UNIVERSITÄT DUISBURG ESSEN

Internet-Technologie & Web Engineering
Data Representation

Dr.-Ing. Matthäus Wander

Universität Duisburg-Essen

Data

- Data transmitted over a network is just bytes
- It can represent:
  - Text
    - ASCII, UTF-8, UTF-16, …
  - Simple data types
    - byte, int, long, double, float, decimal, …
  - Data structures
    - Composite data consisting of simple types
  - Code
    - A *.class file, a DLL, or a Python module
  - Images
    - JPEG, GIF, PNG, …
  - Documents
    - Office Open XML, OpenDocument, PDF, …

Heterogeneity

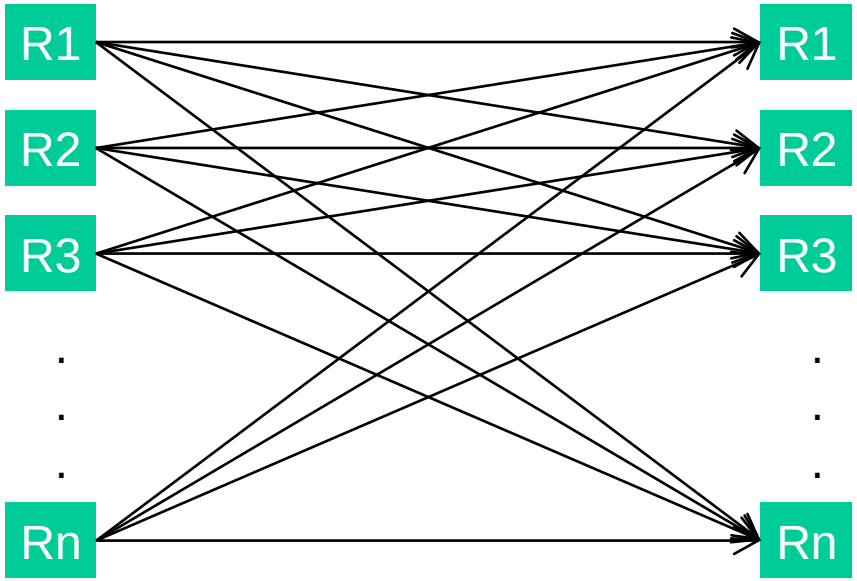
- Data representation can be different from machine to machine
  - Byte order (little-endian vs. big-endian)
  - Text encoding (US-ASCII vs. ISO 8859-1 vs …)
  - Integers (or pointers) may have different lengths on different CPUs
    - Atmel AVR microcontrollers: 8-bit registers
    - Intel 8086, Renesas M16C: 16-bit registers
    - x86: 32-bit registers
    - x86-64: 64-bit registers

Transformation

- A **transformation** is necessary
  - From one data format to another
  - Simple in theory …
  - … but usually with impact on performance
- Two transformation principles:
  1. Pair-wise transformation for every representation
  2. Transform to canonical representation

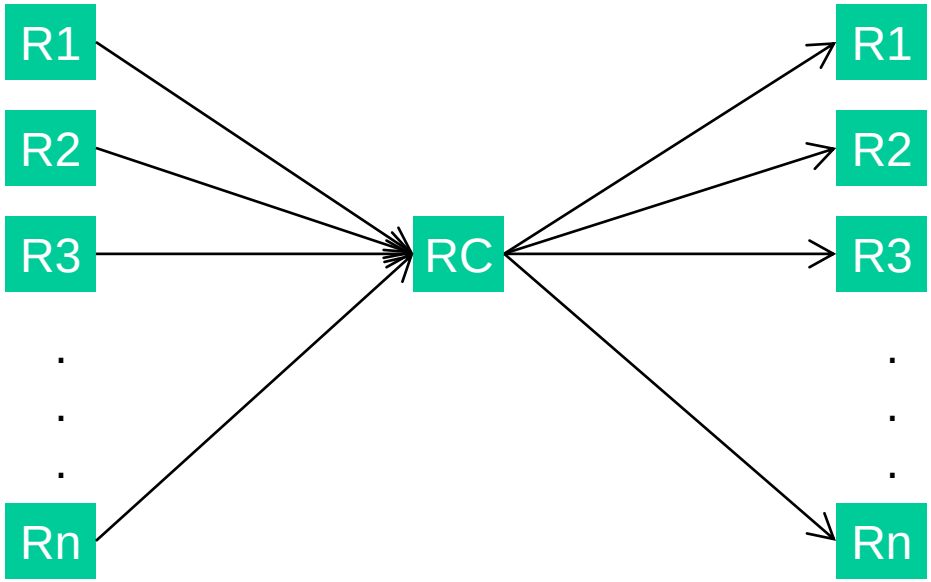
1. Pair-wise Transformation

- More code (n^2 transformations)
- Better performance (runs 1 transformation)
- Usually sender transmits indication of representation and “receiver makes it right”
- Drawback: Receiver must understand all formats



