合肥學院
HEFEI UNIVERSITY

OOP with Java
28. I/O and Streams

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

website

Introduction

- We have learned already a bit about input/output, when using the standard input, output, and error streams in Lesson 6: *Console I/O*

Introduction

- We have learned already a bit about input/output, when using the standard input, output, and error streams in Lesson 6: *Console I/O*
- But what are streams?

Introduction

- We have learned already a bit about input/output, when using the standard input, output, and error streams in Lesson 6: *Console I/O*
- But what are streams?
- Basically, a stream is a sequence of elements of the same type

Introduction

- We have learned already a bit about input/output, when using the standard input, output, and error streams in Lesson 6: *Console I/O*
- But what are streams?
- Basically, a stream is a sequence of elements of the same type
- The elements of a stream are processed exactly in their sequence

Introduction

- We have learned already a bit about input/output, when using the standard input, output, and error streams in Lesson 6: *Console I/O*
- But what are streams?
- Basically, a stream is a sequence of elements of the same type
- The elements of a stream are processed exactly in their sequence
- Streams are either for reading or writing

Introduction

- We have learned already a bit about input/output, when using the standard input, output, and error streams in Lesson 6: *Console I/O*
- But what are streams?
- Basically, a stream is a sequence of elements of the same type
- The elements of a stream are processed exactly in their sequence
- Streams are either for reading or writing
- Streams are the prefered form of I/O in many scenarios

Basic I/O Streams

- In Java, the two most basic types of I/O streams are

Basic I/O Streams

- In Java, the two most basic types of I/O streams are:
  - `byte`-based streams for raw data I/O

Basic I/O Streams

- In Java, the two most basic types of I/O streams are:
  - `byte`-based streams for raw data I/O
  - `char`-based streams for text I/O

Byte Streams

- A byte stream is a sequence of `byte`s

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`, offering, amongst others,
  - the method `int read()` reading a single byte (0...255), returning -1 if the end of the stream is reached

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`, offering, amongst others,
  - the method `int read()` reading a single byte (0...255), returning -1 if the end of the stream is reached
  - the method `int read(byte[] dest)` tries to read up to `dest.length` bytes and store them into `dest` at once, returns the actual number of bytes read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`, offering, amongst others,
  - the method `int read()` reading a single byte (0...255), returning -1 if the end of the stream is reached
  - the method `int read(byte[] dest)` tries to read up to `dest.length` bytes and store them into `dest` at once, returns the actual number of bytes read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `int read(byte[] dest, int off, int len)` tries to read up to `len` bytes at once and store them into `dest` start at index `off`; returns the actual number of bytes read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`, offering, amongst others,
  - the method `int read()` reading a single byte (0...255), returning -1 if the end of the stream is reached
  - the method `int read(byte[] dest)` tries to read up to `dest.length` bytes and store them into `dest` at once, returns the actual number of bytes read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `int read(byte[] dest, int off, int len)` tries to read up to `len` bytes at once and store them into `dest` start at index `off`; returns the actual number of bytes read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `int available()` return a number of bytes available right now (0 does not mean that the stream has ended, more data may come later)

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`, offering, amongst others,
  - the method `int read()` reading a single byte (0...255), returning -1 if the end of the stream is reached
  - the method `int read(byte[] dest)` tries to read up to `dest.length` bytes and store them into `dest` at once, returns the actual number of bytes read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `int read(byte[] dest, int off, int len)` tries to read up to `len` bytes at once and store them into `dest` start at index `off`; returns the actual number of bytes read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `int available()` return a number of bytes available right now (0 does not mean that the stream has ended, more data may come later)
  - the method `void close()` implemented from `java.io.Closeable`

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`
- Byte streams for output are sub-classes of `java.io.OutputStream`

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`
- Byte streams for output are sub-classes of `java.io.OutputStream`, offering, amongst others,
  - the method `void write(int src)` writes a single byte (low-order eight bits of `src`)

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`
- Byte streams for output are sub-classes of `java.io.OutputStream`, offering, amongst others,
  - the method `void write(int src)` writes a single byte (low-order eight bits of `src`)
  - the method `void write(byte[] src)` writes the sequence of bytes from `src`

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`
- Byte streams for output are sub-classes of `java.io.OutputStream`, offering, amongst others,
  - the method `void write(int src)` writes a single byte (low-order eight bits of `src`)
  - the method `void write(byte[] src)` writes the sequence of bytes from `src`
  - the method `void write(byte[] src, int off, int len)` writes the sequence of `len` bytes from `src` starting at index `off`

Byte Streams

- A byte stream is a sequence of `byte`s
- Byte streams for input are sub-classes of `java.io.InputStream`
- Byte streams for output are sub-classes of `java.io.OutputStream`, offering, amongst others,
  - the method `void write(int src)` writes a single byte (low-order eight bits of `src`)
  - the method `void write(byte[] src)` writes the sequence of bytes from `src`
  - the method `void write(byte[] src, int off, int len)` writes the sequence of `len` bytes from `src` starting at index `off`
  - the method `void close()` implemented from `java.io.Closeable`

Byte Stream Implementations

- The concept of the byte stream base classes `InputStream` and `OutputStream` is quite general

Byte Stream Implementations

- The concept of the byte stream base classes `InputStream` and `OutputStream` is quite general
  - There could be implementations of this functionality to deal with actual files

