. It allows other components (like UI elements or game logic) to react when the application's language is updated.

Purpose

The script establishes a centralized system for sending language change notifications. It does this using an event-driven approach, so other parts of the application can subscribe and be informed when a language change occurs.

Namespace

- **LanguageTools**:
  The script is encapsulated inside a logical group or module named "LanguageTools", which organizes it alongside other tools or utilities related to language handling.

Class: LanguageManagerDelegate

This is a static utility class, meaning it is not meant to be instantiated. It acts as a global broadcaster of language change events.

Members Explained

1. **Delegate Type: LanguageUpdateDelegate**
   - Acts like a function signature or template.
   - Any method that wants to respond to a language update must match this structure.
   - It defines what kind of functions can be linked to the event (in this case, functions without parameters and return value).
2. **Event: OnLanguageUpdate**
   - This is the core notification mechanism.
   - External systems (like UI components or settings menus) can subscribe to this event.
   - When a language change occurs, all subscribers are notified at once by invoking the event.
3. **Method: NotifyLanguageUpdate**
   - This is the trigger point.
   - Whenever the language setting changes, calling this method informs all registered listeners.
   - It first checks if there are any subscribers, and only invokes the event if necessary, preventing errors.

How to Use

- **Subscribing**:
  A developer can connect a custom method to the OnLanguageUpdate event. That method will run whenever a language change is announced.

- **Triggering**:
  The system (such as a language settings panel) calls the NotifyLanguageUpdate method when the user changes the language. This sends out a signal to all subscribers.
- **Example Use Case**:
  - A UI text component listens for updates to refresh its displayed text.
  - A settings window closes automatically if it detects a language reload.

Benefits

- Promotes loose coupling: components don't need to know about each other.
- Centralizes update logic: a single point of control for language change events.
- Easy to expand: any number of listeners can subscribe or unsubscribe freely.

This script provides a clean and efficient way to ensure that all parts of the application stay in sync with the language setting, using an event system that's easy to integrate.

Language Class Group

This script defines a series of **data structures** used to manage language localization and UI canvas configurations inside a Unity project. It does **not contain any logic or behavior** — it strictly defines how data is stored and organized, especially for serializing and editing within Unity's Inspector or custom editor tools.
Here's a complete, structured explanation of how it works — class by class, with variables and usage — without using C# syntax:

Purpose

The script enables a Unity-based localization system to:

- Store language data (IDs, texts, metadata).
- Serialize and deserialize UI layout info.
- Maintain editable language components.
- Handle user-defined translation and canvas configuration.

Class Overview & Usage

1. LanguageAvailable

Represents a language option that the user can select.

- culture: Language code (like "en-US").
- nativeName: Native name shown in UI (like "English").).
- name: Friendly name shown in UI (like "English").
- isAvailable: Whether the language is currently available.
- columnIndex: The column used in the TSV table to extract data for this language.

Usage: Listed in the language settings file to define all supported languages.

2. IdData

Stores a localization entry.

- iD: A number that uniquely identifies this entry.
- text: The localized text string for the given language.

Usage: Represents translation rows loaded from TSV or used at runtime.

3. IdMetaData

Stores visual configuration for localized text.

- iD: Identifier for matching this with a translation.
- alignment: How the text is aligned (e.g., left, center).
- fontSize: Font size to apply.

- fontListIndex: Index of the font to use.
- componentType: UI component type this configuration is linked to.

Usage: Controls how a text component looks in the UI.

4. CanvasDataList

Stores a serialized version of a UI GameObject's state.

- Includes transform properties (position, rotation, scale).
- Also includes RectTransform layout values (anchor, pivot, size).

Usage: Saves or restores the UI state when loading or modifying localized UI content.

5. CanvasStructure

Represents the entire structure of a canvas.

- canvasName: Name of the canvas.
- canvasLayers: List of grouped UI objects.
- Also stores serialized versions of Canvas, CanvasScaler, and Raycaster settings.

Usage: Allows full canvas layout export/import for localization testing or prefab generation.

6. CanvasLayers

Represents a single canvas layer.

- CanvasObjectsLayers: Names of GameObjects in this layer.
- rectTransforms: Layout definitions for each object.

Usage: Organizes layout elements inside a canvas into layers.

7. RectTransformData

Serialized version of a RectTransform component.

- Stores layout and anchor values for rebuilding UI positions and sizes.

Usage: Allows for complete UI layout re-application from saved data.

8. CanvasData

Serialized Canvas component values.

- Includes render mode, sorting options, shader channels, etc.

Usage: Preserves Canvas setup between sessions or for editing.

9. CanvasScalerData

Serialized CanvasScaler configuration.

- Includes resolution handling, physical size options, scaling behavior.

Usage: Ensures consistent UI scaling across screen sizes when restoring canvas settings.

10. GraphicRaycasterData

Serialized raycasting properties.

- Defines how raycasts behave on UI elements.

Usage: Saves interactive behavior settings for UI canvases.

11. LanguageOptions

Defines selectable language options with visuals.

- text: Name of the language.
- sprite: Optional icon (e.g., a flag).
- iD: ID used internally to reference this language.

Usage: Used in dropdowns or lists when selecting a language.

12. ScriptText

A text string with an associated event.

- iD: Identifier for the script block.

- text: The message or dialogue line.
- targetScripts: Event triggered when this script is processed.

Usage: Supports localization of dynamic text that also invokes in-game behaviors.

13. LanguageLines
A line of text that may or may not be translated.
- iD: Unique identifier.
- text: The raw or translated text.
- translateText: Boolean flag indicating whether it should be translated.

Usage: Used for scripts or UI elements where some lines are static and others change per language.

14. LanguageForEditingSave
Stores text content and formatting details for one component.
- Contains text, alignment, size, font info, and context.
- componentType: Describes what kind of UI object this is (e.g., button, label).

Usage: Used in the editor when modifying or reviewing text components.

15. CanvasForEditingSave
Stores a canvas's context and its serialized JSON structure.
- canvasID: Unique ID for the canvas.
- textContext: Description or category of this canvas.
- json: Serialized layout state of the canvas.

Usage: Used to save and restore canvas designs during localization sessions.

16. LanguageFileManagerWindowData
Top-level container that holds all editable language and canvas data.
- Includes language name, list of components, canvases, and control flags like:
  - firstTime: Indicates first-time setup.
  - showTextData, showCanvasData: Control panel toggles.
  - fileIsSaved: Tracks save status.
  - idIndex: Used to generate new unique IDs.

Usage: Passed between editor tools and saved to disk to maintain editor state.

17. ManagerLanguageIdData
Used to manage ID-text-context relationships in the editor.
- iD: Identifier.
- text: Display or translation text.
- textContext: Tag that indicates where or how this is used.

Usage: Facilitates organizing and modifying entries in a localization editor window.

✅ **Summary**
This script is the **foundation** of a custom localization and UI layout editor. It defines all the **types of data** the system can handle, and how they are stored and retrieved. These structures allow:
- Safe serialization (saving/loading) of complex UI and text configurations.
- Easy Unity Editor integration with custom tools.
- Seamless runtime support for switching and displaying multiple languages.

Language File Manager

This script is responsible for managing the application's localization system — that is, handling different languages. It loads, saves, and interprets language-related data files (TSV files) and keeps track of the user's selected language preference.

General Function

The script is part of a language management module. Its tasks include:

- Reading configuration data for available languages.
- Determining which language the user selected.
- Parsing language files to extract translated text and UI metadata.
- Caching and saving language preferences.

It works with table-based data stored in .tsv files (tab-separated values), which are structured like spreadsheets. These files contain IDs, culture codes, display names, availability flags, and localized strings.

Class-Level Variables

1. **cachedLanguageData**:
   Keeps the language configuration in memory so it only needs to be loaded once. Improves performance by avoiding redundant file reads.
2. **CultureCodeKey**:
   A fixed identifier used to store and retrieve the user's selected language code from the device's persistent storage.
3. **assetsPath**:
   Stores the complete path to the folder where language data files are located. It's computed once and reused.

Core Methods and Their Roles

1. LoadLanguageSettings

- Loads the main language settings file from a special folder inside the project.
- Caches the result.
- If the file is missing, logs an error and returns nothing.

2. GetSaveCultureCode

- Determines what language the user previously selected.
- Falls back to the default language if no selection is stored.
- Updates the configuration in memory with the chosen culture code.

3. SetSaveCultureCode

- Saves a new language selection to persistent storage.
- Updates the in-memory language settings so that the application uses the new language immediately.

4. GetLanguageAssetsPath

- Constructs the full path to the folder where language files are stored.
- Ensures the folder exists (creates it if needed).
- Returns the complete path string.

5. RowAndColumnCounter

- Measures how many rows and columns are in a provided table.
- Used internally to validate whether a table has enough data to process.

6. GetAvailableLanguages

- Loads the main language data file.
- Reads a specific table structure and extracts information about all available languages.
- Skips unnecessary columns (like IDs or labels).
- Populates a list with data about each language that is marked as available.

7. ExtractIDs

- Takes a table and extracts all text entries related to a given language.

- Locates the appropriate column based on a language code.
- Reads each row, pairing an ID with its localized text.
- Produces a list of ID-text pairs used later for UI translation.

8. GetAllData

- Coordinates the full setup of language data.
- Loads the main translation table, metadata, and canvas-related text from .tsv files.
- Verifies that all files exist.
- Parses each file using helper functions.
- Extracts and stores both translation data and formatting metadata (like font size or alignment).
- Triggers a language update if the selected language is unavailable.

9. GetIDText

- Given a list of translations and an ID, returns the matching localized string.
- If no match is found, returns nothing and warns the developer.

10. GetIDMeta

- Similar to GetIDText, but returns UI metadata instead of text.
- Searches for and returns layout information (e.g., font size or alignment) for a given ID.

🧪 How It's Used in a Project

- **Startup**:
  The application calls GetAllData() to load all translation-related files into memory.
- **Language Switch**:
  When a user changes the language, the system calls SetSaveCultureCode() to store the new preference and GetAllData() to reload all translated content.
- **UI Translation**:
  UI elements use GetIDText() to fetch the correct string and GetIDMeta() to retrieve any associated formatting information.
- **Language List**:
  The settings screen might call GetAvailableLanguages() to list all languages the user can choose from.

📌 Summary

This script is the backbone of the localization system. It:

- Loads settings and data from .tsv files.
- Stores user preferences.
- Extracts translatable content and UI formatting metadata.
- Provides utilities for accessing the translations by ID.

It ensures your application can dynamically change languages and display properly localized content across its interface.

Language Editor Utilities

The script LanguageEditorUtilities is a utility designed for use in the Unity Editor. Its purpose is to assist with managing localized language entries for different keys in a structured way, specifically by adding missing keys from a selected language file to others.
Here's a breakdown of how the script works, including explanations for each variable, method, and its usage:

Overall Purpose

This editor utility provides a custom Unity Editor menu item to ensure all localized language files in a project contain the same keys. It compares a "selected" language file with others and adds any missing keys to those files. This helps developers keep language translations in sync.

Variables

1. **selectedLanguagePath**
    - A string representing the relative path to the primary language JSON file to compare against the others.
    - The script reads from this file to gather the "reference" list of localization keys and values.
2. **targetFolderPath**
    - A string that holds the relative path to the folder where all localization files (including the selected one) are stored.
    - This is where the script searches for other language files that may be missing keys.

Method: AddMissingKeysToOtherLanguages()
Description:
This is the main method invoked by a Unity Editor menu item. When selected, it performs the following:
Step-by-step Logic:

1. **Load the reference dictionary**
    - It reads the JSON content from selectedLanguagePath.
    - The content is deserialized into a dictionary of strings (Dictionary<string, string>) representing the reference key-value pairs.
2. **Iterate through all language files in the target folder**
    - The script loads all .json files in targetFolderPath.
    - It skips the selectedLanguagePath file to avoid modifying it.
3. **Compare and update dictionaries**
    - Each language file is read and converted to a dictionary.
    - The script checks for missing keys by comparing the reference dictionary to the current one.
    - Any keys that are present in the reference but missing in the target are added with their values (from the reference).
4. **Track added keys**
    - A temporary dictionary stores the keys and values that are added to the target file, mainly for logging purposes.
5. **Update the file if changes were made**
    - If any new keys were added, the updated dictionary is written back to the file in JSON format using pretty-printing.
    - Unity's AssetDatabase.Refresh() is called to update the project's file system.
6. **Logging**
    - For each modified file, it logs which keys were added.
    - If no changes are needed, it logs that the file is already complete.

Usage Instructions

- Place this script in an Editor folder within your Unity project.
- Make sure your localization files are stored in Assets/Resources/Localization/ and formatted as .json files with string key-value pairs.
- Right-click in Unity's top menu:
  Navigate to **"Tools" > "Language Editor" > "Add Missing Keys to Other Languages"**.
- This action will:
    - Read the English.json file (or any file designated in selectedLanguagePath).
    - Add any missing keys from it into the other .json files in the same folder.

Summary

This script automates the tedious task of manually checking and adding missing localization entries. It is particularly useful in multilingual Unity projects where each supported language has a separate JSON file, and developers want to ensure consistency across them.

