Getting Started

Importing the package

Unity .NET Framework / Core

1. Add the package to your *Unity Account* from the *Unity Asset Store* (https://assetstore.unity.com/packages/tools/network)
2. Import the package into your project from the *Unity Engine*

Setup

In this example we define a struct that we want serialize/deserialize. Theoretically this can be any class, struct or primitive type of your choosing.

```
public struct PositionVector
{
    float x;
    float y;
    float z;
}
```

NOTE

The library supports most primitive types, this means that it is possible to work with highly complex objects. However, it is recommended to **keep objects as simple as possible**.

Buffer

All forms of serialization/deserialization operate using the **PacketBuffer** class. The PacketBuffer is a simple buffer of fixed size that can be **constructed by specifying number of bytes**. A PacketBuffer may also be **constructed by supplying a byte[]**, this is desirable when for example a byte[] has been received from a remote location.

```
// We first define a buffer to serialize into
PacketBuffer buffer(1024);
```

NOTE

When constructing a PacketBuffer, the **number of bytes need to be evenly divisible by 4**. This is because the library uses an internal scratch buffer to facilitate speed.

👉 WARNING

When constructing a PacketBuffer from a byte[] it is **not allowed** to modify the byte[] externally unless it is first copied.

Serialization

Serialization is performed using the **PacketWriter** class. The class operates on a PacketBuffer to which it sequentially writes bits.

```
// We need a writer to help us with writing into the buffer
PacketWriter writer(buffer);

// Assuming that we have a position vector that we want to serialize
PositionVector positionIn;

// We use the writer to pack each component of the vector
writer.PackFloat(positionIn.x);
writer.PackFloat(positionIn.y);
writer.PackFloat(positionIn.z);

// ...

// In a real environment, we would pack all other objects into the buffer here

// ...
```

When all objects have been packed into the buffer the only thing that remains is to **finalize the writer and buffer**. This is done to ensure that all bits are accounted for.

```
// Once we are done packing all objects we are required to finalize the writer and buffer
writer.FlushFinalize();

// The buffer is now finalized and ready to be read by a PacketReader! (next step)

// In a real environment, we could send the buffer to another computer using buffer.GetByte
s()
```

⚠ IMPORTANT

It is always **required to call FlushFinalize()** on the PacketWriter. Failing to do so is likely to cause some bits not being properly written into the underlying PacketBuffer.

Deserialization

Deserialization is performed using the **PacketReader** class. The class operates on a PacketBuffer from which it sequentially reads bits.

```
// In a real environment, we could construct the buffer from another computer using byte[]

// Assuming that we have access to the buffer from the previous step

// We need a reader to help us with reading from the buffer
PacketReader reader(buffer);

// Assuming that we have a position vector to deserialize into
PositionVector positionOut;

// We use the writer to pack each component of the vector
reader.UnpackFloat(positionOut.x);
reader.UnpackFloat(positionOut.y);
reader.UnpackFloat(positionOut.z);

// ...

// In a real environment, we would unpack all other objects from the buffer here

// ...

// The objects now contain all information that we serialized in the previous step!
```

⚠ IMPORTANT

When reading from the PacketReader, the order of reading is important. More specifically, **the order in which objects were written is the order that the objects must be read**.

👉 WARNING

Before constructing the PacketReader, the buffer is required to be in a finalized state. This means that a **call to FlushFinalize() must have been made prior to construction**.

What's next

This short guide covers the very basics of serialization and deserialization. Further reading includes **advanced topics that provide much more efficient solutions**.

Suggested topics are **Range Quantization** (e.g optimizing integers with limited range) and **Precision Quantization** (e.g optimizing floats with limited precision).