Byte Stream Implementations

- The concept of the byte stream base classes `InputStream` and `OutputStream` is quite general
  - There could be implementations of this functionality to deal with actual files
  - There could be implementations of this functionality for the standard streams

Byte Stream Implementations

- The concept of the byte stream base classes `InputStream` and `OutputStream` is quite general
  - There could be implementations of this functionality to deal with actual files
  - There could be implementations of this functionality for the standard streams (`System.in` is actually this, while `System.out` and `System.err` offer additional functions for text)

Byte Stream Implementations

- The concept of the byte stream base classes `InputStream` and `OutputStream` is quite general
  - There could be implementations of this functionality to deal with actual files
  - There could be implementations of this functionality for the standard streams (`System.in` is actually this, while `System.out` and `System.err` offer additional functions for text)
  - There could be implementations of this functionality for TCP/IP internet connections

Byte Stream Implementations

- The concept of the byte stream base classes `InputStream` and `OutputStream` is quite general
  - There could be implementations of this functionality to deal with actual files
  - There could be implementations of this functionality for the standard streams (`System.in` is actually this, while `System.out` and `System.err` offer additional functions for text)
  - There could be implementations of this functionality for TCP/IP internet connections
  - There could be output streams writing to a buffer in memory or input streams reading from a byte array
  - ...

Byte Stream Implementations

- The concept of the byte stream base classes `InputStream` and `OutputStream` is quite general
  - There could be implementations of this functionality to deal with actual files
  - There could be implementations of this functionality for the standard streams (`System.in` is actually this, while `System.out` and `System.err` offer additional functions for text)
  - There could be implementations of this functionality for TCP/IP internet connections
  - There could be output streams writing to a buffer in memory or input streams reading from a byte array
  - ...
- What the methods actually do depends on the implementations in the corresponding subclasses

Byte Streams for Files

- The byte stream API has been implemented for basic file I/O as follows:

Byte Streams for Files

- The byte stream API has been implemented for basic file I/O as follows:
- `FileInputStream` reads one byte after the other from a file

Byte Streams for Files

- The byte stream API has been implemented for basic file I/O as follows:
- `FileInputStream` reads one byte after the other from a file
- It offers several constructors, one accepts the path to the file to read from as `String`

Byte Streams for Files

- The byte stream API has been implemented for basic file I/O as follows:
- `FileInputStream` reads one byte after the other from a file
- It offers several constructors, one accepts the path to the file to read from as `String`
- `FileOutputStream` writes one byte after the other to a file

Byte Streams for Files

- The byte stream API has been implemented for basic file I/O as follows:
- `FileInputStream` reads one byte after the other from a file
- It offers several constructors, one accepts the path to the file to read from as `String`
- `FileOutputStream` writes one byte after the other to a file
- It offers several constructors, one accepts the path to the file to created and written to as `String`

Copying Files via Byte Streams

- We can use this to write a small program to copy a single file, taking as command line arguments two paths

Copying Files via Byte Streams

- We can use this to write a small program to copy a single file, taking as command line arguments two paths:
  1. the source path to the file to copy

Copying Files via Byte Streams

- We can use this to write a small program to copy a single file, taking as command line arguments two paths:
  1. the source path to the file to copy
  2. the destination path where the file should be copied to

Copying Files via Byte Streams

- We can use this to write a small program to copy a single file, taking as command line arguments two paths:
  1. the source path to the file to copy
  2. the destination path where the file should be copied to
- A first implementation of the file copying procedure could look like this:

Copying Files via Byte Streams

Listing: Copying a file byte-by-byte

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/** a class copying a raw file byte for byte: slow */
public class CopyRawFileBytewise {
  /** The main routine
   *  @param args  args[0]=source file, args[1]=target file */
  public static void main(String[] args) { // we use try-with-resource...
    try (final FileInputStream source = new FileInputStream(args[0])) {
      try (final FileOutputStream target = new FileOutputStream(args[1])) {
        int readByte;
        while ((readByte = source.read()) >= 0) { // while not end-of-stream
          target.write(readByte);              // write the byte we just read
        }
      } // closes target, the "}" in the next line closes source
    } catch (IOException error) { // IOExceptions are checked exceptions
      System.out.println("Copying has failed."); //$NON-NLS-1$
      error.printStackTrace();    // print stack trace
    }
  }
}
```

Copying Files via Byte Streams

- We can use this to write a small program to copy a single file, taking as command line arguments two paths:
  1. the source path to the file to copy
  2. the destination path where the file should be copied to
- A first implementation of the file copying procedure could look like this:
- Copying files byte-by-byte this way means to do a lot of system calls and is slow

Copying Files via Byte Streams

- We can use this to write a small program to copy a single file, taking as command line arguments two paths:
  1. the source path to the file to copy
  2. the destination path where the file should be copied to
- A first implementation of the file copying procedure could look like this:
- Copying files byte-by-byte this way means to do a lot of system calls and is slow
- We could instead allocate a buffer to hold several bytes at once and use the array-based methods

Copying Files via Byte Streams

Listing: Copying a file using a buffer

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/** a class copying a raw file by using a buffer: faster */
public class CopyRawFileUsingBuffer {

  /** The main routine
   *  @param args  args[0]=source file, args[1]=target file */
  public static void main(String[] args) { // we use try-with-resource...
    try (final FileInputStream source = new FileInputStream(args[0])) {
      try (final FileOutputStream target = new FileOutputStream(args[1])) {
        byte[] buffer = new byte[4096]; // a reasonable sized buffer
        int readAmount;                 // the number of bytes actually read

        while ((readAmount = source.read(buffer)) > 0) { // fill buffer
          target.write(buffer, 0, readAmount); // write the bytes we just read
        }
      } // closes target, the "}" in the next line closes source
    } catch (IOException error) { // IOExceptions are checked exceptions
      System.out.println("Copying has failed."); //$NON-NLS-1$
      error.printStackTrace();    // print stack trace
    }
  }
}
```