Canvas Manager

This script is a **utility class** designed to handle **saving, applying, and recreating Unity canvas layouts** using a serialized structure. It helps developers extract UI hierarchies from canvas GameObjects, save their configurations (layout and component data), and reapply or recreate them elsewhere — a useful feature for multilingual UI systems or visual editors.

🧭 General Purpose

The class enables the following:

- **Extracting** the structure and layout of a canvas GameObject into a data object.
- **Applying** saved layout data back to an existing canvas.
- **Rebuilding** a full canvas hierarchy programmatically from saved data.

This provides a **way to serialize and deserialize canvas layouts**, making it easier to handle UI editing, saving, and multilingual layout adjustments.

🔧 Class Overview

This utility class contains only **static methods**, serving as a static toolbox for serializing and reconstructing Canvas layouts. It includes additional helper logic to support hidden or inactive elements and debug visualization during reconstruction.

📌 Main Public Methods

1. ExtractCanvasData(canvasStructure, canvasObject)
What it does:
Populates a data structure (CanvasStructure) with all metadata and hierarchy information from a source canvas GameObject.
Steps:

- Validates the object to ensure no duplicate names exist (which would break path-based mapping).
- Temporarily activates all child GameObjects (including inactive ones) to ensure the full hierarchy is captured.
- Fills out all component settings (Canvas, CanvasScaler, GraphicRaycaster).
- Captures layout paths for all UI elements inside the canvas.
- Converts visual transform data into a structured, serializable format.

Usage:
Used when you want to "snapshot" a canvas and save it to a file or data container.

2. ApplyCanvasData(canvasStructure, canvasObject)
What it does:
Takes saved layout and metadata and **applies it to an existing canvas GameObject**.
Steps:

- Validates if the structure is consistent (same number of objects and transform records).
- Finds each child UI element by name and applies saved transform settings.
- Applies saved canvas component settings as well.

Usage:
Use this when reloading or restoring a canvas to a previously saved layout.

3. CreateCanvasFromStructure(canvasStructure, out canvasObject)

What it does:
Generates a **brand new canvas GameObject** and reconstructs its structure and layout from a saved CanvasStructure.
Steps:

- Validates data integrity and canvas name.
- Creates a new canvas object with the necessary UI components.
- Assigns random debug colors and outline effects to each UI element for visual clarity.
- Reconstructs the full object hierarchy layer by layer.
- Applies transform data and adds visual aids like color and outlines to aid debugging.

Usage:
Use this to programmatically generate new canvases from layout templates or saved designs.

⚙ Supporting Private Methods

4. GenerateCanvasLayers(canvasObject)
What it does:
Recursively explores all UI objects inside the canvas, and converts each path into a linear data structure (CanvasLayers).
Usage:
Used during extraction to flatten and serialize the UI hierarchy.

5. ContainsDuplicateSiblings(parent)
What it does:
Recursively checks for duplicated object names under the same parent, which would break the structure mapping.
Usage:
Used as a safety check before any extraction or building operation.

6. ConvertToRectTransformData(t)
What it does:
Converts Unity's RectTransform into a serializable structure containing all relevant positioning data.
Usage:
Used during data capture and application to ensure accurate layout replication.

7. ActivateAllChildren(parent, states)
What it does:
Recursively enables all GameObjects in a hierarchy and records their original active states to a dictionary.
Usage:
Used before extraction to ensure the entire hierarchy (even inactive objects) is included.
Restores the original active state afterward.

8. FindChildByName(parent, name)
What it does:
Searches through a parent UI object for a child with a matching name.
Usage:
Used during layout application and reconstruction to target the correct GameObject.

9. PopulateCanvasMetadata(ref s, obj)
What it does:
Fills the metadata part of the CanvasStructure from a Unity canvas GameObject, including:

- Canvas properties.

- CanvasScaler settings.
- GraphicRaycaster behavior.

Usage:
Part of the extraction process.

10. UpdateCanvasMetadata(s, obj)
What it does:
Takes saved metadata and applies it to a canvas GameObject.
Usage:
Used during layout restoration or new canvas creation.

📂 **Example Use Case**

1. **Extract**:
   - A developer creates a visual UI layout.
   - Runs ExtractCanvasData() to save the structure to a file or memory.
2. **Apply**:
   - Later, a different layout is needed, so ApplyCanvasData() is called to restore it.
3. **Rebuild**:
   - At runtime or in an editor tool, CreateCanvasFromStructure() is used to build a new UI layout from a predefined design.

✅ **Summary Table**

Function	Purpose	Used When
ExtractCanvasData	Captures canvas layout + settings into a data object	Saving UI layouts
ApplyCanvasData	Applies saved layout to an existing canvas	Updating UI during runtime or in editor
CreateCanvasFromStructure	Builds a new canvas from saved structure	Instantiating layouts dynamically
GenerateCanvasLayers	Converts hierarchy into linear layer paths	Serialization
PopulateCanvasMetadata	Extracts canvas settings into a data object	During extraction
UpdateCanvasMetadata	Applies saved canvas settings to an object	During application or new canvas creation
ContainsDuplicateSiblings	Validates uniqueness of sibling names	During extraction or layout generation
ConvertToRectTransformData	Converts transform to serializable form	During extraction and application
ActivateAllChildren	Temporarily activates all GameObjects in hierarchy	Before extracting hierarchy

ID Exists Attribute

This script creates a **custom validation system** for Unity's Inspector to ensure that integer IDs assigned to components or canvases are **unique**, helping prevent conflicts in a localization system. It uses a **custom attribute and drawer**, and references an external JSON file to perform the validation.

🎯 **General Purpose**

- Ensures each component or canvas in your localized UI system has a unique ID.
- Provides **visual feedback** in the Unity Editor when a duplicate ID is detected.
- Helps avoid bugs caused by overlapping or reused IDs in text or UI elements.

🧩 Components and Usage

🏷️ Attribute: IDExistsAttribute

This is a **marker** that you add to an integer field in your script to enable duplicate-checking.

- **Variable: searchCanvas**
  - A flag that defines whether the attribute should check *canvas IDs* (true) or *component IDs* (false).
- **Constructor**
  - Initializes the attribute with the searchCanvas flag.

Usage Example:

```
[IDExists] public int componentID = 0        → Checks for duplicates among component IDs.
[IDExists(true)] public int canvasID = 0     → Checks among canvas IDs instead.
```

🖼️ Drawer: IDExistsDrawer

This part of the script **renders the property in the Inspector** and adds a **warning box** if the ID is not unique.

🔷 Constant

- **FilePath**:
  Path to the JSON file (LanguageFileData.json) inside Unity's ProjectSettings directory. This file stores existing IDs for both components and canvases.

🔷 Static Cache

- **cachedData**:
  Holds the parsed content of the JSON file so it can be reused without reading the file again on every frame.
- **lastFileWriteTime**:
  Tracks when the JSON file was last modified to determine when to reload it.

🔍 Key Methods

1. OnGUI(position, property, label)

- This is the method responsible for drawing the field in the Unity Editor.
- **Behavior**:
  - Highlights the field in yellow if the ID is already used.
  - Displays a warning box below the field if a duplicate is found.
  - Restores the original GUI color after rendering.

2. GetPropertyHeight(property, label)

- Adjusts the height of the field in the Inspector.
- Adds extra vertical space if a warning is being shown (for duplicated ID).

3. CheckIDExists(id, checkCanvas)

- Loads and parses the JSON file **only if it has been modified** since the last read.
- Depending on the checkCanvas flag:
  - Searches either the list of canvas IDs or component IDs for a match.
- Returns true if the given ID is found in the data, signaling a duplicate.

📁 JSON File (External Dependency)

- The system expects a LanguageFileData.json file located inside ProjectSettings.
- This file stores the current state of IDs in componentSave and canvasSave lists (as defined in LanguageFileManagerWindowData).
- It must be up to date for validation to be accurate.

🧪 Practical Use

- In a Unity localization system where multiple components use language IDs, this script **automates conflict detection**.
- For example, if two buttons are accidentally assigned the same componentID, Unity will **immediately warn the developer** in the Inspector.
- Helps maintain integrity across large UI setups and localization projects.

✅ **Summary**

This script enhances Unity's Inspector by adding a validation layer for language or canvas IDs:

- Makes it easier to detect duplicate IDs visually.
- Prevents localization bugs from overlapping identifiers.
- Keeps checks efficient by caching parsed data from a central configuration file.

Font And Alignment Utility / Font And Alignment Utility TMP

This script is a **static utility tool** used for handling **legacy Unity UI fonts and text alignment** in multilingual projects. It simplifies how fonts and alignments are selected and converted using numeric codes — useful in systems where configuration data (like localization files) uses numbers instead of Unity's enum types.

🎯 **Purpose**

- Provides tools to **retrieve fonts** by index or asset reference.
- Converts **integer alignment codes** into Unity's TextAnchor enum and back again.
- Supports **compatibility** between configuration data and Unity's UI components.

It's located under a namespace for legacy tools, meaning it supports older UI setups like Text and Font, rather than modern TextMeshPro.

🧩 **Overview of Methods and Variables**

🔷 **GetFontByIndex(fontListIndex)**

What it does:

- Takes a number (starting from 1) and returns the corresponding Font object from the list in the language settings.

How it works:

- Loads the font list via a settings loader.
- Checks if the index is valid.
- Returns the corresponding font if within range; otherwise returns nothing.

Usage:

- Used when loading a saved configuration that stores only the font index.

🔷 **GetFontIndex(font)**

What it does:

- Takes a Font asset and returns its 1-based index from the font list.

How it works:

- Loads the font list from the settings.
- Iterates through the list to find a match.
- Returns the position (starting from 1).
- Returns 0 if not found.

Usage:

- Used when saving settings that need to convert a selected font into an index.

🔷 **ConvertToTextAnchor(alignment)**

What it does:

- Takes a numeric code (like 1, 2, 3, etc.) and returns a corresponding TextAnchor value used in Unity.

How it works:

- Maps numbers to anchor positions:
  - Top: 1–3
  - Middle: 7–9
  - Bottom: 13–15
- Returns UpperLeft as a fallback if the input isn't recognized.

Usage:

- Converts layout data from configuration files into actual usable settings for UI elements.

🔷 ConvertToAlignmentCode(alignment)

What it does:

- Takes a TextAnchor enum value and returns the corresponding integer code used in the project's data system.

How it works:

- Uses a switch statement to map TextAnchor values to integer codes.
- Returns 1 by default if not matched.

Usage:

- When saving alignment settings, this method turns them into consistent numeric codes that can be stored easily in JSON, TSV, or similar.

📂 Dependencies

This utility relies on:

- A method called LoadLanguageSettings() from the LanguageFileManager.
- A structure inside that settings file containing a list of Font assets (fontListData.fontList).

🧪 Real-World Scenario

Imagine you have a multilingual game UI and want to:

1. **Save settings**:
   - You store font index = 2 and alignment code = 9 in a config file.
2. **Load settings**:
   - You read index 2, get the correct Font.
   - You read code 9, apply TextAnchor.MiddleRight to your text element.
3. **Editor tools**:
   - When building a localization tool in the Unity Editor, you use these utilities to translate between what's stored in files and what Unity components need.

✅ Summary

This script ensures **smooth conversion and management** of fonts and alignment data in multilingual UI systems, especially when settings are stored as numbers:

Function	Converts or Retrieves	For What
GetFontByIndex	Number → Font asset	Load saved settings
GetFontIndex	Font asset → Number	Save settings
ConvertToTextAnchor	Number → TextAnchor	Load layout data
ConvertToAlignmentCode	TextAnchor → Number	Save layout data

Language Settings Data

This script defines a **ScriptableObject** used to centralize and store all configuration data required for localization in a Unity project. It acts as a **language manager database**, containing folder paths, default language settings, font references, available languages, and localized content.

🎯 Purpose

- Acts as a **configuration hub** for both editor tools and runtime systems.
- Stores everything needed to load and apply localized text and canvas layouts.
- Enables runtime language switching and editor-based language editing tools.

🧩 ScriptableObject: LanguageSettingsData

This is the main data container. It is stored as an asset in Unity's **Resources folder**, which allows it to be loaded at runtime.

📌 Variables and What They Do

🔷 folderName

- String indicating the subdirectory (usually in StreamingAssets or Resources) where language files (e.g., TSV or JSON) are stored.
- Default value is "Languages".

🔷 defaultLanguage

- Culture code for the language used when no other is selected (e.g., "en").
- Used as fallback on first launch or reset.

🔷 fontListData

- Contains a list of legacy Font assets used by Unity's built-in Text component.
- Fonts are assigned to localized elements using an index.

🔷 fontListDataTMP

- Similar to fontListData, but used with **TextMeshPro** (TMP_FontAsset) fonts.
- Allows support for modern UI components.

🔷 errorLanguageTool

- A GameObject reference (usually a prefab).
- Displayed on screen if something goes wrong during language loading (missing file, corrupted data, etc.).

🔷 selectedCulture

- Stores the **currently selected language code**, such as "pt-BR" or "fr-FR".
- Updated during runtime or edited manually in the editor.

🔷 availableLanguages

- A list of language entries the user can choose from.
- Each entry holds:
  - Display name,
  - Culture code,
  - Availability flag,
  - Table column index.