2. Transform to Canonical Representation

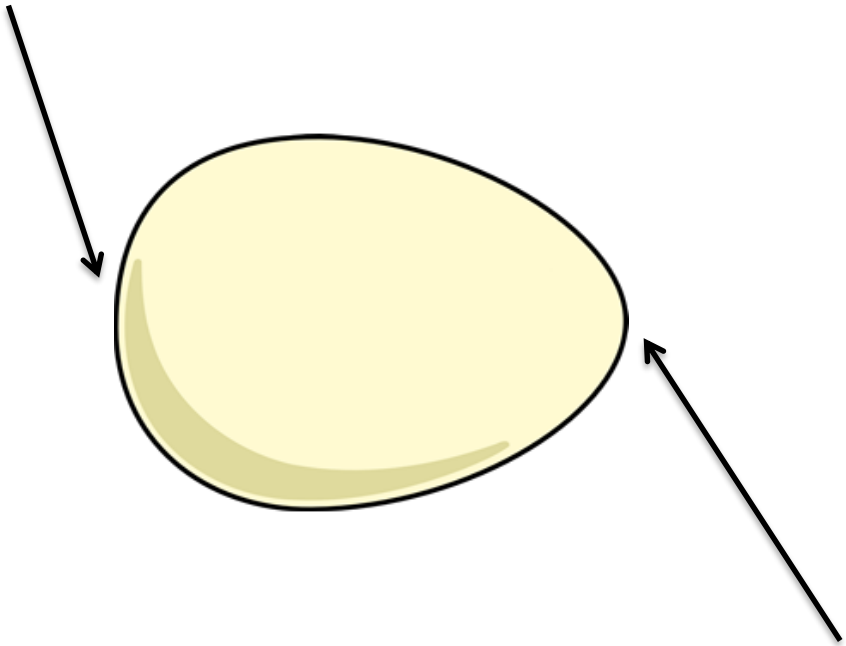
- Less to code (2n transformations)
- Lower performance (runs 2 transformations)
- External representation given by protocol specification or negotiated by communication partners

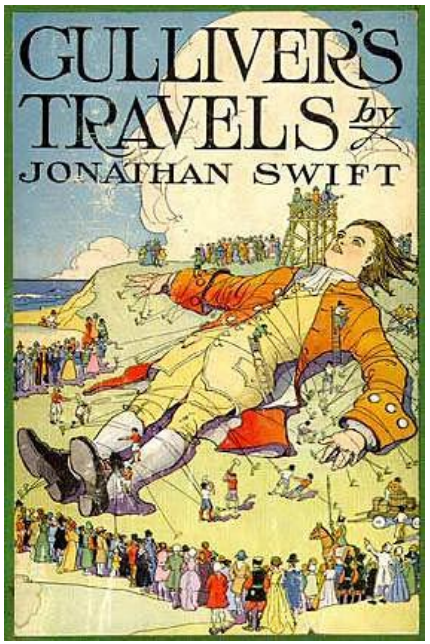


BYTE ORDER

Endianness / Byte Order

- How to crack an egg?
- Big-enders or **big-endians** crack it here

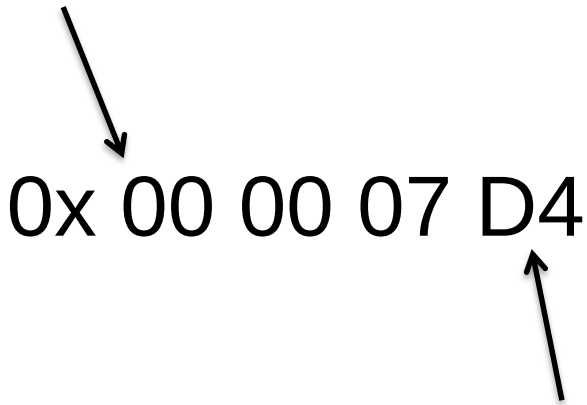




- Little-enders or **little-endians** crack it here

Endianness / Byte Order (2)

- How to store a number as 4-byte integer?
  - 2004 decimal = 0x000007D4 hexadecimal
- If you start here, it's **big-endian**
  - Most significant byte first



0x 00 00 07 D4

- If you start here, it's **little-endian**
  - Least significant byte first

Big-endian

0	1	2	3
00	00	07	D4

Little-endian

0	1	2	3
D4	07	00	00

Endianness / Byte Order (3)

