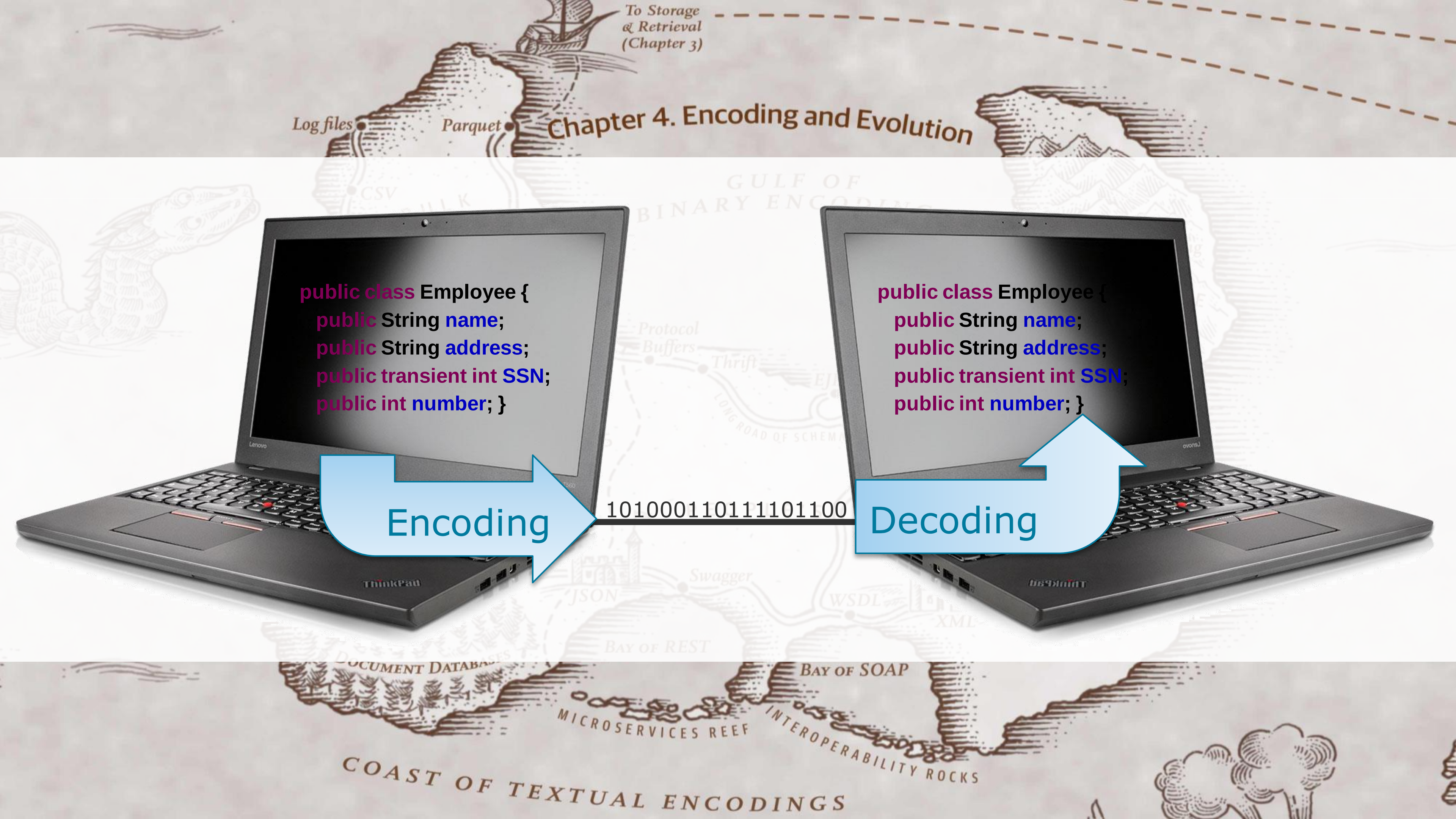
Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Distributed Data Management
Encoding

Thorsten Papenbrock

F-2.04, Campus II

Hasso Plattner Institut


To Storage
& Retrieval
(Chapter 3)
Log files
Parquet
Chapter 4. Encoding and Evolution
GULF OF
BINARY ENCODING
CSV
Protocol
Buffers
Thrift
public class Employee {
public String name;
public String address;
public transient int SSN;
public int number; }
public class Employee {
public String name;
public String address;
public transient int SSN;
public int number; }
Encoding
1010001101111101100
Decoding
Swagger
JSON
WSDL
XML
BAY OF REST
DOCUMENT DATABASES
BAY OF SOAP
MICROSERVICES REEF
INTEROPERABILITY ROCKS
COAST OF TEXTUAL ENCODINGS

Overview

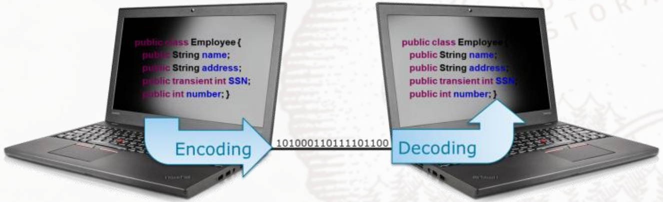
Encoding

- Encoding & Decoding
- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding

Overview

Encoding

- **Encoding & Decoding**
- Language-Specific Encoding
- JSON/XML Encoding
- Binary Encoding



Encoding & Decoding

Layering Data Models

1. Conceptual layer

- Data structures, objects, modules, …
  - ➢ Application code

2. Logical layer

- Relational tables, JSON, XML, graphs, …
  - ➢ Database management system (DBMS) or storage engine

3. Representation layer

- Bytes in memory, on disk, on network, …
  - ➢ Database management system (DBMS) or storage engine