🔷 idData

- Holds localized text entries intended for **non-canvas** components (e.g., scripts, logs, dialogues).
- Each entry consists of an ID and the localized string.

🔷 idMetaData

- Stores extra formatting or context information for each localized ID.
- Used to define text alignment, font size, or usage type (like labels or buttons).

🔷 idCanvasData

- Similar to idData, but specifically for canvas-based UI elements (Text, TMP_Text).
- Stored separately to allow specialized handling and layout loading.

🧩 Editor Class: LanguageSettingsEditor

This is an editor-only helper that adds a **menu item** inside Unity to quickly open the LanguageSettingsData asset.

📌 **Method: OpenLanguageSettingsData()**

What it does:

- Loads the LanguageSettingsData asset from the Resources folder.
- Checks if the asset is already being edited in a Unity window.
- If not, opens the default property editor for the asset.

Usage:

- Accessed from Unity's top menu:
  Window > Language > Language Settings

🧪 **Example Usage Scenario**

1. **Designer sets up localization files** and defines fonts, languages, and metadata.
2. **ScriptableObject is configured** with default values and asset references.
3. At runtime:
   - The system loads LanguageSettingsData.
   - Checks which culture was previously selected.
   - Loads the correct localized text and canvas layouts.
4. In the Editor:
   - The developer opens the settings using the menu shortcut.
   - They can modify available languages or assign fallback fonts.

✅ **Summary Table**

Field	Type	Purpose
folderName	string	Directory for language files
defaultLanguage	string	Fallback language code
fontListData	asset	Fonts for UI.Text
fontListDataTMP	asset	Fonts for TMP_Text
errorLanguageTool	prefab	Shown when loading fails
selectedCulture	string	Currently selected language
availableLanguages	list	Language options
idData	list	Text for scripts/non-canvas
idMetaData	list	Visual and structural metadata
idCanvasData	list	Text used in canvas UI

Language Settings Data Editor

This script defines a **custom Unity Editor Inspector** for the LanguageSettingsData asset — the core configuration object in a multilingual Unity project. It enhances the default Unity Inspector by offering a more intuitive and organized way to **edit localization settings**, including language preferences, font lists, paths, and debug data.

🎯 **Purpose**

The goal of this script is to:

- Simplify editing of language configuration.
- Provide dropdowns for selecting system languages.
- Expose font lists for standard and TextMeshPro UI.
- Display runtime-extracted data in a read-only format.
- Integrate fully with Unity's undo and asset-saving systems.

This tool is **Editor-only** and only runs when inspecting a LanguageSettingsData asset.

🧩 **Core Components**

🔷 **availableCultures**

An internal array that stores **all system languages** supported by the .NET runtime, including regional variants like en-US, pt-BR, etc.

🔷 **availableCultureDisplayNames**

A string array of human-readable names (like "English (United States)", "Portuguese (Brazil)"), shown in the dropdown UI for language selection.

🔷 **currentSelectedCultureIndex**

An index tracking the currently selected default language inside the dropdown.

🔁 **Method Breakdown**

📦 **OnEnable()**

Purpose:

Initializes the dropdown options for default language selection.

What it does:

- Loads every available CultureInfo on the system.
- Converts them into English-readable names.
- Populates internal arrays to be used later by the Inspector GUI.

🖼️ **OnInspectorGUI()**

Purpose:

Draws the custom layout for the Inspector, replacing Unity's default view with a structured editor.

Main Features:

🗂️ **Section: Archives Location**

- Field: folderName
- Describes the subfolder used to load language files.
- Users can manually set or adjust the archive path.

🌐 **Section: Default Language**

- Uses a dropdown populated with availableCultures.
- Updates the defaultLanguage field in LanguageSettingsData with the selected culture code.

🔤 **Section: Font List Data**

- Fields: fontListData, fontListDataTMP
- Exposes object pickers to assign font containers (for both Unity UI and TextMeshPro).
- Uses **serialized property fields** to show and edit the font lists directly inside the Inspector.

❗ **Section: Canvas Log**

- Field: errorLanguageTool
- Lets the developer assign a GameObject prefab that will be shown if a language fails to load correctly.

📊 **Section: Extracted Data (Read-Only)**

These fields reflect runtime or imported data used by the language system.

- selectedCulture: Shows the currently selected culture in use.
- availableLanguages: List of all supported languages and their settings.
- idData: List of ID-to-text pairs for generic usage.
- idMetaData: List of layout/visual metadata for each ID.
- idCanvasData: List of text items specifically used in UI Canvas elements.

All these fields are shown using SerializedProperty rendering and are **non-editable** to avoid accidental changes.

💾 Auto-Save
If any user interface element is modified:
- The asset is marked as dirty.
- Unity's AssetDatabase.SaveAssets() is triggered to write the changes.

🧪 Example Usage
1. The user selects the LanguageSettingsData asset in the **Project window**.
2. A well-organized Inspector panel appears:
    - A dropdown to select default language.
    - Object fields to assign font configurations.
    - A text box to set the language archive folder.
    - A preview of runtime localization data (read-only).
3. If the user changes any values, the changes are immediately saved and tracked by Unity's Undo system.

✅ Summary Table

Section	Purpose	Editable
Archives Location	Set where language files are stored	✅
Default Language	Select system-supported culture	✅
Font List Data	Assign fonts for Unity UI and TMP	✅
Canvas Log	Assign error UI prefab	✅
Extracted Data	View current runtime localization structure	❌

Language Data Creator

This script defines **two Unity Editor tools** for managing the creation of a LanguageSettingsData asset — the central configuration file for the localization system. It provides both **manual and automatic** creation methods to ensure the asset is always present in the project.

🎯 Purpose
- Automatically ensures that a localization settings file exists (Language Data.asset).
- Lets developers create or replace it via the Unity menu.
- Ensures smoother onboarding and fewer errors when setting up the localization system.

🧩 Overview of Components
There are **two main classes** in this script:
1. LanguageDataCreator
   → Handles manual asset creation via a Unity menu.
2. LanguageDataAutoInitializer
   → Automatically creates the asset when the project is loaded if it doesn't exist.

🧩 Class: LanguageDataCreator
📌 Method: CreateLanguageDataAsset()
This method is triggered when the user clicks the menu option:
Assets > Create > Language > Language Data
What it does step-by-step:
1. **Defines target folder**:

    - Sets the destination as Assets/Resources — a Unity folder used for runtime loading.
2. **Ensures the folder exists**:
    - If the Resources folder is missing, it creates it automatically.
3. **Checks if asset already exists**:
    - Looks for an existing file at Assets/Resources/Language Data.asset.
    - If found, prompts the user with a **Yes/No dialog** asking if they want to replace it.
4. **Creates the asset**:
    - If confirmed or not present, it:
        - Instantiates a new LanguageSettingsData object.
        - Saves it to disk.
        - Marks it dirty so Unity registers the change.
        - Refreshes the AssetDatabase.
5. **Focuses the new asset**:
    - Selects and highlights the new asset in the Project window for easy access.

Usage:
Used manually to set up or replace the language data file during development.

🧩 Class: LanguageDataAutoInitializer

This class runs **automatically when the Unity Editor opens**.

📌 Static Constructor

- Registered using the [InitializeOnLoad] attribute, which ensures the code is run on project load.

What it does:

1. Waits until the editor is fully initialized (using EditorApplication.delayCall).
2. Checks if the file Assets/Resources/Language Data.asset exists.
3. If the file is **missing**, it calls CreateLanguageDataAsset() to create it automatically.

Usage:
Ensures the system is never missing its core configuration, reducing the risk of runtime errors or tool crashes.

✅ Summary Table

Feature	What It Does	Trigger
Manual Creation	Adds a Unity menu item to create or replace the asset	User opens the menu: *Assets* > *Create* > *Language* > *Language Data*
Automatic Initialization	Checks and creates the asset when the editor starts	Happens automatically on project load

📁 File Location

- The asset is always saved as: Assets/Resources/Language Data.asset

This ensures it can be loaded at runtime using Resources.Load() without requiring dynamic file path handling.

🧪 Real-World Use Case

When setting up a new project:

1. The developer imports the localization system.
2. The script **automatically creates** the needed configuration file if it's missing.
3. If needed, they can **manually regenerate or replace** the asset using the menu.
4. The localization tools then use this file to load fonts, languages, and other settings.

Language Font List Data / Language Font List Data TMP

This script defines a **data container and custom editor** in Unity that simplifies the management of a list of legacy Font assets (standard Unity Font objects) used in multilingual/localized projects. It consists of two major components:

Scriptable Object: LanguageFontListData

Purpose:

To store a list of legacy Unity fonts (Font) in a reusable and editable way, enabling centralized font management for localization systems.

Key Element:

- **fontList**:
  A list that holds Unity Font objects (like .ttf or .otf files).
  It is initialized as an empty list to avoid null reference issues.

Usage:

This object is created via Unity's **Assets > Create > Language > Language Font List Data (Legacy)** menu. Once created, it becomes a persistent asset that other systems (like language or font managers) can load and reference.

Custom Editor: LanguageFontListDataInspector

Purpose:

To provide a **drag-and-drop interface** inside the Unity Inspector window that allows developers to easily populate the fontList with font assets.

Features:

- **Single-object editing only**:
  The editor disables drag-and-drop interaction if multiple assets are selected to prevent inconsistent changes.
- **Drop zone UI**:
  A custom gray rectangular area labeled "Drop Font Assets Here" is drawn using GUILayoutUtility and EditorGUI.
- **Drag-and-drop handling**:
  - When the user drags one or more objects into the drop zone:
    - If the object is of type Font and not already in the list, it's added.
    - Changes are recorded in the Undo system, so users can revert the operation.
    - The asset is marked dirty to ensure the changes are saved.
- **Fallback to default inspector**:
  After the drag area, the normal inspector fields (like the editable list of fonts) are drawn.

Summary of Usage

1. The developer creates a LanguageFontListData asset in Unity.
2. The asset can be opened in the Inspector.
3. Fonts can be dragged from the Project panel and dropped directly into the drop area.
4. The list is updated, and changes are persisted for runtime use (especially for language-specific font switching).

Why This Is Useful

This tool improves workflow efficiency for managing multiple fonts in localization-heavy projects. It eliminates the need to manually reference fonts via scripting or click-intensive list editing. The drag-and-drop interface and undo support ensure fast, safe iteration directly inside Unity's Editor.

Language Initialization

This script is responsible for managing the initialization of language settings and associated font assets in a Unity application. It ensures that both regular fonts and TextMeshPro (TMP) fonts are correctly loaded from resources or asset bundles when the application starts, and that the appropriate language and font lists are applied.

🧩 Purpose and Use

This class runs automatically when the game launches (outside the Unity Editor) and performs essential setup for a multilingual application. It:

- Loads saved language preferences or defaults to the system culture.
- Loads and prepares font lists for both legacy Unity Text and TMP components.
- Saves and retrieves font data from text files.
- Loads fonts from asset bundles.

🔧 Core Variables

- **settingsData**: Stores the main localization settings loaded from the Resources folder.
- **fontListData**: Holds a list of Unity Font assets.
- **fontListDataTMP**: Holds a list of TMP font assets (TMP_FontAsset).
- **folderPath**: Path on disk where font data files and asset bundles are stored, usually inside the project's Assets/FontData/.

🚀 Initialization Process

InitializeLanguageSettings()

- Called automatically at runtime launch.
- Determines the folderPath for font data.
- Loads localization settings and font lists from the project's Resources.
- If not in the Unity Editor:
  - Ensures the font data folder exists.
  - Saves the current list of fonts to .txt files.
  - Loads .ltbundle and .tmpltbundle asset bundles.
  - Reconstructs the font lists from previously saved files.

📂 Language & Font Loading

LoadLanguageSettings()

- Fetches the LanguageSettingsData ScriptableObject.
- Initializes the variables that hold both regular and TMP font data.

SetupDefaultLanguage()

- Retrieves the current system language (like "en-US").
- Checks if the user has a saved culture preference.
- If not, attempts to use the system culture if it's available; otherwise falls back to the predefined default language.
- Updates the selected culture in settings.
- Loads all localization data tied to the selected language (like texts and canvas layouts).

📦 Asset Bundle Integration

LoadFontsFromAssetBundleLegacy()

- Looks for .ltbundle files inside the font data folder.
- Each file is treated as a Unity AssetBundle.
- Extracts and stores all Font assets from each bundle into the legacy font list.

LoadFontsFromAssetBundleTMP()

- Identical to the legacy version, but loads TMP fonts from .tmpltbundle files.
- Extracts and stores all TMP_FontAsset objects.

💾 Font List Persistence

SaveFontListsToFile()

- Creates two text files (one for each font type) listing the names of the fonts currently in memory.
- Ensures these files are only created if they don't already exist, avoiding overwrite.

LoadFontListsFromFile()

- If the font data folder exists and the list files are present:
  - Reads each line as a font name.
  - Filters the current in-memory font list to only include those listed in the file.
  - This step essentially ensures consistency with what the user previously saved or packed into bundles.

✅ Summary

This class acts as the **language and font bootstrapper** for the application:

- Loads user or system-preferred languages.
- Integrates and manages both traditional and TMP font assets from multiple sources.
- Provides fallback mechanisms for initialization failures.
- Saves and restores font data for use at runtime.

