

# JAVA

## Input/Output

# Overview

- package java.io
  - basic input/output
  - streams
    - bytes
  - since JDK1.1 Reader and Writer
    - chars (Unicode)
- package java.nio – since JDK1.4
  - channels, buffers
  - increased performance
    - classes from java.io internally reimplemented via java.nio
- java.io.Console
  - since JDK 6
  - access to the character console (if exists)
- NIO.2 – since JDK 7
  - mainly the package java.nio.file
  - operations with files, walking trees,...

I/O

Path

# Path

- `java.nio.file.Path`
  - interface
  - represents a path
  - obtaining a path
    - `Paths.get(String first, String... more)`
      - static method
      - ex.  
`Path p = Paths.get("home", "petr", "text.txt");`
    - `Path.of(String first, String... more)`
      - since Java 11
      - the same as `Paths.get()`
      - recommended to use
        - `Paths` maybe deprecated in future
    - `FileSystems.getDefault().getPath(String first, String... more)`
      - `Path.of()` – uses a default filesystem

# Path – methods

- path comparison
  - equals(), startsWith(), endsWith()
- relativization

```
Path p1 = Paths.get("joe");
Path p2 = Paths.get("sally");
Path p1_to_p2 = p1.relativize(p2); // -> ../sally
```
- obtaining actual path of a symlink
  - toRealPath()
- Path implements Iterable<Path>
  - iterates over the path's components
- normalize()
  - removing redundant path elements
    - d1/../d2/ => d1/d2
- ...

# Path – watching for changes

- WatchKey register(WatchService watcher, WatchEvent.Kind<?>... events)

```
WatchService watchService =
FileSystems.getDefault().newWatchService();
WatchKey key = this.path.register(watchService,
ENTRY_CREATE, ENTRY_DELETE);
while (true) {
    for (WatchEvent<?> l : key.pollEvents()) {
        ...
    }
    boolean valid = key.reset();
    if (!valid) {
        ...
    }
}
```

# java.nio.file.Files

- only static methods
  - copy(.. src, .. target, CopyOptions... options)
    - CopyOptions
      - REPLACE\_EXISTING
      - COPY\_ATTRIBUTES
      - NOFOLLOW\_LINKS
  - move(.. src, .. target, CopyOptions... options)
    - CopyOptions
      - ATOMIC\_MOVE
      - REPLACE\_EXISTING
  - delete(), deleteIfExists()
  - byte[] readAllBytes(Path p)
  - List<String> readAllLines(Path path)
  - Path write(Path path, byte[] bytes, OpenOption... options)
  - Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)

# CopyOptions, OpenOptions,...

- interfaces
- used in methods of the Files class
- implementations
  - StandardCopyOptions
    - enum (ATOMIC\_MOVE, COPY\_ATTRIBUTES,...)
  - StandardOpenOptions
    - enum (APPEND, READ, WRITE,...)
  - LinkOptions
    - enum (NOFOLLOW\_LINKS)



# java.nio.file.Files

- (cont.)
  - Path createLink(Path link, Path existing)
  - Path createSymbolicLink(Path link, Path target, FileAttribute<?>... attrs)
  - createDirectory(Path dir, FileAttribute<?>... attrs)
  - createDirectories(Path dir, FileAttribute<?>... attrs)
  - createFile(Path path, FileAttribute<?>... attrs)
  - createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)
  - createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)
  - “test” methods
    - isDirectory()
    - isRegularFile()
    - is....()

# java.nio.file.Files

- walking a file/direcotory tree
  - Path walkFileTree(Path start, FileVisitor<? super Path> visitor)
    - method of Files
  - interface FileVisitor<T>
    - FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
    - FileVisitResult postVisitDirectory(T dir, IOException exc)
    - FileVisitResult visitFile(T file, BasicFileAttributes attrs)
    - FileVisitResult visitFileFailed(T file, IOException exc)

# java.nio.file

- example – deleting a complete tree of files/directories

```
Path start = ...
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    public FileVisitResult visitFile(Path f,
        BasicFileAttributes attrs) throws IOException {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult postVisitDirectory(Path dir,
        IOException e) throws IOException {
        if (e == null) {
            Files.delete(dir);
            return FileVisitResult.CONTINUE;
        } else {
            throw e;
        }
    }
});
```

