

# **Numba's JSON** **Документация**

## Содержимое документации

Как импортировать.....	3
Архитектура классов.....	3
Описание интерфейсов, классов и перечислений.....	4
Сопоставление интерфейсов и классов к данным JSON.....	4
От теории к практике.....	5
Подготовка сцены к изучению.....	5
Работа с классами-обертками.....	6
Работа с JsonObject, JsonField и JsonArray.....	8
Оптимизация.....	14
Сериализация JSON-структуры.....	14
Десериализация JSON-данных.....	15
Проблема десериализации JSON-данных.....	16
Советы.....	18

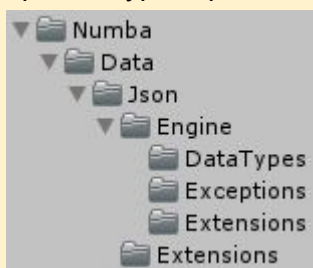
## Как импортировать

Скачайте и откройте **Numba's JSON** пэкэдж в вашем проекте. Вы увидите что **Numba's JSON** пэкэдж это всего лишь директория **Scripts/Numba/Data/Json** с необходимыми классами внутри.

У вас не должно возникнуть конфликтов имен классов при импорте.  
Кликните **Import** для завершения процесса импорта.

## Архитектура классов

Архитектура скриптов движка выглядит так:



Скрипты расположенные в этих директориях одновременно располагаются и в соответствующем пространстве имен.

Ключевыми директориями являются **Engine**, **DataTypes**, **Engine/Extensions** и **Json/Extensions**.

Рассмотрим их по порядку:

Директория	Назначение
Engine	Хранит некоторые основные классы (JsonObject, JsonArray и прочие) для работы с форматом JSON.
DataTypes	Хранит классы-обертки для всех системных типов данных (JsonInt, JsonBool, JsonChar и прочие).
Engine/Extensions	Хранит расширения для JsonField, IJsonValue добавляя в них дополнительные методы.
Json/Extensions	Хранит расширения для unity-классов (Vector2, Vector3, Vector4, Quaternion, Color, Color32 и прочие), добавляя в них дополнительные методы.

## Описание интерфейсов, классов и перечислений

Рассмотрим все необходимые интерфейсы, классы и перечисления подробнее:

Сущность	Описание
IJsonValue	любой json тип данных (JsonInt, JsonBool, JsonObject, JsonArray и прочие) реализует этот интерфейс.
JsonInt, JsonBool и прочие обертки	добавляют некоторое поведение к системным типам данных, json-движок работает только с этими обертками.
JsonObject	Представляет JSON-объект, который состоит из JsonField'ов, каждый из которых состоит из ключа (string) и значения (IJsonValue).
JsonField	Представляет элемент объекта JsonObject, состоит из строкового ключа и значения (IJsonValue). JsonField используется совместно с JsonObject.
JsonArray	Представляет JSON-объект, который хранит значения (IJsonValue).
JsonType	Перечисление JSON-типа (String, Number, Null, Bool, Object, Array, Unknown).
Json	Имеет методы для десериализации JSON-данных.

## Сопоставление интерфейсов и классов к данным JSON

Вот как выглядит сопоставление интерфейсов и классов к конкретным данным JSON:

```
{
  "name": "Zaur",
  "age": 25,
  "growth": 178.7,
  "married": true,
  "vehicle": {
    "brand": "Audi",
    "model": "A8",
    "year": 2017,
    "mileage": 2000,
    "owners": ["Osman", "Sultan", "Alexander"]
  },
  "childs": ["Adam", "Robert", "Adel"]
}
```

- JsonObject
- JsonField
- IJsonValue

Все что в фигурных скобках (серый цвет) представлено с помощью класса JsonObject. JsonObject сам по себе тоже реализует IJsonValue, это можно увидеть глядя на значение поля “vehicle”, которое тоже является JsonObject, однако отмечено голубым цветом.