stdin, stdout, and stderr

- stdin (`Sytem.in`) is a `java.io.InputStream`

stdin, stdout, and stderr

- stdin (`Sytem.in`) is a `java.io.InputStream`
- stdout (`System.out`) and stderr (`System.err`) are `java.io.OutputStream`s (special ones)

stdin, stdout, and stderr

- stdin (`Sytem.in`) is a `java.io.InputStream`
- stdout (`System.out`) and stderr (`System.err`) are `java.io.OutputStream`s (special ones)
- Based on the previous examples, we can now copy data from stdin to a file or from a file to stdout or stderr

Listing: Copying stdin to a file

```
import java.io.FileOutputStream;
import java.io.IOException;

/** a class copying all bytes read from stdin to a file by using a buffer:
    faster */
public class CopyStdInToFileUsingBuffer {

  /** The main routine
   *  @param args  args[0]=target file */
  public static void main(String[] args) { // we use try-with-resource...
    try (final FileOutputStream target = new FileOutputStream(args[0])) {
      byte[] buffer = new byte[4096]; // a reasonable sized buffer
      int readAmount;                 // the number of bytes actually read

      while ((readAmount = System.in.read(buffer)) > 0) { // fill buffer
        target.write(buffer, 0, readAmount); // write the bytes we just read
      }
    } catch (IOException error) { // IOExceptions are checked exceptions
      System.out.println("Copying has failed."); //$NON-NLS-1$
      error.printStackTrace(); // print stack trace
    }
  }
}
```

Listing: Copying a file to stdout

```
import java.io.FileInputStream;
import java.io.IOException;

/** a class copying a raw file to stdout by using a buffer: faster */
public class CopyFileToStdOutUsingBuffer {

  /** The main routine
   *  @param args  args[0]=source file */
  public static void main(String[] args) { // we use try-with-resource...
    try (final FileInputStream source = new FileInputStream(args[0])) {
      byte[] buffer = new byte[4096]; // a reasonable sized buffer
      int readAmount; // the number of bytes actually read

      while ((readAmount = source.read(buffer)) > 0) { // fill buffer
        System.out.write(buffer, 0, readAmount); // write the bytes we just read
      }
    } catch (IOException error) { // IOExceptions are checked exceptions
      System.out.println("Copying has failed."); //$NON-NLS-1$
      error.printStackTrace(); // print stack trace
    }
  }
}
```

Listing: Copying a file to stderr

```
import java.io.FileInputStream;
import java.io.IOException;

/** a class copying a raw file to stdout by using a buffer: faster */
public class CopyFileToStdErrUsingBuffer {

  /** The main routine
   *  @param args  args[0]=source file */
  public static void main(String[] args) { // we use try-with-resource...
    try (final FileInputStream source = new FileInputStream(args[0])) {
      byte[] buffer = new byte[4096]; // a reasonable sized buffer
      int readAmount; // the number of bytes actually read

      while ((readAmount = source.read(buffer)) > 0) { // fill buffer
        System.err.write(buffer, 0, readAmount); // write the bytes we just read
      }
    } catch (IOException error) { // IOExceptions are checked exceptions
      System.out.println("Copying has failed."); //$NON-NLS-1$
      error.printStackTrace(); // print stack trace
    }
  }
}
```

stdin, stdout, and stderr

- stdin (`Sytem.in`) is a `java.io.InputStream`
- stdout (`System.out`) and stderr (`System.err`) are `java.io.OutputStream`s (special ones)
- Based on the previous examples, we can now copy data from stdin to a file or from a file to stdout or stderr
- Warning: This is just an example, *never* use byte streams with text data...

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation:
  - it writes the contents to an internal `byte` array which is re-sized as needed

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation:
  - it writes the contents to an internal `byte` array which is re-sized as needed
  - we can get a copy of this byte array via `byte[] toByteArray()`

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation:
  - it writes the contents to an internal `byte` array which is re-sized as needed
  - we can get a copy of this byte array via `byte[] toByteArray()`
  - we can write the buffered contents to another output stream via `writeTo(OutputStream)`

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation:
  - it writes the contents to an internal `byte` array which is re-sized as needed
  - we can get a copy of this byte array via `byte[] toByteArray()`
  - we can write the buffered contents to another output stream via `writeTo(OutputStream)`
  - we can reset the stream via `reset()` to use it again

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation:
  - it writes the contents to an internal `byte` array which is re-sized as needed
  - we can get a copy of this byte array via `byte[] toByteArray()`
  - we can write the buffered contents to another output stream via `writeTo(OutputStream)`
  - we can reset the stream via `reset()` to use it again
- `java.io.ByteArrayInputStream` is an `InputStream` implementation