It ensures that the application always starts with correct fonts and localized settings, while maintaining high flexibility for asset management and platform compatibility.

Access Permission Checker

This script is a **runtime utility** designed to **check access permissions** for critical Unity project directories—specifically the **Assets** and **StreamingAssets** folders. If access is denied, it automatically displays an in-game warning using a predefined UI object from the language settings.

🧠 What the Script Does (Overview)

1. **Runs on game startup**.
2. **Attempts to write and delete a file** in two folders:
   - Assets
   - StreamingAssets
3. If either check fails, it **instantiates a warning UI**.
4. The warning is configured via the localization settings.

🔍 Breakdown of Variables and Methods

🔶 InitializeCheckSettings()

- Marked with [RuntimeInitializeOnLoadMethod], so it runs automatically when the game starts (before Start()).
- Calls CheckAccessPermissions().

🔶 CheckAccessPermissions()

- Calls CheckFolderAccess() for both:
  - Application.dataPath (the root Assets folder)
  - Application.streamingAssetsPath
- If either folder can't be accessed, it logs an error and calls ShowWarning().

🔶 CheckFolderAccess(string path)

- Accepts a folder path.
- Checks if the folder exists.
- Tries to:

1. **Write a temporary file** (TestAccessFile.tmp) to the folder.
2. **Delete that file**.

- If both operations succeed, returns true.
- If writing or deleting fails (due to permission issues), returns false and logs the error.

🔶 **ShowWarning()**

- Loads localization settings using the LanguageFileManager.
- Gets the warning prefab object from the setting: errorLanguageTool.
- Instantiates the warning GameObject.
- Uses DontDestroyOnLoad to ensure it remains visible across scene changes.
- Logs confirmation or failure.

🛠️ **Setup and Usage**

✅ **When to use:**

- On platforms where file system permissions might be restricted (e.g., mobile, consoles, or sandboxed OS environments).
- To ensure localized assets and configuration files can be read/written at runtime.

🔧 **Requirements:**

- A prefab UI GameObject should be assigned in the LanguageSettingsData.errorLanguageTool field. This acts as the warning UI.

📦 **Example Use Case**

Imagine your game downloads language data or loads localized assets from StreamingAssets. If permission is denied—perhaps due to platform rules or user settings—this script will **warn the player** through a UI popup, allowing developers to handle it gracefully rather than silently failing.

✅ **Summary**

Role	Purpose
InitializeCheckSettings	Automatically triggers the permission check on game start.
CheckFolderAccess	Attempts file creation/deletion to determine read/write permissions.
ShowWarning	Displays a persistent UI warning when permission is denied.
errorLanguageTool	Prefab used to visually warn users (loaded from language settings).

This script ensures a robust and user-aware experience in multilingual applications by preemptively catching critical I/O failures and informing users through a unified language-aware warning system.

Language Text / Language Text TMP

This script provides automatic localization for legacy UnityEngine.UI.Text components using the **LanguageTools** system. It updates the **text content**, **font**, **font size**, and **alignment** based on the current language configuration.

🧩 **Core Purpose**

The script attaches to a GameObject with a Text component and ensures that its content is:

- Translated into the currently selected language.
- Styled (font, size, alignment) according to localization metadata.
- Updated live when the language changes.

Variables Overview

Public & Serialized Fields

- **textComponent**
  The Text component that will display the localized content.
- **translateText**
  A flag indicating whether the system should replace the Text content with a translated string. If false, only style updates will occur.
- **iD**
  A unique identifier (typically negative for custom IDs) that maps to the correct entry in the localization database for both the text and its metadata.

Private Field

- **languageData**
  Stores the currently active language configuration loaded from a LanguageSettingsData ScriptableObject. It contains the full set of translation strings, fonts, and layout rules.

Lifecycle Flow

On Enable

- Subscribes to the OnLanguageUpdate event.
- Immediately calls LanguageUpdate() to apply the correct language and styling.

On Disable

- Unsubscribes from the language update event to prevent errors when the object is inactive.

Main Method: LanguageUpdate()

This is the core method that handles all localization logic for the Text component.

Step-by-step:

1. **Null Check**
   If textComponent isn't assigned, it logs an error and exits.
2. **Load Language Settings**
   Retrieves the current language data (e.g., "en-US" or "pt-BR") with translations and metadata.
3. **Apply Translation (if enabled)**
   Uses the stored iD to fetch the translated text from languageData.idData.
   If valid, updates the Text.text property.
4. **Apply Metadata**
   Retrieves metadata for font styling from languageData.idMetaData, including:
   - **Alignment** (horizontal/vertical via TextAnchor)
   - **Font size**
   - **Font asset** (from a list of fonts by index)

Editor Integration

Custom Inspector (LanguageTextEditor)

Adds an "Import Settings" button that allows developers to:

- Open a custom editor window.
- Automatically pull the current Text component's:
  - Text
  - Alignment (converted to internal code)
  - Font size
  - Font reference (as index in font list)
- Edit or save these values into the localization system.

Behavior:

- Prevents overwriting an existing ID unless the user explicitly approves.
- Ensures all values can be exported easily for translators or for saving to a localization database.

📌 Typical Use Case

1. Attach this script to a UI GameObject with a Text component (e.g., a button label).
2. Assign a unique ID and enable "translateText".
3. Use the editor to import and store the default text and style.
4. At runtime:
    - The system detects the selected language.
    - The component updates the text and appearance accordingly.

✅ Summary Table

Feature	Purpose
textComponent	UI text element to localize.
translateText	Toggles text translation on/off.
iD	Lookup key for localized text and style metadata.
LanguageUpdate()	Applies text, font, alignment, and font size.
Editor Button	Extracts UI values for localization entry.
Runtime Language Change	Auto-updates the UI when language changes.

This component is ideal for **legacy Unity UI projects** needing multilingual support while keeping the workflow smooth and editor-friendly.

Language Text Input Field / Language Text Input Field TMP

This script enables automatic localization for Unity's legacy InputField UI component using the **LanguageTools** system. It updates the placeholder text as well as font properties (alignment, size, and font asset) based on the currently selected language configuration.

🔍 Purpose and Usage

The component is designed to:

- Translate the **placeholder text** dynamically.
- Apply styling (font, size, alignment) from localization metadata.
- Integrate with Unity's event system to update when the user changes the language.

It works at **runtime** and can also be configured via a custom **Editor inspector**.

📦 Variable Breakdown

Component Fields

- **inputField**
  A reference to the Unity InputField component that this script will localize.
- **translateText**
  Boolean toggle. If true, the placeholder text will be translated using the LanguageTools system.
- **iD**
  An identifier used to locate both the translated string and the metadata (like font info) from the localization data.

Internal Fields

- **languageData**
  Holds the currently loaded LanguageSettingsData which includes the culture, font list, metadata, and available languages.

- **text**
  The actual text entry field of the InputField (where user input appears).
- **placeholder**
  The placeholder Text component inside the InputField, which displays localized instruction text.

🔄 Runtime Lifecycle

OnEnable()

- Subscribes to the LanguageManagerDelegate.OnLanguageUpdate event.
- Immediately calls LanguageUpdate() to localize the input field.

OnDisable()

- Unsubscribes from the language event to prevent dangling references when the object is deactivated.

🔧 LanguageUpdate() Logic

This is the method that performs the actual localization:

1. **Validation**
   Checks if inputField, placeholder, and text references are correctly assigned. Logs errors otherwise.
2. **Load Language Settings**
   Retrieves the LanguageSettingsData asset from resources.
3. **Apply Translated Text**
   If translateText is enabled:
   - It fetches a localized string using the iD.
   - If found, applies it to the placeholder.text.
4. **Apply Style Metadata**
   Pulls metadata for:
   - **Text alignment** (converted to Unity's TextAnchor)
   - **Font size**
   - **Font asset**

Then applies those to both the placeholder and text components.

🧰 Editor Features

Custom Inspector (LanguageTextInputFieldEditor)

Enhances the Unity Editor with a dedicated **“Import Settings”** button:

- Extracts:
  - Current placeholder text
  - Alignment
  - Font size
  - Font reference
- Opens a custom LanguageTools editor window with this data.
- Allows you to edit and save these values directly to the localization database.

The import action is protected by a confirmation dialog to avoid unintentional overwrites.

💡 Typical Workflow

1. Add this script to a GameObject that includes an InputField.
2. Assign the inputField field in the Inspector.
3. Enable translateText if you want the placeholder to be translated.
4. Assign a unique iD for the placeholder localization entry.
5. Use the **Import Settings** button to capture and store placeholder text and styling.

At runtime, when the user changes the language:

- The placeholder and text styling will update to match the selected language configuration.

✅ **Summary Table**

Element	Purpose
inputField	The Unity UI input field to be localized.
translateText	Determines if placeholder text will be translated.
iD	ID to look up localized text and styling metadata.
LanguageUpdate()	Applies all translations and style updates.
OnEnable()/OnDisable()	Subscribes/unsubscribes to language change event.
Custom Inspector Button	Allows easy import of current settings into the localization system.

This script provides essential localization support for input fields in legacy Unity UIs, making it easy to maintain multi-language experiences across your project.

Language Dropdown / Language Dropdown TMP

This script integrates a legacy Unity Dropdown UI component with the **LanguageTools** system, allowing it to dynamically translate its option texts and apply styling (font, font size, alignment) based on the current language settings. It's designed for **runtime localization** and includes **custom Editor support** for authoring and testing localized dropdown options inside the Unity Editor.

🔧 Core Features

- Automatically updates the dropdown's option texts based on selected language.
- Applies visual formatting from metadata: alignment, font size, and font.
- Synchronizes dropdown display when the language changes.
- Includes a custom Inspector for importing and registering option settings with the localization system.

🧩 Variables Explained

UI References

- **dropdown**:
  The Unity UI Dropdown component that will be localized.
- **captionText**:
  The text shown in the collapsed (selected) state of the dropdown.
- **itemText**:
  The text shown for each individual dropdown option when expanded.

Localization Data

- **translateText**:
  A toggle that controls whether to localize option texts using LanguageTools.
- **options**:
  A list of LanguageOptions, where each option has:
  - text: Display string.
  - sprite: Optional image.
  - iD: Unique identifier used to fetch localized text and styling.
- **languageData**:
  Loaded configuration from the LanguageSettingsData ScriptableObject, which includes available languages and style metadata.

🔄 Runtime Flow

OnEnable()

- Subscribes to language change notifications.
- Immediately applies localization by calling LanguageUpdate().

OnDisable()

- Unsubscribes from the language update delegate.

LanguageUpdate()

This is the main method responsible for updating the dropdown's appearance and content based on language settings:

1. **Validation**
   Ensures dropdown, captionText, and itemText are properly assigned. Logs errors otherwise.
2. **Load Settings**
   Loads language configuration using LoadLanguageSettings().
3. **Translate Options**
   If translateText is enabled, it calls UpdateLocalizedOptions() which updates the options list with translated text using the iD.
4. **Apply Styling**
   Retrieves styling metadata from the first option's ID:
   - Text alignment (converted to Unity's TextAnchor)
   - Font size
   - Font asset

These styles are applied to both captionText and itemText.

UpdateLocalizedOptions()

Handles the translation and repopulation of dropdown entries:

- Iterates through the options list.
- Replaces each text field with a localized string from the language data.
- Stores the current dropdown value.
- Clears and repopulates dropdown entries with translated texts and sprites.
- Restores the previous selection using SetValueWithoutNotify() to prevent triggering UI callbacks.
- Updates the captionText with the correct value.

🛠 Editor Integration

LanguageDropdownEditor

Provides a custom inspector interface for easier configuration:

- Adds a **"Import Settings"** button at the top of the Inspector.
- When clicked:
  1. Asks if the user wants to replace existing IDs (if any exist).
  2. Opens the **Language Editor Window**:
     - The first option is opened with styling info.
     - Remaining options are opened without styling info (text only).
- Prevents accidental edits when multiple objects are selected.

🧪 Typical Usage Flow

1. Attach this component to a GameObject that has a Unity Dropdown.
2. Assign the dropdown reference in the Inspector.
3. Add each dropdown option to the options list with its iD and display text.
4. Set translateText to true to enable automatic translation.
5. Use the **"Import Settings"** button to send the option texts and metadata into the localization system.
6. At runtime, the dropdown will update automatically when the language changes.

✅ Summary Table

Element	Purpose
dropdown	The target Unity Dropdown component to be localized.
translateText	Toggles whether to localize dropdown option texts.
options	Stores the options, their text, sprites, and localization IDs.
LanguageUpdate()	Applies localization and styling based on selected language.
UpdateLocalizedOptions()	Rebuilds dropdown with localized entries and restores selection.
captionText / itemText	Display elements for styling and translated text.
Editor script	Allows easy import of options into the localization tool.

This script brings full localization support to legacy Dropdown UI components and ensures both content and presentation are dynamically adapted to the selected language using LanguageTools.

Language Manager / Language Manager TMP

This script manages a **legacy Unity dropdown UI** for language selection and is part of a localization system using LanguageTools. It allows players to choose their preferred language, updates the application accordingly, and remembers their selection across sessions.

