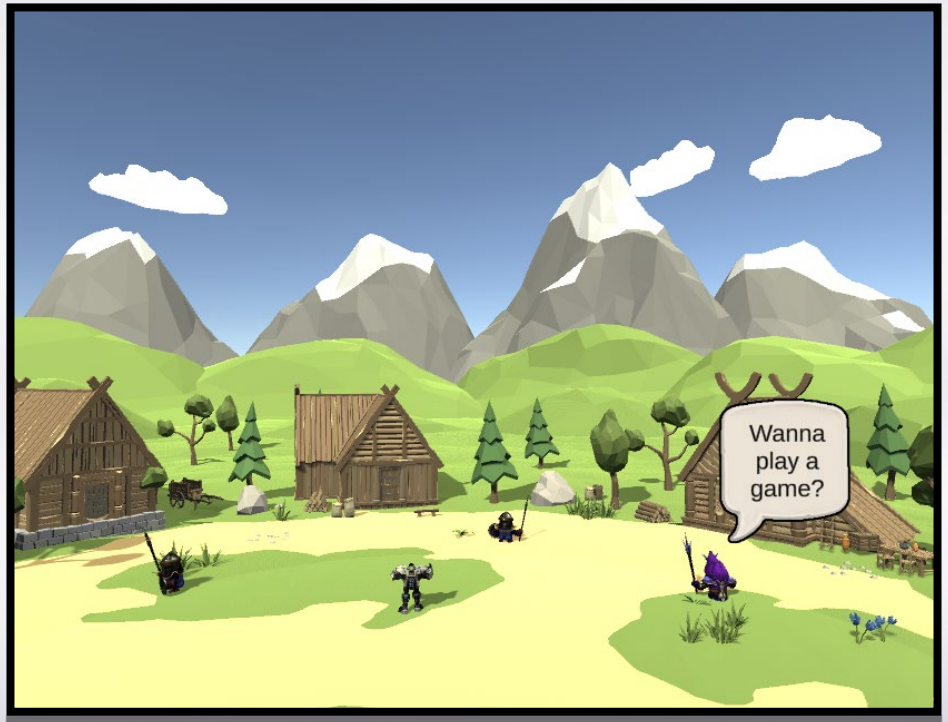
Networking lecture 3

Protocol design and implementation



Recap & plan for today

- Recap
  - Basic networking
  - "Correct" TCP communication (bounded / non blocking)

- Plan for today: protocol design and implementation
  - Explanation of object based application protocols & (de)serialization of objects
  - Explanation of assignment 3 – 3D Avatar chat lobby

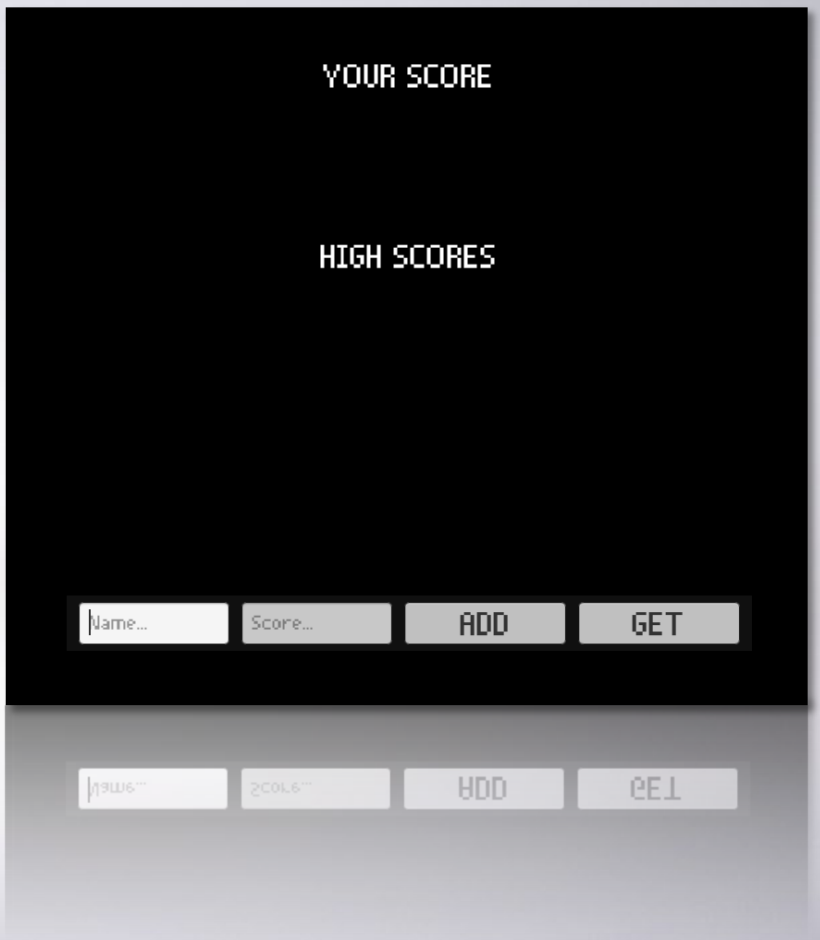
What is an application protocol?