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation:
  - it writes the contents to an internal `byte` array which is re-sized as needed
  - we can get a copy of this byte array via `byte[] toByteArray()`
  - we can write the buffered contents to another output stream via `writeTo(OutputStream)`
  - we can reset the stream via `reset()` to use it again
- `java.io.ByteArrayInputStream` is an `InputStream` implementation:
  - reads the contents from an `byte` array passed to its constructor (and ends when the end of the array is reached)

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation:
  - it writes the contents to an internal `byte` array which is re-sized as needed
  - we can get a copy of this byte array via `byte[] toByteArray()`
  - we can write the buffered contents to another output stream via `writeTo(OutputStream)`
  - we can reset the stream via `reset()` to use it again
- `java.io.ByteArrayInputStream` is an `InputStream` implementation:
  - reads the contents from an `byte` array passed to its constructor (and ends when the end of the array is reached)
  - `ByteArrayInputStream(byte[] b)` reads from the whole array

Byte Array Streams

- `java.io.ByteArrayOutputStream` is an `OutputStream` implementation:
  - it writes the contents to an internal `byte` array which is re-sized as needed
  - we can get a copy of this byte array via `byte[] toByteArray()`
  - we can write the buffered contents to another output stream via `writeTo(OutputStream)`
  - we can reset the stream via `reset()` to use it again
- `java.io.ByteArrayInputStream` is an `InputStream` implementation:
  - reads the contents from an `byte` array passed to its constructor (and ends when the end of the array is reached)
  - `ByteArrayInputStream(byte[] b)` reads from the whole array
  - `ByteArrayInputStream(byte[] b, int off, int len)` reads only in the `len` bytes starting at offset `off`

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?
- We need a standardized mapping from these types to raw bytes

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?
- We need a standardized mapping from these types to raw bytes
- Such a mapping exists (we won't discuss it here) and is implemented in the `Data*Streams`

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?
- We need a standardized mapping from these types to raw bytes
- Such a mapping exists (we won't discuss it here) and is implemented in the `Data*Streams`:
  - `DataInputStream` is a subclass of `InputStream`

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?
- We need a standardized mapping from these types to raw bytes
- Such a mapping exists (we won't discuss it here) and is implemented in the `Data*Streams`:
  - `DataInputStream` is a subclass of `InputStream`:
    - its constructor takes an `InputStream` as parameter from which it will read

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?
- We need a standardized mapping from these types to raw bytes
- Such a mapping exists (we won't discuss it here) and is implemented in the `Data*Streams`:
  - `DataInputStream` is a subclass of `InputStream`:
    - its constructor takes an `InputStream` as parameter from which it will read
    - it additionally offers a method of the form `readXXX` for reading one instance of each primitive type

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?
- We need a standardized mapping from these types to raw bytes
- Such a mapping exists (we won't discuss it here) and is implemented in the `Data*Streams`:
  - `DataInputStream` is a subclass of `InputStream`:
    - its constructor takes an `InputStream` as parameter from which it will read
    - it additionally offers a method of the form `readXXX` for reading one instance of each primitive type
  - `DataOutputStream` is a subclass of `OutputStream`

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?
- We need a standardized mapping from these types to raw bytes
- Such a mapping exists (we won't discuss it here) and is implemented in the `Data*Streams`:
  - `DataInputStream` is a subclass of `InputStream`:
    - its constructor takes an `InputStream` as parameter from which it will read
    - it additionally offers a method of the form `readXXX` for reading one instance of each primitive type
  - `DataOutputStream` is a subclass of `OutputStream`:
    - its constructor takes an `OutputStream` as parameter to which it will write

Reading/Writing Structured Data

- What if you want to not just write single `byte` values, but other primitive types such as `int`, `double`, etc?
- We need a standardized mapping from these types to raw bytes
- Such a mapping exists (we won't discuss it here) and is implemented in the `Data*Streams`:
  - `DataInputStream` is a subclass of `InputStream`:
    - its constructor takes an `InputStream` as parameter from which it will read
    - it additionally offers a method of the form `readXXX` for reading one instance of each primitive type
  - `DataOutputStream` is a subclass of `OutputStream`:
    - its constructor takes an `OutputStream` as parameter to which it will write
    - it additionally offers a method of the form `writeXXX` for writing one instance of each primitive type

Listing: Using Byte Array Streams and Data Streams

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

/** a class writing some primitive types to a buffer, printing the buffer, then reading the values again */
public class DataAndByteIOStreams {

  /** The main routine
   *  @param args  args[0]=target file */
  public static void main(String[] args) { // we use try-with-resource...
    byte[] buffer;

    try { // wrap all code in a huge try-catch clause
      try (ByteArrayOutputStream bos = new ByteArrayOutputStream()) {
        try (DataOutputStream dos = new DataOutputStream(bos)) {
          dos.writeLong(0x88_99_aa_bb_cc_dd_ee_ffL); // write 64bit long to dos, results in 8 bytes to bos
          dos.writeBoolean(true);  // write true to dos, results in byte value 1 to bos
          dos.writeFloat(2f);      // write float 2f to dos, results in 4 bytes (0x40_00_00_00) to bos
          dos.writeInt(8192 | 32); // 8192 | 32 = 0x00002020 to dos, resulting in 4 to bos
        } // automatically close the data output stream
        buffer = bos.toByteArray(); // get a copy of the buffer holding all writtendata
      } // close the byte array output stream