Оранжевого цвета отмечены поля данного объекта, которые всегда имеют ключ (слева от двоеточия, и значение (справа от двоеточия), они представлены классом JsonField.

Значение поля подсвечено голубым цветом, оно всегда представлено каким-либо классом реализующим интерфейс IJsonValue, это могут быть и JsonInt, и JsonBool, и JsonObject и прочие.

## От теории к практике

Давайте двигаться от простого к сложному.

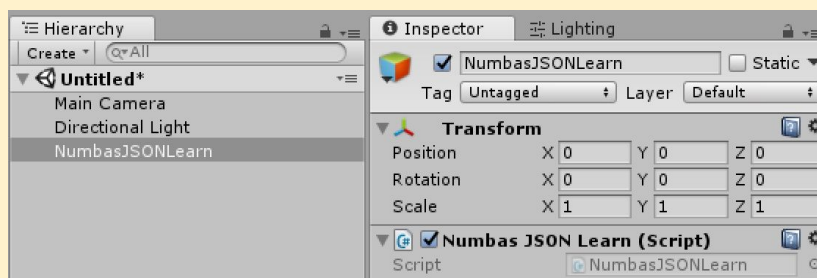
Мы будем изучать **Numba's JSON** в такой последовательности:

1. Научимся работать с классами-обертками системных типов данных;
2. Научимся использовать их в купе с такими классами как JsonField, JsonObject и JsonArray;
3. Изучим различные способы работы с JsonObject, JsonField и JsonArray;
4. Изучим сериализацию json-структуры и десериализацию json-данных.

## Подготовка сцены к изучению

Перейдите в папку Scripts и создайте C#-скрипт, назовите его “NumbasJSONLearn”, в самом скрипте оставьте только метод Start, остальное удалите.

Создайте новую сцену (если еще не создали), создайте новый пустой объект в сцене, назовите его “NumbasJSONLearn” и добавьте на него одноименный скрипт.



Все, теперь можно начинать изучение

## Работа с классами-обертками

Чтобы получить доступ к классам-оберткам подключите следующее пространство имен:

```
using Numba.Data.Json.Engine.DataTypes;
```

Теперь у вас есть доступ к классам-оберткам. Всего существуют следующие классы-обертки:

- JsonBool;
- JsonByte;
- JsonChar;
- JsonDecimal;
- JsonDouble;
- JsonFloat;
- JsonInt;
- JsonLong;
- JsonNull;
- JsonNumber;
- JsonSByte;
- JsonShort;
- JsonString;
- JsonUInt;
- JsonULong;
- JsonUShort;

Несмотря на их большое количество, работать с ними не составит труда, потому что работа с ними почти ничем не отличается от работы с соответствующими системными классами, а также в большинстве случаев, вы вообще не будете работать с этими обертками, так как везде где это возможно системные классы автоматически будут преобразовываться в классы-обертки и наоборот.

Особо внимательные наверно уже заметили 2 класса которые вызывают некое подозрение: JsonNull и JsonNumber.

JsonNull ничего не оборачивает и представляет значение null в терминах JSON, а JsonNumber оборачивает все числовые типы данных с помощью преобразования числа в строку. Они понадобятся при десериализации json-данных, в остальных случаях от них мало пользы.

Почему их наличие является необходимым будет объяснено в разделе “проблема десериализации JSON-данных”.

Все классы-обертки работают не с чистыми системными классами (int, float, bool и прочие), а с Nullable системными классами (int?, float?, bool?). Сделано это для того чтобы вы легко могли целочисленным полям присваивать null значение, иногда это бывает необходимо при работе с JSON.

И так, теперь, когда вы узнали что такое класс-обертка и как он работает с системным типом который оборачивает, самое время начать практиковаться.