"An agreement on the messages and the format of the messages that go back and forth between client and server in order to implement the required features for a specific application"

- Compared with something like a chatbox, that usually means:
  - more message types
  - more complicated messages
  - messages have meaning

Simple application example: high score server

- Very basic functionality:
  - We can add a score
  - We can get scores
- Let's look at the non-networked version first:
  - 001_highscoreview_ui_only

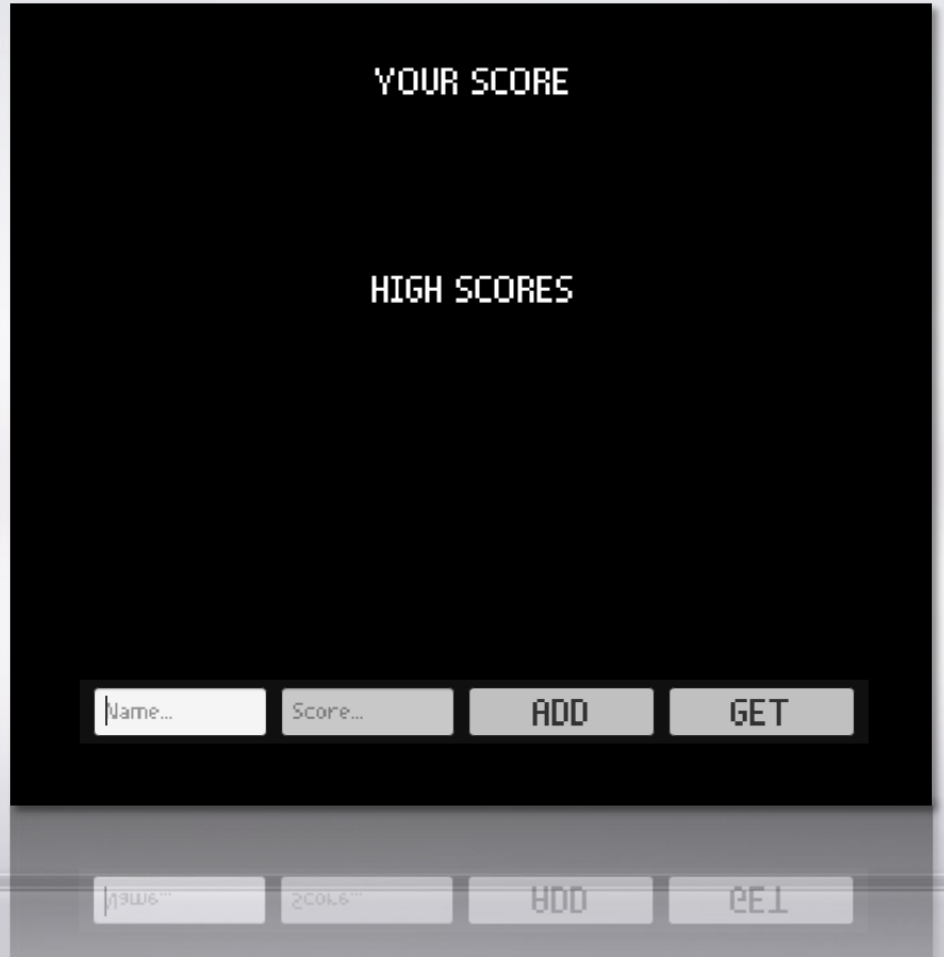


Example 001 setup

- Main scripts:
  - TcpHighScoreClient → main script (at the moment, a stub for actual client code)
  - Score → instances store a player name and player score
- Some UI helper scripts that you can ignore for now:
  - HighScoreView → wraps all the UI logic for the high score view
  - ScoreRow → wraps a single row in the HighScoreView
- Most of the UI communication is event based
- No actual networking code yet

Assume we'd like to 'network' this application…

Which application protocol messages would you need?



Protocol (design)

- Client → Server: 1. AddScore {name:string, score:int}
  2. GetScores {}
- Server → Client: 1. HighScores[{name:string,score:int}]

- Details/decisions to make:
  - Does the server reply with all highscores or just the highest 3?
  - Do we already sort the highscores or not?
  - How do we send/receive/process these messages?

  (Note that these are a different sort of choices!)

Communication options

Option 1 – Using strings

Client → Server

"addscore,BillyM,3333360"

"getscores"

Server → Client

"highscores,5,BillyM,3333360,Hans,0,
StevenW,695500,BruceD,666,L00tz0r,10"

Example 002_stringbased_highscore_server

Example 002 setup

- Different approach to code sharing:
  - there is a 'shared' project with all classes that are shared between client and server (at this point only the Score and StreamUtil class).
  - when you build this shared project, the resulting dll is automatically copied to the unity client (into the Assets/Scripts/shared folder)
  - the server solution includes the shared project,
    which triggers build on the both of them when started.

Evaluation

- Simple setup, but...
  - Strings are “hard” to construct/parse (leads to ugly unreadable constructions)
  - Duplication of parsing code on client & server
  - Easy to make mistakes, yet hard to debug
  - Not strongly typed
  - Wrong request formats will cause exceptions & crashing
    (in other words, parsing and handling could be improved ☺)
  - It is not clear what the protocol is unless you investigate the code for both client and server and reverse engineer the protocol from that code.