- Identical number, different sequence of bytes
  - But what if sender and receiver use different orders?
- Most computer systems use little-endian
  - Intel x86, x86-64, …
- Some use big endian
  - Motorola 68k, PowerPC,…
- Most network protocols use big-endian
  - IP, TCP, UDP, DNS, HTTP/2, …
  - Network Byte Order

TEXT ENCODING

Text Encoding

- Characters and symbols are encoded as bytes
  - E.g. **"A" = 0x41, "B" = 0x42**
- Number of available characters (character set) depends on size of encoded character
  - E.g. one character = one byte → maximum of 256 characters
  - What about Chinese, Japanese, Greek, … characters?
- Multiple common encodings (character sets)
  - US-ASCII, ISO 8859, Unicode, …

US-ASCII

- 7-bit encoding: 128 characters
- Non-printable
  - Carriage return (CR)
  - Newline (LF)
  - Tab
- Printable
  - Latin alphabet
  - Numbers
  - Some symbols

Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

ISO 8859

- 8-bit encoding: 256 characters
  - Bottom 128: characters identical with ASCII
  - Upper 128: new characters
- Several versions
  - e.g. ISO 8859-1: Western European (Ó, Ä, â, ü, ß)
  - e.g. ISO 8859-5: Cyrillic (Й, д, Б)

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	*nicht belegt*															
1...																
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8...	*nicht belegt*															
9...																
A...	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
B...	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C...	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D...	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E...	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F...	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Unicode

- Unicode assigns a code point for each character
  - 1.1 million code points in *Universal Coded Character Set* (UCS, defined in ISO 10646)
  - First 128 code points: characters identical with ASCII
- Multiple encoding formats available
  - UTF-8: variable length 1-4 bytes
  - UTF-16: variable length 2 or 4 bytes
  - UCS-2: fixed length 2 bytes
    - Cannot represent all possible unicode characters
  - UTF-32: fixed length 4 bytes
- You can use Unicode e.g. in Java: "\u2260"

UTF-8

- Variable length 1-4 bytes per characters
- Single byte for code points U+0000 to U+007F
  - ASCII characters encoded as single byte 0x00 to 0x7F
  - Backward compatible with ASCII
- Multiple bytes for code points ≥U+0080

U-00000000 - U-0000007F	**0***xxxxxxx*
U-00000080 - U-000007FF	**110***xxxxx* **10***xxxxxx*
U-00000800 - U-0000FFFF	**1110***xxxx* **10***xxxxxx* **10***xxxxxx*
U-00010000 - U-001FFFFF	**11110***xxx* **10***xxxxxx* **10***xxxxxx* **10***xxxxxx*

UTF-8 by Example

- Unicode character U+00A9 (“©”) = $1010\ 1001_{(2)}$ is encoded in UTF-8 as:
  - $\underline{110}00010_{(2)}\ \underline{10}101001_{(2)}$

  = 0xC2 0xA9
- U+2260 (“≠“) = $0010\ 0010\ 0110\ 0000_{(2)}$ is encoded in UTF-8 as:
  - $\underline{1110}0010_{(2)}\ \underline{10}001001_{(2)}\ \underline{10}100000_{(2)}$

  = 0xE2 0x89 0xA0
- UTF-8 “wastes” bits on non-ASCII characters

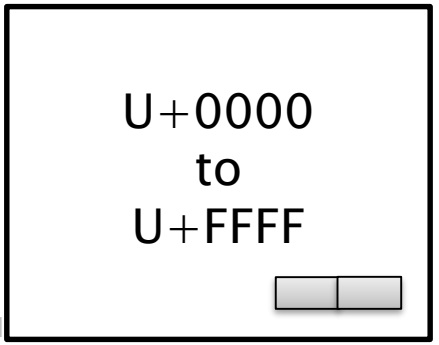
UTF-8 Decoding Algorithm

- First bit 0?
  - ⇨ process 1 byte and **return** character
- Second bit 0? (when in the middle of a character)
  - ⇨ skip 1 byte and **goto** start
- Third bit 0?
  - ⇨ process 2 bytes and **return** character
- … and so on

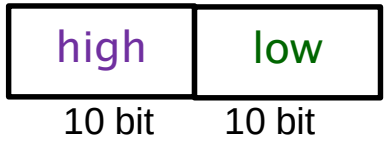
U-00000000 - U-0000007F	**0***xxxxxxx*
U-00000080 - U-000007FF	**110***xxxxx* **10***xxxxxx*
U-00000800 - U-0000FFFF	**1110***xxxx* **10***xxxxxx* **10***xxxxxx*
U-00010000 - U-001FFFFF	**11110***xxx* **10***xxxxxx* **10***xxxxxx* **10***xxxxxx*