      System.out.print(buffer.length); // how many bytes were written? 8+1+4+4 = 17
      System.out.print(':');
      for (byte b : buffer) { // fast read-only iteration over buffer
        System.out.print('␣');
        System.out.print(Integer.toHexString(b & 0xff)); // write hex value of current byte
      }                      //     ________long___________   __float_ ___int___
      System.out.println(); // 17: 88 99 aa bb cc dd ee ff 1 40 0 0 0 0 0 20 20
                            //                        boolean-^
      try (ByteArrayInputStream bis = new ByteArrayInputStream(buffer)) { // now we read again from the buffer
        try (DataInputStream dis = new DataInputStream(bis)) { // and wrap bis into data input stream
          System.out.println(Long.toHexString(dis.readLong())); // read the long
          System.out.println(dis.readBoolean());                // read the boolean
          System.out.println(dis.readFloat());                  // read the float
          System.out.println(dis.readInt());                    // read the int
        } // automatically close dis
      } // automatically close bis

    } catch (IOException error) { // if something failed (that should really not happen here) ...
      error.printStackTrace();    // ... print the stack trace
    }
  }
}
```

More Byte Streams

- There are several more useful implementations of byte-based streams

More Byte Streams

- There are several more useful implementations of byte-based streams, e.g.,
  - `java.io.BufferedInputStream` / `java.io.BufferedOutputStream` wrap `java.io.InputStream` / `java.io.OutputStream` to provide buffered I/O which makes the single-byte operations faster

More Byte Streams

- There are several more useful implementations of byte-based streams, e.g.,
  - `java.io.BufferedInputStream` / `java.io.BufferedOutputStream` wrap `java.io.InputStream` / `java.io.OutputStream` to provide buffered I/O which makes the single-byte operations faster
  - `java.net.Socket`, implementing TCP sockets in Java, provides `java.io.InputStream` / `java.io.OutputStream` to read/write from an internet connection

More Byte Streams

- There are several more useful implementations of byte-based streams, e.g.,
  - `java.io.BufferedInputStream` / `java.io.BufferedOutputStream` wrap `java.io.InputStream` / `java.io.OutputStream` to provide buffered I/O which makes the single-byte operations faster
  - `java.net.Socket`, implementing TCP sockets in Java, provides `java.io.InputStream` / `java.io.OutputStream` to read/write from an internet connection
  - `java.io.ObjectInputStream` / `java.io.ObjectOutputStream` are similar to the data input/output streams, but additionally allow for reading/writing whole (serializable) objects

Character Encoding

- Different languages have different characters

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language (for German, we can e.g., replace some less important characters with "ä" and "ß"...)

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)
- These approaches are insufficient for other languages

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)
- These approaches are insufficient for other languages
- Universal Coded Character Set (UCS) [3] and Unicode [4–6]

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)
- These approaches are insufficient for other languages
- Universal Coded Character Set (UCS) [3] and Unicode [4–6]
- Encoded as `UTF-7`, `UTF-8` [7] (compatible to `ASCII`), `UTF-16`, and `UTF-32`

Character Encoding

- Different languages have different characters
- Originally, storage of text data mainly designed for US English
- Here, 1B per character is sufficient: `ASCII` / `ISO/IEC 8859-1` [1]
- Original idea: bytes have different meaning, depending on language
- `GB2312` [2] encoding especially for Chinese characters (2B for each non-`ASCII` char)
- These approaches are insufficient for other languages
- Universal Coded Character Set (UCS) [3] and Unicode [4–6]
- Encoded as `UTF-7`, `UTF-8` [7] (compatible to `ASCII`), `UTF-16`, and `UTF-32`
- When dealing with text data, we *must* make sure to use the right encoding!

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char`s

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`, offering, amongst others,
  - the method `int read()` reading a single character, returning -1 if the end of the stream is reached

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`, offering, amongst others,
  - the method `int read()` reading a single character, returning -1 if the end of the stream is reached
  - the method `int read(char[] dest)` tries to read up to `dest.length` characters and store them into `dest` at once, returns the actual number of characters read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`, offering, amongst others,
  - the method `int read()` reading a single character, returning -1 if the end of the stream is reached
  - the method `int read(char[] dest)` tries to read up to `dest.length` characters and store them into `dest` at once, returns the actual number of characters read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `int read(char[] dest, int off, int len)` tries to read up to `len` characters at once and store them into `dest` start at index `off`; returns the actual number of characters read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`, offering, amongst others,
  - the method `int read()` reading a single character, returning -1 if the end of the stream is reached
  - the method `int read(char[] dest)` tries to read up to `dest.length` characters and store them into `dest` at once, returns the actual number of characters read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `int read(char[] dest, int off, int len)` tries to read up to `len` characters at once and store them into `dest` start at index `off`; returns the actual number of characters read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `boolean ready()` return `true` if characters are ready to read and `read` won't block

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`, offering, amongst others,
  - the method `int read()` reading a single character, returning -1 if the end of the stream is reached
  - the method `int read(char[] dest)` tries to read up to `dest.length` characters and store them into `dest` at once, returns the actual number of characters read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `int read(char[] dest, int off, int len)` tries to read up to `len` characters at once and store them into `dest` start at index `off`; returns the actual number of characters read (may be less, e.g., if end of stream is reached), or -1 if end of stream already reached
  - the method `boolean ready()` return `true` if characters are ready to read and `read` won't block
  - the method `void close()` implemented from `java.io.Closeable`

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`
- Character streams for output are sub-classes of `java.io.Writer`

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`
- Character streams for output are sub-classes of `java.io.Writer`, offering, amongst others,