Option 2 – Using objects

Would it not be great if we could do something like:

```
(Client sends message)
AddRequest request = new AddRequest();
request.score = new Score (“Blah”,10);
client.Send (addRequest);
```

```
(Server receives message)
object request = client.Receive();
if (request is AddRequest) {
      //…..process AddRequest
}
```

Option 2 – Using objects

... and similarly ...

(Server sends message)

```
HighScoresUpdate scoresUpdate = new HighScoresUpdate();
scoresUpdate.scores = <….list of score objects here ….>;
client.Send (scoresUpdate);
```

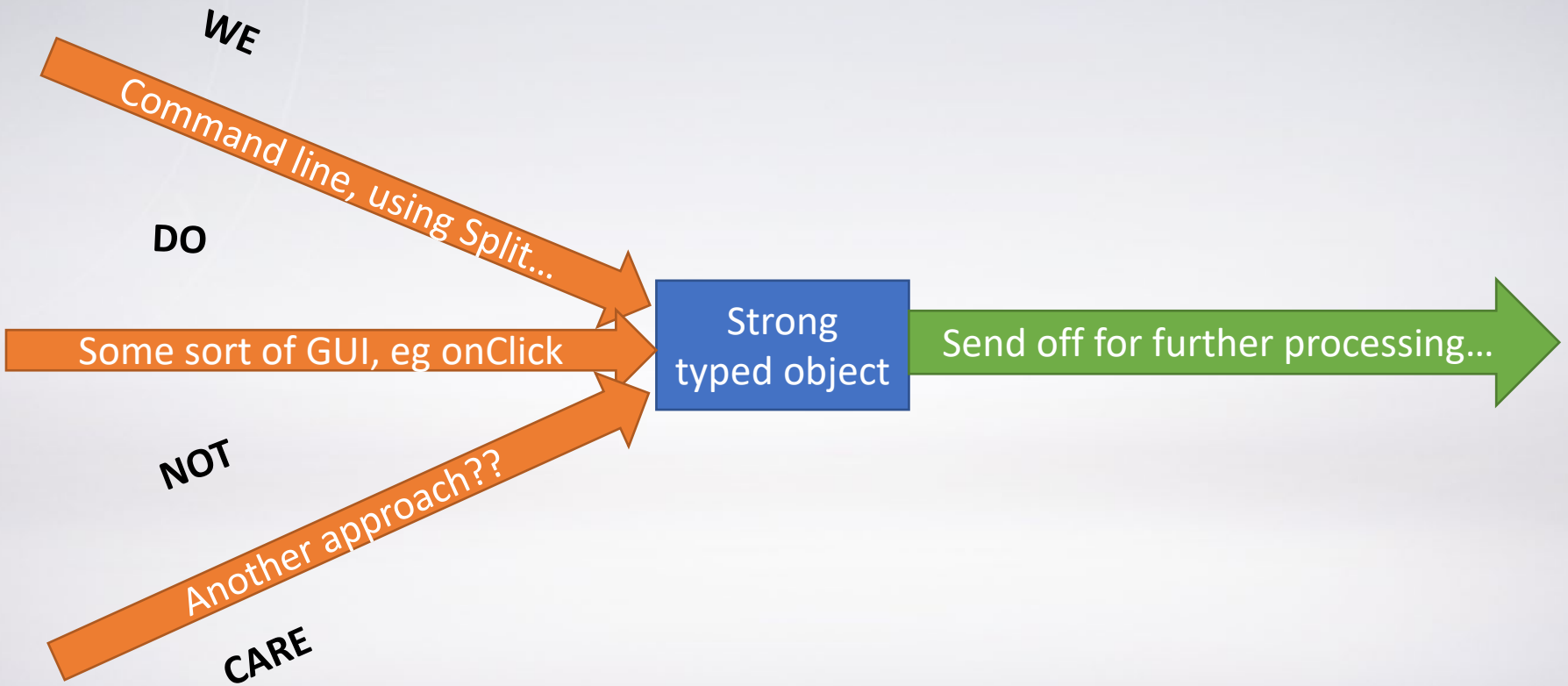
(Client receives message)

```
object request = client.Receive();
if (request is HighScoresUpdate) {
    //….. process HighScoresUpdate
}
```

Advantages of object based protocols

- Everything is strong typed
  - It’s clear how they are constructed
  - It’s easier to process them
  - It’s easy to validate correct input / harder to construct invalid requests
- Parsing is separated from main logic
  - Format is not directly tied to user input
- It’s clear what messages are supported (assuming all 'protocol' classes are in 1 folder)

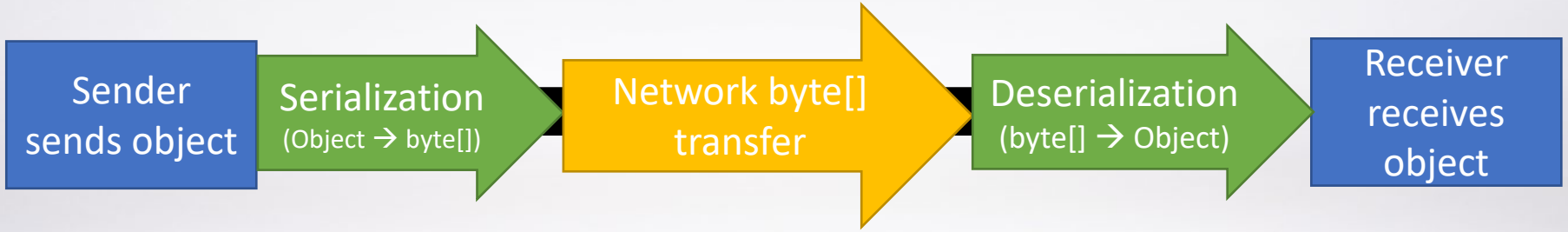
Object construction



Conclusion

Using an object based protocol is ***awesome***

The only question ofcourse is:
"How do we implement diz magiiicz?!"