Давайте начнем с JsonBool. Для того чтобы создать объект данного класса используйте конструктор по умолчанию, а затем выведите в консоль его строковое представление:

```
private void Start()
{
    JsonBool jBool = new JsonBool();
    print(jBool);
}
```

Нажмите Play в Unity и вы увидите что в консоль выведено значение null.

```
! null
UnityEngine.MonoBehaviour:print(Object)
```

Это потому что как мы уже говорили, все классы-обертки работают с Nullable системными типами, а они по умолчанию имеют значение null. Чтобы установить значение в JsonBool, можно передать его в конструктор:

```
JsonBool jBool = new JsonBool(true);
print(jBool);
```

Снова нажмите Play и вы увидите следующее:

```
! true
UnityEngine.MonoBehaviour:print(Object)
```

Все классы-обертки имеют перегруженный конструктор который принимает конкретное значение для оборачиваемого Nullable системного типа.

Другой способ изменить оборачиваемое значение это использовать свойство **Value** класса-обертки:

```
JsonBool jBool = new JsonBool(true);
jBool.Value = false;
print(jBool);
```

Результат в консоли:

```
! false
UnityEngine.MonoBehaviour:print(Object)
```

Есть еще один способ установить значение в объект класса-обертки - просто присвоить значение:

```
JsonBool jBool = true;
print(jBool);
```

Результат:

```
! true
UnityEngine.MonoBehaviour:print(Object)
```

Это наиболее прозрачный и предпочтительный вариант установки значения в класс-обертку.

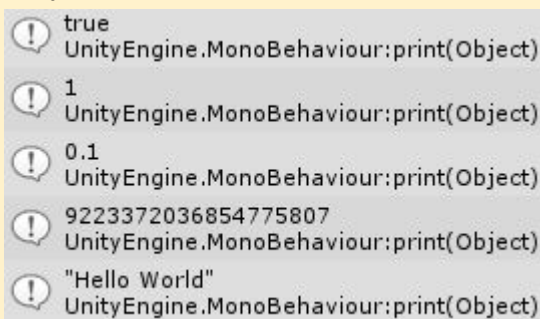
Все классы-обертки имеют неявный оператор преобразования Nullable системного типа в себя (в класс-обертку), кроме JsonNull.

Все что мы сделали с JsonBool можно сделать и с любым другим классом-оберткой (кроме JsonNull):

```
JsonBool jBool = true;
JsonInt jInt = 1;
JsonFloat jFloat = 0.1f;
JsonLong jLong = long.MaxValue;
JsonString jString = "Hello World";

print(jBool);
print(jInt);
print(jFloat);
print(jLong);
print(jString);
```

Результат:



The screenshot shows the Unity console with five log entries, each preceded by an exclamation mark icon. The entries are: 'true' from 'UnityEngine.MonoBehaviour:print(Object)', '1' from 'UnityEngine.MonoBehaviour:print(Object)', '0.1' from 'UnityEngine.MonoBehaviour:print(Object)', '9223372036854775807' from 'UnityEngine.MonoBehaviour:print(Object)', and '"Hello World"' from 'UnityEngine.MonoBehaviour:print(Object)'.

Можете поэкспериментировать со всеми классами-обертками самостоятельно.

## Работа с JsonObject, JsonField и JsonArray

Для того чтобы работать с JsonObject, JsonField и JsonArray необходимо подключить следующее пространство имен:

```
using Numba.Data.Json.Engine;
```

### JsonField

Принцип работы с JsonField следующий:

1. Создать JsonField<T> (T - тип значения поля), передав значение имени поля и значения поля.

Вот так выглядит создание строкового и числового полей:

```
JsonField<JsonString> jStringField = new JsonField<JsonString>("name", "Zaur");
JsonField<JsonInt> jIntField = new JsonField<JsonInt>("age", 25);
```

Можно менять имя и значение поля после его создания:



```
jStringField.Name = "brand";
jStringField.Value = "Audi";

print(jStringField);
print(jIntField);
```

Результат вывода полей в консоль:

```
! "brand":"audi"
  UnityEngine.MonoBehaviour:print(Object)
! "age":25
  UnityEngine.MonoBehaviour:print(Object)
```

## JsonObject

Принцип работы с JsonObject следующий:

1. Создайте JsonObject;
2. Добавляете в него поля.

