SpreadsheetEasy

SpreadsheetEasy is a powerful Unity asset that simplifies reading and accessing Spreadsheet files (.xlsx, .xls) in your Unity projects. It provides an intuitive API for querying Spreadsheet data by column names, making it perfect for game configuration files, localization data, and other data-driven content.

Features

- **Easy Spreadsheet Access**: Read Spreadsheet files with simple API calls
- **Type-Safe Queries**: Query data by column names instead of cell coordinates
- **Multiple Data Types**: Support for string, int, float, double, Vector3, Color, and arrays
- **TryGet API**: Clean error handling without try-catch blocks - returns bool and outputs result via out parameter
- **Row-Based Access**: Get entire rows as `SpreadsheetRow` objects for efficient multiple value retrieval
- **JSON Export**: Convert Spreadsheet data to JSON format for easy integration
- **Runtime & Editor Support**: Works in both Editor and runtime builds
- **High Performance**: Index cache system provides 50-100x speedup for frequent queries
- **Automatic Caching**: Data and indices are cached after first load for optimal performance
- **JSON Validation**: Validate JSON format in Spreadsheet cells with automatic detection and comprehensive error reporting
- **Spreadsheet Data JSON Checker**: Standalone JSON validation tool that validates JSON format in Spreadsheet files directly from any folder

Requirements

- Unity 2020.3 or later
- ExcelDataReader (included)

Quick Start

1. Setup

Place your Spreadsheet files (`.xlsx` or `.xls`) in the StreamingAssets folder:

StreamingAssets Folder

- Create a `StreamingAssets` folder: `Assets/StreamingAssets/`
- Unity supports only one StreamingAssets folder, but you can organize files in subdirectories
- Place Spreadsheet files directly in `Assets/StreamingAssets/` or any subdirectory
- Example:
  - `Assets/StreamingAssets/Config.xlsx` (or `.xls`)
  - `Assets/StreamingAssets/GameData/Items.xlsx` (or `.xls`)
  - `Assets/StreamingAssets/GameData/Enemies.xls` (or `.xlsx`)

2. Load Spreadsheet Files

In your code, call `LoadAllSpreadsheet()` to load all Spreadsheet files:

```
using SpreadsheetEasy;

void Awake()
{
    // Load all Spreadsheet files from StreamingAssets
    SpreadsheetReader.LoadAllSpreadsheet();
}
```

Optional - Lazy Loading: If you have many Spreadsheet files and don't want to load them all at once, you can skip calling `LoadAllSpreadsheet()` . When you use any query method (like `GetInt()` , `GetRow()` , etc.), SpreadsheetEasy will automatically load the required file on first access and cache it for subsequent queries. The first read operation will be slightly slower, but all subsequent queries will use the cached data for optimal performance.

Note: `LoadAllSpreadsheet()` automatically scans:

- **StreamingAssets folder** (including subdirectories) - Works in both Editor and Runtime
  - Unity has only one StreamingAssets folder (`Assets/StreamingAssets/`)
  - You can organize files in subdirectories within StreamingAssets

Important: File Name Requirements

WARNING: File names must be unique across all subdirectories!

SpreadsheetEasy uses file names (without path and extension) as cache keys. This means:

- **Allowed**: Different paths, same file name
  - `StreamingAssets/Config/PlayerConfig.xlsx` → cached as `"PlayerConfig"`
  - `StreamingAssets/GameData/PlayerConfig.xlsx` → cached as `"PlayerConfig"` (**CONFLICT!**)

If you have multiple Spreadsheet files with the same name in different subdirectories, only the first one loaded will be accessible. The later files will be ignored or overwrite the cache!

Best Practice: Ensure all Spreadsheet file names are unique, regardless of their folder location:

- **OK**: `StreamingAssets/Config/PlayerConfig.xlsx` and `StreamingAssets/GameData/EnemyConfig.xlsx` (different names)
- **OK**: `StreamingAssets/Config/PlayerConfig.xlsx` and `StreamingAssets/GameData/PlayerData.xlsx` (different names)
- **ERROR**: `StreamingAssets/Config/Config.xlsx` and `StreamingAssets/GameData/Config.xlsx` (same name - conflict!)

File Name Normalization: SpreadsheetEasy automatically normalizes file names by removing paths and extensions:

- `"SpreadsheetEasy/PlayerConfig.xlsx"` → `"PlayerConfig"`
- `"PlayerConfig.xlsx"` → `"PlayerConfig"`
- `"PlayerConfig"` → `"PlayerConfig"`

This means you can use any of these formats when querying, and they will all resolve to the same cache key:

```
using SpreadsheetEasy;

// All these are equivalent:
SpreadsheetReader.GetString("SpreadsheetEasy/PlayerConfig", "Players",
"ID", "P001", "Name");
SpreadsheetReader.GetString("PlayerConfig.xlsx", "Players", "ID",
"P001", "Name");
SpreadsheetReader.GetString("PlayerConfig", "Players", "ID", "P001",
"Name");
```

3. Basic Usage

Important: The first row of your Spreadsheet sheet must contain column headers (column names). Data rows start from the second row.

Example Spreadsheet Structure:

ID	Name	Rarity	HP	Speed	Position	Rotation	Color	Items	Scores	Tags	Active
001	Player1	Epic	100	5.5	10,20,30	0,90,0	#FF0000	1,2,3	95.5,88.0,92.3	warrior,tank	true
002	Player2	Rare	80	6.0	15,25,35	0,180,0	#00FF00	4,5,6	78.2,85.1,90.0	mage,healer	false

```
using SpreadsheetEasy;

// Get a string value
string playerName = SpreadsheetReader.GetString("PlayerConfig",
"Players", "ID", "P001", "Name");
```

```
// Get an integer value
int health = SpreadsheetReader.GetInt("PlayerConfig", "Players", "ID",
"P001", "HP");

// Get a float value
float speed = SpreadsheetReader.GetFloat("PlayerConfig", "Players",
"ID", "P001", "Speed");

// Get a bool value (supports: true/false, 1/0, yes/no, case-
insensitive)
bool isActive = SpreadsheetReader.GetBool("PlayerConfig", "Players",
"ID", "P001", "Active");

// Get an Enum value (supports string name case-insensitive or integer
value)
// Note: GetEnum<T>() returns only the first matching result
//       Use GetAllEnums<T>() to get all matching results when multiple
rows match
public enum PlayerRarity { Common, Rare, Epic, Legendary }
PlayerRarity rarity = SpreadsheetReader.GetEnum<PlayerRarity>
("PlayerConfig", "Players", "ID", "P001", "Rarity");

// Get all Enum values when multiple rows match the search condition
List<PlayerRarity> allRarities =
SpreadsheetReader.GetAllEnums<PlayerRarity>("PlayerConfig", "Players",
"HP", "100", "Rarity");

// You can also use Enum.ToString() as search value
string playerName = SpreadsheetReader.GetString("PlayerConfig",
"Players", "Rarity", PlayerRarity.Epic.ToString(), "Name");

// Get a Vector3 (format: "x,y,z" or "x;y;z")
Vector3 position = SpreadsheetReader.GetVector3("PlayerConfig",
"Players", "ID", "P001", "Position");
Vector3 rotation = SpreadsheetReader.GetVector3("PlayerConfig",
"Players", "ID", "P001", "Rotation");
```

```
// Get a Color (supports hex #RRGGBB, #RRGGBBAA, or comma/semicolon-
separated r,g,b,a or r;g;b;a)
Color playerColor = SpreadsheetReader.GetColor("PlayerConfig",
"Players", "ID", "P001", "Color");

// Get an int array (format: "1,2,3" or "1;2;3")
int[] items = SpreadsheetReader.GetIntArray("PlayerConfig", "Players",
"ID", "P001", "Items");

// Get a float array (format: "1.5,2.5,3.5" or "1.5;2.5;3.5")
float[] scores = SpreadsheetReader.GetFloatArray("PlayerConfig",
"Players", "ID", "P001", "Scores");

// Get a string array (format: "a,b,c" or "a;b;c")
string[] tags = SpreadsheetReader.GetStringArray("PlayerConfig",
"Players", "ID", "P001", "Tags");
```