Answer: by serializing and deserializing the protocol objects.
In normal people speak: by converting objects to byte[] and back again.

(De)serialization approaches

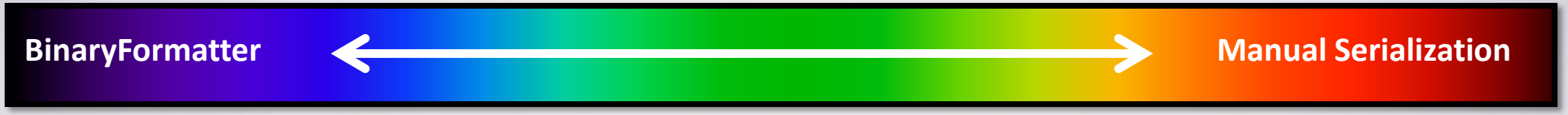
Lot of different approaches possible

- BinaryFormatter
- XmlSerializer
- SoapFormatter
- DataContractSerializer
- DataContractJsonSerializer
- NewtonSoft Json library
- Google Protocol Buffers
- Some form of DIY serialization
- …and probably a **ton** more…

How do they differ?

- Human readable (XML, JSON, ...) or not (some form of binary protocol)
- Ease of use (everything automatic vs DIY)
- Dependencies (.Net / 3rd party / none?)
- Speed of (de)serialization
- Size of resulting byte[]
- TCP/UDP agnostic?
- Platform independent or not
- Mainstream (recommended / well known / industry standard)

E.g. BinaryFormatter vs custom serialization



- BinaryFormatter:
  - Easy to use, no manual labour: you just instantiate a class and BinaryFormatter (de)serializes it for you
  - Slow, with very large byte[] results
- Custom serialization:
  - Little bit more work/complicated: you have to implement the whole (de)serialization mechanism yourself
  - Really fast & small
  - Lots of control

BinaryFormatter vs custom serialization

(De)serializing the objects 100.000 times:

- SimpleMessage – just a string and int
- ArrayMessage – a simple collection
- DictionaryMessage – a “complex” collection
- NestedMessage – contains SimpleMessages

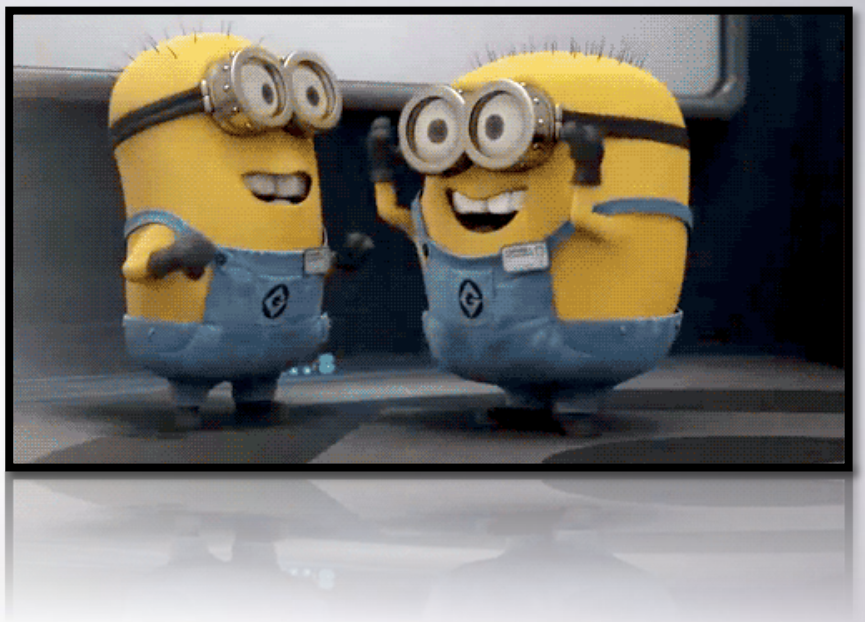
		Simple	Array	Nested	Dictionary
BinaryFormatter	Serialization time	909 ms	1307 ms	2050 ms	4811 ms
	Deserialization time	1036 ms	1304 ms	2286 ms	6138 ms
	Size	167 bytes	189 bytes	332 bytes	1760 bytes
Custom	Serialization time	46 ms	77 ms	72 ms	100 ms
	Deserialization time	76 ms	77 ms	64 ms	104 ms
	Size	13 bytes	28 bytes	18 bytes	35 bytes
Comparison	Serialization speed ratio	19.76 x	16.97 x	28.47 x	48.11 x
	Deserialization speed ratio	13.63 x	16.94 x	35.72 x	59.02 x
	Size ratio	12.85 x	6.75 x	18.44 x	50.29 x

Does that mean you should never use something like BinaryFormatter?