# java.io.File

- since Java 1.0
  - java.nio.files.Path – since Java 7
  - java.io.File is not deprecated
    - used in many places in the std. library
- also represents a path
  - similar usage as Path
  - but Path has better functionality
- conversions between them
  - File.toPath()
  - Path.toFile()

always prefer Path  
(if possible)

# Path / file separators

- fields of `java.io.File`
  - static `String pathSeparator`
  - static `char pathSeparatorChar`
    - path separator
  - static `String separator`
  - static `char separatorChar`
    - name separator in paths
- a method of `java.nio.file.FileSystem`
  - `String getSeparator()`

## Streams

# Overview

- since Java 1.0
- **InputStream**
  - `int read()`
    - reads one byte from an input (returns -1 if the end is reached)
  - `int read(byte[] b)`
    - reads several bytes (returns number of read bytes or -1)
- **OutputStream**
  - `void write(int b)`
  - `void write(byte[] a)`
- all other I/O classes derived from the **InputStream/OutputStream**
  - children are used
  - **InputStream** and **OutputStream** are abstract

# Input streams

- `ByteArrayInputStream`
  - reads from a buffer in memory
- `StringArrayInputStream`
  - converts a string to an input stream
- `FileInputStream`
  - reads from a file
- `PipedInputStream`
  - "reading" end of a pipe
  - for data passing between threads
- `SequenceInputStream`
  - concatenation of several streams
- all of them has only basic `read()` methods
  - reading by bytes



# Output streams

- `ByteArrayOutputStream`
  - writes to a buffer in memory
- `FileOutputStream`
  - writes to a file
- `PipedOutputStream`
  - "writing" end of a pipe
  - for data passing between threads
- **no** `StringArrayOutputStream`
  - `ByteArrayOutputStream` can be used
- all of them has only basic `write()` methods
  - writing by bytes

# Filters

- `FilterInputStream`
- `FilterOutputStream`
- **abstract classes**
  - many children
- via filters, further functionality is added to the basic streams
  - a filter receives another stream as a parameter of the constructor
  - data are read/written through the filter
- basic streams are almost always used via a stream
- several filters can be applied over a single stream

# Types of filters

- `DataOutputStream`
  - defines the write method for all primitive types
- `DataInputStream`
  - defines the read method for all primitive types
  - reads data in the same format as written by `DataOutputStream`
    - the format is platform independent
- `BufferedInputStream`
- `BufferedOutputStream`
  - do not add new read/write methods
  - I/O will be buffered
    - basic streams are not
  - capacity of the buffer can be specified

# Types of filters

- `LineNumberInputStream`
  - **information about current line**
- `PushbackInputStream`
  - **can return data back to the stream**

# Types of filters

- `PrintStream`
  - writes data in a human readable form
    - `DataOutputStream` writes data to be read by `DataInputStream`
  - defines methods `print()` and `println()` for "all" types
  - method `printf()`
    - as `printf` in C
  - method `flush()`
    - writes the buffer to an underlying stream
    - `PrintStream` is automatically buffered
    - `flush()` is called automatically when a new line is written
      - autoflush after each write can be set in a constructor
  - methods do not throw `IOException`
    - method `checkError()`

# Usage

- layering filters over basic I/O streams

```
DataInputStream di = new DataInputStream(  
    new BufferedInputStream (  
        new FileInputStream("file.txt")));  
int a = di.readInt();  
long b = di.readLong();
```

```
DataOutputStream ds = new DataOutputStream(  
    new BufferedOutputStream (  
        new FileOutputStream("file.txt")));  
ds.writeInt(100);  
ds.writeLong(1234L);
```

on `java.nio.file.Files`  
there are “shortcut”  
methods  
for opening

## Reader & Writer

# Overview

- since Java 1.1
- char-oriented I/O
  - char = 2 bytes
- streams are not deprecated
  - few of them are
- `Reader`
  - defines the `read` method for reading a char and array of chars
- `Writer`
  - defines the `write` method for writing a char and array of chars
- `Reader` and `Writer` – **abstract classes**
- `InputStreamReader`, `OutputStreamWriter`
  - creating `Reader/Writer` from a stream



# Types of I/O

- similar to streams

InputStream	Reader InputStreamReader
OutputStream	Writer OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
-	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