🔧 Purpose and Functionality

- Loads the list of available languages from project settings.
- Populates a Dropdown component with language names.
- Applies the selected language.
- Saves the user's choice to PlayerPrefs.
- Broadcasts an update to all localized components when a new language is selected.

📦 Variables and Their Roles

Serialized Fields

- **languageDropdown**
  A reference to the **legacy Dropdown** UI element. This is the main UI used by users to pick a language.

Private Runtime Fields

- **availableLanguages**
  A list of all languages that are available and marked as usable, pulled from the settings asset.
- **languageData**
  The current configuration loaded from the LanguageSettingsData resource, which includes default language, available languages, and font data.

🔄 Execution Flow

1. Start()

When the GameObject is initialized, this method:

- Checks if the dropdown is assigned.
- Loads the language settings asset.
- Retrieves the list of supported languages.
- Populates the dropdown with language names.
- Sets up the listener for language changes.

2. PopulateDropdown()

This method:

- Clears old dropdown options.
- Builds a list of display names for available languages.
- Compares each language's **culture code** with the one saved in PlayerPrefs.
- Sets the dropdown's value to match the saved language without triggering events (to avoid unnecessary updates).

Example:
If "en-US" is saved and present in the availableLanguages list, it will pre-select that.

3. SetupDropdownSelection()
Ensures only one listener is active by:
- Removing any previous listeners (to avoid double invocation).
- Adding the method OnLanguageChanged() as the listener for when the user selects a language.

4. OnLanguageChanged(int index)
Triggered when the user changes the dropdown selection:
- Checks if the index is valid.
- Gets the corresponding language's **culture code**.
- Saves that selection using PlayerPrefs.
- Reloads all language-related data.
- Broadcasts a language update event so other scripts like localized text or images can refresh themselves.

🌍 **Example Usage**
Assume you have a dropdown with:
- English (en-US)
- Portuguese (pt-BR)
- Spanish (es-ES)

After setting up this component:
- The dropdown will show these names.
- The selected option will persist between play sessions.
- When a player changes the selection, the UI and text components in the scene will update instantly to match the selected language.

✅ **Summary**

Element	Role
Dropdown	UI control to pick the language
LanguageSettingsData	Source of all available language info
PopulateDropdown()	Fills the dropdown based on supported languages
SetupDropdownSelection()	Hooks up the language-change listener
OnLanguageChanged(index)	Applies language change and triggers UI refresh
PlayerPrefs Integration	Saves and restores the selected language
Broadcast System	Triggers global update for all language-aware components

This component is ideal for multilingual Unity projects using **legacy Unity UI (Dropdown)** where a lightweight, persistent, and responsive language selector is needed.

Automatic Language Font Validator / Automatic Language Font Validator TMP

🎯 **Purpose and Use Case**
This script is designed to **automatically validate the font used in a UI Text component** in Unity to ensure that it supports **all characters** present in the text. If it finds unsupported characters, it will attempt to:

- Replace the font with a fallback one that does support them.
- If that fails and language management is enabled, replace the native language name with a more compatible fallback name.

🦴 Variables and What They Do

1. textComponent

This is the **UI Text element** that the script monitors. It must be assigned in the Inspector. The script will inspect the text displayed by this component to check if its font supports all characters.

2. isLanguageManager

This boolean toggle determines whether the script should also handle **fallback names** for languages. If enabled and the font cannot display a native language name (e.g., Chinese or Arabic), it will fall back to an alternative name that can be displayed properly.

3. supportedLanguages

A **list of available languages**, populated from the language settings when isLanguageManager is enabled. It is used to look up fallback names when font validation fails.

4. localizationSettings

Stores the app's **language and font configuration data**, loaded at startup. This includes the list of fallback fonts that might be used if the primary one doesn't support all characters.

5. lastValidatedText

Stores the **last text** that was checked to prevent unnecessary repeated validations every frame. It helps optimize performance by only running validation when the text actually changes.

🧠 Methods and Logic Flow

Start

- Runs once when the script is initialized.
- Verifies that the textComponent is assigned.
- Loads localization settings (which include fallback fonts).
- If isLanguageManager is enabled, also loads the list of supported languages.
- Performs an initial validation on the current text of the component.

Update

- Runs every frame.
- Checks whether the text has changed since the last frame using reference equality.
- If it changed, re-validates the new text.

ValidateFontSupport

This is the **core method** of the script.

- It first checks if everything is properly initialized (text component, localization settings, and font list).
- It inspects every character in the text to determine whether the current font can render it.
- If the font supports all characters, nothing is done.
- If it doesn't:
  - It loops through a list of fallback fonts.
  - For each font, it checks if **all characters** in the text are supported.
  - If a compatible fallback is found, it assigns that font to the text component.
- If no fallback font is suitable and language fallback is enabled:
  - It searches the supported languages for a match based on the **native language name** (which may contain unsupported characters).
  - If a match is found, it replaces the current text with a more compatible fallback name.

📌 Summary of Behavior

Condition	Action Taken
Font supports all characters	No change
Font does **not** support all characters	Try fallback fonts

No fallback font works and language manager is enabled	Replace native name with fallback name

✅ When to Use
Use this script if your application:
- Supports multiple languages with distinct character sets (e.g., Latin, Cyrillic, Chinese).
- Uses custom fonts that may not include all characters.
- Needs to **automatically resolve font-related display issues** without developer intervention.

Adjust Size To Dropdown / Adjust Size To Dropdown TMP

The **AdjustSizeToDropdown** script is a runtime UI utility designed to automatically resize and reposition a UI element (typically a dropdown container) based on the size and wrapping behavior of its text content.

🔧 Purpose
This component ensures that dropdowns using **legacy Unity UI Text** are fully visible and dynamically adjusted:
- Expands width when texts wrap into multiple lines.
- Realigns the dropdown left or right to avoid overflowing off the canvas.
- Automatically scrolls to the selected item.
- Adjusts scroll speed for long content lists.

🧩 Key Variables and Their Roles
- **manualSizeAdjustment**: A fixed value added to any automatic width resizing (used for spacing or compensation).
- **sizeMultiplier**: A value added when at least one text wraps to a second line or more.
- **margin**: How close the dropdown is allowed to get to the edge of the canvas before being repositioned.
- **fitDirection**: Default direction in which the dropdown should expand (either left or right).
- **parentRect**: The RectTransform of the current UI element being resized.
- **canvasRectTransform**: The RectTransform of the Canvas the dropdown is within.
- **scrollRect**: Reference to the ScrollRect component handling vertical scrolling of dropdown options.
- **canvasWidth / objectWidth**: Widths used for comparison when checking for overflow.
- **textList**: A list of all child Text elements in the dropdown, along with whether each one wraps lines.
- **textIsBroken**: True if any text in textList exceeds one line.
- **sizeAdjustment**: The calculated amount of extra width needed based on wrapping and manual adjustments.

🔄 Execution Flow
✅ Start (Initialization)
- Finds and stores the nearest Canvas in the hierarchy.
- Initializes parentRect and canvasRectTransform.
- Collects all child Text components and disables their vertical text cutoff.
- Each text is wrapped into a BrokenTexts entry with a flag indicating line wrapping.
- Adjusts the scroll sensitivity based on the number of dropdown items.
- Scrolls to ensure the currently selected option is in view.

🔄 Update (Each Frame)
- Checks for line wrapping on each Text.

- If any are wrapped, increases the size adjustment using sizeMultiplier.
- Applies both dynamic and manual adjustments to the dropdown width.
- Expands the element in the specified fitDirection using offsetMin or offsetMax.
- Converts the dropdown's position to the Canvas coordinate space.
- Detects if the dropdown overflows beyond the canvas boundary:
    - If going off to the **right**, it switches expansion to the **left**.
    - If going off to the **left**, it switches expansion to the **right**.

🔍 **CheckLineBreaks**

- Iterates over each Text object in textList.
- Uses Unity's internal text generator to determine if line breaks occurred.
- Flags that text as brokenText = true if it wrapped.

🎯 **ScrollToSelected**

- Looks for the currently selected dropdown item (based on the Toggle that's active).
- Calculates the vertical scroll position so that the selected item is visible.

📦 **Data Structure**

BrokenTexts

- itemLabel: The actual Text UI component.
- brokenText: A boolean indicating if the text spans multiple lines.

🧪 **Example Use Case**

You have a dropdown UI with long, localized texts. Depending on the language, some items may wrap or become wider. This script ensures:

- Texts are never cut off.
- Scroll sensitivity matches list length.
- The dropdown stays fully visible on-screen.

📌 **Summary Table**

Feature	Description
Auto-resize	Expands the dropdown container width based on content wrapping
Fit direction	Supports dynamic switching between expanding left or right
Scroll auto-adjust	Scrolls to the selected item automatically
Margin guard	Keeps dropdown from clipping at canvas edges
Manual adjustment	Allows extra spacing defined by the designer
Legacy UI support	Works with built-in Text components, not TextMeshPro

This script is particularly useful in **localized games or apps**, where text length varies by language, helping avoid layout issues without needing manual adjustments for each language.

Language Image

This script dynamically updates a **Unity UI Image** with a **localized texture** based on the currently selected language. It is part of a multilingual system where assets like images change depending on language settings (e.g., different flags, icons, or culturally adapted visuals).

🔍 **Purpose**

- Automatically replaces a UI image with a version specific to the **selected language** or **culture**.

- Ensures that only one load occurs per language switch, avoiding redundant operations.
- Loads the image **asynchronously** from disk using UnityWebRequest.

Variables and Their Roles

Configuration

- **fileName**
  Name of the image file to load (e.g., "Example.png"). Should match a real image file under the structured language directory.
- **useImage**
  Boolean toggle to determine if the Image component should be updated. If disabled, the image won't be visually applied.

UI References

- **image**
  Reference to the Unity UI Image component that should display the localized image.
- **imageTexture**
  Stores the raw Texture2D loaded from the file.
- **spriteTexture**
  A Sprite object created from the loaded texture, assigned to the Image component.

Internal State

- **languageData**
  Loaded language settings, which include the currently selected culture (e.g., "fr-FR" or "en-US").
- **previousFilePath**
  Remembers the last file path loaded. Prevents reloading the same image unnecessarily.
- **filePath**
  The full system path to the localized image file, built based on the current language.

How It Works

1. When the Object Is Enabled

- **OnEnable()**:
  - Subscribes to language change events via LanguageManagerDelegate.OnLanguageUpdate.
  - Immediately triggers LanguageUpdate() to apply the current language's image.

2. When the Object Is Disabled

- **OnDisable()**:
  - Unsubscribes from the language change delegate to avoid memory leaks or repeated updates.

3. Language Change Handler

- **LanguageUpdate()**:
  - Verifies that the Image component is assigned (if enabled).
  - Loads the current language data to get the active culture.
  - Constructs a full file path to the localized image:
  - {LanguageAssetsPath}/Assets/{Culture}/Image/{fileName}
  - Compares with previousFilePath. If it's the same, the function exits.
  - If a new image is needed:
    - Destroys previous Texture2D and Sprite to free memory.
    - Starts an asynchronous coroutine to load the new image.

4. Image Loading Coroutine

- **LoadFileCoroutine()**:
  - Uses UnityWebRequestTexture.GetTexture to load the image from the disk.
  - If loading fails, logs an error and stops.
  - On success:
    - Creates a Sprite using the loaded Texture2D.
    - If the image component is active, it assigns the new sprite to it.

📁 Example Use Case

You're developing a multilingual app where:

- **Buttons**, **icons**, or **labels** must appear in different styles depending on language.
- For example, "PlayButton_en.png" vs. "PlayButton_ja.png" for English and Japanese users.

In this case:

- Attach LanguageImage to a GameObject with an Image component.
- Provide a shared file name like "PlayButton.png".
- Place localized versions in:
- StreamingAssets/Assets/en-US/Image/PlayButton.png
- StreamingAssets/Assets/ja-JP/Image/PlayButton.png

The system loads the correct version at runtime based on the selected language.

✅ Summary Table

Feature	Description
Culture-aware file path	Builds dynamic image paths based on language
Async file loading	Uses UnityWebRequest to fetch image data from disk
Sprite conversion	Creates a Sprite from the loaded texture
Image assignment	Assigns the new sprite to a Unity UI Image component
Resource cleanup	Destroys old texture and sprite to avoid memory bloat
Language event integration	Reacts automatically when the language is changed

This script is a powerful utility for maintaining fully localized visual content across UI scenes, ensuring a seamless and professional user experience in multi-language Unity applications.

Language Raw Image

This script is designed to dynamically **load and display localized images** in Unity using the RawImage component. It ensures the correct image is shown based on the **selected language or culture**, updating automatically when the language changes.

🎯 Purpose

To support multilingual UI by loading **culture-specific image files** (like flags, backgrounds, or logos) and displaying them via a RawImage UI element. It works both during runtime and in multi-language projects.

🔧 Components and Their Roles

📁 Serialized Fields

- **fileName**
  The name of the image file to be loaded (e.g., "banner.png"). It is expected to exist in a language-specific subfolder.
- **useRawImage**
  If true, the script will assign the loaded texture to a RawImage component. If false, it will only load the texture but not apply it.

Public Fields

- **rawImage**
  A reference to the RawImage component on which the localized image will be shown.
- **imageTexture**
  Stores the texture loaded from disk, which is later assigned to the RawImage.