4. Physical layer

- Electrical currents, pulses of light, magnetic fields, …
  - ➢ Operating system and hardware drivers

```
class TestSerial {
  public byte version = 100;
  public byte count = 0;
}
```

```
{
  "class": TestSerial,
  "version": "100",
  "count": "0"
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Network Connections are Physical

Node 1

1. Conceptual layer
   - Data structures, objects, modules, …
     - ➢ Application code
2. Logical layer
   - Relational tables, JSON, XML, graphs, …
     - ➢ Database management system (DBMS) or storage engine
3. Representation layer
   - Bytes in memory, on disk, on network, …
     - ➢ Database management system (DBMS) or storage engine
4. Physical layer
   - Electrical currents, pulses of light, magnetic fields, …
     - ➢ Operating system and hardware drivers

Node 2

1. Conceptual layer
   - Data structures, objects, modules, …
     - ➢ Application code
2. Logical layer
   - Relational tables, JSON, XML, graphs, …
     - ➢ Database management system (DBMS) or storage engine
3. Representation layer
   - Bytes in memory, on disk, on network, …
     - ➢ Database management system (DBMS) or storage engine
4. Physical layer
   - Electrical currents, pulses of light, magnetic fields, …
     - ➢ Operating system and hardware drivers

Encoding & Decoding

Network Connections are Physical

Encoding

- "Representation of the data"
  (or "Process of changing the representation")

Serialization

- "Serial encoding of the data"
  (or "Process of serializing a representation")
- i.e. special case of encoding
- Serial formats:
  - Char arrays in layer 2 (JSON, XML, …)
  - Byte arrays in layer 3
  - Sequences of signals in layer 4

Decoding (and Deserialization)

- The reverse of encoding (and serialization)

1. Conceptual layer
   Data structures, objects, modules, …
   ➢ Application code
2. Logical layer
   Relational tables, JSON, XML, graphs, …
   ➢ Database management system (DBMS) or storage engine
3. Representation layer
   Bytes in memory, on disk, on network, …
   ➢ Database management system (DBMS) or storage engine
4. Physical layer
   Electrical currents, pulses of light, magnetic fields, …
   ➢ Operating system and hardware drivers

Overview

Encoding

- Encoding & Decoding
- **Language-Specific Encoding**
- JSON/XML Encoding
- Binary Encoding

```
public class Employee {
  public String name;
  public String address;
  public transient int SSN;
  public int number; }
```

```
public class Employee {
  public String name;
  public String address;
  public transient int SSN;
  public int number; }
```

Two Different Representations

Application In-memory Data

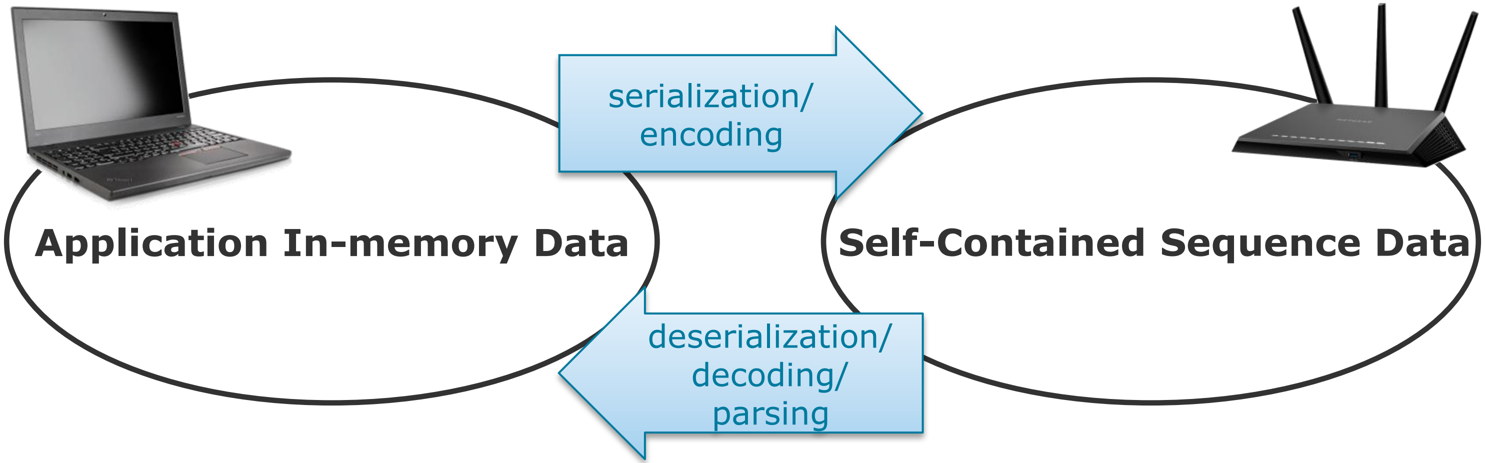
Self-Contained Sequence Data

- Language specific formats
- Logical structures: objects, structs, lists, arrays, hash tables, trees, …
- Optimized for efficient manipulation by the CPU

- Standardized encoding formats
- Byte sequences: Native, JSON, XML, protocol buffers, Avro, …
- Optimized for disk persistence, network transmission, and inter-process communication

Language-Specific Encoding

Two Different Representations



Problems:

- Tied to a programming language (language-specific data structures)
- Tied to an address space (process-specific pointers)

Problems:

- Inefficient and complicated access and manipulation operations due to lack of pointers and serial byte representation

Language-Specific Encoding

Serialization/Encoding

1. Conceptual layer
2. Logical layer
3. Representation layer
4. Physical layer

Language-specific serialization formats

- Goal: convert in-memory data into byte sequence data back and forth
- Examples:
  `Serializable` (Java), `Kryo` (Java), `Marshal` (Ruby), `pickle` (Python), …

Advantages

- Native language support; easy to use
- Default implementation for intra-language (distributed) communication

Problems

- Serialized data is still tied to a programming language.
- Deserialization of arbitrary, byte-encoded objects can cause security issues.
- Data versioning is complicated, i.e., lack of forward/backward compatibility.
- Performance is often an issue, because arbitrary object serialization can be costly (e.g., Java `Serializable` is known to be inefficient).

Example: java.io.Serializable

```
import java.io.*;

public class Employee implements java.io.Serializable {
  public String name;
  public String address;
  public transient int SSN;
  public int number;

  public Employee(String name, String address, int SSN, int number) {
    this.name = name;
    this.address = address;
    this.SSN = SSN;
    this.number = number;
  }
}
```

Java can serialize any class that implement the Serializable interface (serialization via reflection)

All fields must also be serializable or explicitly marked as transient, i.e., non-serializable

https://www.tutorialspoint.com/java/java_serialization.htm

Example: java.io.Serializable

```
import java.io.*;
public class SerializeDemo {
  public static void main(String [ ] args) {
    Employee e = new Employee("Diana Brown", "Citystreet 8, Jamestown", 42, 123);
    try {
      FileOutputStream fileOut = new FileOutputStream("/tmp/employee.ser");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      out.writeObject(e);
      out.close();
      fileOut.close();
    } catch(IOException i) {
      i.printStackTrace();
    }
  }
}
```

Can be any output stream; also to network etc.

Performs the actual serialization using reflection

https://www.tutorialspoint.com/java/java_serialization.htm

Example: java.io.Serializable

```
import java.io.*;
public class DeserializeDemo {
  public static void main(String [ ] args) {
    Employee e = null;
    try {
      FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      e = (Employee) in.readObject();
      in.close();
      fileIn.close();
    } catch(IOException | ClassNotFoundException i) {
      i.printStackTrace();
    }
  }
}
```

Performs the actual deserialization; result is an object

https://www.tutorialspoint.com/java/java_serialization.htm

Example: java.io.Serializable

Surprise!

- The serialized objects are much larger than expected:

Hexadecimal Code

```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

serialization/ encoding

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

2 bytes + header (?)