- No, helper classes like BinaryFormatter are great for Q&D prototyping
- Best thing would be to make your application serialization layer agnostic so you can easily switch between approaches
- However that is a bit beyond the scope of this lecture,
  for now we will just focus on 1 way to get those pesky bytes across

Our approach

- Custom (manual) serialization
  - You learn more about working with byte[]'s
  - You learn more about essential helper classes
  - It is fast (once you got it working)
  - The resulting byte[] are small
  - Easy to use and adapt
- Would you also do it this way in 'real life' ?
  - Yes, but for big projects you would use some sort of auto code generation approach like Google Protocol buffers



Other reasons for custom serialization

- You cannot just serialize any object and expect it to work:
  - GameObject?
  - Transform?
  - Rigidbody?
- Objects often contain pointers/references which are hard to serialize:
  - Address X on System Y may contain garbage on System Z
  - Serializing shared object references, will introduce copies on deserialization
- Some form of manual handling is usually required anyway!

In the end, whatever solution you choose...

- ... the main thing, the core concept, the mooost important thingamabob: **is that we can send and receive typed objects**.

- How you do that is less important than the principle itself:
  - This lecture demonstrates the custom approach
  - The samples provide basic code for this approach
  - The assignments demand using a custom approach
  - After this course, you are free to use whatever object based approach you want, but I would still very strongly recommend against using 'just' strings.

Custom serialization

But how?

- Again choices, choices, choices
- Custom serialization can also be done in x ways

- Again keep main thing in mind, we need to be able to convert:
  - an object to a byte[]
  - a byte[] to an object

Some possible approaches…

```
interface ISerializable {
    byte[] Serialize();
    void Deserialize(byte[]);
}
```

```
class MessageConverter {
    public byte[] Serialize (Object pObject);
    public Object Deserialize (byte[] pBytes);
}
```

```
class MessageConverter {
    public void Serialize (Object pObject, Stream pStream);
    public Object Deserialize (Stream pStream);
}
```

Code design goals

- No ‘over’ design
- Easy handling of byte[]
- Minimum amount of code
- TCP/UDP agnostic (can be used with both)
- Use an approach that can send multiple objects in one 'go'

Intermediate step

- In order to satisfy those goals, instead of jumping straight from string to object based communication ...
- ... we're making an in between step, using a custom Packet class:

Packet class?

Wrapper class around a byte[] that helps to (de)serialize values:

Write(int)	<->	ReadInt()
Write(string)	<->	ReadString()
Write(bool)	<->	ReadBool()
Write(?)	<->	Read?()

Packet class send example

```
private void onScoreAdded(Score pScore)
{
    //store the last score the player wants
    _lastAddedPlayerScore = pScore;

    //send the add score command
    Packet outPacket = new Packet();
    outPacket.Write("addscore");
    outPacket.Write(pScore.name);
    outPacket.Write(pScore.score);
    sendPacket(outPacket);
}
```

Packet class receive example

```
byte[] inBytes = StreamUtil.Read(client.GetStream());
Packet inPacket = new Packet(inBytes);

//get the command
string command = inPacket.ReadString();
Console.WriteLine("Received command:" + command);

//process it
if (command == "addscore")
{
    handleAddScore(client, inPacket);
}
```

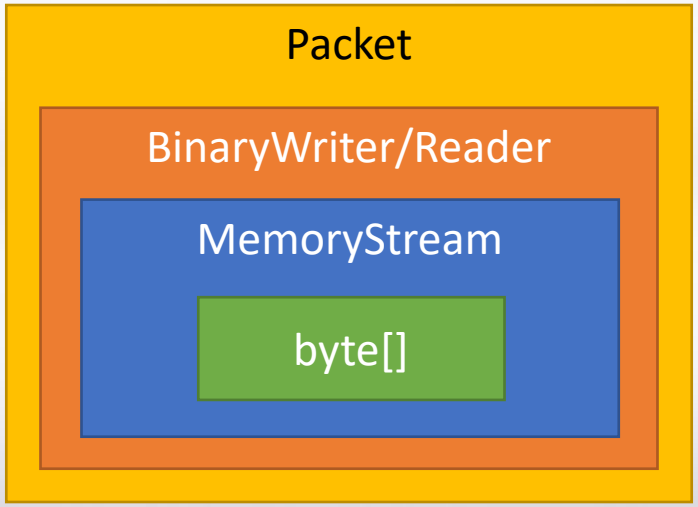
```
private void handleAddScore(TcpClient pClient, Packet pInPacket)
{
    _scores.Add(
        new Score(
            pInPacket.ReadString(),
            pInPacket.ReadInt()
        )
    );
}
```

Packet class advantages

- No typeless string constructing/parsing anymore
- We have custom/fast (de)serialization,
  but at the same time, we don’t have to deal with byte[] directly
- Decouples rest of application from the nitty gritty details of (de)serialization
- Can be used as a standalone class,
  if you don't care about the awesomeness of objects ;)