  - the method `void write(int)` writes a single character

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`
- Character streams for output are sub-classes of `java.io.Writer`, offering, amongst others,
  - the method `void write(int)` writes a single character
  - the method `void write(char[] src)` writes the sequence of characters from `src`

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`
- Character streams for output are sub-classes of `java.io.Writer`, offering, amongst others,
  - the method `void write(int)` writes a single character
  - the method `void write(char[] src)` writes the sequence of characters from `src`
  - the method `void write(char[] src, int off, int len)` writes the sequence of `len` characters from `src` starting at index `off`

Character Streams

- Java provides a stream API for character: a character stream is a sequence of `char` s
- Character streams for input are sub-classes of `java.io.Reader`
- Character streams for output are sub-classes of `java.io.Writer`, offering, amongst others,
  - the method `void write(int)` writes a single character
  - the method `void write(char[] src)` writes the sequence of characters from `src`
  - the method `void write(char[] src, int off, int len)` writes the sequence of `len` characters from `src` starting at index `off`
  - the method `void close()` implemented from `java.io.Closeable`

Character Streams for Files

- The character stream API has been implemented for basic file I/O as follows:

Character Streams for Files

- The character stream API has been implemented for basic file I/O as follows:
- `FileReader` reads one character after the other from a file

Character Streams for Files

- The character stream API has been implemented for basic file I/O as follows:
- `FileReader` reads one character after the other from a file:
  - It offers several constructors, one accepts the path to the file to read from as `String`

Character Streams for Files

- The character stream API has been implemented for basic file I/O as follows:
- `FileReader` reads one character after the other from a file:
  - It offers several constructors, one accepts the path to the file to read from as `String`
  - It assumes that the file is in the system's default character encoding and decodes the binary data read from the file accordingly

Character Streams for Files

- The character stream API has been implemented for basic file I/O as follows:
- `FileReader` reads one character after the other from a file:
  - It offers several constructors, one accepts the path to the file to read from as `String`
  - It assumes that the file is in the system's default character encoding and decodes the binary data read from the file accordingly
- `FileWriter` writes one character after the other to a file

Character Streams for Files

- The character stream API has been implemented for basic file I/O as follows:
- `FileReader` reads one character after the other from a file:
  - It offers several constructors, one accepts the path to the file to read from as `String`
  - It assumes that the file is in the system's default character encoding and decodes the binary data read from the file accordingly
- `FileWriter` writes one character after the other to a file
  - It offers several constructors, one accepts the path to the file to created and written to as `String`

Character Streams for Files

- The character stream API has been implemented for basic file I/O as follows:
- `FileReader` reads one character after the other from a file:
  - It offers several constructors, one accepts the path to the file to read from as `String`
  - It assumes that the file is in the system's default character encoding and decodes the binary data read from the file accordingly
- `FileWriter` writes one character after the other to a file
  - It offers several constructors, one accepts the path to the file to created and written to as `String`
  - It will transform the characters to raw binary data using the system's default character encoding

Copying Text Files via Character Streams

- We can use this to write a small program to copy a single text file in the system's default encoding, taking as command line arguments two paths

Copying Text Files via Character Streams

- We can use this to write a small program to copy a single text file in the system's default encoding, taking as command line arguments two paths:
  1. the source path to the file to copy

Copying Text Files via Character Streams

- We can use this to write a small program to copy a single text file in the system's default encoding, taking as command line arguments two paths:
  1. the source path to the file to copy
  2. the destination path where the file should be copied to

Copying Text Files via Character Streams

- We can use this to write a small program to copy a single text file in the system's default encoding, taking as command line arguments two paths:
  1. the source path to the file to copy
  2. the destination path where the file should be copied to
- A first implementation of the file copying procedure could look like this:

Copying Text Files via Character Streams

Listing: Copying a text file character-by-character

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/** a class copying a text file character by character: slow */
public class CopyTextFileCharacterwise {

  /** The main routine
   *  @param args  args[0]=source file, args[1]=target file */
  public static void main(String[] args) { // we use try-with-resource...
    try (final FileReader source = new FileReader(args[0])) {
      try (final FileWriter target = new FileWriter(args[1])) {
        int readCharacter;
        while ((readCharacter = source.read()) >= 0) { // while not end-of-stream
          target.write(readCharacter);        // write the character we just read
        }
      } // closes target, the "}" in the next line closes source
    } catch (IOException error) { // IOExceptions are checked exceptions
      System.out.println("Copying has failed."); //$NON-NLS-1$
      error.printStackTrace();    // print stack trace
    }
  }
}
```

Copying Text Files via Character Streams

- We can use this to write a small program to copy a single text file in the system's default encoding, taking as command line arguments two paths:
  1. the source path to the file to copy
  2. the destination path where the file should be copied to
- A first implementation of the file copying procedure could look like this:
- Copying files character-by-character this way means to do a lot of system calls and is slow

Copying Text Files via Character Streams

- We can use this to write a small program to copy a single text file in the system's default encoding, taking as command line arguments two paths:
  1. the source path to the file to copy
  2. the destination path where the file should be copied to
- A first implementation of the file copying procedure could look like this:
- Copying files character-by-character this way means to do a lot of system calls and is slow
- We could instead allocate a buffer to hold several characters at once and use the array-based methods