51 bytes

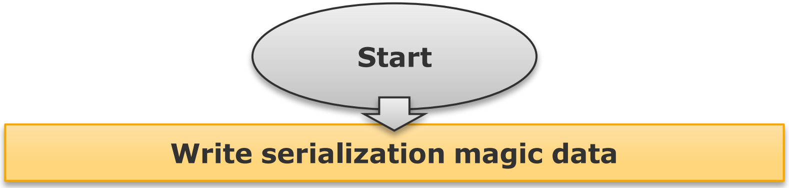
Why?

- Size scales linearly (but not favorably)

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

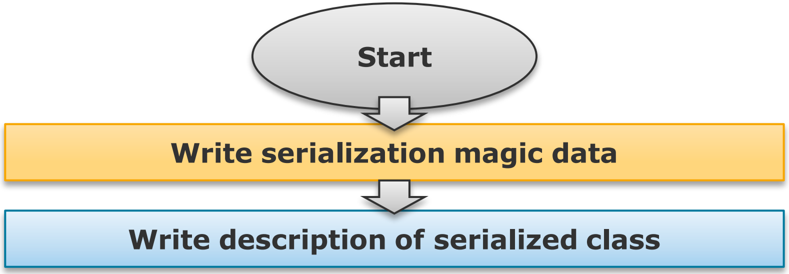
Serialization magic data specifies …

1. the serialization protocol (AC ED)
2. the serialization version (00 05)
3. the beginning of a new Object (0x73).

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```java
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Description of serialized class specifies …

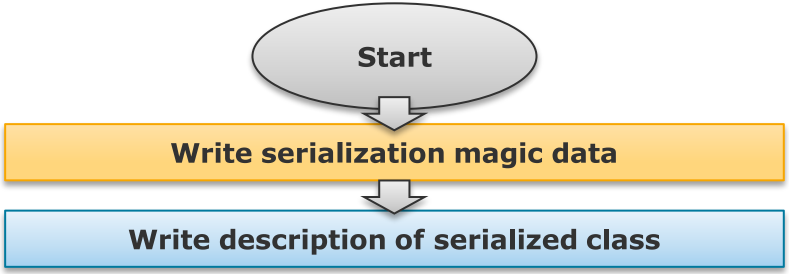
1. beginning of a new class (0x72)
2. length of the class name (00 0A)
3. name of the class (53 […] 74)
4. serial version identifier (A0 […] F9)
5. various flags (e.g. 0x02 = serialization support)
6. number of fields in this class (00 02)

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Language-Specific Encoding

Example: java.io.Serializable

Java serialization algorithm:




```java
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Description of serialized class specifies …

7. field code of “count” representing “byte” (0x42)
8. length of the field name (00 05)
9. name of the field (63 […] 74 which is “count”)
10. field code of “version” representing “byte” (0x42)
11. length of the field name (00 07)
12. name of the field (76 […] 6E which is “version”)

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:

Start

Write serialization magic data

Write description of serialized class

```
class TestSerial implements Serializable {
   public byte version = 100;
   public byte count = 0;
}
```

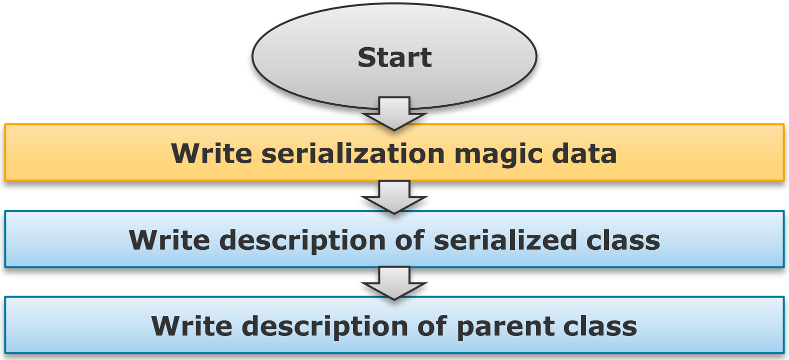
AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Description of serialized class specifies …
13. end of class (0x78)

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

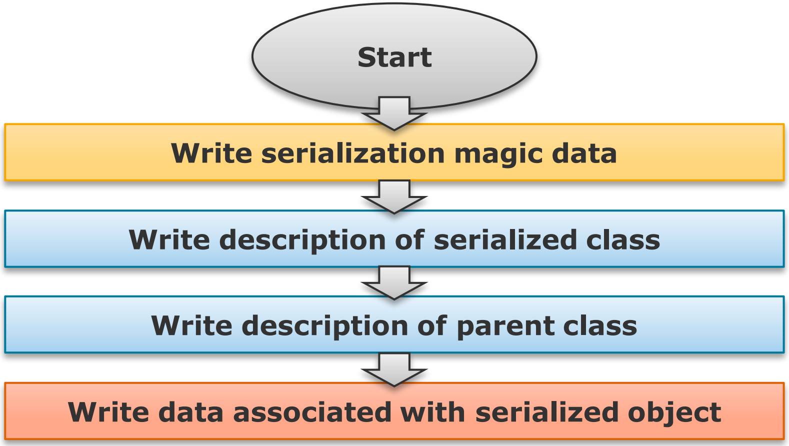
Description of parent class specification:

- Follows the same pattern as shown for the serialized class: (1) class definition and (2) field definitions
- Recursively adds the parent's parents until parent class is `Object`
- No parent here (0x70), because parent is already `Object`

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Data associated with serialized object …

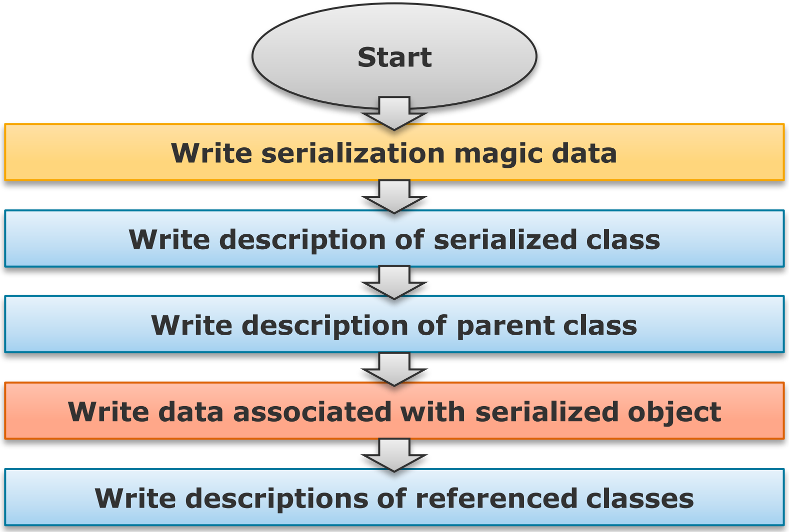
1. The first value (0x00 which is 0 for "count")
2. The second value (0x64 which is 100 for "version")

- Byte-length of the values is known from their types.
- Fields are own and inherited fields.

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:




```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Description of referenced classes specification:

- Follow the same pattern as shown for the serialized class: (1) class definition and (2) field definitions
- No specifications here, because the class `TestSerial` has no referenced classes

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Example: java.io.Serializable

Java serialization algorithm:

Start
↓
Write serialization magic data
↓
Write description of serialized class
↓
Write description of parent class
↓
Write data associated with serialized object
↓
Write descriptions of referenced classes
↓
Write data associated with referenced objects
↓
Stop

```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

AC ED 00 05 73 72 00 0A 53 65
72 69 61 6C 54 65 73 74 A0 0C
34 00 FE B1 DD F9 02 00 02 42
00 05 63 6F 75 6E 74 42 00 07
76 65 72 73 69 6F 6E 78 70 00
64

Data associated with referenced objects …

- Follows the same pattern as shown for the serialized object: concatenation of byte encoded values
- No values here, because the class `TestSerial` has no referenced classes

https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

Language-Specific Encoding

Example: java.io.Serializable

```
class Message implements Serializable {
   private static final long serialVersionUID = 6455048433435395034L;
   int[] data = {1,2,3};
   String name = "message42";
   boolean validity = true;
   Map<String, String> map = Stream.of(new String[][] {
           { "key1", "value1" },
           { "key2", "value2" },
     }).collect(Collectors.toMap(data -> data[0], data -> data[1]));
}
```

290 byte

ACED00057372002464652E6870692E6F63746F7075732E74657374696E672E5465737
424324D657373616765994F05992DBC3DA0200045A000876616C69646974795B000
46461746174000025B494C00036D617074000F4C6A6176612F7574696C2F4D61703B4
C00046E616D657400124C6A6176612F6C616E672F537472696E673B787001757200002
5B494DBA602676EAB2A5020000787000000003000000010000000200000003737200
116A6176612E7574696C2E486173684D61700507DAC1C31660D103000246000A6C6F
6164466163746F72490009746872657368686F6C6478703F4000000000000C770800000
01000000027400046B65793174000676616C75653174000046B65793274000676616C
756532787400096D657373616765343 2

Distributed Data Analytics

Encoding

Example: Kryo

Optimized, Java-specific serializer that uses some optimizations, which we will see for other serializers in a minute!

```
class Message implements Serializable {
  private static final long serialVersionUID = 6455048433435395034L;
  int[] data = {1,2,3};
  String name = "message42";
  boolean validity = true;
  Map<String, String> map = Stream.of(new String[][] {
        { "key1", "value1" },
        { "key2", "value2" },
    }).collect(Collectors.toMap(data -> data[0], data -> data[1]));
}
```

104 byte

010064652E6870692E6F63746F7075732E74657374696E672E5465737424324D65737
36167E5010104020406010 16A6176612E7574696C2E486173684D61F0010203016B65
79B1030176616C7565B103016B6579B2030176616C7565B2016D65737361676534B2
01

Language-Specific Encoding

Example: Kryo without Class-Serialization

```
class Message implements Serializable {
  private static final long serialVersionUID = 6455048433435395034L;
  int[] data = {1,2,3};
  String name = "message42";
  boolean validity = true;
  Map<String, String> map = Stream.of(new String[][] {
        { "key1", "value1" },
        { "key2", "value2" },
    }).collect(Collectors.toMap(data -> data[0], data -> data[1]));
}
```

66 byte

01010402040601006A6176612E7574696C2E486173684D61F0010203016B6579B1030
176616C7565B103016B6579B2030176616C7565B2016D65737361676534B201

Overview

Encoding

- Encoding & Decoding
- Language-Specific Encoding
- **JSON/XML Encoding**
- Binary Encoding


```
{
  "_id": 1,
  "username": "ben",
  "password": "ughiwuv",
  "contact": {
              "phone": 0331-1781471,
              "email": "ben87@gmx.de",
              "skype": "benno.miller"
            },
  "access": {
              "level": 3,
              "group": "user"
            }
}
```


JSON Format

Encoding Strategy

1. Conceptual layer
2. Logical layer
3. Representation
4. Physical layer

Example: XML

```
class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

encoding

```
<TestSerial>
  <version>100</version>
  <count>0</count>
</TestSerial>
```

serialization

```
3C 54 65 73 74 53 65 72 69 61
6C 3E A2 02 03 C7 66 57 27 36
96 F6 E3 E3 13 03 03 C2 F7 66
57 27 36 96 F6 E3 EA 20 20 3C
63 6F 75 6E 74 3E 30 3C 2F 63
6F 75 6E 74 3E A3 C2 F5 46 57
37 45 36 57 26 96 16 C3 EA
```

69 bytes

>51 bytes Java serialization but language independent!