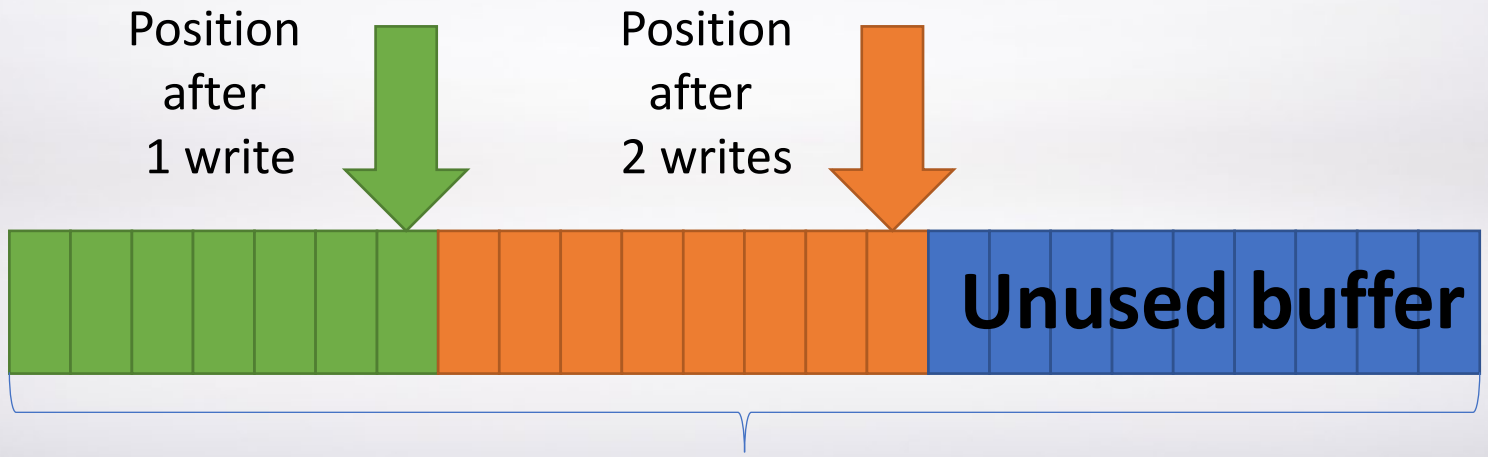
Packet class setup

- Easiest way to implement this, is by using a couple of helper classes:
  - MemoryStream → wraps a byte[] so it can be treated as a stream
  - BinaryWriter → writes literal values into a stream as bytes
  - BinaryReader → reads bytes from a stream as a literal value



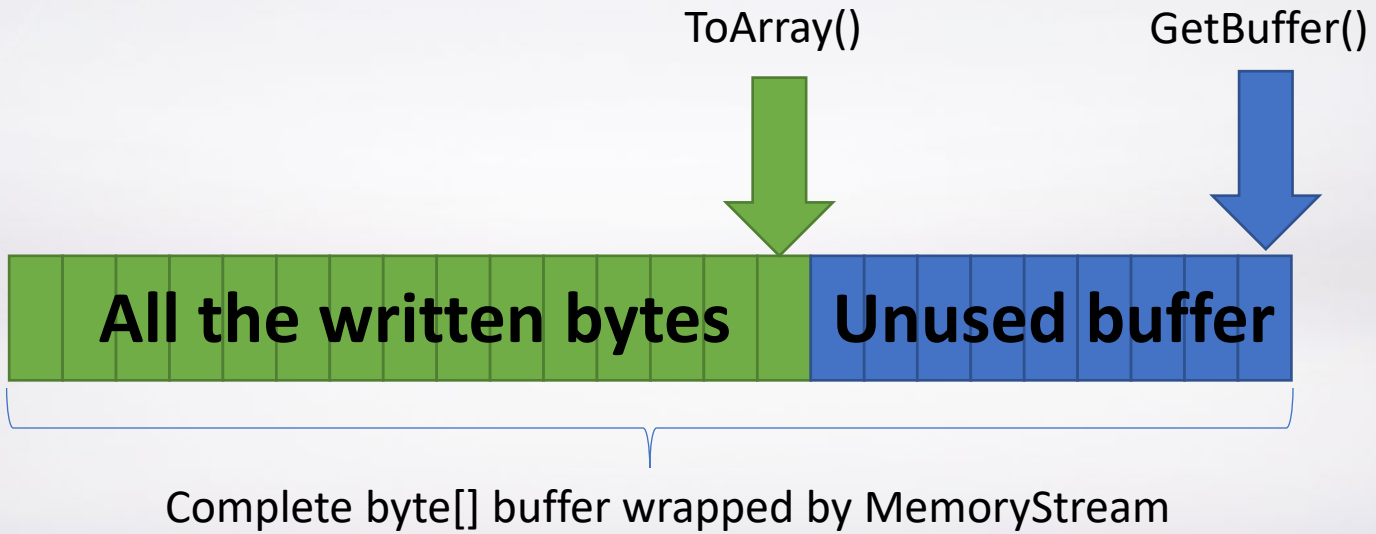
MemoryStream

- Wraps a single byte[] buffer (default size = 0)
- Allows you to write bytes to the buffer, automatically maintaining the write position
- Automatically resizes the buffer when required
  (resize is always x 2, try to avoid resizing !!)