UTF-16

- 2 bytes per character U+0000 to U+FFFF
  - e.g. “≠” ⇨ U+2260 ⇨ 0x2260 (16-bit number)
- 4 bytes per character >U+FFFF
  - Encoded as two 16-bit numbers called surrogate pairs
  - e.g. “𝍢” ⇨ U+24B62 ⇨ 0xD852 0xDF62 (2× 16-bit)
- Unicode reserves 2x 1024 code points for surrogates without any character definition
  - High surrogates: U+D800 to U+DBFF
  - Low surrogates: U+DC00 to U+DFFF

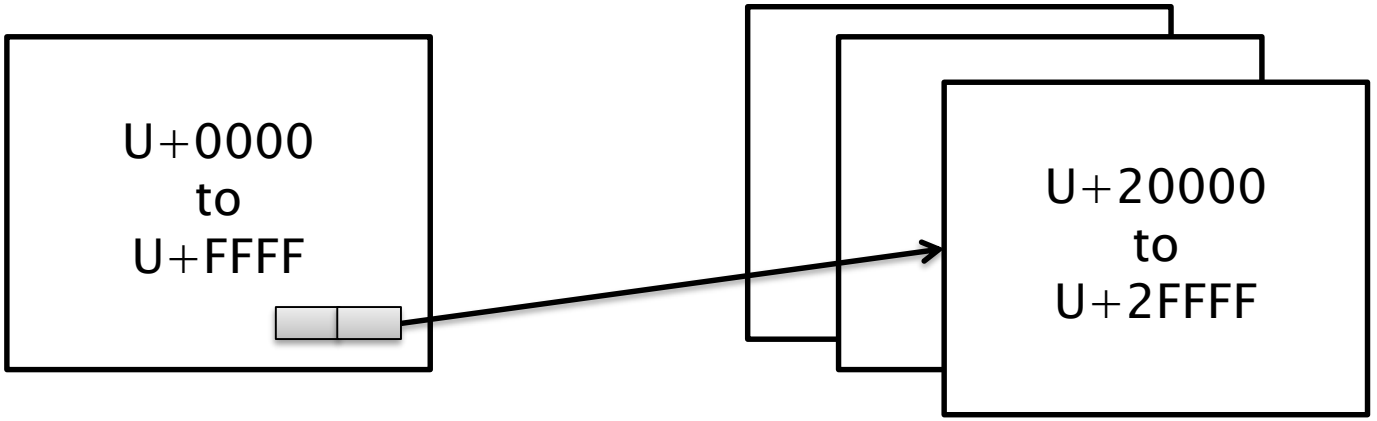


Decoding UTF-16 Surrogates

- Each surrogate encodes 10 bit of information
- UTF-16 example: 0xD852 0xDF62
  - 0xD852 – 0xD800 = 0x052 = 00 0101 0010$_{(2)}$
  - 0xDF62 – 0xDC00 = 0x362 = 11 0110 0010$_{(2)}$
- Convert surrogate pair to Unicode code point
  - 0x10000 + 

⇨ Unicode code point
  - 0001 0100 1011 0110 0010$_{(2)}$ = 0x14B62
  - 0x10000 + 0x14B62 ⇨ U+24B62 („𤭢“)

Unicode Planes



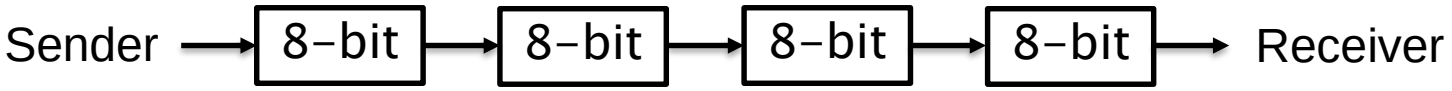
- Basic Multilingual Plane (BMP)
  - 2^{16} code points
- Surrogate code points are reserved and refer to characters in one of 16 supplementary planes
  - 2^{16} additional code points per supplementary plane
  - 16×2^{16} supplementary planes $= 2^{20}$ additional codes

UTF-32

- Uses fixed-length encoding
  - 32-bit numbers for all characters
- Easy to program and use
  - No variable length conversion like in UTF-8 or UTF-16
- But: wastes space for many characters
  - U+24B62 ⇨ 0x00024B62 = 0000 0000 0000 0010 0100 1011 0110 0010$_{(2)}$
- Sometimes used as internal format
  - Efficient addressing of single characters
  - Access 100th character: data[4*100]