Private Variables

- **languageData**
  Holds the currently loaded language settings, including the selected culture (like "pt-BR" or "fr-FR").
- **previousFilePath**
  Caches the last image file path used. If the current path is the same, it avoids reloading the image.
- **filePath**
  The full system path to the localized image file. This is calculated based on the current language.

Lifecycle and Functionality

1. OnEnable()

- Subscribes to the **language update event**.
- Immediately calls LanguageUpdate() to ensure the correct image is shown when the object becomes active.

2. OnDisable()

- Unsubscribes from the language update event to prevent memory leaks or redundant operations when disabled.

3. LanguageUpdate()

This is the **core logic** method responsible for:

- Validating that a RawImage is provided if required.
- Loading current language settings.
- Building the absolute file path based on this structure:
- {LanguageAssetsPath}/Assets/{Culture}/Image/{fileName}
- Skipping the load if the image path hasn't changed.
- Verifying the file exists at that location.
- Destroying the previously loaded texture (to free memory).
- Starting the LoadFileCoroutine() to load the new image asynchronously.

4. LoadFileCoroutine()

This is a **Unity coroutine** that:

- Uses UnityWebRequestTexture.GetTexture to asynchronously fetch the image from the disk.
- Waits for the request to finish (yield return).
- If the request fails, logs an error and exits.
- If it succeeds:
  - Extracts the Texture2D from the request.
  - Assigns it to the RawImage.texture, if useRawImage is true.

Example Use Case

Let’s say you’re making a multilingual app and want to show different background illustrations for different languages.
You could:

- Create subfolders like:

- StreamingAssets/Assets/en-US/Image/background.png
- StreamingAssets/Assets/pt-BR/Image/background.png
- Attach this script to a GameObject with a RawImage and set:
    - fileName = "background.png"
    - useRawImage = true

When the language is switched in your app, the correct version of the background image is automatically loaded and displayed.

✅ Summary Table

Feature	Description
Language-aware image path	Builds dynamic file paths based on current culture
Memory-safe loading	Destroys previous textures before loading new ones
Async loading	Uses UnityWebRequest for non-blocking I/O
Component safety checks	Ensures required references are valid before use
Automatic language sync	Reacts to language changes via global delegate
Flexible usage	Can work with or without displaying the texture (useRawImage toggle)

This component is part of a broader **LanguageTools** localization system, helping to visually adapt a Unity project to different audiences by updating graphical assets in sync with language preferences.

Language Text Mesh / Language Text Mesh TMP

This script enables automatic translation and localization for legacy **TextMesh** components in Unity using the **LanguageTools** system. It dynamically updates the text content, font, and font size at runtime based on the currently selected language and associated metadata.

🔧 Purpose

The component is designed for **3D TextMesh objects**, enabling them to:

- Display localized text using a language ID.
- Automatically update when the language changes.
- Use appropriate font and size for the selected language.
- Be managed in the Unity Editor via a custom inspector.

🧩 Fields and Variables

🧱 Component Configuration

- **textComponent**
  Reference to the legacy Unity TextMesh that will display the localized text.
- **translateText**
  Determines whether the component should replace the text with a translated version using the assigned ID.
- **iD**
  A unique integer that identifies the text in the localization system. Used to fetch both the translation and metadata (e.g., font, font size).

🔒 Internal Use

- **languageData**
  Holds the loaded LanguageSettingsData, which contains selected culture, translations, fonts, and metadata for localization.

🔁 Runtime Workflow

▶ OnEnable()

- Subscribes this component to a global language update event.
- Calls LanguageUpdate() immediately so the text is correct on startup.

■ OnDisable()

- Unsubscribes from the language update event to prevent unnecessary calls or memory leaks when disabled.

🌐 LanguageUpdate()

This is the **main logic** that applies the localization to the TextMesh:

1. **Checks if the TextMesh is assigned**. Logs an error if it's missing.
2. **Loads the current language data** (selected language, font lists, and metadata).
3. If translateText is enabled:
    - Looks up the translated string using the iD.
    - Replaces the textComponent.text with the translated version (if found).
4. **Applies metadata**:
    - If fontSize is defined, updates the font size.
    - If fontListIndex is valid, loads the font from the legacy font list and assigns it to the TextMesh.
    - Also updates the font texture on the associated MeshRenderer to ensure the material is consistent.

🧪 Example Use Case

Imagine a 3D label or sign in your scene that should say "Welcome" in the player's selected language. Using this component:

- You set iD = 101, which corresponds to a multilingual entry.
- It updates automatically based on the player's chosen language.
- The font changes appropriately (e.g., to support Arabic or Japanese).
- Font size can also be adjusted per language for readability.

🧰 Editor Integration (Custom Inspector)

The script includes a Unity Editor inspector with enhanced functionality:

✨ LanguageTextMeshEditor

- Provides a custom UI in the Inspector when this component is selected.
- Adds a button: **"Import Settings"**:
    - Checks if the iD already exists.
    - If not (or if confirmed), it extracts:
        - The current text from the TextMesh.
        - The current font size.
        - The current font index (from the localization font list).
    - Opens the **Language File Manager Window** to save these values under that ID for localization.

This workflow simplifies translation setup and helps prevent manual errors.

📜 Summary Table

Field/Method	Description
textComponent	The legacy TextMesh to apply localization to
translateText	Enables translation from ID
iD	ID used to fetch translated text and font metadata
languageData	Current language settings and data loaded from configuration
OnEnable()	Subscribes to language change event and updates immediately

OnDisable()	Unsubscribes from language updates
LanguageUpdate()	Applies the translation, font, and font size to the TextMesh
LanguageTextMeshEditor	Unity Editor UI for importing text, size, and font index as translation metadata

✅ **When to Use**

Use this component if you're working with **legacy 3D text** in Unity (TextMesh, not TextMeshPro) and want to localize it dynamically with full control over appearance based on the language selected.

It integrates cleanly with the rest of the **LanguageTools** framework, offering both runtime flexibility and editor-time tooling for translation management.

Language Create File

This script is a **localization utility** for Unity that dynamically **creates text files containing translated or static text lines**, based on the active language settings. It supports use during **runtime** and inside the **Unity Editor**, making it useful for generating language-specific files such as instructions, logs, or subtitles.

🔧 **What the Script Does**

- **Generates a text file** (.txt or custom extension) in a specified folder.
- **Supports localization**: If a line is marked for translation, it fetches the corresponding translated text by its ID.
- **Runs updates automatically** when the language changes.
- Includes a **custom inspector UI** with a **live preview** of the file content.

🧩 **Main Script: LanguageCreateFile**

🔷 **Public Fields**

- **fileName**
  The name of the file to be created (e.g., "Test File").
- **fileExtension**
  The file extension (e.g., .txt, .log, .md).
- **folderInUnity / folderInBuild**
  Internal folders where the file is saved:
  - In **Editor**, it's saved under Assets/folderInUnity.
  - In **builds**, it's saved under StreamingAssets/folderInBuild.
- **fileLines**
  A list of lines to write into the file. Each line is a LanguageLines object, which includes:
  - The line's text
  - An optional **ID** used to fetch a translation
  - A flag translateText indicating whether to replace the line with a translated version.

🔷 **Private Field**

- **languageData**
  Holds the loaded localization settings (via LanguageSettingsData), containing current culture, available translations, and more.

🧠 **Method Breakdown**

▶️ **OnEnable()**

- Subscribes the object to language updates (LanguageManagerDelegate.OnLanguageUpdate).
- Immediately calls LanguageUpdate() to initialize the content.

⏹️ OnDisable()

- Unsubscribes from the language update delegate.

🌐 LanguageUpdate()

- Loads language settings using LoadLanguageSettings().
- Iterates through fileLines. If translateText is true, replaces the text with its localized version using the assigned iD.
- Calls CreateFile() to save the result as a file.

📂 GetFolderPath()

- Returns the correct directory path depending on whether the project is running in the Unity Editor or as a built app.
    - Uses Application.dataPath for Editor.
    - Uses Application.streamingAssetsPath for builds.

📝 CreateFile(string folderPath)

- Ensures the output directory exists.
- Writes all lines from fileLines into a file at the composed path using a StreamWriter.

🧩 Custom Inspector: LanguageCreateFileEditor

🔷 Key Features

- Adds a **button** to open an editor window for all translatable lines (Import Settings).
- Displays a **preview box** with the current content of the generated file.
- Provides a custom UI for managing and visualizing the content directly inside Unity's Inspector.

🛠️ Inspector GUI Workflow

- Import Settings lets the user edit each translatable line individually.
- A "File Preview" section displays:
    - File name (with extension)
    - Each text line, updated live

🧪 Example Usage Scenario

Imagine you are developing a multilingual game or app that:

- Saves localized **instructions** or **dialogue lines** to .txt files.
- These files are used by an external system or engine (like a TTS reader or dialogue parser).
- You can hook this component to an in-scene GameObject and automatically create updated files every time the language changes.

✅ Summary Table

Component	Role
fileLines	Holds static and translatable lines to write to the file.
LanguageUpdate()	Replaces lines with localized versions and writes the file.
CreateFile()	Handles the physical creation of the file on disk.
GetFolderPath()	Decides where to write the file depending on platform.
LanguageCreateFileEditor	Custom inspector that previews and imports content visually.

This tool bridges localization and file generation, ideal for exporting text assets based on selected languages. It ensures consistent formatting, supports translation IDs, and integrates seamlessly into Unity workflows.

Warning this component can be used for malicious purposes depending on the file format you define in its settings. It is your responsibility not to leave formats that can be used in malicious ways.

Language Script

This script is a component for managing localized text dynamically in a Unity scene. It connects to the localization system and updates strings in UI or 3D elements based on the active language. It works using assigned **IDs**, which map to translated strings defined in external settings. It also triggers **UnityEvents** so that other components can respond to these updates.

🔍 What This Script Does

- It lets you **assign text IDs** to elements (like UI labels or 3D text).
- When the language changes, it **fetches translations** for those IDs.
- It then **invokes UnityEvents**, passing the localized string to all linked targets (e.g., SetText() on a Text or TMP_Text component).
- It's designed to support **both in-editor setup** and **runtime behavior**.

🔧 Component: LanguageScript

🔸 Public Fields

- **debug**
  Enables console logging during updates to show which text is being applied and where.
- **scriptTexts**
  A list of ScriptText objects. Each one contains:
  - An iD: a unique numeric identifier for the localized string.
  - A text: the actual translated or placeholder text.
  - A UnityEvent<string>: a list of methods to call when the string updates.

🔸 Private Fields

- **languageData**
  Stores the current language configuration loaded from the system. This includes all the text and IDs available.

🔄 Life Cycle Events

✅ OnEnable()

- Called automatically when the component becomes active.
- Subscribes to the **language update event**.
- Immediately triggers a manual update to apply the correct translations.

❌ OnDisable()

- Unsubscribes from the language update event when the component is disabled.

🔄 Main Functionality

🌐 LanguageUpdate()

- This is triggered either on enable or when the language system notifies of a change.
- It does the following:
  1. Loads the language settings.
  2. For each ScriptText, it uses the iD to find the corresponding translation.
  3. Updates the text field with the translated value.
  4. Triggers all UnityEvents (targetScripts) attached to that ScriptText, passing in the localized text.
  5. Optionally logs the change if debug is true.

Support Method

ApplyUnityEvent(UnityEvent<string> unityEvent, string value)

- Loops through each method connected to the UnityEvent.
- Calls the method using reflection only if it accepts a **single string parameter**.
- This is how the system passes the translated text to other components (e.g., Text.text = value or SetSubtitle(value)).

Custom Inspector: LanguageScriptEditor

This section improves the user interface in the Unity Editor.

Features:

- Adds an **"Import Settings"** button:
  - When clicked, opens the language editor window for each ScriptText.
  - Lets users configure or replace the text assigned to each iD.
  - Prompts before overwriting an existing ID.
- Uses the default Unity inspector to show standard fields (debug, scriptTexts).

Usage Example

1. Attach this component to any GameObject that controls dynamic text (e.g., subtitle system, info popup, billboard).
2. Add entries to scriptTexts, each with:
   - An ID (linked to a translation).
   - A UnityEvent that passes the translated string to a component, such as TMP_Text.SetText.
3. When the language changes (either at runtime or through the editor), the UI updates instantly.

Summary Table

Element	Description
scriptTexts	Holds a list of localizable entries (ID + value + event).
LanguageUpdate()	Fetches translated text and applies it via events.
ApplyUnityEvent()	Calls all methods listening for updated text with the translated string.
OnEnable/OnDisable()	Hooks into the language update lifecycle.
LanguageScriptEditor	Adds editor tools to make editing and importing easier.

This script is essential for **runtime language switching**, ensuring your UI and text elements are synced with the current localization configuration, while giving you flexibility to trigger any custom behaviors (animations, logs, UI refreshes) when a translation is applied.

Language Audio Player

This script manages **language-specific audio playback** in Unity. It dynamically loads and plays a localized audio file depending on the player's selected language, supporting formats like WAV or MP3. It also integrates with Unity's AudioSource to play the audio automatically if configured.