ToArray() vs GetBuffer()

- ToArray(): All the **written** bytes
- GetBuffer(): Whole **buffer** wrapped by the stream



BinaryReader / Writer

- Stream wrappers, eg new BinaryReader (stream);
- BinaryReader/Writer have a lot of helper methods, for example:

BinaryWriter:

Write (Int)
Write (Bool)
Write (String)
+18 overloads

BinaryReader:

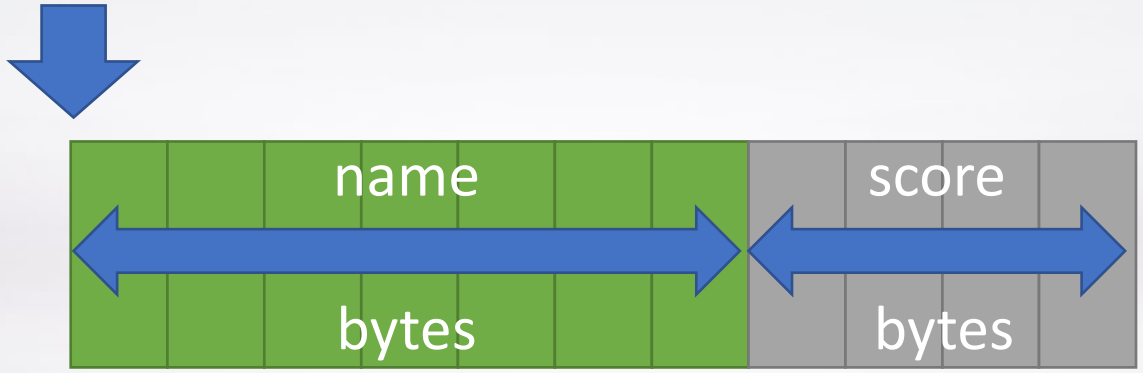
ReadInt()
ReadBool()
ReadString()
etc...

BinaryReader / Writer

- The BinaryReader/Writer know exactly how many bytes to read/write:
  - Int32 → 4
  - Int16 → 2
  - String → writes/reads length in the stream first (similar to our StreamUtil class)
  - etc
- There is a 1 on 1 relation between what you *write* into the packet & what you *read* from it and *the order* in which you do so

Writing into a Packet ...

... causes x bytes to be written to the underlying MemoryStream in a specific order:



Reading from a Packet

After transferring these bytes to the "other" side, we recreate a Packet from them and need to read from the Packet in *exactly the same* order:

What if ...

...we try to read values from the Packet in the wrong order, or use the wrong ReadX call, or use a different amount of Reads than Writes?

Always make sure your writes and reads are *symmetrical.*

Example 003_packetbased_highscore_server

- Things to note:
  - No awkward string parsing anymore
  - Not fully OO yet either, but first step has been made
  - Lost some debugging functionality,
    but that can be fixed by augmenting the Packet class

Next step, objects!

- We had ***sendString (awkwardStringHere);***
- Now we have ***sendPacket (betterPacketHere);***
- Next step ***sendObject (awesomeObjectHere);***

- How about just wrapping this piece of code in a class?

```
Packet outPacket = new Packet();
outPacket.Write("addscore");
outPacket.Write(pScore.name);
outPacket.Write(pScore.score);
```

For example, something like ....

```
public class AddRequest : ISerializable {

    public string name;
    public int score;

    public Packet Serialize () {
        Packet packet = new Packet();
        packet.Write("addscore");
        packet.Write(name);
        packet.Write(score);
        return packet;
    }

    public AddRequest Deserialize(Packet pPacket) {
        ..... ?? ....
    }
}
```

Every request/response can be treated the same way!

But how about Deserialize?
What is the issue with this approach?

The issue with this approach...

- ... is that we don't know *which* class instance to call Deserialize on:
  - e.g. do we call AddRequest.Deserialize or GetRequest.Deserialize?

- It's what we call a chicken and egg problem: we need to deserialize an object to find out which class to use to de-serialize that object...

- In other words:
  - we need to add some sort object identification somewhere!
  - object identification needs to be handled outside of the object itself ...

Fix – part 1

Make it possible to pass in a Packet to the (de)serialization methods:
(so we have access to the Packet before and after (de)serialization...)

```
public class AddRequest : ISerializable {

    //....

    public void Serialize (Packet pPacket) {
        pPacket.Write (.....);
    }

    public void Deserialize(Packet pPacket) {
        .... = pPacket.Read... ();
    }
}
```

Fix – part 2

Make it possible to write ISerializable objects into a Packet, by adding these two methods to the Packet class:

```
public void Write (ISerializable pSerializable)
{
    Write(pSerializable.GetType().FullName);
    pSerializable.Serialize(this);
}

public ISerializable ReadObject()
{
    Type type = Type.GetType(ReadString());
    ISerializable obj = (ISerializable)Activator.CreateInstance(type);
    obj.Deserialize(this);
    return obj;
}
```

This approach also has a downside: we need a parameterless constructor.

Again, different approaches possible, most important thing: you must decide which type to deserialize outside of the object.