Structural Elements

```
{
  "_id": 1,
  "username": "ben",
  "password": "ughiwuv",
  "contact": {
        "phone": 0331-1781471,
        "email": "ben87@gmx.de",
        "skype": "benno.miller"
      },
  "access": {
        "level": 3,
        "group": "user"
      },
  "supervisor": {
        "$ref": "AnnaMT",
        "$id": 1,
        "$db": "users"
      }
}
```

JSON Format

Nested key-value pairs

```
<_id>Benno87</_id>
<username>ben</username>
<password>ughiwuv</password>
<contact>
    <phone>0331-1781254</phone>
    <email>ben87@gmx.de</email>
    <skype>benno.miller</skype>
</contact>
<access>
    <level>3</level>
    <group>user</group>
</access>
<supervisor>
    <ref>AnnaMT</ref>
    <id>1</id>
    <db>users</db>
</supervisor>
```

XML Format

Nested tagged values

Structural Elements


```
{
  "_id": 1,
  "username": "ben",
  "password": "ughiwuv",
  "contact": {
        "phone": 0331-1781471,
        "email": "ben87@gmx.de",
        "skype": "benno.miller"
      },
  "access": {
        "level": 3,
        "group": "user"
      },
  "supervisor": {
        "$ref": "AnnaMT",
        "$id": 1,
        "$db": "users"
      }
}
```


JSON Format


```
<_id>Benno87</_id>
<username>ben</username>
<password>ughiwuv</password>
<contact phone = "0331-1781254
         email = "ben87@gmx.de"
         skype = "benno.miller" />
<access level = "3"
        group = "user" />
<supervisor ref = "AnnaMT"
            id = "1"
            db = "users" />
```


XML Format

Using attributes makes XML much smaller, but the mix of tags and attributes is also harder to read.

JSON/XML Encoding

Lists

```
{
  "employees": [
    {
      "firstName": "John",
      "lastName": "Doe"
    },
    {
      "firstName": "Anna",
      "lastName": "Smith"
    },
    {
      "firstName": "Peter",
      "lastName": "Jones"
    }]
}
```

JSON Format

```
<employees>
  <employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName>
    <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName>
    <lastName>Jones</lastName>
  </employee>
</employees>
```

XML Format

Both formats are similarly expressive.

Some Standardized Encodings

HPI Hasso Plattner Institut

```xml
<UnitTemplate name="Tank" speed="0.5" health="100">
        <weapons>
                <weapon name="big cannon" />
                <weapon name="small turret" />
        </weapons>
        <abilities>
                <cloak cooldown="10" />
                <regenerate healthPerSecond="1" />
        </abilities>
</UnitTemplate>
```

e.g., for Web Services

XML
Extensible Markup Language

```
UnitTemplate name="Tank" speed=0.5 health=100 {
        weapons "big cannon" "small turret"
        abilities {
                "cloak" cooldown=10
                "regenerate" healthPerSecond=1
        }
}
```

SDL
Simple Declarative Language

```json
{ "UnitTemplate" : {
        "name" : "Tank",
        "speed" : 0.5,
        "health" : 100,
        "weapons" : [ "big cannon", "small turret" ],
        "abilities" : [
                { "cloak" : { "cooldown" : 10 } },
                { "regenerate" : { "healthPerSecond" : 1 } }
        ]
} }
```

e.g., for REST-based services

JSON
JavaScript Object Notation

```toml
[UnitTemplate]
name = "Tank"
speed = 0.5
health = 100
weapons = [ "big cannon", "small turret" ]

[[UnitTemplate.Ability]]
type = "cloak"
cooldown = 10

[[UnitTemplate.Ability]]
type = "regenerate"
healthPerSecond = 1
```

TOML
Tom's Obvious, Minimal Language

And many more: YAML, CSV, …

Standardized Encodings

Advantages

- Language and address-space independence
- Human readability (sometimes)
- Ability to query and store in a structured manner

Problems

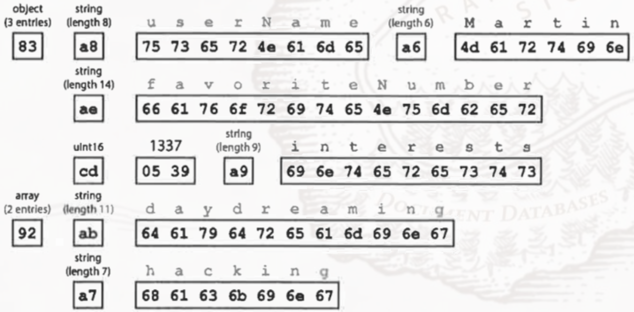
- No or only weak typing
  - Number encoding is ambiguous and imprecise.
- No support for binary strings (only Unicode)
  - Storing binary strings in Unicode increases data size (>33%).
- Schemata, if needed, require optional (complicated) schema support (e.g., XML Schema)
  - Without explicit schema definition, applications must define schemata.
- Large binary representation (if String is directly serialized into binary)
  - Native encoding formats are typically more concise.

JSON distinguishes only strings and numbers, but not int, float, or double; XML sees all values as strings

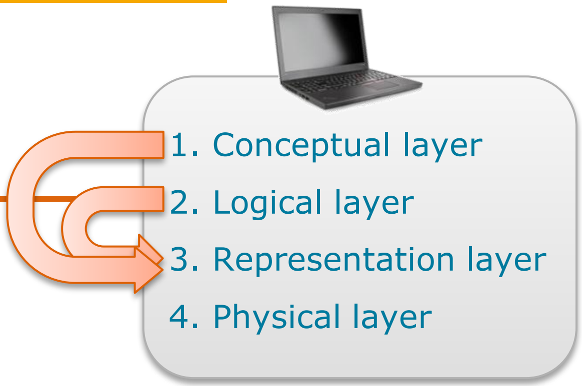
Overview

Encoding

- Encoding & Decoding
- Language-Specific Encoding
- JSON/XML Encoding
- **Binary Encoding**



Motivation



Problems

- Unicode formats and their naïve binary encodings are large.
- Data types are lost.

Idea

- Encode Unicode formats into binary strings with format-specific encodings.
- Keep the original structure (attribute names, nesting, …).