Using TryGet Methods (No try-catch needed):

```
using SpreadsheetEasy;

// TryGet methods return bool and output the result via out parameter
// No need for try-catch blocks - cleaner code!

if (SpreadsheetReader.TryGetInt("PlayerConfig", "Players", "ID", "1001",
"HP", out int hp))
{
    Debug.Log($"HP is {hp}");
}
else
{
    Debug.Log("HP not found or invalid format");
}
```

```
// Works with all data types
if (SpreadsheetReader.TryGetString("PlayerConfig", "Players", "ID",
"P001", "Name", out string name))
{
    Debug.Log($"Player name: {name}");
}

if (SpreadsheetReader.TryGetFloat("PlayerConfig", "Players", "ID",
"1001", "Speed", out float speed))
{
    Debug.Log($"Speed: {speed}");
}

if (SpreadsheetReader.TryGetBool("PlayerConfig", "Players", "ID",
"1001", "Active", out bool isActive))
{
    Debug.Log($"Item is active: {isActive}");
}
```

4. SpreadsheetRow for Multiple Values

When you need multiple values from the same row, use `SpreadsheetRow` for better performance:

```
using SpreadsheetEasy;

SpreadsheetRow playerRow = SpreadsheetReader.GetRow("PlayerConfig",
"Players", "ID", "P001");
if (playerRow != null)
{
    string name = playerRow.GetString("Name");
    int hp = playerRow.GetInt("HP");
    float speed = playerRow.GetFloat("Speed");
    bool active = playerRow.GetBool("Active");
    // Note: Enum values should be read using
SpreadsheetReader.GetEnum<T>() or SpreadsheetReader.TryGetEnum<T>()
```

```
    // SpreadsheetRow does not have GetEnum method - use
SpreadsheetReader methods instead
    Vector3 pos = playerRow.GetVector3("Position");
    Vector3 rot = playerRow.GetVector3("Rotation");
    Color col = playerRow.GetColor("Color");
    int[] items = playerRow.GetIntArray("Items");
    float[] scores = playerRow.GetFloatArray("Scores");
    string[] tags = playerRow.GetStringArray("Tags");
}
```

5. Multiple Search Conditions

You can search with multiple conditions using `Dictionary<string, string>`. All conditions must be satisfied (AND logic):

```
using SpreadsheetEasy;

// Create search conditions
Dictionary<string, string> conditions = new Dictionary<string, string>
{
    { "HP", "100" },
    { "Speed", "3" }
};

// Get single row with multiple conditions
SpreadsheetRow playerRow = SpreadsheetReader.GetRow("PlayerConfig",
"Players", conditions);
if (playerRow != null)
{
    string name = playerRow.GetString("Name");
    int hp = playerRow.GetInt("HP");
    float speed = playerRow.GetFloat("Speed");
    Debug.Log($"Player: {name}, HP: {hp}, Speed: {speed}");
}

// Get all rows matching multiple conditions
```

```
List<SpreadsheetRow> playerRows =
SpreadsheetReader.GetAllRows("PlayerConfig", "Players", conditions);
foreach (SpreadsheetRow row in playerRows)
{
    Debug.Log($"Player: {row.GetString("Name")}");
}

// Get single value with multiple conditions
string playerName = SpreadsheetReader.GetString("PlayerConfig",
"Players", conditions, "Name");
int level = SpreadsheetReader.GetInt("PlayerConfig", "Players",
conditions, "Level");

// Get all values with multiple conditions
List<string> playerNames =
SpreadsheetReader.GetAllStrings("PlayerConfig", "Players", conditions,
"Name");
List<int> levels = SpreadsheetReader.GetAllInts("PlayerConfig",
"Players", conditions, "Level");
```

API Reference

GetValue Methods

All GetValue methods follow this pattern:

```
SpreadsheetReader.Get[Type](fileName, sheetName, searchColumn,
searchValue, targetColumn)
```

Parameters:

- `fileName` : Name of the Spreadsheet file (supports full path like `"SpreadsheetEasy/PlayerConfig"` or file name only like `"PlayerConfig"` , with or without extension)
    - **Important**: File names must be unique across all subdirectories!

SpreadsheetEasy uses file names (without path and extension) as cache keys.
    - Examples: `"PlayerConfig"`, `"SpreadsheetEasy/PlayerConfig"`, `"PlayerConfig.xlsx"` all resolve to the same cache key `"PlayerConfig"`
- `sheetName`: Name of the sheet within the Spreadsheet file
- `searchColumn`: Column name to search for matching values
- `searchValue`: Value to match in the search column
- `targetColumn`: Column name to retrieve the value from

Available Methods (Single Condition):

Note: All methods throw exceptions when the row is not found (see Error Handling section). Default values are returned only when the row is found but the cell value is empty or cannot be parsed.

- `GetString()` - Returns string value (throws `KeyNotFoundException` if row not found, returns empty string if cell is empty)
- `GetInt()` - Returns int value (throws `KeyNotFoundException` if row not found, returns 0 if cell is empty or parse fails)
- `GetFloat()` - Returns float value (throws `KeyNotFoundException` if row not found, returns 0f if cell is empty or parse fails)
- `GetDouble()` - Returns double value (throws `KeyNotFoundException` if row not found, returns 0.0 if cell is empty or parse fails)
- `GetBool()` - Returns bool value (throws `KeyNotFoundException` if row not found, returns false if cell is empty or parse fails)
    - Accepts: `"true"`, `"false"`, `"1"`, `"0"`, `"yes"`, `"no"` (all case-insensitive, e.g., TRUE, False, YES work too)
- `GetEnum<T>()` - Returns Enum value (throws `KeyNotFoundException` if row not found, returns default enum value if cell is empty or parse fails)
    - **Important**: Returns only the **first** matching result. If multiple rows match the search condition, use `GetAllEnums<T>()` instead to get all matching results.
    - Supports parsing from string name (case-insensitive) or integer value

    - Example: `GetEnum<PlayerRarity>("PlayerConfig", "Players", "ID", "P001", "Rarity")`
    - Spreadsheet cell value can be "Common", "common", "COMMON", or "0" (integer value)
- `GetVector3()` - Returns Vector3 (throws `KeyNotFoundException` if row not found, returns zero if cell is empty or parse fails)
    - Format: `"x,y,z"` or `"x;y;z"`
- `GetColor()` - Returns Color (throws `KeyNotFoundException` if row not found, returns white if cell is empty or parse fails)
    - Formats: `"r,g,b,a"`, `"r;g;b;a"`, `"#RRGGBB"`, or `"#RRGGBBAA"`
- `GetIntArray()` - Returns int[] (throws `KeyNotFoundException` if row not found, returns empty array if cell is empty or parse fails)
    - Format: `"1,2,3"` or `"1;2;3"`
- `GetFloatArray()` - Returns float[] (throws `KeyNotFoundException` if row not found, returns empty array if cell is empty or parse fails)
    - Format: `"1.5,2.5,3.5"` or `"1.5;2.5;3.5"`
- `GetStringArray()` - Returns string[] (throws `KeyNotFoundException` if row not found, returns empty array if cell is empty or parse fails)
    - Format: `"a,b,c"` or `"a;b;c"`

Available Methods (Multiple Conditions):

All `Get*` methods support multiple search conditions using `Dictionary<string, string>`:

```
// Get value with multiple conditions
SpreadsheetReader.Get[Type](fileName, sheetName, Dictionary<string, string> searchConditions, targetColumn)
```

Example:

```
// Create search conditions: HP = 100 AND Speed = 3
Dictionary<string, string> conditions = new Dictionary<string, string>
{
    { "HP", "100" },
    { "Speed", "3" }
};

// Get single value with multiple conditions
string playerName = SpreadsheetReader.GetString("PlayerConfig",
"Players", conditions, "Name");
int level = SpreadsheetReader.GetInt("PlayerConfig", "Players",
conditions, "Level");
float speed = SpreadsheetReader.GetFloat("PlayerConfig", "Players",
conditions, "Speed");
```

Note: All conditions must be satisfied (AND logic). The method returns the first matching row's value.

Important: Multiple condition search supports **numeric-aware comparison**. For example, searching with `"3"` will match Spreadsheet values like `"3.0"` or `"3"`, and vice versa. This makes it easier to search numeric columns without worrying about exact string format.

Available GetAll Methods (Multiple Values):

When multiple rows match your search criteria, use `GetAll*` methods to retrieve all matching values:

- `GetAllStrings()` - Returns `List<string>` containing all matching string values
- `GetAllInts()` - Returns `List<int>` containing all matching int values
- `GetAllFloats()` - Returns `List<float>` containing all matching float values
- `GetAllDoubles()` - Returns `List<double>` containing all matching double values
- `GetAllBools()` - Returns `List<bool>` containing all matching bool values
- `GetAllEnums<T>()` - Returns `List<T>` containing all matching Enum values
- `GetAllVector3s()` - Returns `List<Vector3>` containing all matching Vector3

values

- `GetAllColors()` - Returns `List<Color>` containing all matching Color values

Note: All `GetAll*` methods throw exceptions when the file/sheet is not found. They return empty lists only when no matching rows are found (but file/sheet exists). For Enum values, use `TryGetAllEnums<T>()` to avoid exception handling.