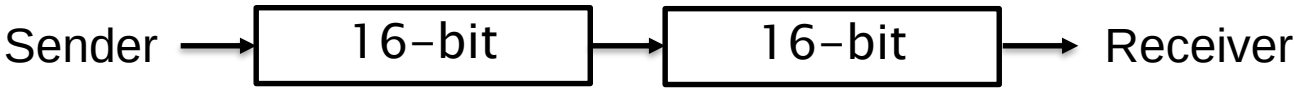
Byte Order in Unicode

- UTF-8: sequence of 1 to 4 bytes per char



  - Order of bytes guaranteed e.g. by stream socket
  - Each byte is an 8-bit number on its own
    ⇨ endianness irrelevant for UTF-8

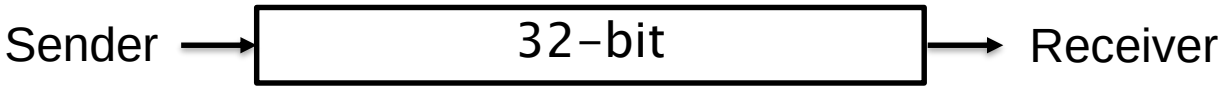
- UTF-16: sequence of 16-bit numbers



  - Order by bytes guaranteed e.g. by stream socket
  - But: what is the byte order of each 16-bit number?

Byte Order in Unicode (2)

- With UTF-16, byte order must be specified
  - Explicitly negotiate UTF-16 LE or BE (Little/Big Endian)
  - Use Byte Order Mark (BOM) before the first character
    - U+FEFF = Byte Order Mark (invisible character)
    - U+FFFE = reserved, not used
  - BOM has no meaning in UTF-8, but appears sometimes as 0xEF 0xBB 0xBF
- UTF-32: sequence of 32-bit numbers



  - Same issue about byte order as with UTF-16

TEXT-BASED AND BINARY ENCODING

Text-based vs. Binary Network Protocols

- What is the wire format of our network protocol?
  - Text, e.g. „Content-Length: 1028374“
  - Binary, e.g. 0x00 0x0F 0xB1 0x16
- Text-based protocols
  - Easy to understand and debug
  - Examples: SMTP, FTP, HTTP/1.1
- Binary protocols
  - More efficient (message size and conversion)
  - Examples: DNS, HTTP/2

JavaScript Object Notation

EXAMPLE: JSON

JSON by Example

```
{
  "bookid": 1234,
  "title": "JSON in 5 minutes",
  "author": "Guy Incognito",
  "price": 0.50,
  "availability": 1,
  "bookoftheyear": true,
}
```

JavaScript Object Notation (JSON)

- Text-based data interchange format
  - Principle: canonical representation
- Key-value data structure with basic data types
  - Supports nesting for hierarchical tree structures
  - Not well suitable for object graphs
- Self-describing: text format indicates tag length and nesting
- Platform-independent and widely supported
  - Originates from JavaScript for Web usage

JSON by Example (2)

```
{
  "libraries": [ "BA", "LK", "MC" ],
  "books": [
              {  "title": "JSON in 5 minutes",
                 "author": "Guy Incognito",
                 "availability": 1
              },
              {  "title": "Make Data Representation Great Again",
                 "author": "Jason Drumpf",
                 "availability": 17
              }
          ]
}
```

JSON Data Types

- Numbers: (un)signed integer, float/double
- String
- Boolean
- Array
  - **"someArray": [1, 2, 3, 4]**
- Object (key/value map)
  - **{ "foo": "bar", "moep": { "nested": "structure" } }**
- null

Extensible Markup Language

EXAMPLE: XML

XML by Example

```
<book bookid="1234">
   <title>XML in 5 minutes</title>
   <author>Guy Incognito</author>
   <price>0.50</price>
   <availability>1</availability>
   <bookoftheyear/>
</book>
```

Extensible Markup Language (XML)

- Text-based data interchange format
  - Principle: canonical representation
- Data structure with tags and attributes
  - Supports nesting for hierarchical tree structures
  - Not suitable for binary data (e.g. PNG image)
- Self-describing: text format indicates tag length and nesting
- Platform-independent and widely supported
  - Specified by W3C for the World Wide Web

XML Basics I

- XML is based on markup
  - Opening and closing tags

```
<tag>…</tag>
```

- A tag has …
  - … a name
  - … optional content between opening and closing tag
  - … an optional namespace
  - … optional attributes

```
<tag key="value">…</tag>
```

XML Basics II

- XML documents must be well-formed
  - Only Unicode characters
  - Have one unique root element
  - All start tags must match end tags (case sensitive)
  - All elements must be closed
  - All elements must be properly nested
  - All attribute values must be quoted
  - Reserved characters must be escaped
    - e.g. replace < in with XML entity `<`