Binary Encodings

- For JSON: MessagePack, BSON, BJSON, UBJSON, BISON, Smile, …
- For XML: WBXML, Fast Infoset, …
- For Code: Apache Thrift, Protocol Buffers, Apache Avro

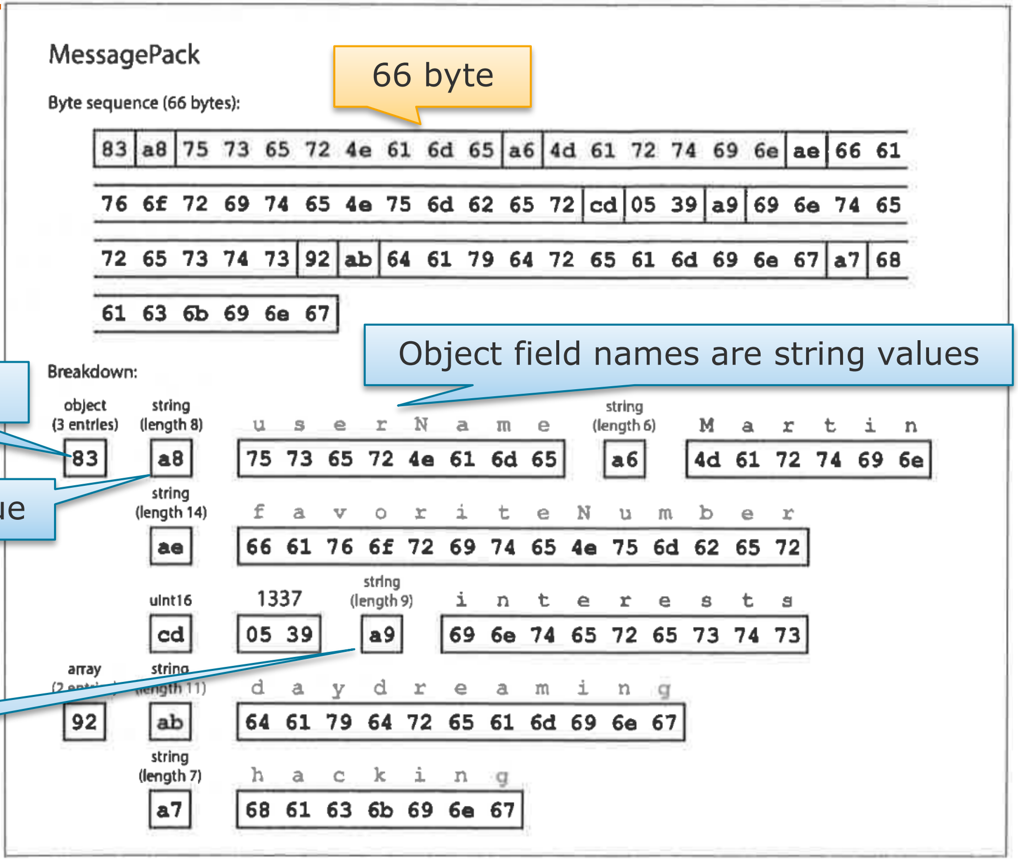
Binary Encoding

MessagePack

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

JSON Format

81 byte



66 byte

Object field names are string values

Object preamble

Alternating: data type (+ length) and value

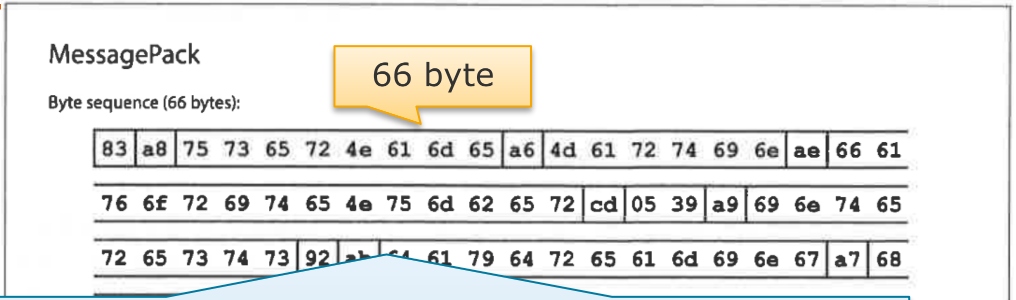
Here: string-length < 16

Binary Encoding
MessagePack

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

JSON Format

81 byte

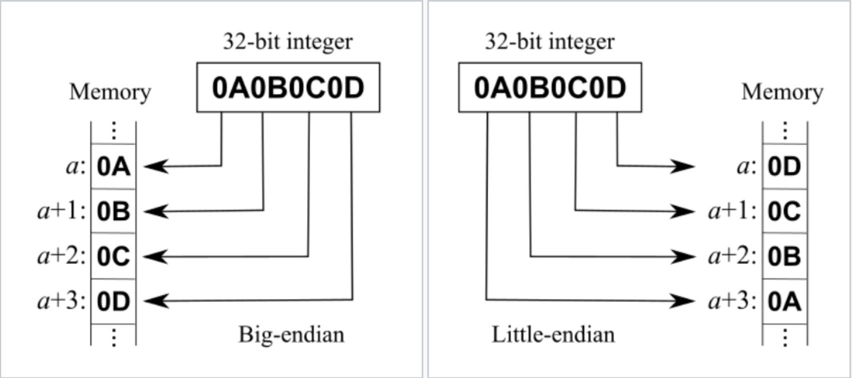


66 byte

Caution: Mind **Endianness**!

- Denotes the order of bytes of a string/int/… in memory.
- Big-endian: most significant byte first
- Little-endian: least significant byte first
- Most systems use big-endian (as depicted on the slides).
- Order matters for transmission and (de)serialization

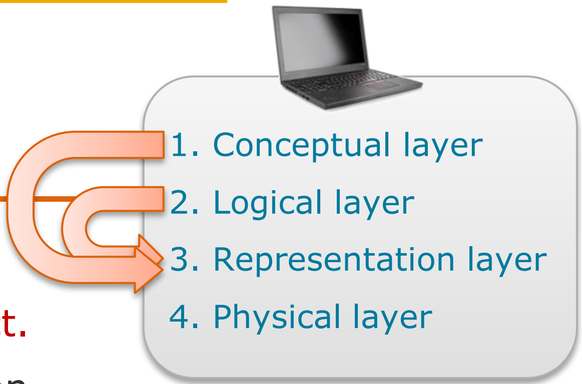
Endian example



Big-endian Little-endian

Binary Encoding

Schema-based Binary Encoding



Motivation

- MessagePack stores attribute names (and types) for each object.
  - ➢ Redundant information that increases memory consumption

Idea

- Define the attributes (= fields) once for all objects.
  - ➢ Define a schema!
  - ➢ No need to encode the attributes and their size

Binary Encoding Libraries

- Apache Thrift (by Facebook)
  - https://thrift.apache.org/
- Protocol Buffers (by Google)
  - https://developers.google.com/protocol-buffers/
- …