Имена полей не должны совпадать, иначе будет брошено исключение.

Самый примитивный вариант добавить поля в JsonObject выглядит так:

```
JsonObject jobject = new JsonObject();
jobject.Add("name", "Zaur");
jobject.Add("age", 25);
```

Но можно передать массив полей прямо в конструктор:

```
JsonField<JsonString> jStringField = new JsonField<JsonString>("name", "Zaur");
JsonField<JsonInt> jIntField = new JsonField<JsonInt>("age", 25);

JsonField[] fields = new JsonField[] { jStringField, jIntField };

JsonObject jobject = new JsonObject(fields);

print(jobject);
```

Или вызвать метод AddRange после конструктора:

```
JsonField<JsonString> jStringField = new JsonField<JsonString>("name", "Zaur");
JsonField<JsonInt> jIntField = new JsonField<JsonInt>("age", 25);

JsonField[] fields = new JsonField[] { jStringField, jIntField };

JsonObject jobject = new JsonObject();
jobject.AddRange(fields);

print(jobject);
```

Или использовать синтаксис инициализации объектов:

```
JsonObject jobject = new JsonObject() { { "name", "Zaur" }, { "age", 25 } };

print(jobject);
```

Результат для всех примеров:

```
! {"name": "Zaur", "age": 25}
UnityEngine.MonoBehaviour:print(Object)
```

Вот основные методы и индексаторы класса JsonObject:

**Add** - добавляет новое или уже существующее поле в объект;

**AddRange** - добавляет множество полей за раз;

**Insert** - вставляет новое или уже существующее поле в объект;

**InsertRange** - вставляет множество полей за раз начиная с указанной позиции;

**RemoveField** - удаляет поле по названию или ссылке;

**RemoveFieldAt** - удаляет поле по индексу;

**RemoveAllFields** - удаляет поля по фильтру;

**Clear** - удаляет все поля;

**GetField** - находит поле по имени;

**Contains** - Проверяет имеется-ли поле с указанным именем;

**GetFieldIndex** - возвращает индекс поля по имени;

**Find** - ищет первое попавшееся поле по фильтру;

**Get + имя типа** - возвращает значение поля по его имени;

**Set + имя типа** - устанавливает значение поля по его имени;

**this[int index]** - возвращает поле по индексу;

**this[string fieldName]** - возвращает значение поля (как IJsonValue) по его имени;

**IsNullField** - проверяет является ли значение поля с указанным именем JsonNull'ом;

**ToString** - сериализует данный объект.

Индексатор **this[int index]** в качестве результата возвращает поле необобщенного типа **JsonField**. С помощью **JsonField** невозможно изменять значение поля, это можно сделать только с помощью обобщенной версии **JsonField<T>**. Для того чтобы легко преобразовывать **JsonField** в нужный тип подключите следующее пространство имен:

```
using Numba.Data.Json.Engine.Extensions;
```

Теперь для того чтобы привести тип **JsonField** к **JsonField<T>** достаточно вызвать метод **As+тип** на объекте:

```
JsonObject jobject = new JsonObject() { { "name", "Zaur" }, { "age", 25 } };
jobject[1].AsInt().Value = 26;
print(jobject);
```

результат:

```
! {"name": "Zaur", "age": 26}
UnityEngine.MonoBehaviour:print(Object)
```

То-же самое касается и индексатора **this[string fieldName]**, который вернет вам не конкретный тип представляющий значение поля, а более общий - **IJsonValue**:

```
JsonObject jobject = new JsonObject() { { "name", "Zaur" }, { "age", 25 } };
int age = jobject["age"].AsInt().Value;
```