Component Purpose

The component is used to:

- **Automatically play a localized audio file** when the scene starts or the language changes.
- **Load the correct version** of the audio file based on the selected culture (e.g., en-US, pt-BR).
- **Avoid redundant loading** by caching previously loaded paths.

- **Play audio** through a Unity AudioSource component if desired.

Fields (Variables)

Public / Serialized

- **fileName**
  The name of the audio file to load (e.g., "Intro.wav"). Should not include special characters.
- **audioType**
  The format of the file (e.g., WAV, MP3). Used by Unity to interpret the file type correctly.
- **useAudioSource**
  Boolean flag that determines whether the audio should be played through an AudioSource.
- **audioSource**
  Optional component that plays the loaded audio. Only required if useAudioSource is true.
- **audioClip**
  The audio file loaded from disk, stored in memory.

Private

- **languageData**
  A reference to the current language settings (which includes the selected culture).
- **previousFilePath**
  Caches the last loaded audio file path to prevent reloading the same clip multiple times.
- **filePath**
  The full path to the current audio file, based on the culture and folder structure.

Runtime Behavior

OnEnable()

- Subscribes to a language update event.
- Immediately calls LanguageUpdate() to apply localization as soon as the object is active.

OnDisable()

- Unsubscribes from the language update event to avoid memory leaks or errors when the object is disabled.

LanguageUpdate()

Main logic to update and play the localized audio:

1. **Checks prerequisites**:
   - Ensures AudioSource is assigned if it's going to be used.
   - Loads LanguageSettingsData to know the currently selected culture.
2. **Constructs the audio file path**:
   - Combines the base path, language culture folder, and Sounds subfolder.
   - For example: .../Assets/en-US/Sounds/Intro.wav
3. **Skips redundant loading**:
   - If this audio file was already loaded previously, it avoids reloading.
4. **Checks if file exists**:
   - Logs an error if the file is missing.
5. **Destroys the old clip**:
   - If another clip is already in memory, it clears it first.
6. **Starts a coroutine to load and play** the audio.

LoadFileCoroutine()

Handles asynchronous loading of the audio:

- Uses UnityWebRequestMultimedia.GetAudioClip to read the file from local disk.
- Waits for download to finish.
- Logs any error if the download fails.
- If successful:
    - Assigns the loaded clip to the audioClip field.
    - Plays it through the AudioSource if configured to do so.

📁 Folder & File Structure

Expected folder layout (for each language):

Assets/
- en-US/Sounds/Intro.wav
- pt-BR/Sounds/Intro.wav

The script will switch the file it loads based on the selected language's culture code.

🧪 Use Case

- You have audio instructions or dialogues recorded in multiple languages.
- You want Unity to **automatically select the correct version** based on the player's language.
- You want to **play it via AudioSource**, or manage the AudioClip manually for other uses (like syncing).

✅ Summary Table

Element	Description
fileName	Name of the audio file to load
audioType	Format of the audio file (e.g., WAV, MP3)
useAudioSource	Whether to play the audio immediately using an AudioSource
audioSource	Component responsible for playing audio
audioClip	Holds the currently loaded audio file
LanguageUpdate()	Handles file path resolution, validation, and triggers playback
LoadFileCoroutine()	Loads the clip from file and plays it if conditions are met

This script is ideal for **audio localization** in dialogue-driven games, tutorials, or voice-over UI systems, and integrates seamlessly into the rest of the LanguageTools system.

Language Canvas

This script provides a **runtime and editor solution** for managing **language-specific layouts of Unity canvas objects**. It allows a canvas to automatically update its structure (such as positions, sizes, and hierarchy) based on the selected language. It also offers tools within the Unity Editor to **capture and save canvas layouts** for later reuse.

🎯 Purpose

- Dynamically apply canvas layout changes when the language is updated.
- Associate a canvas with a unique ID for localization purposes.
- Enable designers to **save and update layouts** from within the Unity Editor using JSON.

🧩 Component Overview

This script is composed of two main parts:

1. **The Runtime MonoBehaviour** (LanguageCanvas)
2. **The Unity Editor Extension** (LanguageCanvasEditor)

🧩 MonoBehaviour: LanguageCanvas

Attached to canvas GameObjects in the scene. Handles runtime behavior and integrates with Unity's event system for language changes.

📌 Variables

- **canvasID**
  A unique number that links this canvas to its saved layout in the language system.
  Used to fetch language-specific canvas data.
- **canvasStructure**
  An object that stores the current layout of the canvas — including transform, hierarchy, and component metadata.
  It can be serialized into JSON or restored from JSON.

📌 Method: OnEnable()

- Automatically runs when the canvas is enabled in the scene.
- Subscribes the canvas to a global language change event.
- Immediately calls LanguageUpdate() to apply the correct layout.

📌 Method: OnDisable()

- Removes the canvas from the language change event subscription list.

📌 Method: LanguageUpdate()

- Called when a language change occurs.
- Retrieves the canvas layout data associated with canvasID from the language settings.
- Passes the JSON data to LoadCanvasData() to apply the layout.

📌 Method: LoadCanvasData(json)

Purpose:
Reads a JSON string and applies its content (canvas structure) to the GameObject.
Steps:

1. Validates that the input is not empty.
2. Parses the JSON into a CanvasStructure object.
3. If parsing is successful, stores the data and applies it to the canvas using a utility method.

Usage:
Used to switch canvas layout based on selected language.

📌 Method: SaveCanvasData() (Editor-only)

Purpose:
Used to extract and save the current layout of the canvas into a structured JSON string.
Steps:

1. Records an undo action (for editor support).
2. Extracts data from the canvas using a utility method.
3. Serializes the result into JSON and returns it.

Usage:
Allows designers to capture canvas configurations directly in the Unity Editor.

🧩 Editor Extension: LanguageCanvasEditor

This defines a custom UI for the LanguageCanvas component inside the Unity Inspector.

📌 Method: OnInspectorGUI()

Purpose:
Customizes the Inspector to provide buttons for managing canvas layout capture and editing.
Features:

- Shows a helper message about activating all canvas objects before importing layout data.
- Provides an **"Import Settings"** button that:
  - Checks if the canvas ID already exists.
  - Prompts the user before overwriting.
  - Captures the current canvas data and opens an editor window for further editing.
- Falls back to displaying the standard Unity Inspector fields for all properties.

Usage:
Gives designers interactive tools to save or update layout data from within the Unity Editor.

🧪 Example Workflow

1. A LanguageCanvas component is added to a UI canvas in a scene.
2. In the **Inspector**, the designer clicks **"Import Settings"**.
3. The tool captures the canvas layout and saves it under the given canvasID.
4. At runtime, when the app changes language, LanguageUpdate() is triggered.
5. The canvas automatically updates its layout using the saved JSON.

✅ Summary Table

Element	Role	When It's Used
canvasID	Links the canvas to language-specific layout data	Always
canvasStructure	Stores layout data for export/import	During save/load
LanguageUpdate()	Loads and applies canvas layout	On language change
LoadCanvasData(json)	Parses JSON and applies to canvas	At runtime
SaveCanvasData()	Captures canvas into JSON	In editor
LanguageCanvasEditor	Custom UI for capturing layouts	In editor

Rebuild Canvas

This script provides **Unity Editor tools** for managing and rebuilding UI canvas hierarchies using structured JSON data, typically in the context of multilingual applications. It enables you to:

- **Load canvas data** from JSON.
- **Recreate a canvas** in the scene using that data.
- **Save canvas structures** back to JSON for future reuse or editing.

🎯 Overall Purpose

This script is built to work inside the Unity Editor. Its main goal is to **bridge the gap between visual canvas layouts and saved data files**, making it easy to:

- Export UI designs.
- Import and regenerate canvases from configuration files.
- Edit and track canvas layouts per language or localization.

🧩 Part 1: MonoBehaviour - RebuildCanvas

This component is added to GameObjects in a Unity scene. It allows loading/saving of a canvas layout using Unity's built-in JSON serialization system.

📌 Public Variables

- **canvasObject**:
  A reference to the root GameObject representing the generated canvas hierarchy in the scene.
- **canvasStructure**:
  An in-memory object that represents the layout and metadata of the canvas (like positions, anchors, and scale settings), which can be serialized to or deserialized from JSON.
- **canvasID**:
  An identifier to associate this instance with a specific entry in a list of saved canvas layouts. Used for tracking and referencing.

📌 Method: CreateCanvasData(json)

Purpose:
Takes a JSON string that contains canvas data and regenerates the canvas in the scene.
How it works:

1. Checks if the input JSON is valid.
2. Deserializes it into a CanvasStructure object.
3. Deletes the current canvas in the scene (if any).
4. Uses utility logic (CreateCanvasFromStructure) to rebuild a canvas using the parsed structure.
5. Registers undo steps for editor integration.

Usage:
Called when a user chooses a saved layout to apply to the scene.

📌 Method: SaveCanvasData()

Purpose:
Reads the current canvas in the scene and converts it into a JSON string for saving.
How it works:

1. Uses ExtractCanvasData to turn the current canvas into a CanvasStructure.
2. Serializes that object into JSON.
3. Returns the result to be saved or reused.

Usage:
Called when the user wants to update the saved canvas configuration.

🧩 Part 2: Custom Editor - RebuildCanvasEditor

This is the custom Inspector interface that appears when a RebuildCanvas component is selected in the Unity Editor. It replaces the default UI with a more functional and user-friendly interface.

📌 Key Variables

- **fileData**:
  The path to the JSON file in Unity's ProjectSettings folder that stores saved canvas data.
- **canvasSave**:
  A list of canvas records loaded from the JSON file. Each item includes layout info, context, and a unique canvas ID.
- **script**:
  A reference to the currently selected RebuildCanvas component being edited.

📌 Method: OnInspectorGUI()

Purpose:
Defines what appears in the Inspector when the RebuildCanvas component is selected.
Key features:

- Loads saved canvas data from JSON.

- Displays a **dropdown** to select which canvas to load (by ID and context).
- Includes buttons to:
  - Rebuild the canvas in the scene from selected JSON.
  - Save the current canvas to overwrite the selected entry.
- Displays warnings and helper messages for better UX.

Usage:
Provides an editor interface to manage canvas extraction and regeneration, driven by JSON files.

📌 Method: LoadDataJson()

Purpose:
Reads the saved JSON file and loads all canvas entries into memory.

How it works:
- Checks if the JSON file exists.
- Deserializes its contents into LanguageFileManagerWindowData.
- Stores the list of canvases in canvasSave.

Usage:
Automatically called when the custom editor is drawn, to populate the dropdown menu with available options.

🧪 Workflow Example

1. **Author designs a canvas manually in Unity.**
2. Adds the RebuildCanvas component to a GameObject.
3. Presses **"Replace canvasID"** to extract the current canvas into JSON and save it.
4. Later, selects the saved canvas from the dropdown.
5. Presses **"Create Canvas Hierarchy"** to rebuild the layout from JSON.

This allows version-controlled, editable canvas layouts that can be tied to different languages or UI contexts.

✅ Summary Table

Element	Purpose	Type
canvasObject	Holds the generated canvas GameObject	Variable
canvasStructure	Stores layout and metadata of a canvas	Variable
canvasID	Links to saved layout entry	Variable
CreateCanvasData	Loads JSON → Creates canvas in scene	Method
SaveCanvasData	Extracts current canvas → JSON	Method
RebuildCanvasEditor	Custom Inspector for interaction	Editor class
LoadDataJson	Loads saved layouts from disk	Editor method

Language File Manager Window

The script defines a Unity Editor window tool for managing multilingual localization in a project. It enables developers to visualize, edit, and organize language-related data such as UI texts and canvas layouts. Here's a breakdown of how it works, its components, and intended usage — explained without code:

🧩 Purpose and Usage

This tool is accessed via the Unity menu as **"Window > Language > Language File Manager"**. It provides a user interface for managing localization files, primarily in .tsv and .json formats. It allows:

- Editing localized text components.
- Editing canvas structure data.
- Detecting duplicate IDs.
- Saving/Loading localization data from disk.
- Rebuilding UI canvases with localized layout data.

Data Managed

- **Text Components**: Stored as a list of localized text entries, each with an ID, text, font settings, alignment, and type.
- **Canvas Components**: UI layout configurations stored as JSON strings mapped to unique canvas IDs.
- **Languages**: List of available languages with culture info and availability status.

Main Variables

- languageForEditing: Indicates which language is currently being edited.
- componentSave: Holds text components and their metadata.
- canvasSave: Holds canvas layout data as JSON.
- availableLanguages: Languages that are detected and can be edited.
- scrollPosition: Scroll state of the editor window.
- firstTime: Marks whether the window just opened (initialization step).
- idIndex: Currently selected ID for preview or editing.
- showTextData, showCanvasData: Toggles to show/hide data sections.
- fileIsSaved: Tracks whether the data needs to be saved.
- fileData: Path to the JSON file used for saving editor state.
- duplicateID, canvasID: Lists of IDs detected as duplicates.

Editor Interface Functions

- **OnEnable**: Initializes state, loads culture info, and file paths.
- **OnGUI**: Renders the editor window UI and handles user interaction.
- **DrawActionButtons**: Shows buttons for opening folders, saving/loading, rebuilding canvas, etc.
- **DrawLanguageDropdown**: Allows switching the editing language with culture names.
- **DrawIdDisplayPanel**: Displays details for the selected ID and duplicate detection.
- **RenderComponentDisplay**: Renders editable fields for each text component.
- **DrawButton**: Utility method to create a styled button with action binding.