All `GetAll*` methods support both single condition and multiple conditions:

```
// Single condition
List<T> GetAll[Type](fileName, sheetName, searchColumn, searchValue,
targetColumn)

// Multiple conditions
List<T> GetAll[Type](fileName, sheetName, Dictionary<string, string>
searchConditions, targetColumn)
```

TryGetValue Methods

The `TryGet*` methods provide a cleaner way to query Spreadsheet data without using try-catch blocks. These methods return `bool` to indicate success or failure, and output the result through an `out` parameter.

All TryGetValue methods follow this pattern:

```
bool SpreadsheetReader.TryGet[Type](fileName, sheetName, searchColumn,
searchValue, targetColumn, out [Type] result)
```

Parameters:

- Same as `Get*` methods, plus:
- `result` : Output parameter that receives the value if the operation succeeds

Return Value:

- Returns `true` if the value was found and successfully parsed
- Returns `false` if the row was not found, the file/sheet/column doesn't exist, or the value cannot be parsed

Available TryGet Methods (Single Condition):

- `TryGetString()` - Returns bool, outputs string value
- `TryGetInt()` - Returns bool, outputs int value
- `TryGetFloat()` - Returns bool, outputs float value
- `TryGetDouble()` - Returns bool, outputs double value
- `TryGetBool()` - Returns bool, outputs bool value
  - Accepts: `"true"`, `"false"`, `"1"`, `"0"`, `"yes"`, `"no"` (all case-insensitive)
- `TryGetEnum<T>()` - Returns bool, outputs Enum value
  - Supports parsing from string name (case-insensitive) or integer value
- `TryGetVector3()` - Returns bool, outputs Vector3 value
  - Format: `"x,y,z"` or `"x;y;z"`
- `TryGetColor()` - Returns bool, outputs Color value
  - Formats: `"r,g,b,a"`, `"r;g;b;a"`, `"#RRGGBB"`, or `"#RRGGBBAA"`
- `TryGetIntArray()` - Returns bool, outputs int[] value
  - Format: `"1,2,3"` or `"1;2;3"`
- `TryGetFloatArray()` - Returns bool, outputs float[] value
  - Format: `"1.5,2.5,3.5"` or `"1.5;2.5;3.5"`
- `TryGetStringArray()` - Returns bool, outputs string[] value
  - Format: `"a,b,c"` or `"a;b;c"`

Available TryGet Methods (Multiple Conditions):

All `TryGet*` methods support multiple search conditions using `Dictionary<string, string>`:

```
bool SpreadsheetReader.TryGet[Type](fileName, sheetName,
Dictionary<string, string> searchConditions, targetColumn, out [Type]
result)
```

Example:

```
// Single condition - no try-catch needed!
if (SpreadsheetReader.TryGetInt("PlayerConfig", "Players", "ID", "1001",
"HP", out int hp))
{
    Debug.Log($"HP is {hp}");
}
else
{
    Debug.Log("HP not found or invalid format");
}

// Multiple conditions
Dictionary<string, string> conditions = new Dictionary<string, string>
{
    { "HP", "100" },
    { "Speed", "3" }
};

if (SpreadsheetReader.TryGetString("PlayerConfig", "Players",
conditions, "Name", out string name))
{
    Debug.Log($"Player name is {name}");
}
else
```

```
{
    Debug.Log("Player not found");
}

// TryGet with different types
if (SpreadsheetReader.TryGetFloat("PlayerConfig", "Players", "ID",
"1001", "Speed", out float speed))
{
    Debug.Log($"Speed is {speed}");
}

if (SpreadsheetReader.TryGetBool("PlayerConfig", "Players", "ID",
"1001", "Active", out bool isActive))
{
    Debug.Log($"Item is active: {isActive}");
}

if (SpreadsheetReader.TryGetEnum<PlayerRarity>("PlayerConfig",
"Players", "ID", "P001", "Rarity", out PlayerRarity rarity))
{
    Debug.Log($"Player rarity: {rarity}");
}
else
{
    Debug.Log("Player rarity not found or invalid format");
}

if (SpreadsheetReader.TryGetVector3("PlayerConfig", "Players", "ID",
"1001", "Position", out Vector3 position))
{
    Debug.Log($"Position is {position}");
}

if (SpreadsheetReader.TryGetColor("PlayerConfig", "Players", "ID",
"1001", "Color", out Color color))
{
```

```
    Debug.Log($"Color is {color}");
}

if (SpreadsheetReader.TryGetIntArray("PlayerConfig", "Players", "ID",
"1001", "Items", out int[] items))
{
    Debug.Log($"Items array has {items.Length} elements");
}

// TryGetAllEnums: Get all Enum values without exception handling
if (SpreadsheetReader.TryGetAllEnums<PlayerRarity>("PlayerConfig",
"Players", "HP", "100", "Rarity", out List<PlayerRarity> rarities))
{
    Debug.Log($"Found {rarities.Count} matching rarities");
    foreach (PlayerRarity rarity in rarities)
    {
        Debug.Log($"Rarity: {rarity}");
    }
}
else
{
    Debug.Log("Failed to get Enum values (file/sheet not found or
invalid)");
}

// TryGetAllStrings: Get all string values without exception handling
if (SpreadsheetReader.TryGetAllStrings("PlayerConfig", "Players", "HP",
"100", "Name", out List<string> names))
{
    Debug.Log($"Found {names.Count} matching names");
    foreach (string name in names)
    {
        Debug.Log($"Name: {name}");
    }
}
```

```
// TryGetAllInts: Get all int values without exception handling
if (SpreadsheetReader.TryGetAllInts("PlayerConfig", "Players", "Rarity",
"Epic", "HP", out List<int> hps))
{
    Debug.Log($"Found {hps.Count} HP values");
}

// TryGetAllFloats: Get all float values without exception handling
if (SpreadsheetReader.TryGetAllFloats("PlayerConfig", "Players", "HP",
"100", "Speed", out List<float> speeds))
{
    Debug.Log($"Found {speeds.Count} speed values");
}

// TryGetAllBools: Get all bool values without exception handling
if (SpreadsheetReader.TryGetAllBools("PlayerConfig", "Players", "HP",
"100", "Active", out List<bool> actives))
{
    Debug.Log($"Found {actives.Count} active status values");
}

// TryGetAllVector3s: Get all Vector3 values without exception handling
if (SpreadsheetReader.TryGetAllVector3s("PlayerConfig", "Players", "HP",
"100", "Position", out List<Vector3> positions))
{
    Debug.Log($"Found {positions.Count} position values");
}

// TryGetAllColors: Get all Color values without exception handling
if (SpreadsheetReader.TryGetAllColors("PlayerConfig", "Players", "HP",
"100", "Color", out List<Color> colors))
{
    Debug.Log($"Found {colors.Count} color values");
}
```

Benefits of TryGet Methods:

- **No try-catch blocks**: Cleaner code without exception handling
- **Explicit failure handling**: Return value clearly indicates success or failure
- **Safe defaults**: Output parameter is set to default value (0, false, null, etc.) on failure
- **Performance**: Same performance as `Get*` methods (uses the same index cache system)

Available TryGetAll Methods (Multiple Values):

All `TryGetAll*` methods provide exception-free access to multiple matching values:

- `TryGetAllStrings()` - Returns bool, outputs `List<string>` containing all matching string values
- `TryGetAllInts()` - Returns bool, outputs `List<int>` containing all matching int values
- `TryGetAllFloats()` - Returns bool, outputs `List<float>` containing all matching float values
- `TryGetAllDoubles()` - Returns bool, outputs `List<double>` containing all matching double values
- `TryGetAllBools()` - Returns bool, outputs `List<bool>` containing all matching bool values
- `TryGetAllEnums<T>()` - Returns bool, outputs `List<T>` containing all matching Enum values
- `TryGetAllVector3s()` - Returns bool, outputs `List<Vector3>` containing all matching Vector3 values
- `TryGetAllColors()` - Returns bool, outputs `List<Color>` containing all matching Color values

Behavior:

- Returns `true` if successful (file/sheet exists), `false` otherwise
- Outputs empty list if no matches found (but file/sheet exists)
- Outputs empty list and returns `false` if file/sheet not found or invalid

All `TryGetAll*` methods support both single condition and multiple conditions:

```
// Single condition
bool TryGetAll[Type](fileName, sheetName, searchColumn, searchValue,
targetColumn, out List<[Type]> result)

// Multiple conditions
bool TryGetAll[Type](fileName, sheetName, Dictionary<string, string>
searchConditions, targetColumn, out List<[Type]> result)
```

When to Use TryGet vs Get:

- Use `TryGet*` when you want to handle missing data gracefully without exceptions
- Use `Get*` when you want exceptions to be thrown for error handling or when data must exist

GetRow Method

```
SpreadsheetRow GetRow(string fileName, string sheetName, string
searchColumn, string searchValue)
```

Returns an `SpreadsheetRow` object that provides access to all columns in the matched row. Use this when you need multiple values from the same row.