Вот пример того как можно работать с JsonObject:

```
// Here we create and use initialize syntax
JsonObject person = new JsonObject()
{
    { "name", "Zaur" },
    { "age", 25 },
    { "growth", 128.7f },
    { "married", true },
    { "vehicle", new JsonObject(){
        { "brand", "Audi" },
        { "model", "A8" },
        { "year", 2017 },
        { "mileage", 2000f },
        { "owners", new JsonArray(){ "Osman", "Sultan", "Alexander" } } } },
    { "childs", new JsonArray(){ "Adam", "Robert", "Adel" } }
};

// Add one more field
person.Add("speciality", "Builder");
// Change field value
person.SetString("speciality", "Programmer");
// Change field name
person.GetField("speciality").Name = "credo";
// Remove fields with name "growth" and "married"
person.RemoveField("growth");
person.RemoveField("married");
// Get int value from field with "age" name
int age = person.GetInt("age").Value;
// Set value to field with "age" name
person.SetInt("age", age + 1);
// Get vehicle object
JsonObject vehicle = person.GetObject("vehicle");
// Insert color field at 3 position
vehicle.Insert(2, "color", "Black");
// Get "age" field by index and set old value again
person[1].As<JsonInt>().Value = 25;
// Get person vehicle owners by 2 ways
JsonArray ownersBy1Way = person["vehicle"].AsObject()["owners"].AsArray();
JsonArray ownersBy2Way = person.GetObject("vehicle").GetArray("owners");
// Get fields count
int fieldsCount = person.Count;
// Cycle for all fields
foreach (JsonField field in person)
{
    // Do something
}

// Implicit call ToString and log to console
print(person);
```

Результат в консоли:

```
{
  "name": "Zaur",
  "age": 25,
  "vehicle": {
    "brand": "Audi",
    "model": "A8",
    "color": "Black",
    "year": 2017,
    "mileage": 2000,
    "owners": ["Osman", "Sultan", "Alexander"]
  },
  "childs": ["Adam", "Robert", "Adel"],
  "credo": "Programmer"
}
```

## JsonArray

Принцип работы с классом JsonArray:

1. Создать JsonArray;
2. Добавить элементы;

Самый примитивный вариант работы с JsonArray выглядит так:

```
JsonArray jArray = new JsonArray();
jArray.Add(true);
```

Можно передать массив элементов через конструктор:

```
IJsonValue[] elements = new IJsonValue[] { new JsonInt(1), new JsonBool(true), new JsonString("Hi") };
JsonArray jArray = new JsonArray(elements);

print(jArray);
```

Или вызвать метод AddRange:

```
IJsonValue[] elements = new IJsonValue[] { new JsonInt(1), new JsonBool(true), new JsonString("Hi") };

JsonArray jArray = new JsonArray() { };
jArray.AddRange(elements);

print(jArray);
```

Или через синтаксис инициализации объектов:

```
JsonArray jArray = new JsonArray() { 1, true, "Hi" };

print(jArray);
```

Результат для всех:

```
[1,true,"Hi"]
UnityEngine.MonoBehaviour:print(Object)
```

Основными методами и индексаторами для JSONArray являются:

**Add** - добавляет новый элемент в конец;

**AddRange** - добавляет множество элементов в конец;

**Insert** - вставляет элемент в позицию;

**InsertRange** - вставляет множество элементов в позицию;

**Remove** - удаляет первый совпадающий элемент из списка;

**RemoveAt** - удаляет элемент по индексу;

**RemoveAll** - удаляет элементы по фильтру;

**Clear** - удаляет все элементы из списка;

**Find** - находит первый совпадающий элемент по фильтру;

**FindAll** - находит все совпадающие элементы по фильтру;

**Contains** - проверяет содержится ли переданный элемент в списке;

**this[int index]** - возвращает элемент по индексу или записывает по индексу;

**ToString** - сериализует данный объект.

Для некоторых методов возвращающих IJsonValue, а также для индексатора удобно подключать расширяющие методы из пространства:

```
using Numba.Data.Json.Engine.Extensions;
```

Это позволит вам использовать метод As+тип для приведения к конкретному типу:

```
JSONArray jArray = new JSONArray() { 1, true, 3.14f, "Hi" };  
float pi = jArray[2].AsFloat().Value;
```