File Operations

- **SaveLanguageFile**:
  - Sorts entries.
  - Validates data (e.g., duplicate IDs).
  - Builds .tsv tables for canvas, language text, and metadata.
  - Writes to files after confirming with the user.
- **LoadLanguageFile**:
  - Parses .tsv files into structured data.
  - Rebuilds text and canvas entries.
  - Loads and applies metadata like font size, alignment, etc.
- **SaveDataJson / LoadDataJson**:
  - Persists UI state, like scroll position and selected ID, in a .json file inside ProjectSettings.

🔄 Utility Functions

- **CompareID**: Sorts the component and canvas entries by their IDs.
- **FindDuplicateIds**: Identifies duplicate IDs and stores them for display and warning.
- **RemoveID**: Deletes the selected ID after optional user confirmation.
- **GetAllavailableLanguages**: Reads all the languages listed in a .tsv file and updates the dropdown list.
- **AddComponent / AddCanvas**: Adds or replaces an entry in the list and updates the .json state.

🎨 Visual Elements

- Uses color-coded boxes and labels to distinguish data groups.
- Draws tooltips and trash icons for better UX.
- Adds real-time Undo/Redo support.

✅ Summary

This script serves as a robust **localization management tool inside Unity**, allowing real-time editing, validation, and organization of multilingual content. It integrates tightly with Unity's serialization, undo system, and file handling mechanisms, making it highly suitable for multilingual UI-heavy projects.

Language Prefab Creator

This script defines a powerful and automated **Editor utility** for Unity that helps developers **quickly instantiate language-related prefabs**—for both 3D objects and UI components—directly from the Unity Editor menu. It streamlines the creation process, including **Canvas setup**, **Event System creation**, **Undo support**, and **Prefab unpacking** for immediate editing.

🎯 Purpose

- Allow language-related prefabs (like multilingual text, toggles, input fields) to be created via menu commands.
- Automatically **set up necessary UI infrastructure** (like Canvas and EventSystem).
- Make prefabs editable immediately by **unpacking** them.
- Integrate with Unity's **Undo and Selection systems** for user-friendly editing.

🧩 Key Functional Areas

There are **three core types of functionality**:

1. **Utility functions** (Canvas creation, prefab instantiation, configuration)
2. **Prefab menu integrations** for 3D objects
3. **Prefab menu integrations** for UI (both TMP and Legacy)

🔧 Core Utility Methods

🔷 CreateUICanvas()

Creates a complete UI Canvas setup **if one doesn't already exist**.

- Creates a GameObject named "Canvas" with:
  - Canvas component (set to ScreenSpaceOverlay)
  - CanvasScaler
  - GraphicRaycaster
- Adds a separate EventSystem object with:
  - EventSystem and StandaloneInputModule
- Registers both objects with Unity's Undo system so they can be reverted.

Usage: Automatically called when a UI prefab is created and no canvas exists.

🔷 **CreateAndConfigurePrefab(fileName, selectedGameObject, isUI)**
Handles the full process of **instantiating and configuring** a prefab.
Parameters:
- fileName: Name of the prefab (e.g., "Toggle (LT) [TMP]")
- selectedGameObject: The currently selected object in the Hierarchy to use as a parent
- isUI: Whether this prefab is for UI (if true, ensures a Canvas exists)

Steps:
1. Ensures there's a Canvas if it's a UI prefab.
2. Locates the prefab by name using FindPrefabByName() (assumed utility method).
3. Instantiates the prefab under the correct parent.
4. Calls FinalizePrefabSetup() to complete configuration.

🔷 **FinalizePrefabSetup(fileName, newGameObject)**
Finalizes newly created prefab instances:
- Registers the object with Undo system.
- Unpacks the prefab so it becomes a regular editable GameObject (not a prefab instance).
- Automatically triggers **rename mode** in the hierarchy using Unity's delayed call system.

Usage: Called after prefab is created, ensuring smooth integration into the editor workflow.

🧩 **Menu Integration: Prefab Creation Shortcuts**
The [MenuItem] attributes bind functions to Unity's **top menu bar**, under:
- **GameObject > Language > 3D Object**
- **GameObject > Language > UI**
- **GameObject > Language > UI > Legacy**

Each of these functions calls CreateAndConfigurePrefab() with a specific prefab name.
✅ **Example Menu Items:**
📦 **3D Prefabs:**
- Language Create File (LT)
- Language Script (LT)
- Audio Source (LT)
- New Text (LT) [Legacy]
- New Text (LT) [TMP]

🖼️ **UI (TMP):**
- RawImage (LT)
- Image (LT)
- Language Manager (LT) [TMP]
- Dropdown (LT) [TMP]
- InputField (LT) [TMP]
- Toggle (LT) [TMP]
- Button (LT) [TMP]
- Text (LT) [TMP]

📋 **UI (Legacy):**
- Language Manager (LT) [Legacy]
- Dropdown (LT) [Legacy]
- InputField (LT) [Legacy]
- Toggle (LT) [Legacy]
- Button (LT) [Legacy]
- Text (LT) [Legacy]

🧪 **Example Usage**
1. A developer opens Unity and wants to create a localized button.

2. From the menu:
   GameObject > Language > UI > Button (LT) [TMP]
3. The script:
   - Checks if a Canvas exists (creates one if needed).
   - Instantiates the TMP button prefab under the canvas.
   - Unpacks it for editing.
   - Highlights and selects it, ready to rename and configure.

✅ **Summary Table**

Method	Purpose	Editor Integration
CreateUICanvas()	Creates Canvas + EventSystem	Internal only
CreateAndConfigurePrefab()	Loads and instantiates prefab under a parent	Called by menu handlers
FinalizePrefabSetup()	Unpacks prefab, enables undo and rename	Internal only
MenuItem: CreateTMPButtonPrefab() (and others)	Adds prefab creation to the Unity menu	Public menu commands

Component Converter

This script defines a **Unity Editor tool** that automatically **converts standard Unity UI and 3D components** into **LanguageTools-compatible components**. It's designed for localization workflows and simplifies the process of making GameObjects compatible with the multilingual system.

🎯 **Purpose**

- Converts common Unity components (like Text, Image, Button, TMP_Text, etc.) into their LanguageTools equivalents.
- Ensures compatibility with a language management system for dynamic text, images, and audio.
- Handles **Undo support**, **prefab safety**, and **duplicate prevention**.
- Operates on **multi-selection** via a single menu command.

🧩 **How It Works**

🔧 **Trigger: Menu Command**

- The conversion process is launched via Unity's top menu:
  GameObject > Language > Converter to Language Tool
- Method: ConvertComponents()
  This is the **entry point**, triggered when the menu item is clicked.

🔢 **Internal Counters**

- callCount and expectedCalls are used to avoid **multiple executions** when Unity calls the menu handler multiple times during multi-selection.
  This ensures the conversion logic only runs once per batch.

🔁 **Main Conversion Logic**

🔷 **ConvertComponents()**

1. Collects all selected GameObjects.
2. Waits until Unity finishes triggering all handler calls for selection.
3. Iterates over each object and calls ObjectAnalysis().

🔷 ObjectAnalysis(GameObject)

This method performs **component detection** and selectively calls the appropriate conversion method. It checks for:

- Unity UI components: Text, Image, RawImage, Button, Toggle, Dropdown, InputField
- TextMeshPro components: TMP_Text, TMP_Dropdown, TMP_InputField
- 3D text/audio: TextMesh, AudioSource, MeshRenderer

Each valid component is passed to a specialized conversion method.

🔁 Individual Component Conversions

Each ConvertX() method:

1. **Checks if the LanguageTools component already exists** to prevent duplicates.
2. **Adds the new component** using Undo.AddComponent().
3. **Assigns the original Unity component** (like text, image, etc.) to a reference field in the new component.
4. Registers the creation with Unity's Undo system.
5. Logs the result to the console.

📌 Examples

Component Type	LanguageTools Equivalent
Text	LanguageText
TMP_Text	LanguageTextTMP
TextMesh	LanguageTextMesh
TMP + MeshRenderer	LanguageTextMeshTMP
Image	LanguageImage
RawImage	LanguageRawImage
AudioSource	LanguageAudioPlayer
Dropdown	LanguageDropdown + AdjustSizeToDropdown
TMP_Dropdown	LanguageDropdownTMP + AdjustSizeToDropdownTMP
InputField	LanguageTextInputField
TMP_InputField	LanguageTextInputFieldTMP

🧠 Additional Utilities

🔷 ConvertTextComponent<T>()

Used to convert Button and Toggle by scanning their children for Text or TMP_Text.

🔷 Dropdown Conversion

- Creates a list of LanguageOptions from dropdown choices.
- Ensures the dropdown's template also gets a sizing component (AdjustSizeToDropdown*).

🧪 Example Workflow

1. A developer selects several UI elements in the hierarchy (e.g., a button, an image, and a TMP text).
2. From the top menu:
   GameObject > Language > Converter to Language Tool
3. The script:
   - Detects all relevant components.
   - Adds the appropriate LanguageTools wrapper components.
   - Logs messages and errors as needed.
4. The converted objects now support localization features (e.g., dynamic text updates, switching language, etc.).

✅ **Summary Table**

Feature	Purpose
ConvertComponents()	Starts the conversion process
ObjectAnalysis()	Detects and dispatches components for conversion
ConvertX() methods	Add and link LanguageTools wrappers
Undo.Register...	Ensures all operations are undoable
Logging	Reports success or duplicate detection

Font Asset Bundle Builder

This script defines an **Editor utility for Unity** that automatically builds **AssetBundles** from Font and TMP_FontAsset assets found in a specific project folder called "AssetBundles". These bundles can later be used to dynamically load fonts at runtime — an essential feature for localized applications supporting multiple languages and font styles.

🎯 Purpose

- Converts font assets into **AssetBundles**, which can be **dynamically loaded** during gameplay.
- Separates regular Unity fonts and TextMeshPro (TMP) fonts into distinct bundles.
- Uses **chunk-based compression** and applies appropriate file extensions for identification.
- Ensures asset integrity and provides developer feedback with logs.

🧩 Usage Overview

To use this tool:

1. Place Font or TMP_FontAsset assets inside a folder named **"AssetBundles"** in the **Unity project**.
2. Use the Unity menu:
   Assets > Build Font Asset Bundles [Language Tool Unity]
3. Unity will:
   - Locate all relevant font assets.
   - Create an individual AssetBundle for each.
   - Save them in the same "AssetBundles" folder.
   - Print success or error messages to the Console.

🔧 Main Components

🔷 BuildFontAssetBundles()

Triggered from the menu.
It's the entry point that initiates the bundle creation process.
Steps:

1. Searches the project for a folder named **"AssetBundles"** using AssetDatabase.FindAssets("AssetBundles t:Folder").
2. Converts the folder's GUID to a usable path.
3. Finds all:
   - **Standard Fonts** (t:Font)
   - **TMP Fonts** (t:TMP_FontAsset)
4. Calls BuildFontAssetBundle() for each asset found.
5. Calls AssetDatabase.Refresh() to make sure Unity sees the new bundles.

🔷 BuildFontAssetBundle(fontGuid, outputPath, isTMP)

Processes a single font asset and builds an AssetBundle for it.
Parameters:

- fontGuid: Unique ID for the asset in the Unity database.
- outputPath: Destination path for the generated bundle.
- isTMP: Whether the font is a TextMeshPro font.

What it does:

1. Converts the GUID to an actual file path (assetPath).
2. Checks if the file has a valid font extension (.ttf, .otf, .ttc) if it's not a TMP font.
3. Builds a custom bundle name:
    - Ends in .ltbundle for standard fonts
    - Ends in .tmpltbundle for TMP fonts
4. Prepares an AssetBundleBuild object describing what to bundle.
5. Uses BuildPipeline.BuildAssetBundles() with:
    - **Chunk-based compression**
    - **Windows standalone target**
6. Logs success or error.

🔷 IsValidFontFile(extension)

Returns true if the extension is a valid font type:
.ttf, .otf, .ttc

📂 Output

Each font or TMP font asset becomes a **separate file** in the "AssetBundles" folder:

- Regular fonts: FontName.ltbundle
- TMP fonts: FontName.tmpltbundle

These can later be loaded at runtime using Unity's AssetBundle.LoadFromFile() method.

✅ Summary Table

Element	Description
Menu Command	Adds option under **Assets > Build Font Asset Bundles [Language Tool Unity]**
Asset Folder	Must be named "AssetBundles"
Supported Types	Unity Font, TextMeshPro TMP_FontAsset
Output Format	.ltbundle (regular fonts), .tmpltbundle (TMP fonts)
Compression	Chunk-based
Undo/Errors	Logs conversion status and errors in the Console

🧪 Example Scenario

A multilingual project needs to support:

- Arabic (with its own .ttf)
- Japanese (with TMP fallback font)

1. The developer places these files inside the "AssetBundles" folder.
2. Runs the menu command.
3. Two bundles are created:
    - arabic-font.ltbundle
    - japanese-font.tmpltbundle

These can be shipped separately or downloaded as needed.