Note: This method returns only the **first** matching row. If multiple rows match the search criteria, use `GetAllRows()` instead.

GetRow Method (Multiple Search Conditions)

```
SpreadsheetRow GetRow(string fileName, string sheetName,
Dictionary<string, string> searchConditions)
```

Search for a row using multiple conditions. All conditions must be satisfied (AND logic).

Note: Multiple condition search supports **numeric-aware comparison**. For example, searching with `"3"` will match Spreadsheet values like `"3.0"` or `"3"`, and vice versa.

Example:

```
// Search with multiple conditions: HP = 100 AND Speed = 3
Dictionary<string, string> conditions = new Dictionary<string, string>
{
    { "HP", "100" },
    { "Speed", "3" }
};

SpreadsheetRow playerRow = SpreadsheetReader.GetRow("PlayerConfig",
"Players", conditions);
if (playerRow != null)
{
    string name = playerRow.GetString("Name");
    int hp = playerRow.GetInt("HP");
    float speed = playerRow.GetFloat("Speed");
    Debug.Log($"Player: {name}, HP: {hp}, Speed: {speed}");
}
```

GetAllRows Method (No Search Conditions)

Get all rows in a sheet without any search conditions:

```
List<SpreadsheetRow> GetAllRows(string fileName, string sheetName)
```

Example:

```
// Get all players in the sheet
List<SpreadsheetRow> allPlayerRows =
SpreadsheetReader.GetAllRows("PlayerConfig", "Players");

foreach (SpreadsheetRow row in allPlayerRows)
{
    string name = row.GetString("Name");
    int hp = row.GetInt("HP");
    Debug.Log($"Player: {name}, HP: {hp}");
}
```

GetAllRows Method (Multiple Matches)

When multiple rows match your search criteria (e.g., multiple players with HP = 100), use `GetAllRows()` to get all matching rows:

```
List<SpreadsheetRow> GetAllRows(string fileName, string sheetName,
string searchColumn, string searchValue)
```

Example:

```
// Get all players with HP = 100
List<SpreadsheetRow> playerRows =
SpreadsheetReader.GetAllRows("PlayerConfig", "Players", "HP", "100");

foreach (SpreadsheetRow row in playerRows)
{
    string name = row.GetString("Name");
    int hp = row.GetInt("HP");
    Debug.Log($"Player: {name}, HP: {hp}");
}
```

GetAllRows Method (Multiple Search Conditions)

```
List<SpreadsheetRow> GetAllRows(string fileName, string sheetName,
Dictionary<string, string> searchConditions)
```

Get all rows matching multiple conditions. All conditions must be satisfied (AND logic).

Note: Multiple condition search supports **numeric-aware comparison**. For example, searching with `"3"` will match Spreadsheet values like `"3.0"` or `"3"`, and vice versa.

Example:

```
// Get all players with HP = 100 AND Speed = 3
Dictionary<string, string> conditions = new Dictionary<string, string>
{
    { "HP", "100" },
    { "Speed", "3" }
};

List<SpreadsheetRow> playerRows =
SpreadsheetReader.GetAllRows("PlayerConfig", "Players", conditions);

foreach (SpreadsheetRow row in playerRows)
{
    string name = row.GetString("Name");
    int hp = row.GetInt("HP");
    float speed = row.GetFloat("Speed");
    Debug.Log($"Player: {name}, HP: {hp}, Speed: {speed}");
}
```

GetRowCount Method

Get the total number of rows in a sheet. By default, it excludes the header row (row 0). Set `includeHeaderRow` to `true` to include it.

```
int GetRowCount(string fileName, string sheetName, bool includeHeaderRow
= false)
```

Example:

```
// Get row count excluding header row (default)
int dataRowCount = SpreadsheetReader.GetRowCount("PlayerConfig",
"Players");
Debug.Log($"Total data rows: {dataRowCount}"); // Excludes header row

// Get row count including header row
int totalRowCount = SpreadsheetReader.GetRowCount("PlayerConfig",
"Players", includeHeaderRow: true);
Debug.Log($"Total rows (including header): {totalRowCount}");
```

GetAll Methods (Multiple Values)

For convenience, you can also get all matching values from a specific column directly:

Note: All `GetAll*` methods throw exceptions when the file/sheet is not found. If you want to avoid exception handling, use the corresponding `TryGetAll*` methods (e.g., `TryGetAllStrings()`, `TryGetAllInts()`, `TryGetAllEnums<T>()`, etc.) which are available for all data types.

```
// Get all string values
List<string> GetAllStrings(string fileName, string sheetName, string
searchColumn, string searchValue, string targetColumn)

// Get all int values
List<int> GetAllInts(string fileName, string sheetName, string
searchColumn, string searchValue, string targetColumn)

// Get all float values
List<float> GetAllFloats(string fileName, string sheetName, string
searchColumn, string searchValue, string targetColumn)
```

```
// Get all double values
List<double> GetAllDoubles(string fileName, string sheetName, string searchColumn, string searchValue, string targetColumn)

// Get all bool values
List<bool> GetAllBools(string fileName, string sheetName, string searchColumn, string searchValue, string targetColumn)

// Get all Enum values (returns all matching results as List<T>)
// Important: Use GetAllEnums<T>() instead of GetEnum<T>() when you need all matching results
// Note: GetAllEnums<T>() throws exceptions when file/sheet not found
//       Use TryGetAllEnums<T>() to avoid exception handling
List<T> GetAllEnums<T>(string fileName, string sheetName, string searchColumn, string searchValue, string targetColumn) where T : struct, Enum

// Try to get all Enum values (no exception handling needed)
// Returns true if successful, false otherwise
// Outputs List<T> containing all matching Enum values (empty list if no matches found)
bool TryGetAllEnums<T>(string fileName, string sheetName, string searchColumn, string searchValue, string targetColumn, out List<T> result) where T : struct, Enum

// Get all Vector3 values
List<Vector3> GetAllVector3s(string fileName, string sheetName, string searchColumn, string searchValue, string targetColumn)

// Get all Color values
List<Color> GetAllColors(string fileName, string sheetName, string searchColumn, string searchValue, string targetColumn)
```

Example:

```
// Get all player names with HP = 100
List<string> playerNames =
SpreadsheetReader.GetAllStrings("PlayerConfig", "Players", "HP", "100",
"Name");

// Get all HP values where Name = "Player1"
List<int> hps = SpreadsheetReader.GetAllInts("PlayerConfig", "Players",
"Name", "Player1", "HP");

// Get all Enum values
// Note: GetEnum<T>() returns only the first matching result
//       GetAllEnums<T>() returns all matching results as List<T>
public enum PlayerRarity { Common, Rare, Epic, Legendary }
List<PlayerRarity> rarities =
SpreadsheetReader.GetAllEnums<PlayerRarity>("PlayerConfig", "Players",
"HP", "100", "Rarity");
foreach (PlayerRarity rarity in rarities)
{
    Debug.Log($"Player Rarity: {rarity}");
}

// Using TryGetAllEnums (no exception handling needed)
if (SpreadsheetReader.TryGetAllEnums<PlayerRarity>("PlayerConfig",
"Players", "HP", "100", "Rarity", out List<PlayerRarity> rarities2))
{
    Debug.Log($"Found {rarities2.Count} matching rarities");
    foreach (PlayerRarity rarity in rarities2)
    {
        Debug.Log($"Player Rarity: {rarity}");
    }
}
else
{
    Debug.Log("Failed to get Enum values");
}
```

GetAll Methods (Multiple Search Conditions)

All `GetAll*` methods support multiple search conditions:

```
// Get all string values with multiple conditions
List<string> GetAllStrings(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn)

// Get all int values with multiple conditions
List<int> GetAllInts(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn)

// Get all float values with multiple conditions
List<float> GetAllFloats(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn)

// Get all double values with multiple conditions
List<double> GetAllDoubles(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn)

// Get all bool values with multiple conditions
List<bool> GetAllBools(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn)

// Get all Enum values with multiple conditions
// Note: GetAllEnums<T>() throws exceptions when file/sheet not found
//       Use TryGetAllEnums<T>() to avoid exception handling
List<T> GetAllEnums<T>(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn) where
T : struct, Enum

// Try to get all Enum values with multiple conditions (no exception
handling needed)
bool TryGetAllEnums<T>(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn, out
List<T> result) where T : struct, Enum
```

```
// Get all Vector3 values with multiple conditions
List<Vector3> GetAllVector3s(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn)

// Get all Color values with multiple conditions
List<Color> GetAllColors(string fileName, string sheetName,
Dictionary<string, string> searchConditions, string targetColumn)
```