XML by Example (2)

```
<root>
  <places>
    <library>BA</library> <library>LK</library> <library>MC</library>
  </places>
  <books>
    <book bookid="1234">
      <title>XML in 5 minutes</title>
      <author>Guy Incognito</author>
      <availability>1</availability>
    </book>
    <book bookid="5678">
      <title>Make Data Representation Great Again</title>
      <author>Jason Dumb</author>
      <availability>17</availability>
    </book>
  </books>
</root>
```

XML Features

- Structure somewhat evident for humans
  - Only syntax, semantics left for guessing
  - Human-readable or human-understandable?
- Verbose text representation is inefficient
  - Expensive conversion from binary data to text
- XML is a metalanguage that lets you derive your own data format
  - e.g. in natural text: "*there must be a tag 'book', which has one attribute 'bookid' and child tag 'author', …*"
  - e.g. as XML schema with automatic validation

XML Schema by Example

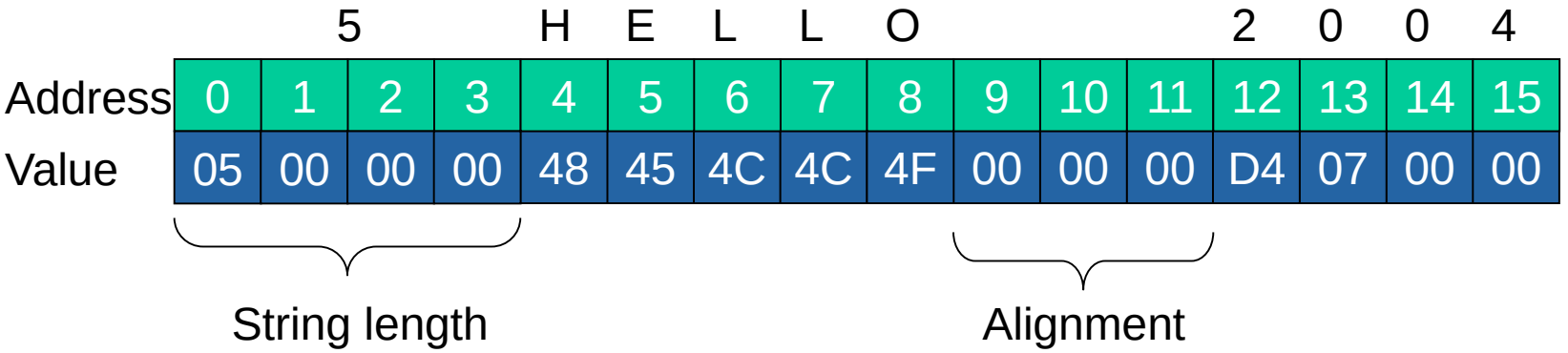
```
<xs:element name="book">

  <xs:complexType>

  <xs:attribute name="bookid" type="xs:positiveInteger"
   use="required" />

    <xs:sequence>

      <xs:element name="title" type="xs:string" />

      <xs:element name="author" type="xs:string" minOccurs="1"
   maxOccurs="unbounded" />

      <xs:element name="price" type="xs:decimal" />

      <xs:element name="availability"
   type="xs:nonNegativeInteger" />

    </xs:sequence>

  </xs:complexType>

</xs:element>
```

Common Data Representation

EXAMPLE: CDR

CDR by Example

- struct <string, unsigned long>
  - “Hello”, 2004
- Sender’s byte order: little-endian



Common Data Representation (CDR)

- Binary data format used by CORBA middleware
  - Principle: “Receiver makes it right”
  - Sender declares byte order in protocol headers
  - Data is aligned to multiples of given size value (remainder is zero-filled)
- Data structure with basic data types
  - Simple types (**`short`**, **`long`**, **`float`**, **`char`**, …)
  - Complex types (**`sequence`**, **`string`**, **`union`**, **`struct`**, …)
- No in-band indication of data structure
  - Requires external schema specification

Abstract Syntax Notation One

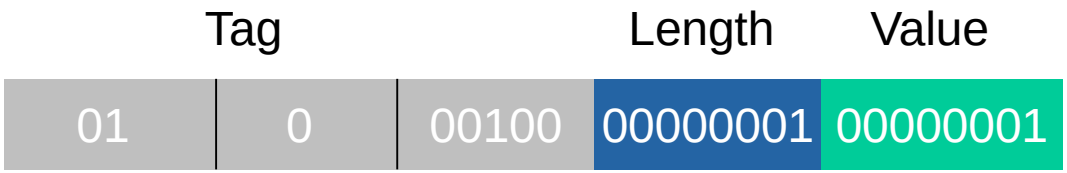
EXAMPLE: ASN.1

ASN.1 by Example

- Example definition:

```
BookType ::= INTEGER {
    paperback(0),
    hardcover(1),
    leatherback(2),
}
hardcover BookType ::= hardcover(1)
```