} both open source since 2007

Binary Encoding

Thrift with BinaryProtocol

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

JSON Format

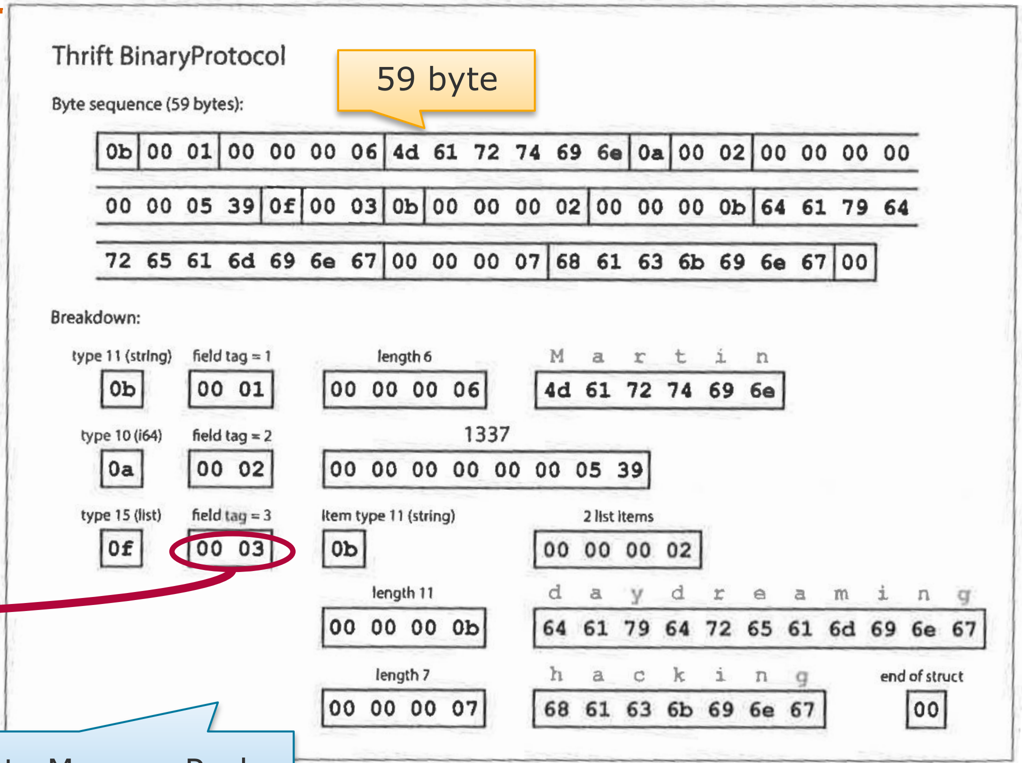
81 byte

```
struct Person {
  1: required string       username,
  2: optional i64          favoriteNumber,
  3: optional list<string> interests
}
```

Thrift Struct

Schema definition

59 byte



Backward compatibility

- Adding optional fields and changing field names possible
- Changing field tags (or types) breaks reading of old data

Very similar to MessagePack, but without field names

Binary Encoding
Thrift with CompactProtocol

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

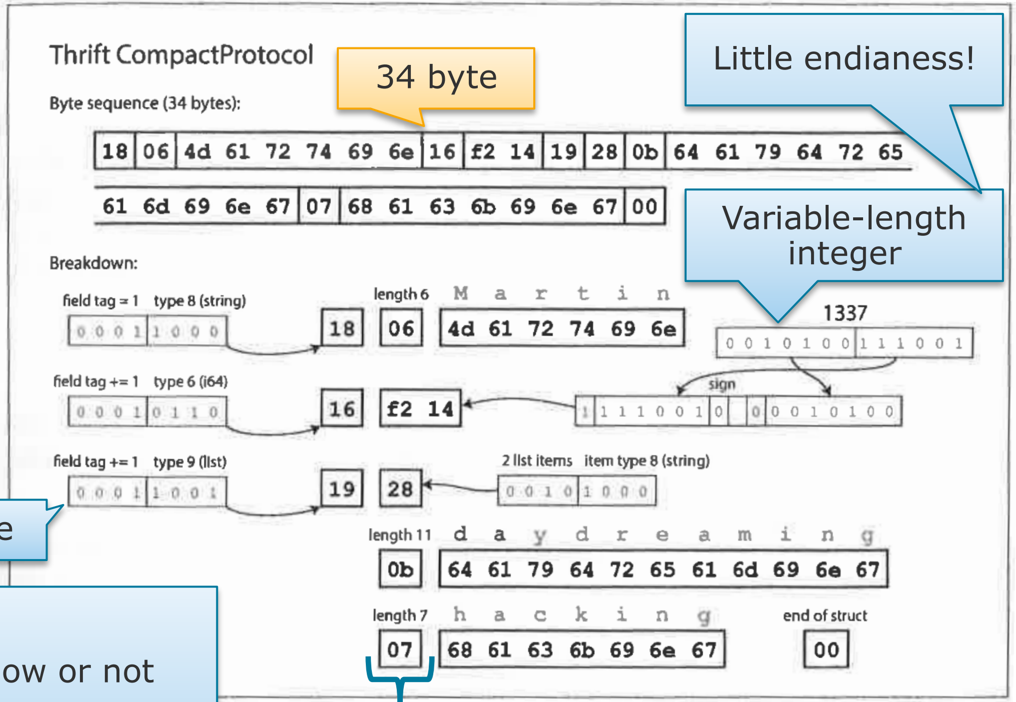
JSON Format

81 byte

```
struct Person {
  1: required string       username,
  2: optional i64          favoriteNumber,
  3: optional list<string> interests
}
```

Thrift Struct

Field tag + type in one byte



Variable-length integers

- First bit in each byte encodes if more bytes follow or not (0 = "last byte", 1 = "more bytes to come")
- Last bit of first byte encodes the integers sign (0 = "+", 1 = "–")