Example:

```
// Create search conditions
Dictionary<string, string> conditions = new Dictionary<string, string>
{
    { "HP", "100" },
    { "Speed", "3" }
};

// Get all player names with HP = 100 AND Speed = 3
List<string> playerNames =
SpreadsheetReader.GetAllStrings("PlayerConfig", "Players", conditions,
"Name");

// Get all levels with HP = 100 AND Speed = 3
List<int> levels = SpreadsheetReader.GetAllInts("PlayerConfig",
"Players", conditions, "Level");

// Get all Enum values with multiple conditions
// Note: GetEnum<T>() returns only the first matching result
//       GetAllEnums<T>() returns all matching results as List<T>
public enum PlayerRarity { Common, Rare, Epic, Legendary }
List<PlayerRarity> rarities =
SpreadsheetReader.GetAllEnums<PlayerRarity>("PlayerConfig", "Players",
conditions, "Rarity");
```

```
// Using TryGetAllEnums with multiple conditions (no exception handling
needed)
if (SpreadsheetReader.TryGetAllEnums<PlayerRarity>("PlayerConfig",
"Players", conditions, "Rarity", out List<PlayerRarity> rarities2))
{
    Debug.Log($"Found {rarities2.Count} matching rarities");
    foreach (PlayerRarity rarity in rarities2)
    {
        Debug.Log($"Player Rarity: {rarity}");
    }
}
else
{
    Debug.Log("Failed to get Enum values");
}

// Example: Process all Enum values
foreach (PlayerRarity rarity in rarities)
{
    switch (rarity)
    {
        case PlayerRarity.Common:
            Debug.Log("Player Rarity: Common");
            break;
        case PlayerRarity.Rare:
            Debug.Log("Player Rarity: Rare");
            break;
        case PlayerRarity.Epic:
            Debug.Log("Player Rarity: Epic");
            break;
        case PlayerRarity.Legendary:
            Debug.Log("Player Rarity: Legendary");
            break;
    }
}
```

JSON Export Methods

```
// Get single row as JSON (throws KeyNotFoundException if row not found)
string rowJson = SpreadsheetReader.GetRowJson("PlayerConfig", "Players",
"ID", "P001");

// Get entire table as JSON array (returns "[]" if file/sheet not found
or empty)
string tableJson = SpreadsheetReader.GetTableJson("PlayerConfig",
"Players");
```

Note:

- `GetRowJson()` throws exceptions when the row is not found (same behavior as other Get methods)
- `GetTableJson()` returns `"[]"` (empty JSON array) if the file or sheet is not found, or if the sheet is empty (does not throw exceptions)

Utility Methods

```
// Load all Spreadsheet files and rebuild indices (called automatically
on first access)
SpreadsheetReader.LoadAllSpreadsheet();

// Manually rebuild indices (useful after clearing index cache)
SpreadsheetReader.RebuildIndices();

// Clear only index cache (keeps data loaded)
SpreadsheetReader.ClearIndexCache();

// Clear all cache (data + indices)
SpreadsheetReader.ClearCache();
```

Spreadsheet File Format Requirements

1. **First row must be headers** (column names)
2. Column names are case-insensitive
3. Empty cells return default values (empty string for strings, 0 for numbers, false for bool, empty arrays for arrays, etc.) when the row is found. If the row is not found, methods throw exceptions.
4. Supported formats: `.xlsx` , `.xls`

Error Handling

SpreadsheetEasy provides two approaches for error handling:

Approach 1: TryGet Methods (Recommended for Missing Data)

Use `TryGet*` methods when you want to handle missing data gracefully without exceptions:

```
// Clean code without try-catch
if (SpreadsheetReader.TryGetInt("PlayerConfig", "Players", "ID", "P001",
"HP", out int hp))
{
    Debug.Log($"Player HP: {hp}");
}
else
{
    Debug.Log("HP not found or invalid format");
}
```

Benefits:

- No try-catch blocks needed
- Explicit success/failure indication
- Cleaner, more readable code

Approach 2: Get Methods with Exception Handling

All `Get*` methods throw exceptions on error instead of returning default values. Use try-catch blocks to handle errors:

```
try
{
    int hp = SpreadsheetReader.GetInt("PlayerConfig", "Players", "ID",
"P001", "HP");
    Debug.Log($"Player HP: {hp}");
}
catch (FileNotFoundException ex)
{
    Debug.LogError($"Spreadsheet file not found: {ex.Message}");
}
catch (KeyNotFoundException ex)
{
    Debug.LogWarning($"Row not found: {ex.Message}");
}
catch (ArgumentException ex)
{
    Debug.LogError($"Invalid argument: {ex.Message}");
}
catch (FormatException ex)
{
    Debug.LogError($"Format error: {ex.Message}");
}
```

Exception Types:

- `FileNotFoundException` : Spreadsheet file not found
- `ArgumentException` : Invalid parameters (e.g., column not found, sheet not found)
- `InvalidOperationException` : Operation failed (e.g., sheet is empty)
- `KeyNotFoundException` : No matching row found

- `FormatException` : Value format error (e.g., cannot parse string to int)
- `IndexOutOfRangeException` : Index out of range (SpreadsheetRow only)

When to Use Each Approach:

- **Use `**TryGet***` methods**: When data may not exist and you want to handle it gracefully (e.g., optional configuration, user-generated content)
- **Use `**Get***` methods**: When data must exist and exceptions should be thrown (e.g., critical game data, configuration that must be present)

Example: Using TryGet to check if value exists:

```
// Using TryGet (no try-catch needed)
if (SpreadsheetReader.TryGetString("PlayerConfig", "Sheet1", "ID",
"999", "Name", out string value))
{
    Debug.Log($"Value found: {value}");
}
else
{
    Debug.Log("Value not found!");
}

// Using Get with try-catch (alternative approach)
try
{
    string value = SpreadsheetReader.GetString("PlayerConfig", "Sheet1",
"ID", "999", "Name");
    Debug.Log($"Value found: {value}");
}
catch (KeyNotFoundException)
{
    Debug.Log("Value not found!");
}
```

Advanced Usage

SpreadsheetEasy Settings

SpreadsheetEasy provides a settings system that allows you to configure how Spreadsheet files are loaded at runtime. The settings are stored in a ScriptableObject asset located at `Assets/Resources/SpreadsheetEasySettings.asset`.

Accessing Settings

To access or create the settings:

1. **Via Menu**: `Assets` → `Create` → `Spreadsheet Easy` → `Settings`
2. **Via Code**: The settings are automatically loaded from `Resources/SpreadsheetEasySettings.asset` at runtime

Loading Mode

The `LoadingMode` setting determines how Spreadsheet files are loaded:

RuntimeParsing (Default):

- Parse Spreadsheet files at runtime from StreamingAssets folder
- Files are parsed on first access and cached for subsequent queries
- Best for: Development, testing, or when Spreadsheet files need to be updated without rebuilding
- Requires: Spreadsheet files in `StreamingAssets` folder

ScriptableObject (Performance Mode):

- Load pre-baked ScriptableObject assets for zero parsing cost
- Spreadsheet files are converted to ScriptableObject assets at build time
- Best for: Production builds where maximum performance is required
- Requires: Spreadsheet source folder configured in settings, ScriptableObject assets generated before build

Configuring Settings

1. Create or open `SpreadsheetEasySettings.asset` (via menu: `Assets` → `Create` → `Spreadsheet Easy` → `Settings`)
2. Select the desired `Mode`:
    - **RuntimeParsing**: Parse Spreadsheet files at runtime (default)
    - **ScriptableObject**: Use pre-baked ScriptableObject assets
3. If using **ScriptableObject** mode:
    - Set `Spreadsheet Source Folder` to the folder containing your Spreadsheet files
    - ScriptableObject assets will be automatically generated from Spreadsheet files in this folder and saved to `Resources/SpreadsheetEasyBaked` folder
    - **Important**: If `Spreadsheet Source Folder` is not set, ScriptableObject assets will NOT be auto-generated

How Loading Mode Affects Behavior

RuntimeParsing Mode:

- `LoadAllSpreadsheet()` loads all Spreadsheet files from StreamingAssets folder
- Files are parsed on first access if not preloaded
- Data is cached in memory after first load

ScriptableObject Mode:

- `LoadAllSpreadsheet()` skips StreamingAssets loading
- Files are loaded on-demand from `Resources/SpreadsheetEasyBaked` folder as ScriptableObject assets
- No parsing overhead - data is already serialized in Unity's binary format
- Faster load times, especially for large Spreadsheet files
- ScriptableObject assets are automatically generated when Spreadsheet files in the source folder are added, modified, or deleted

Note: The API usage remains the same regardless of loading mode. All `Get*`, `TryGet*`, and `GetRow` methods work identically in both modes.