- **`hardcover`** is transferred as:



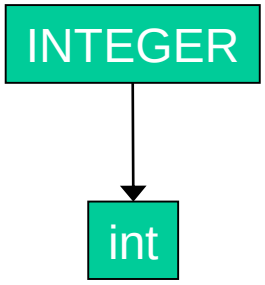
ISO ASN.1

- Standard for representation of data structures
  - Used e.g. in UMTS, SNMP, LDAP, X.509
  - Principle: canonical representation
- Provides syntax for schema of a data structure
  - Language-agnostic: abstract syntax is mapped to implementation-specific concrete syntax
- Several encoding formats (transfer syntax)
  - Binary: e.g. BER, CER, DER, …
  - Text: e.g. XER (XML-based), JER (JSON-based)

ASN.1 Abstract and Concrete Syntax

- Abstract Syntax
  - Describes data structure in its own abstract language
  - Based on predefined data types

    `BOOLEAN, INTEGER, ENUMERATED, STRING,…`
- Concrete Syntax
  - Describes representation of data structure in a selected programming language



ASN.1 abstract syntax

Java syntax

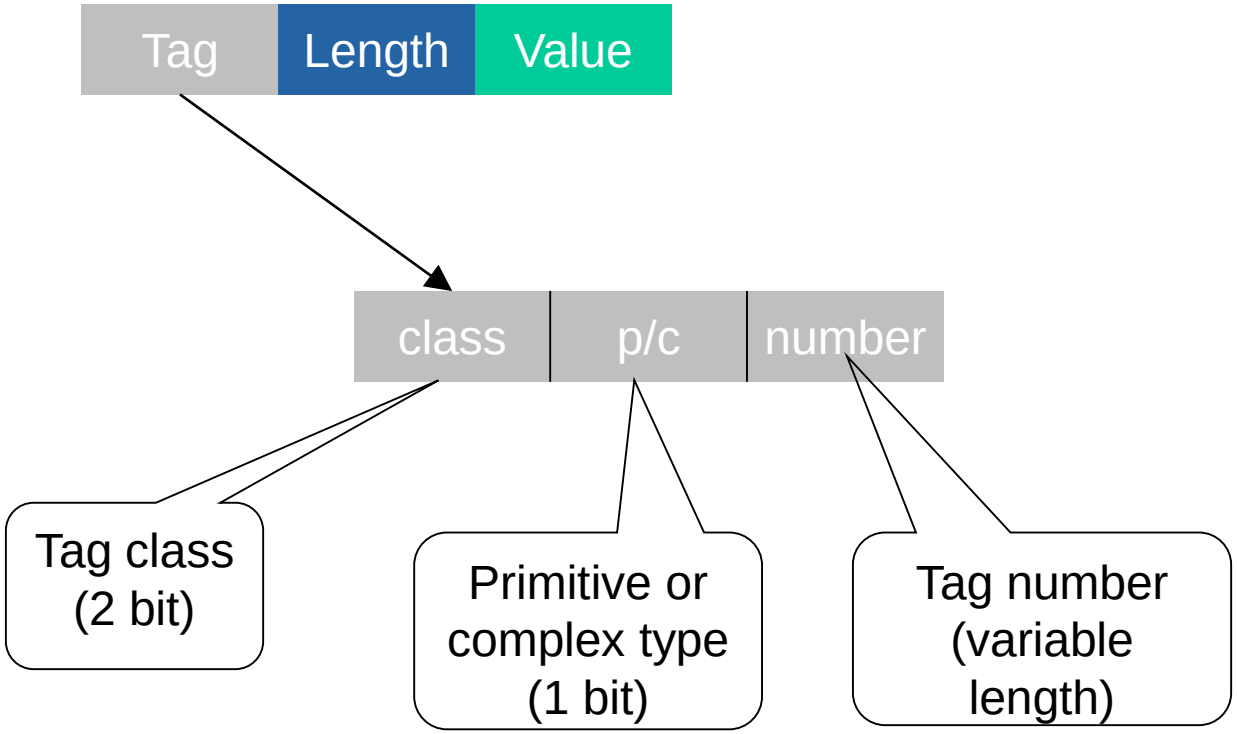
Concrete syntax

ASN.1 Transfer Syntax

- Example: Basic Encoding Rules (BER)
  - Binary representation of data structure
- Self-describing: data type is indicated for each field
- Self-delimiting: length of data is indicated for each field
- Principle: Canonical representation
  - However: same data structure can be represented with different BER encodings (different bytes on the wire)
  - DER is a unique representation of BER