Protocol Buffers

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

JSON Format

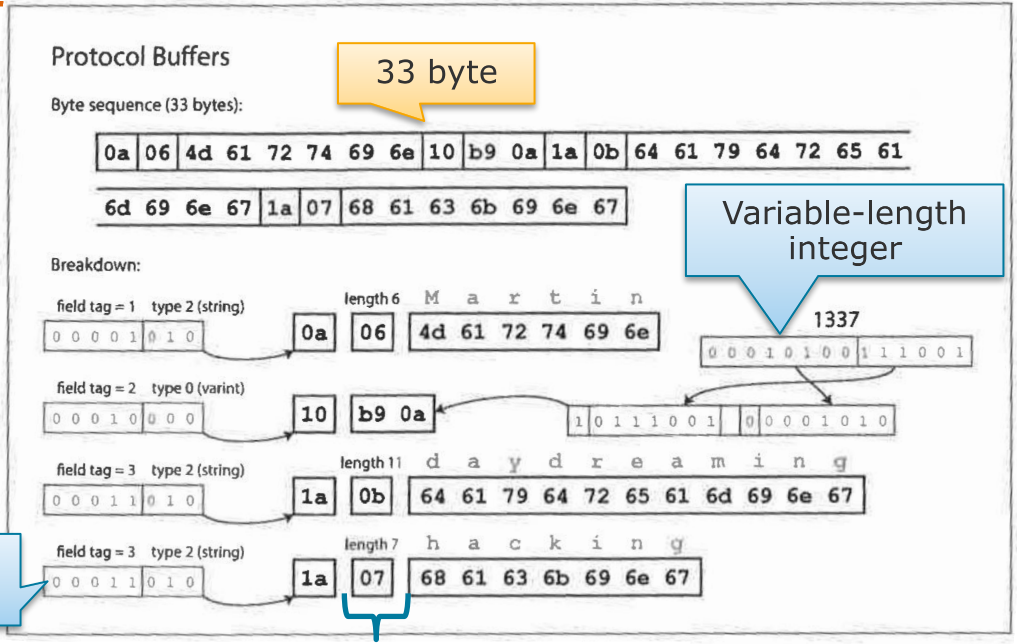
81 byte

```
message Person {
  required   string  username,        = 1;
  optional   i64     favoriteNumber,  = 2;
  repeated   string  interests        = 3;
}
```

P.B. message

Schema definition

Put values with same field tag in a list



Protocol Buffers are very similar to Thrift's CompactProtocol

Variable-length integers

Avro

Apache Avro

- A binary encoding format developed as a sub-project of Hadoop in 2009
  - https://avro.apache.org/
- Differences to Thrift and Protocol Buffers:
  - No tag numbers: fields are matched by order in schema and byte sequence
  - No field modifiers `optional` or `required`: optional fields have default values
  - Special data type `union`: specifies multiple data types (and `null` if allowed)
  - Nullable fields must have type `union`

```
record Person {
  string            username;
  union{null,long}  favoriteNumber=null;
  array<string>     interests;
}
```

Avro record

Schema definition

Uses of Avro

- Apache Pig (query engine for Hadoop)
- Espresso (database management system)
- Avro RPC (remote procedure call protocol)
- …

Binary Encoding
Avro

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

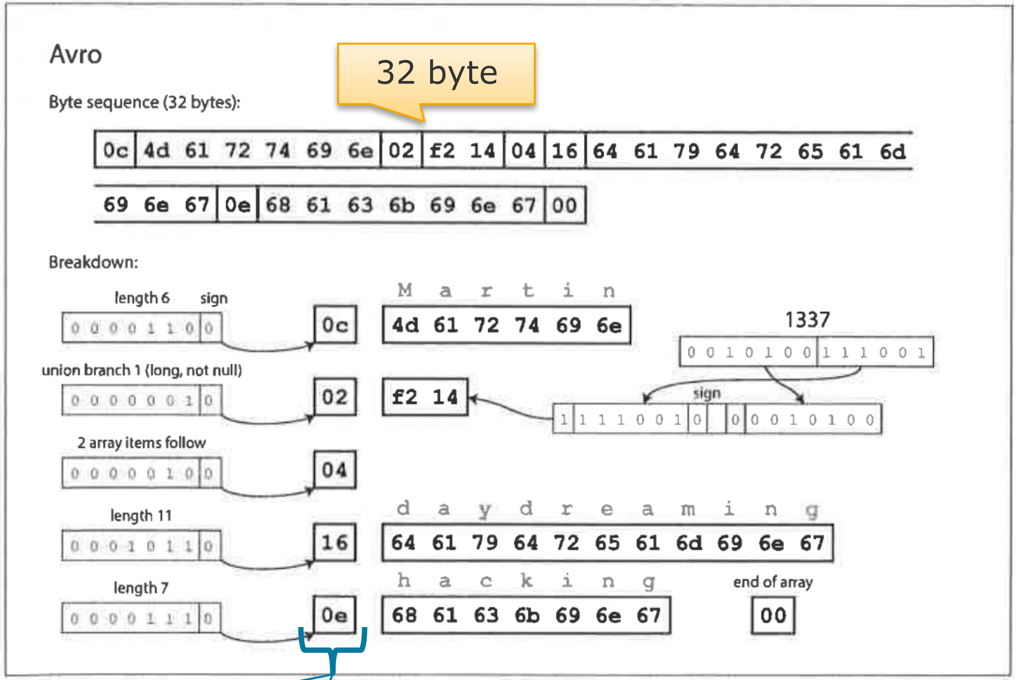
JSON Format

81 byte

```
record Person {
  string              username;
  union{null,long}    favoriteNumber=null;
  array<string>       interests;
}
```

Avro record

Schema definition



The data type is known;
the field is matched by sequence
→ we only need the length!

Avro: Writer's and Reader's schemata

- Avro associates data with two different schemata:
  - Writer's schema:
    - The schema with which the data was written
    - Fix for written data; might differ for other (newer/older) datasets
    - Stored with the data (in same file, database, or connection handshake)
  - Reader's schema:
    - The schema of the application reading the data
    - Might change with the version of the application
    - Stored in application
- When reading: Avro dynamically maps Reader's and Writer's schemata
- When writing: Avro uses the Reader's schema

"self-describing data"

Binary Encoding

Avro: W-R Mapping

Writer's schema

Data type	Field name
string	userName
union{null,long}	favoriteNumber
array<string>	interests
string	photoURL

Reader's schema

Data type	Field name
long	userID
union{null,int}	favoriteNumber
string	userName
array<string>	interests

Advantages

- Most compact binary encoding (compared with previous formats)
- Backward compatibility:
  - Avro dynamically maps schemata at read-time and resolves differences
  - Fields are mapped by name; no field tags that can break the encoding
  - Default values account for missing fields
  - Data types can change if conversion is possible (e.g. int → long, float → string)
- Schema generation:
  - Reader's schemata can be generated from existing data (no need to generate field tags that match a Writer's schema)

Check yourself

Suppose we have a linked list that is implemented as shown in the following code snippet:

```
public class IntLinkedList {
    int size;
    IntNode first;
    IntNode last;
    …

    private static class IntNode {
        int item;
        IntNode next;
        IntNode prev;
    }
    …
}
```

Question 1:

Give reasons why the default Java serializer should not be used here.

Question 2:

How would a more reasonable serialization look like?

Chapter 4. Encoding and Evolution