Вот как можно использовать JSONArray:

```
// Create JSONArray with initialize syntax  
JSONArray jArray = new JSONArray() { true, 3.14f, true, "Hi", null };  
// Remove first true element, now result is [3.14f, true, "Hi"]  
jArray.Remove(true);  
// Add new element  
jArray.Add(decimal.MaxValue);  
// Insert some value between 3 and 4 elements  
jArray.Insert(3, "Between true and Hi");  
// Create new JsonObject  
JsonObject jobject = new JsonObject() { { "name", "Zaur" } };  
// Add jArray to jobject through field  
jobject.Add("jArray", jArray);  
// Foreach cycle for elements  
foreach (IJsonValue value in jArray)  
{  
    // Do something  
}  
// For cycle for elements  
for (int i = 0; i < jArray.Count; i++)  
{  
    // Do something  
}  
  
print(jobject);
```

Результат:

```
! {"name":"Zaur","jArray":[3.14,true,"Hi","Between Hi and null",null,79228162514264337593543950335]}  
UnityEngine.MonoBehaviour:print(Object)
```

Запрещается использовать ключевое слово null вместо JsonNull, в противном случае будет брошено исключение. Это касается и JsonObject, и JsonField, и JSONArray.



## Оптимизация

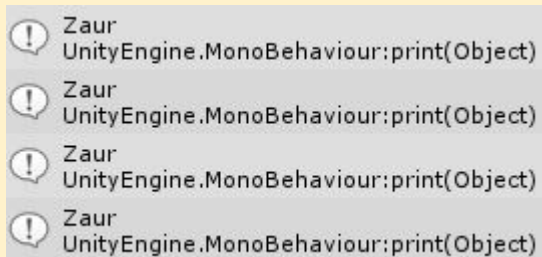
**Numba's JSON** использует линейный алгоритм для сериализации/десериализации данных. Это означает что вы можете не волноваться на этот счет. Единственное на счет чего вам стоит немного волноваться - код который пишете вы, потому что одну и ту же цель можно достичь несколькими способами, вот пример:

```
JsonObject jobject = new JsonObject() { { "name", "Zaur" } };

string s1 = jobject.GetField("name").BaseValue.AsString();
string s2 = jobject.GetField("name").AsString().Value.Value;
string s3 = jobject.GetString("name");
string s4 = jobject["name"].AsString();

print(s1);
print(s2);
print(s3);
print(s4);
```

Результат:



```
Zaur
UnityEngine.MonoBehaviour:print(Object)
Zaur
UnityEngine.MonoBehaviour:print(Object)
Zaur
UnityEngine.MonoBehaviour:print(Object)
Zaur
UnityEngine.MonoBehaviour:print(Object)
```

Как видите результат один и тот-же. Обычно вам следует использовать самый короткий способ, потому что обычно он и есть самый оптимизированный. Для того чтобы понимать какой способ лучше использовать вам придется внимательно изучить методы классов JsonObject, JsonField, JsonArray и некоторых других классов.

## Сериализация JSON-структуры

Для того чтобы сериализовать какой-либо json объект, достаточно вызвать метод ToString на нем:

```
JsonObject person = new JsonObject()
{
    { "name", "Zaur" },
    { "age", 25 },
    { "growth", 178.7f },
    { "married", true },
    { "haters", new JsonNull() },
    { "Vehicle", new JsonObject(){
        { "brand", "Audi" },
        { "model", "A8" },
        { "color", "Black" },
        { "owners", new JsonArray(){ "Osman", "Sultan", "Alexander" } } } }
};

string serializedPerson = person.ToString();
```

Метод ToString вернет строку, которая хранит сериализованные данные, то есть json-данные. Эти данные затем можно сохранить на диске, отправить по почте, записать в поток и прочее.