Copying Text Files via Character Streams

Listing: Copying a text file using a buffer

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/** a class copying a text file by using a buffer: faster */
public class CopyTextFileUsingBuffer {

  /** The main routine
   *  @param args  args[0]=source file, args[1]=target file */
  public static void main(String[] args) { // we use try-with-resource...
    try (final FileReader source = new FileReader(args[0])) {
      try (final FileWriter target = new FileWriter(args[1])) {
        char[] buffer = new char[4096]; // a reasonable sized buffer
        int readAmount;                 // the number of characters actually read

        while ((readAmount = source.read(buffer)) > 0) { // fill buffer
          target.write(buffer, 0, readAmount); // write the characters we just read
        }
      } // closes target, the "}" in the next line closes source
    } catch (IOException error) { // IOExceptions are checked exceptions
      System.out.println("Copying has failed."); //$NON-NLS-1$
      error.printStackTrace();    // print stack trace
    }
  }
}
```

Character Streams wrapping Byte Streams

- The most basic character stream implementations directly wrap a byte stream

Character Streams wrapping Byte Streams

- The most basic character stream implementations directly wrap a byte stream
- `java.io.InputStreamReader`

Character Streams wrapping Byte Streams

- The most basic character stream implementations directly wrap a byte stream
- `java.io.InputStreamReader`:
  - reads its data from an `java.io.InputStream` passed to it in its constructor
  - as optional second parameter, the name of a text encoding can be provided (otherwise, the system's default encoding is used)

Character Streams wrapping Byte Streams

- The most basic character stream implementations directly wrap a byte stream
- `java.io.InputStreamReader`:
  - reads its data from an `java.io.InputStream` passed to it in its constructor
  - as optional second parameter, the name of a text encoding can be provided (otherwise, the system's default encoding is used)
- `java.io.OutputStreamWriter`

Character Streams wrapping Byte Streams

- The most basic character stream implementations directly wrap a byte stream
- `java.io.InputStreamReader`:
  - reads its data from an `java.io.InputStream` passed to it in its constructor
  - as optional second parameter, the name of a text encoding can be provided (otherwise, the system's default encoding is used)
- `java.io.OutputStreamWriter`:
  - writes its data to an `java.io.OutputStream` passed to it in its constructor

Character Streams wrapping Byte Streams

- The most basic character stream implementations directly wrap a byte stream
- `java.io.InputStreamReader`:
  - reads its data from an `java.io.InputStream` passed to it in its constructor
  - as optional second parameter, the name of a text encoding can be provided (otherwise, the system's default encoding is used)
- `java.io.OutputStreamWriter`:
  - writes its data to an `java.io.OutputStream` passed to it in its constructor
  - as optional second parameter, the name of a text encoding can be provided (otherwise, the system's default encoding is used)

Character Streams wrapping Byte Streams

- The most basic character stream implementations directly wrap a byte stream
- `java.io.InputStreamReader`:
  - reads its data from an `java.io.InputStream` passed to it in its constructor
  - as optional second parameter, the name of a text encoding can be provided (otherwise, the system's default encoding is used)
- `java.io.OutputStreamWriter`:
  - writes its data to an `java.io.OutputStream` passed to it in its constructor
  - as optional second parameter, the name of a text encoding can be provided (otherwise, the system's default encoding is used)
- These character streams thus can be used in any situation where we have byte streams, e.g., to work on `stdin`/`stdout` or on socket-provided streams of a TCP/IP internet connection

Character Streams wrapped around Byte Streams

- The code below is fully equivalent to the previous example...

Listing: Text File Copying using Character Streams wrapped around Byte Streams

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

/** a class copying a text file by using character streams wrapped around byte streams */
public class CopyTextFileUsingBufferAndWrappedStreams {

  /** The main routine
   *  @param args  args[0]=source file, args[1]=target file */
  public static void main(String[] args) { // we use try-with-resource...

    try (final FileInputStream fis = new FileInputStream(args[0])) {
      try (final InputStreamReader source = new InputStreamReader(fis)) {

        try (final FileOutputStream fos = new FileOutputStream(args[1])) {
          try (final OutputStreamWriter target = new OutputStreamWriter(fos)) {

            char[] buffer = new char[4096]; // a reasonable sized buffer
            int readAmount; // the number of characters actually read

            while ((readAmount = source.read(buffer)) > 0) { // fill buffer
              target.write(buffer, 0, readAmount); // write the characters we just read
            }
          }
        }
      }
    } catch (IOException error) { // IOExceptions are checked exceptions
      System.out.println("Copying has failed."); //$NON-NLS-1$
      error.printStackTrace(); // print stack trace
    }
  }
}
```

Advanced Character Streams

- There are several more useful implementations of character-based streams

Advanced Character Streams

- There are several more useful implementations of character-based streams, e.g.,
  - `java.io.BufferedReader` is wrapped around a `java.io.Reader` and offers not just faster, buffered reading, but also the ability to read a complete *line* of text via the method `String readLine()` returning a `String` containing a full line of text from its source (or `null` if the end of stream has been reached)

Advanced Character Streams

- There are several more useful implementations of character-based streams, e.g.,
  - `java.io.BufferedReader` is wrapped around a `java.io.Reader` and offers not just faster, buffered reading, but also the ability to read a complete *line* of text via the method `String readLine()` returning a `String` containing a full line of text from its source (or `null` if the end of stream has been reached)
  - `java.io.BufferedWriter` is wrapped around a `java.io.Writer` offers buffered writing and the method `newLine()` which starts a new line in the text output