Runtime File Loading

At runtime, SpreadsheetEasy loads files from the StreamingAssets folder:

StreamingAssets folder: Automatically scanned when `LoadAllSpreadsheet()` is called

- Unity has only one StreamingAssets folder (`Assets/StreamingAssets/`)
- Supports subdirectories within StreamingAssets
- Files are accessible at runtime via file system
- Files can be updated without rebuilding the application

SpreadsheetEasyIndex.txt generation and fallback

- The index file `StreamingAssets/SpreadsheetEasyIndex.txt` is automatically generated before builds by the `AutoSpreadsheetIndexer` (`IPreprocessBuildWithReport`). You can also regenerate it manually via menu: `Tools` → `Spreadsheet Easy` → `Generate Spreadsheet Index (Recursive)`.
- The index file is automatically regenerated whenever files in the StreamingAssets folder are added, deleted, or moved. This is handled by the `SpreadsheetIndexAssetPostprocessor` which monitors asset changes. Performance optimizations are in place (debounce mechanism) to prevent unnecessary regenerations during rapid file operations, so you don't need to worry about performance impact.
- `LoadAllSpreadsheet()` uses `SpreadsheetEasyIndex.txt` to preload all Spreadsheet files in one pass. If the file is missing or empty at runtime, SpreadsheetEasy falls back to on-demand loading: the first access to a file performs a direct load (slower once), and subsequent accesses use the cache. If the target Spreadsheet file itself is missing, the first on-demand load will throw a file-not-found error.

Best Practice:

- Place Spreadsheet files in `StreamingAssets` folder for runtime access
- Organize files in subdirectories within StreamingAssets for better file management

Loading from Subdirectories

If your Spreadsheet files are organized in subdirectories within the StreamingAssets folder, you can use the file name (without extension) or the relative path (without extension) when querying:

```
// File location: StreamingAssets/MyTable/EnemyData.xlsx
// You can use just the file name (no extension needed):
string enemyName = SpreadsheetReader.GetString("EnemyData", "Sheet1",
"ID", "P001", "Name");

// Or use the relative path (no extension needed):
string enemyName = SpreadsheetReader.GetString("MyTable/EnemyData",
"Sheet1", "ID", "P001", "Name");

// File location: StreamingAssets/GameData/Items.xlsx
int itemSpeed = SpreadsheetReader.GetInt("Items", "Items", "ItemID",
"sword_01", "Speed");
// Or: SpreadsheetReader.GetInt("GameData/Items", "Items", "ItemID",
"sword_01", "Speed");

// File location: StreamingAssets/Config/Levels/Stage1.xlsx
float difficulty = SpreadsheetReader.GetFloat("Stage1", "Settings",
"Key", "difficulty", "Value");
// Or: SpreadsheetReader.GetFloat("Config/Levels/Stage1", "Settings",
"Key", "difficulty", "Value");
```

Important:

- **No extension needed**: You don't need to include `.xlsx` or `.xls` in the file name parameter
- **File names must be unique**: SpreadsheetEasy uses file names (without path and

extension) as cache keys. If you have multiple files with the same name in different folders, only the first one loaded will be accessible. See File Name Requirements for details.

Tools

Spreadsheet Data JSON Checker

A standalone JSON validation tool that works independently and directly reads Spreadsheet files. This tool allows you to quickly validate JSON format in Spreadsheet files from any folder in your project.

Access the Tool

- **Menu Path**: `Tools` → `Spreadsheet Easy` → `Spreadsheet Data JSON Checker`
- Opens an EditorWindow where you can select a folder to validate

Features

- **Direct File Access**: Works directly with Spreadsheet files, no pre-processing or setup required
- **Folder Selection**: Drag and drop folders from Project window or use the object field to browse
- **Recursive Search**: Automatically scans all subdirectories for Spreadsheet files
- **Automatic JSON Detection**: Detects columns containing JSON data by checking rows 2-52
- **Comprehensive Validation**: Validates all JSON cells in detected columns
- **Detailed Error Reporting**: Shows file path, sheet name, row number, column name, and error message for each invalid JSON cell
- **Results Display**: Validation results displayed directly in the EditorWindow with:
  - Summary statistics (files checked, sheets checked, JSON cells validated, errors found)
  - Completion time and duration

    - List of all checked files and their sheets
- **Persistent Selection**: Remembers your last selected folder for convenience

How to Use

1. Open the tool via `Tools` → `Spreadsheet Easy` → `Spreadsheet Data JSON Checker`
2. Drag a folder from the Project window to the "Spreadsheet Folder" field, or click the field to browse
3. Click "Validate JSON" button
4. Review the validation results in the window:
    - Check the summary message for overall status
    - View completion time and duration
    - Expand "Checked Files and Sheets" to see all processed files
    - Check Console for detailed error messages if any errors are found

Validation Process

The tool automatically:

1. Scans the selected folder and all subdirectories for `.xlsx` and `.xls` files
2. Reads each Spreadsheet file and processes all sheets
3. Detects JSON columns by checking if cells in rows 2-52 contain JSON-like content (starts with `{` or `[`)
4. Validates all non-empty cells in detected JSON columns
5. Reports any JSON format errors with detailed location information

Error Messages

When JSON validation errors are found, the tool provides:

- **File Path**: Assets-relative path to the Spreadsheet file
- **Sheet Name**: Name of the sheet containing the error

- **Row Number**: 1-based row number (for user-friendly display)
- **Column Name**: Header name of the column
- **Error Message**: Specific JSON validation error
- **Cell Value**: Truncated cell content (first 200 characters)

All errors are logged to the Unity Console for easy review and debugging.

Use Cases

- **Quick Validation**: Quickly check JSON format in Spreadsheet files with no setup required
- **Batch Validation**: Validate multiple Spreadsheet files in a folder at once
- **Pre-commit Checks**: Verify JSON format before committing changes
- **Troubleshooting**: Identify JSON format errors when Spreadsheet data fails to parse correctly

Performance Optimization

SpreadsheetEasy includes a powerful **Index Cache System** that provides **50-100x speedup** for frequent data access.

Benchmark Results

Based on tests with a 1000-row Spreadsheet file:

Operation	Without Index	With Index	Speedup
Single Query	~0.5ms	~0.01ms	**50x faster**
1000 Queries	~500ms	~10ms	**50x faster**
GetRow (multiple columns)	~1.5ms	~0.02ms	**75x faster**

How It Works

1. **Column Index Cache**: Column names mapped to indices (O(1) lookup)
2. **Row Index Cache**: Search results cached for instant retrieval
3. **Lazy Loading**: Indices built only when needed
4. **Automatic**: Works automatically, no code changes required

Recent Performance Improvements

SpreadsheetEasy has been significantly optimized for large Spreadsheet files:

- **Pre-built SharedString Lookup**: O(1) access instead of O(n) per cell lookup
- **Optimized Column Index Parsing**: Eliminated string concatenation overhead
- **Large File Support**: Spreadsheet files with 5000+ rows now process in milliseconds instead of tens of seconds