AddRequest variation 1/3

Store name & score as two separate values and (de)serialize them as two separate values:

```
public class AddRequest : ISerializable {

    public string name;
    public int score;

    public void Serialize (Packet pPacket) {
        packet.Write (name);
        packet.Write (score);
    }

    public void Deserialize(Packet pPacket) {
        name = pPacket.ReadString ();
        score = pPacket.ReadInt();
    }

}
```

AddRequest variation 2/3

Store score as a single object but (de)serialize it as two separate values:

```
public class AddRequest : ISerializable {

    public Score score;

    public void Serialize (Packet pPacket) {
        packet.Write (score.name);
        packet.Write (score.score);
    }

    public void Deserialize(Packet pPacket) {
        score = new Score();
        score.name = pPacket.ReadString ();
        score.score = pPacket.ReadInt();
    }

}
```

AddRequest variation 3/3

Make Score : ISerializable too, so we can do this:

```
public class AddRequest : ISerializable {

    public Score score;

    public void Serialize (Packet pPacket) {
        packet.Write (score);
    }

    public void Deserialize(Packet pPacket) {
        score = pPacket.ReadObject() as Score;
        //or even: score = pPacket.Read<Score>();
    }

}
```

Example 004_objectbased_highscore_server

- Things to note:
  - GetRequest is empty → why does this still work?
  - (De)serialization of lists and nested objects
  - Our protocol is completely clear from looking at the classes in the 'protocol' folder
- There is a whole range of improvements possible:
  - outside of the scope of this lecture,
    since the issues & fixes clutter the basic concepts
  - included at the end of the lecture for those of you who are interested

Assignment 3 – 3D Chat lobby

Assignment 3 – Introduction of the setup

- Client overview
- Server overview
- Current state of the implementation
- Your assignment: "network" this application
  - Decide what messages/protocol you need to support the required functionality
  - Implement your protocol ***step by step*** using Packets/Objects

Assignment 3 – Sufficient requirements

- A client can join a server and will get a random avatar assigned at a random (but valid!) position
- Clients can see all avatars from all other clients
- Every time a client joins or a disconnect is detected, the client's lobby is updated accordingly
- Clients can "speak" through their own avatar only
- (See bb download for all the details on the extra requirements for a higher grade)

Possible improvements

Some possible improvements…

- (De)serialization mechanism is hard to replace
- Less duplicated code
- Disallow parameterless constructors to improve validation
- All protocol object fields are public
- Better debugging (add ToString to each object!)
- Not every ISerializable object is a protocol object!

Replacing the serialization layer

- Wrap all communication in another class:

```
TcpChannel (TcpClient) {
        Send (Object pObject)
        Receive():Object
}
```

- You can serialize however you like beneath the surface
- Also prevents current code duplication
- More info on this in the next lecture!

Avoid parameterless constructors

- Register external serializers for the protocol objects:

```
class AddRequestHelper : ISerializer<AddRequest> {
    public void Serialize (AddRequest pRequest, Packet pPacket);
    public AddRequest Deserialize (Packet pPacket);
}
```

- Don't write classname into packet, but use a class id (int) with which to identify the correct (de)serialization helper.
- The helpers know exactly how to create the protocol objects, so we don't have to use parameterless constructors through reflection anymore.

Do not use public fields

- Well, public fields in this case aren't so bad, but it would be good to make them at least readonly.
- This wasn't possible while using parameterless constructors, but if you apply the previous 'fix', you can also fix this one.

Better debugging

- Options?
  - Implement a ToString for the Packet class
  - Implement a ToString for each protocol object
- How can you implement a ToString for the Packet class?
  - Add info about the sort of values you are writing
  - Use this for introspection
  - Don't forget to reset the stream position to 0 after
- However, assuming the Packet class has been thoroughly tested and works correctly, it is actually *much easier and more logical* to add debugging to the protocol object. Either by hand or automated through a base class (!).

Example protocol object base class with debugging

```
public abstract class ASerializable
{
    abstract public void Serialize(Packet pPacket);
    abstract public void Deserialize(Packet pPacket);

    public override string ToString()
    {
        StringBuilder builder = new StringBuilder();
        builder.Append("\n" + GetType().Name + ":");
        builder.Append("\n------------------------------------------------------------");

        IEnumerable<FieldInfo> publicFields = GetType().GetFields().Where(f => f.IsPublic);
        foreach (FieldInfo field in publicFields)
        {
            object value = field.GetValue(this);
            if (value is ICollection)
            {
                ICollection collection = value as ICollection;
                foreach (object item in collection) builder.Append(item.ToString());
            }
            else
            {
                builder.Append(String.Format("\nName: {0} \t\t\t Value: {1}", field.Name, value) + "");
            }
        }
        builder.Append("\n------------------------------------------------------------");
        return builder.ToString();
    }
}
```

Not every ISerializable is a 'message'

- The send and receive methods in the last example allow you to send any ISerializable object: AddRequest, GetRequest, HighScoresUpdate, but *also Score, even though* Score is not a protocol message in itself (only as part of the other messages).

/ design the protocol

- One solution is to:
  - add an IMessage interface where IMessage : ISerializable
  - wrap all communication in a custom TcpChannel that only allows you to send & receive IMessage instances.