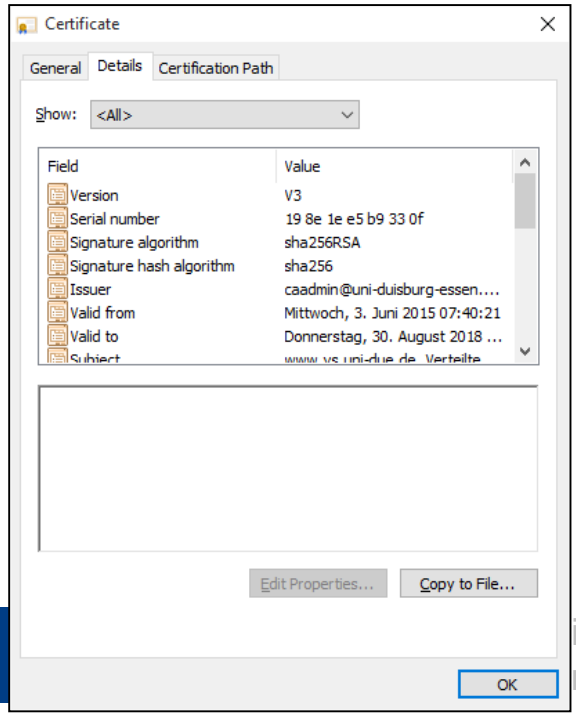
ASN.1 Basic Encoding Rules

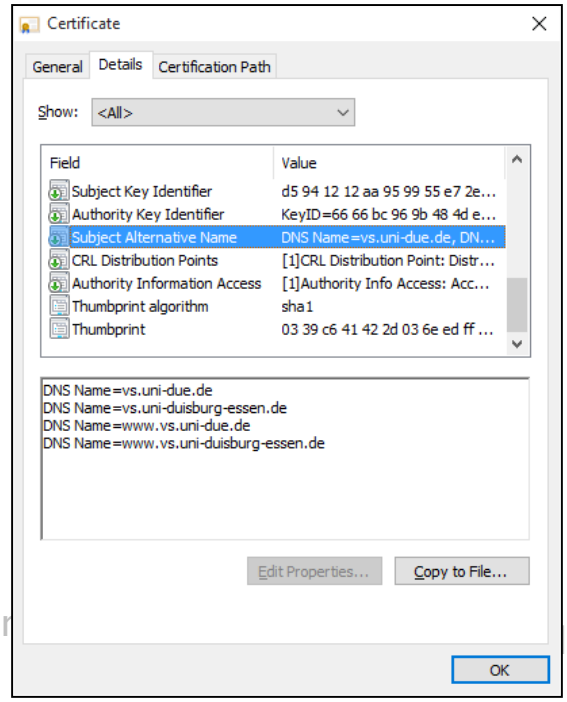
- Data is transferred in TLV coding (tag, length, value)



ASN.1 Example Application: X.509 Certificates

- X.509 certificates are ASN.1 data structures
  - Encoded with DER (unique representation of BER)
- Can include optional extension fields
  - Ignored by applications that don‘t know them





ENCODING PRINCIPLES

CDR versus ASN.1/BER

- CDR encodes values only
  - A CDR-encoded sequence of bytes cannot be decoded without the corresponding structure definition
  - However, the receiver most likely knows what to expect and how to interpret it
  - Advantage: efficient encoding (data only, no tags)
- ASN.1/BER encodes data as Tag & Value
  - Any BER-encoded sequence of bytes can be decomposed into its values
  - However, the semantics of these values is not encoded
  - Disadvantage: tags consume space

External Schema Specification

- How to indicate the schema of data structures?
  - e.g. <String, int>
- Static schema in external protocol specification
  - Syntax: server must send string, then number
  - Semantics: string is text message, number is user id
- How to determine the length of a field?
- What if the protocol changes?
  - Server sends <int, String, int>
  - Old implementation will interpret int as a String!

Self-describing Data Structures

- In-band indication of schema
  - <<Type, Length, String>, <Type, Length, int>>
- Receiver can detect and handle unexpected data
  - Unknown Type? Skip and continue
- But: schema does not define semantics
  - What is the meaning of the String or the int?
  - Still relying on external protocol specification
- Send version field in your protocol to recognize changes in semantics

Conclusion

- We need data representation for interoperability
  - Challenges: byte order, integer length, text encoding
- Text encoding: use UTF-8 or (subset of) ASCII
- Data format may be self-describing or rely completely on an external schema
- Use standards in open heterogeneous environments (e.g. Internet)
  - Interoperable but inefficient, e.g. XML, JSON
  - Binary formats in general more efficient than text