Performance API

```
// Load all Spreadsheet files (automatically rebuilds indices)
SpreadsheetReader.LoadAllSpreadsheet();

// Manually rebuild all indices (useful after clearing index cache)
SpreadsheetReader.RebuildIndices();

// Get cache statistics for monitoring
SpreadsheetReader.CacheStats stats = SpreadsheetReader.GetCacheStats();
Debug.Log(stats.ToString());

// CacheStats structure contains:
// - DataFileCount: Number of cached Spreadsheet files
// - DataSheetCount: Total number of cached sheets
// - DataRowCount: Total number of data rows
// - DataColumnCount: Total number of columns
// - ColumnIndexCount: Number of column index entries
// - RowIndexCount: Number of row index entries
```

```
// - EstimatedMemoryBytes: Estimated memory usage in bytes

// Example output of stats.ToString():
// ========== Cache Statistics ==========
// Data Cache:
//   Files: 3
//   Sheets: 5
//   Total Rows: 1,250
//   Total Columns: 120
//   Average Rows per Sheet: 250.0
//   Average Columns per Sheet: 24.0
//
// Index Cache:
//   Column Index Maps: 5
//   Row Index Entries: 450
//
// Memory Usage:
//   Estimated Memory: 125KB
// ========================================

// Cache hit tracking (diagnostic)
SpreadsheetReader.ShowCacheTrackingLogs = true;     // Enable detailed
cache hit/miss logs
SpreadsheetReader.CacheHitStats hitStats =
SpreadsheetReader.GetCacheHitStats();
Debug.Log(hitStats.ToString());             // Row/column index cache
hit rates
SpreadsheetReader.ResetCacheHitStats();             // Reset counters

// Example output of hitStats.ToString():
// ========== Cache Hit Statistics ==========
// Row Index Cache:
//   Hits: 4
//   Misses: 1
//   Total Queries: 5
//   Hit Rate: 80.0%
```

```
//
// Column Index Cache:
//   Hits: 18
//   Misses: 4
//   Total Queries: 22
//   Hit Rate: 81.8%
//
// Overall Statistics:
//   Total Queries: 27
//   Total Hits: 22
//   Total Misses: 5
//   Overall Hit Rate: 81.5%
// ========================================

// Clear only indices (keep data loaded)
SpreadsheetReader.ClearIndexCache();

// Clear everything (data + indices)
SpreadsheetReader.ClearCache();
```

Best Practices

DO: Load Data at Startup

```
public class GameManager : MonoBehaviour
{
    void Awake()
    {
        // LoadAllSpreadsheet automatically rebuilds indices for optimal
performance
        SpreadsheetReader.LoadAllSpreadsheet();
        Debug.Log("Spreadsheet data loaded!");
    }
}
```

Note: `LoadAllSpreadsheet()` automatically rebuilds indices, so you don't need to call `RebuildIndices()` separately unless you've cleared the index cache.

DO: Use GetRow for Multiple Columns

```
// Bad: Multiple queries for same row (slow)
string name = SpreadsheetReader.GetString("Items", "Sheet1", "ID",
"1001", "Name");
int hp = SpreadsheetReader.GetInt("Items", "Sheet1", "ID", "1001",
"HP");
float speed = SpreadsheetReader.GetFloat("Items", "Sheet1", "ID",
"1001", "Speed");

// Good: GetRow once, access multiple columns (75x faster)
SpreadsheetRow row = SpreadsheetReader.GetRow("Items", "Sheet1", "ID",
"1001");
string name = row.GetString("Name");
int hp = row.GetInt("HP");
float speed = row.GetFloat("Speed");
```

DO: Use in Update() - It's Fast Enough!

```
// Good: With index cache, queries are extremely fast (~0.005ms)
void Update()
{
    // Safe to use in Update! Only ~0.005ms per query
    int hp = SpreadsheetReader.GetInt("Players", "Sheet1", "ID", "P001",
"HP");
    float speed = SpreadsheetReader.GetFloat("PlayerConfig", "Settings",
"Key", "speed", "Value");

    // 60 FPS = 16.67ms per frame
    // You can do ~3000 queries per frame before performance issues!
}
```

```
// Even better: Cache SpreadsheetRow for multiple columns
SpreadsheetRow _playerRow;

void Start()
{
    _playerRow = SpreadsheetReader.GetRow("Players", "Sheet1", "ID",
"P001");
}

void Update()
{
    // Extremely fast! Just memory access
    int hp = _playerRow.GetInt("HP");
    float speed = _playerRow.GetFloat("Speed");
}
```

DO: Cache Frequently Used Data

```
public class ItemDatabase : MonoBehaviour
{
    private Dictionary<string, SpreadsheetRow> _itemCache = new
Dictionary<string, SpreadsheetRow>();

    public SpreadsheetRow GetItem(string itemId)
    {
        if (!_itemCache.ContainsKey(itemId))
        {
            _itemCache[itemId] = SpreadsheetReader.GetRow("Items",
"Sheet1", "ID", itemId);
        }
        return _itemCache[itemId];
    }
}
```

CAUTION: Extremely Large Loops (1000+)

For very large datasets (1000+ items per frame), consider pre-caching:

```
// Still works, but not optimal: 1000 queries × 0.005ms = 5ms per frame
void Update()
{
    for (int i = 0; i < 1000; i++)
    {
        string name = SpreadsheetReader.GetString("Items", "Sheet1",
"ID", i.ToString(), "Name");
        ProcessItem(name);
    }
}

// Better for extreme cases: Pre-cache for maximum performance
Dictionary<int, SpreadsheetRow> _items;

void Start()
{
    // Load all items once (1000 queries × 0.005ms = 5ms, one-time cost)
    _items = new Dictionary<int, SpreadsheetRow>();
    for (int i = 0; i < 1000; i++)
    {
        _items[i] = SpreadsheetReader.GetRow("Items", "Sheet1", "ID",
i.ToString());
    }
}

void Update()
{
    // 50x faster: Dictionary lookup ~0.0001ms × 1000 = 0.1ms per frame
    for (int i = 0; i < 1000; i++)
    {
        string name = _items[i].GetString("Name");
        ProcessItem(name);
```

```
    }
}
```

Note: For loops with < 100 iterations, the cached version (5ms) is already fast enough for most games!

DON'T: Clear Cache Unnecessarily

```
// Bad: Clears cache every frame
void Update()
{
    SpreadsheetReader.ClearCache();
    int hp = SpreadsheetReader.GetInt("Items", "Sheet1", "ID", "1001",
"HP");
}

// Good: Cache persists across frames
void Start()
{
    SpreadsheetReader.LoadAllSpreadsheet(); // Automatically rebuilds
indices
}

void Update()
{
    int hp = SpreadsheetReader.GetInt("Items", "Sheet1", "ID", "1001",
"HP"); // Fast!
}
```

Memory Usage

For a typical Spreadsheet file with 1000 rows and 20 columns:

- **Column Index Cache**: ~2 KB per sheet
- **Row Index Cache**: ~50 KB per sheet

- **Total Overhead**: ~52 KB per sheet (minimal!)

Performance Testing

Use `SpreadsheetEasyPerformanceTest.cs` to benchmark your data:

1. Open Assets/SpreadsheetEasy/Examples/SpreadsheetEasyPerformanceTest.unity

Example output:

```
[SpreadsheetEasy] Speedup: 51.57x faster
[SpreadsheetEasy] Time saved: 477.87ms (98.1%)
```

Advanced Optimization Strategies

For very large datasets (10,000+ rows), consider these strategies:

1. **Partition Data**: Split into multiple sheets by category
2. **Lazy Loading**: Load sheets only when needed
3. **Application Cache**: Cache SpreadsheetRow objects at application level
4. **Preload Critical Data**: Use `RebuildIndices()` for frequently accessed sheets

Example: Lazy Loading with Application Cache

```
public class DataManager : MonoBehaviour
{
    private Dictionary<string, Dictionary<string, SpreadsheetRow>>
_cache;

    void Awake()
    {
        _cache = new Dictionary<string, Dictionary<string,
SpreadsheetRow>>();
    }
```

```
        public SpreadsheetRow GetData(string sheetName, string id)
        {
            // Ensure sheet cache exists
            if (!_cache.ContainsKey(sheetName))
            {
                _cache[sheetName] = new Dictionary<string, SpreadsheetRow>
    ();
            }

            // Check application cache
            if (!_cache[sheetName].ContainsKey(id))
            {
                // Query SpreadsheetReader (uses index cache)
                _cache[sheetName][id] = SpreadsheetReader.GetRow("GameData",
    sheetName, "ID", id);
            }

            return _cache[sheetName][id];
        }
    }
```

Troubleshooting

Q: Spreadsheet files not loading
A: Make sure your Spreadsheet files are placed in `StreamingAssets` folder, and call `LoadAllSpreadsheet()` at startup. Check the Console for any error messages.

Q: Example scenes cannot run or show errors
A: If example scenes cannot execute, it may be because the example Spreadsheet files are missing from the StreamingAssets folder. Example files are automatically copied to the StreamingAssets folder when you first import the asset. If the automatic copy did not occur, or you accidentally deleted the example files, you can manually execute the menu item `Tools` → `Spreadsheet Easy` → `Copy Example Files To StreamingAssets` to copy all example Spreadsheet files to the StreamingAssets folder.

Note: The original example files are stored in `Assets/SpreadsheetEasy/Examples/SpreadsheetExamples`. If you don't need the example files for the demo scenes, you can safely delete the `Assets/StreamingAssets/SpreadsheetExamples` folder. When you need them again, simply execute the menu item `Tools` → `Spreadsheet Easy` → `Copy Example Files To StreamingAssets` once to restore the example files.