Advanced Character Streams

- There are several more useful implementations of character-based streams, e.g.,
  - `java.io.BufferedReader` is wrapped around a `java.io.Reader` and offers not just faster, buffered reading, but also the ability to read a complete *line* of text via the method `String readLine()` returning a `String` containing a full line of text from its source (or `null` if the end of stream has been reached)
  - `java.io.BufferedWriter` is wrapped around a `java.io.Writer` offers buffered writing and the method `newLine()` which starts a new line in the text output
  - `java.io.CharArrayReader` and `java.io.CharArrayWriter` are the character stream equivalent of the byte stream `java.io.ByteArrayInputStream` and `java.io.ByteArrayOutputStream`

Summary

- We have learned about the concept of streams, sequential sources or destinations of data

Summary

- We have learned about the concept of streams, sequential sources or destinations of data
- We have learned that Java offers `byte`-based streams based on `java.io.InputStream` and `java.io.OutputStream`

Summary

- We have learned about the concept of streams, sequential sources or destinations of data
- We have learned that Java offers `byte`-based streams based on `java.io.InputStream` and `java.io.OutputStream`
- We have learned that text is actually a very complicated thing to deal with and we cannot handle it just with `byte`-based I/O

Summary

- We have learned about the concept of streams, sequential sources or destinations of data
- We have learned that Java offers `byte`-based streams based on `java.io.InputStream` and `java.io.OutputStream`
- We have learned that text is actually a very complicated thing to deal with and we cannot handle it just with `byte`-based I/O
- We have learned that Java offers character-based streams based on `java.io.Reader` and `java.io.Writer`

Summary

- We have learned about the concept of streams, sequential sources or destinations of data
- We have learned that Java offers `byte`-based streams based on `java.io.InputStream` and `java.io.OutputStream`
- We have learned that text is actually a very complicated thing to deal with and we cannot handle it just with `byte`-based I/O
- We have learned that Java offers character-based streams based on `java.io.Reader` and `java.io.Writer`
- We have seen that the concept of streams can be implemented with many different source and destination types, e.g., files, standard streams, memory buffers, internet connections, ...

Summary

- We have learned about the concept of streams, sequential sources or destinations of data
- We have learned that Java offers `byte`-based streams based on `java.io.InputStream` and `java.io.OutputStream`
- We have learned that text is actually a very complicated thing to deal with and we cannot handle it just with `byte`-based I/O
- We have learned that Java offers character-based streams based on `java.io.Reader` and `java.io.Writer`
- We have seen that the concept of streams can be implemented with many different source and destination types, e.g., files, standard streams, memory buffers, internet connections, ...
- Algorithms working on streams are thus naturally versatile

Summary

- We have learned about the concept of streams, sequential sources or destinations of data
- We have learned that Java offers `byte`-based streams based on `java.io.InputStream` and `java.io.OutputStream`
- We have learned that text is actually a very complicated thing to deal with and we cannot handle it just with `byte`-based I/O
- We have learned that Java offers character-based streams based on `java.io.Reader` and `java.io.Writer`
- We have seen that the concept of streams can be implemented with many different source and destination types, e.g., files, standard streams, memory buffers, internet connections, ...
- Algorithms working on streams are thus naturally versatile
- Java further makes heavy use of the concept of plugging streams together, e.g., we would normally hava an `java.io.InputStream`, wrap it into a `java.io.Reader`, which we would then wrap into a `java.io.BufferedReader` to be able to read text line-by-line

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog

Bibliography

Bibliography I

1. *ISO/IEC 8859-1 – Final Text of DIS 8859-1, 8-bit Single-Byte Coded Graphic Character Sets – Part 1: Latin Alphabet No.1*, volume ISO/IEC 8859-1:1997 (E). Geneva, Switzerland: International Organization for Standardization (ISO), February 12, 1998. URL `http://std.dkuug.dk/jtc1/sc2/wg3/docs/n411.pdf`.
2. Ken Lunde. *CJKV Information Processing*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999. ISBN 0-596-51447-6 and 1-56592-224-7. URL `http://examples.oreilly.de/english_examples/cjkvinfo/AppE/gb2312.pdf`.
3. *Information Technology – Universal Coded Character Set (UCS) (ISO/IEC 10646:2011)*. Geneva, Switzerland: International Organization for Standardization (ISO), 2011.
4. USA: The Unicode Consortium Mountain View, CA and Julie D. Allen. *The Unicode Standard, Version 5.0*. Reading, MA, USA: Addison-Wesley Professional, fifth edition, 2007. ISBN 0-321-48091-0 and 978-0-321-48091-0. URL `http://books.google.de/books?id=Yn1UAAAAMAAJ`.
5. The unicode consortium, 2011. URL `http://www.unicode.org/`.
6. Jukka K. Korpela. *Unicode Explained*. Internationalize Documents, Programs, and Web Sites. Sebastopol, CA, USA: O'Reilly Media, Inc., June 28, 2006. ISBN 059610121X and 9780596101213. URL `http://books.google.de/books?id=PcWU2yxc8WkC`.
7. François Yergeau. *STD 63: UTF-8, A Transformation Format of ISO 10646*, volume 3629 of *Request for Comments (RFC)*. Network Working Group, November 2003. URL `https://tools.ietf.org/html/rfc3629`.