## Десериализация JSON-данных

Чтобы десериализовать JSON-данные используется класс Json. Он располагается в следующем пространстве имен:

```
using Numba.Data.Json.Engine;
```

Класс Json имеет 3 основных метода:

1. **GetSupposedType** - вернет предполагаемый тип JSON данных;
2. **Parse** - десериализует JSON-данные в структуру и вернет IJsonValue;
3. **Parse<T>** - десериализует JSON-данные в структуру и вернет IJsonValue как T

Для десериализации используйте метод Parse<T>. Обобщенный параметр T должен соответствовать типу JSON-данных которые вы передали. Если JSON-данные представлены как объект, а вы вместо T укажете JsonArray - будет брошено исключение.

Давайте десериализуем в структуру JSON-данные которые мы сериализовали прежде:

```
JsonObject person = new JsonObject()
{
    { "name", "Zaur" },
    { "age", 25 },
    { "growth", 178.7f },
    { "married", true },
    { "haters", new JsonNull() },
    { "Vehicle", new JsonObject(){
        { "brand", "Audi" },
        { "model", "A8" },
        { "color", "Black" },
        { "owners", new JsonArray(){ "Osman", "Sultan", "Alexander" } } } }
};

string serializedPerson = person.ToString();

JsonObject deserializedPerson = Json.Parse<JsonObject>(serializedPerson);

print(deserializedPerson);
```

Результат:

```
! {"name":"Zaur","age":25,"growth":178.7,"married":true,"haters":null,"Vehicle":{"brand":"Audi","model":"A8","color":"Black","owners":["Osman","Sultan","Alexander"]}}
UnityEngine.MonoBehaviour:print(Object)
```

Как видим десериализация прошла отлично.

## Проблема десериализации JSON-данных

Десериализация прошла отлично. Но есть одна тонкость. Если при десериализации встречается числовое поле, то невозможно узнать какого типа конкретно (byte, short, int, long, float, double, decimal) было это поле до сериализации. То-же самое касается и полей, значением которых при десериализации оказался null - невозможно узнать какой тип был у этого поля до сериализации. По этой причине существуют 2 класса **JsonNumber** и **JsonNull**.

Любое число в процессе десериализации обортывается классом **JsonNumber** и считывание или запись этого поля осуществляется посредством этого класса.

Любое null-поле в процессе десериализации обортывается **JsonNull** классом и считывание этого поля осуществляется посредством этого класса.

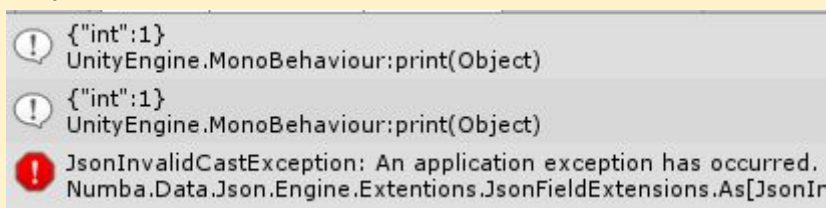
Давайте напишем маленький пример в котором создадим JsonObject с числовым полем типа int, сериализуем его, а затем попробуем десериализовать и вытащить то самое число из него как int:

```
JsonObject jobject = new JsonObject() { { "int", 1 } };
string jsonData = jobject.ToString();
print(jsonData);

JsonObject parsedJObject = Json.Parse<JsonObject>(jsonData);
print(parsedJObject);

int value = parsedJObject.GetInt("int").Value;
```

Результат:



```
{\"int\":1}
UnityEngine.MonoBehaviour:print(Object)
{\"int\":1}
UnityEngine.MonoBehaviour:print(Object)
JsonInvalidCastException: An application exception has occurred.
Numba.Data.Json.Engine.Extentions.JsonFieldExtensions.As[JsonIr]
```

Как видим сериализация и десериализация прошли успешно, а вот извлечение числа как тип int привело к исключению. Это потому что при десериализации данное число трактуется как **JsonNumber** а не как **JsonInt**. **Запомните**, любое число вне зависимости от типа, при десериализации обортывается классом **JsonNumber**, и работать с ним надо, естественно, как с **JsonNumber**.