Q: Queries are still slow
A: Make sure you're not clearing cache unnecessarily. Check for `ClearCache()` or `ClearIndexCache()` calls in your code. Alternatively, you can use ScriptableObject mode for better performance:

1. Open `SpreadsheetEasySettings.asset` (via menu: `Tools` → `Spreadsheet Easy` → `Settings`)
2. Change `Mode` to `ScriptableObject`
3. Set `Spreadsheet Source Folder` to your Spreadsheet files folder
4. ScriptableObject assets will be automatically generated, providing zero parsing cost at runtime

Q: How do I switch between RuntimeParsing and ScriptableObject modes?
A: You can switch modes using the Settings ScriptableObject:

1. Open `SpreadsheetEasySettings.asset` via menu: `Tools` → `Spreadsheet Easy` → `Settings`
2. Change the `Mode` dropdown to your desired mode:
    - **RuntimeParsing**: Parse Spreadsheet files at runtime from StreamingAssets (default)
    - **ScriptableObject**: Load pre-baked ScriptableObject assets for zero parsing cost
3. If switching to ScriptableObject mode, make sure to set `Spreadsheet Source Folder` to your Spreadsheet files folder
4. The settings are automatically loaded at runtime, so you can switch modes without code changes

Q: Memory usage is high
A: Use `ClearIndexCache()` to free index memory while keeping data loaded. Or use application-level caching with selective loading.

Q: First query is slow
A: Make sure you call `LoadAllSpreadsheet()` at startup. It automatically rebuilds indices for optimal performance.

Q: How do I know if index cache is working?
A: Use `SpreadsheetEasyPerformanceTest` to measure performance. You should see 50-100x speedup.

Q: Crash on iOS build with System.ExecutionEngineException
A: This is caused by Code Stripping removing necessary assemblies. You must provide a `link.xml` file in your plugin root directory to prevent stripping of required assemblies.

Create a file named `link.xml` in `Assets/SpreadsheetEasy/` (or your plugin root directory) with the following content:

```
<linker>
    <assembly fullname="ExcelDataReader" preserve="all"/>
</linker>
```

This prevents Unity's code stripper from removing required types and methods, which can cause crashes on iOS builds.

Q: Multiple precompiled assemblies with the same name error
A: This error occurs when multiple plugins in your project include the same DLL (e.g., ExcelDataReader.dll). If you encounter this error, please delete the `ExcelDataReader.dll` from this asset's folder and use the one already in your project.

To fix:

1. Locate the ExcelDataReader.dll in `Assets/SpreadsheetEasy/ExcelDataReader/` (or similar location in this asset)

2. Delete or remove it from the project
3. Use the ExcelDataReader.dll that is already present in another plugin or in your project

Performance Test Example Output

```
[SpreadsheetEasy] Starting performance test...
[SpreadsheetEasy] File: GameData, Sheet: Items
[SpreadsheetEasy] Query: ID='1001' -> Name
[SpreadsheetEasy] Query count: 1000

[SpreadsheetEasy] Test 1: Without index cache (clearing cache each time)
[SpreadsheetEasy] Time without index: 487.32ms
[SpreadsheetEasy] Average per query: 0.4873ms

[SpreadsheetEasy] Test 2: With index cache (indices cached and reused)
[SpreadsheetEasy] Time with index: 9.45ms
[SpreadsheetEasy] Average per query: 0.0095ms

[SpreadsheetEasy] ========== PERFORMANCE RESULTS ==========
[SpreadsheetEasy] Speedup: 51.57x faster
[SpreadsheetEasy] Time saved: 477.87ms (98.1%)
[SpreadsheetEasy] ========================================
```

Examples

Two example scenes are provided to help you get started with SpreadsheetEasy:

SpreadsheetEasyDemo.unity

A comprehensive demo scene that demonstrates all basic features of SpreadsheetEasy. This scene includes:

Location: `Assets/SpreadsheetEasy/Examples/SpreadsheetEasyDemo.unity`

Script: `SpreadsheetEasyDemo.cs`

Features Demonstrated:

1. **Single Value Reading**: Read individual values (string, int, float, bool, Enum) from Spreadsheet
2. **Multiple Values from Same Row**: Use `SpreadsheetRow` to efficiently retrieve multiple columns
3. **Different Data Types**: Examples for Vector3, Color, arrays (int\], $float\[$, string[]), and Enum
4. **Multiple Rows Query**: Get all rows matching a search condition using `GetAllRows()`
5. **Multiple Search Conditions**: Query with multiple conditions using `Dictionary<string, string>`
6. **TryGet Methods**: Clean error handling without try-catch blocks
7. **Enum Support**: Read Enum values from Spreadsheet with case-insensitive string parsing or integer values

How to Use:

1. Open `Assets/SpreadsheetEasy/Examples/SpreadsheetEasyDemo.unity` in Unity
2. Make sure example Spreadsheet files are in `StreamingAssets` folder (see Troubleshooting if files are missing)
3. Play the scene and check the Console for output
4. Examine `SpreadsheetEasyDemo.cs` to see code examples for each feature

Example Spreadsheet File: Uses `PlayerConfig.xlsx` with sheet `Players`

SpreadsheetEasyPerformanceTest.unity

A performance testing scene that demonstrates the speedup provided by SpreadsheetEasy's index cache system.

Location: `Assets/SpreadsheetEasy/Examples/SpreadsheetEasyPerformanceTest.unity`

Script: `SpreadsheetEasyPerformanceTest.cs`

Features:

1. **Performance Benchmarking**: Compare query performance with and without index cache
2. **GetRow Performance Test**: Measure performance when accessing multiple columns
3. **Cache Statistics**: Display detailed cache statistics (files, sheets, rows, columns, memory usage)
4. **Benchmark Comparison**: Compare `GetString` vs `GetRow` for multiple column access

How to Use:

1. Open `Assets/SpreadsheetEasy/Examples/SpreadsheetEasyPerformanceTest.unity` in Unity
2. Select the `SpreadsheetEasyPerformanceTest` GameObject in the scene
3. Configure test settings in the Inspector:
    - `TestFileName` : Spreadsheet file name to test (e.g., "ItemConfig")
    - `TestSheetName` : Sheet name to test (e.g., "Items")
    - `SearchColumn` : Column name to search (e.g., "Rarity")
    - `SearchValue` : Value to find (e.g., "Rare")
    - `TargetColumn` : Column to retrieve (e.g., "Name")
    - `QueryCount` : Number of queries to perform (default: 1000)
4. Enable `RunTestOnStart` to run test automatically when scene starts, or use Context Menu:
    - Right-click the component → `Run Performance Test`
    - Right-click the component → `Test GetRow Performance`
    - Right-click the component → `Show Cache Statistics`
    - Right-click the component → `Benchmark: GetString vs GetRow`
5. Check the Console for detailed performance results

Expected Results:

You should see **50-100x speedup** when using index cache compared to clearing cache each time. Example output:

```
[SpreadsheetEasy] ========== PERFORMANCE RESULTS ==========
[SpreadsheetEasy] Speedup: 51.57x faster
[SpreadsheetEasy] Time saved: 477.87ms (98.1%)
[SpreadsheetEasy] ========================================
```

Example Spreadsheet File: Uses `ItemConfig.xlsx` with sheet `Items`

Example Spreadsheet Files

Example Spreadsheet files are located in `Assets/SpreadsheetEasy/Examples/SpreadsheetExamples/` and are automatically copied to `StreamingAssets/SpreadsheetExamples/` when you first import the asset.

If example files are missing, you can manually copy them using:

- Menu: `Tools` → `Spreadsheet Easy` → `Copy Example Files To StreamingAssets`

Note: In production environments, you can safely delete the `StreamingAssets/SpreadsheetExamples/` folder if you don't need the example files.

For more information, see the Troubleshooting section.

Notes

- Spreadsheet files must be placed in the `StreamingAssets` folder (or subdirectories within it).
- Spreadsheet file names must be unique across all subdirectories (file names without path/extension are used as cache keys).
- The first row of each sheet must contain column headers; data rows start from the second row.

- Sheet names and column names are case-insensitive but must match exactly (ignoring case).

Third-Party Licenses

ExcelDataReader

This asset uses ExcelDataReader, a lightweight and fast library written in C# for reading Microsoft Spreadsheet files.

ExcelDataReader is licensed under the MIT License:

```
MIT License

Copyright (c) ExcelDataReader contributors

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
```

```
    AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
    OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
    IN THE
    SOFTWARE.
```

For more information, visit: https://github.com/ExcelDataReader/ExcelDataReader