Вот корректный пример строчки с извлечением числа из **JsonNumber**:

```
int value = parsedJObject.GetNumber("int").ToInt();
```

**GetNumber** вернет нам число обернутое в класс **JsonNumber**, у этого класса есть ряд методов преобразования себя в конкретный числовой тип, имена таких методов начинаются на **To+тип**, в данном случае мы использовали **ToInt**.



А теперь давайте тоже самое сделаем с null-полем. Переделаем немного пример в соответствии с этим рисунком:

```
JsonObject jobject = new JsonObject() { { "int", 1 } };
jobject.SetInt("int", null);
string jsonData = jobject.ToString();
print(jsonData);

JsonObject parsedJObject = Json.Parse<JsonObject>(jsonData);
print(parsedJObject);
```

Тут мы перезаписываем значение поля "int" в null. Это значит что при десериализации значение данного поля будет установлено в JsonNull. А теперь задумайтесь, есть ли смысл извлекать null из объекта поле значение которого null и работать с ним? Что с таким полем можно сделать? Ответ такой - смысла нет и ничего с таким полем не сделаешь, оно ведь null.

По этой причине у JsonObject класса нет метода GetNull, но есть метод IsNullField, который по имени поля проверяет является ли его значение типом JsonNull.

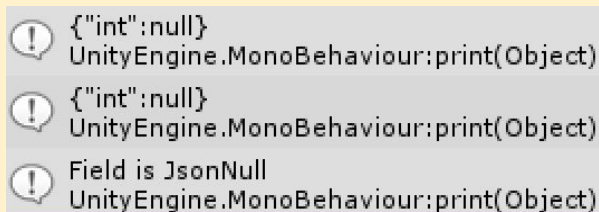
Пример с использованием IsNullField метода выглядит так:

```
JsonObject jobject = new JsonObject() { { "int", 1 } };
jobject.SetInt("int", null);
string jsonData = jobject.ToString();
print(jsonData);

JsonObject parsedJObject = Json.Parse<JsonObject>(jsonData);
print(parsedJObject);

if (parsedJObject.IsNullField("int"))
{
    print("Field is JsonNull");
}
else
{
    print(string.Format("Field is JsonNumber with value: {0}", parsedJObject.GetNumber("int")));
}
```

Результат:



```
{ "int": null }
UnityEngine.MonoBehaviour:print(Object)
{ "int": null }
UnityEngine.MonoBehaviour:print(Object)
Field is JsonNull
UnityEngine.MonoBehaviour:print(Object)
```

Как видим, метод IsNullField вернул значение true, указывая тем самым что в поле "int" записано значение JsonNull.

Несмотря на то, что у JsonObject класса нет метода GetNull, все-же есть один способ извлечь из поля JsonNull:

```
JsonNull jNull = parsedJObject.GetField("int").AsNull().Value;
```

Этим способом мы сначала извлекаем само поле "int", затем с помощью расширяющего метода AsNull приводим его к типу JsonField<JsonNull>, а уже затем извлекаем из этого обобщенного поля значение JsonNull. Но в этом нет никакого смысла, потому что с JsonNull ничего сделать нельзя, именно поэтому существует метод IsNullField.

## Советы

Ни в коем случае не создавайте циклические ссылки между JsonObject и JsonArray объектами, иначе это приведет к зависанию приложения на стадии сериализации и бесконечному выделению памяти.

Не извлекайте JsonNull из полей, просто проверяйте поле на JsonNull с помощью метода IsNullField.

Не забывайте, что при сериализации происходит потеря конкретных числовых типов данных у полей, а десериализация будет использовать JsonNumber.

Везде где это возможно, пользуйтесь неявным приведением системных типов данных к классам-оберткам.

Не забывайте, пользоваться самым коротким путем достижения цели, потому как самый короткий путь - самый производительный.