# Filters

- again similar to streams

FilterInputStream	FilterReader
FilterOutputStream	FilterWriter
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
PushbackInputStream	PushbackReader

# Exception management

# Exceptions

- almost “everything” in java.io throws IOException
  - extends Exception
  - needs to be caught/declared

# File copy

```
InputStream is;
OutputStream os;
try {
    is = new FileInputStream(finNm);
    os = new FileOutputStream(foutNm);
    int c;
    while ((c = is.read()) != -1) {
        os.write(c);
    }
    os.close();
    is.close();
} catch (IOException ex) {
    System.out.println("Exception occurred");
}
```

- Is this code OK?

# File copy

```
InputStream is;
OutputStream os;
try {
    is = new FileInputStream(finNm);
    os = new FileOutputStream(foutNm);
    int c;
    while ((c = is.read()) != -1) {
        os.write(c);
    }
    os.close();
    is.close();
} catch (IOException ex) {
    System.out.println("Exception occurred");
}
```

- Is this code OK?

**NO**

# Exceptions

- streams and readers/writers implement `AutoCloseable`
  - always use *try with resources*

```
try (InputStream is = new FileInputStream(finNm);
     OutputStream os = new FileOutputStream(foutNm)) {
    int c;
    while ((c = is.read()) != -1) {
        os.write(c);
    }
} catch (IOException ex) {
    System.out.println("Exception occurred");
}
```

# RandomAccessFile



# Overview

- reading and writing records from/to files
- movement over the file
- outside hierarchy of streams
- implements the interfaces `DataInput` and `DataOutput`
  - these interfaces are implemented by `DataInputStream` resp. `DataOutputStream`
  - methods `read` and `write` for primitive types
- opens the file for either reading only or reading and writing
  - the constructor parameter
    - "r" or "rw"

## NIO

# Overview

- “new I/O”
- since JDK1.4
- better performance
  - closer to structures of I/O in OS
- classes from `java.io` (stream and reader/writer) reimplemented by classes from `java.nio`
- defines *channels* and *buffers*
  - communication with a channel is by buffer only
- `FileInputStream`, `FileOutputStream` and `RandomAccessFile`
  - **method** `FileChannel getChannel()`
  - **since Java 7 also** `FileChannel.open(Path....)`
- `java.nio.channels.Channels`
  - methods for creation of Readers and Writers from channels

# Usage

- `java.nio.ByteBuffer`
  - only possibility for communication with a channel

```
FileChannel fc =  
    new FileOutputStream("data.txt").getChannel();  
fc.write(ByteBuffer.wrap("Some text ".getBytes()));  
fc.close();
```

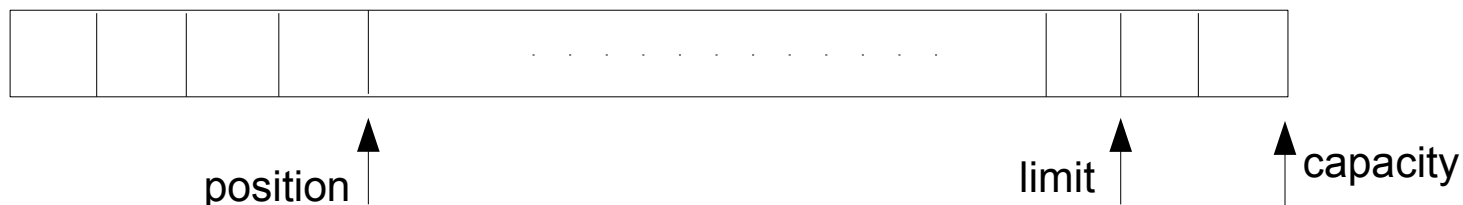
```
fc = new FileInputStream("data.txt").getChannel();  
ByteBuffer buff = ByteBuffer.allocate(1024);  
fc.read(buff);  
buff.flip();  
while(buff.hasRemaining())  
    System.out.print((char)buff.get());
```

# Buffer creation

- `ByteBuffer.wrap(byte[] b)`
  - static method
  - creates a buffer from an array of bytes
  - buffer is interconnected with the array
  - buffer capacity = `b.length`
- `ByteBuffer.allocate(int capacity)`
  - static method
  - allocates an empty buffer with specified capacity
- `ByteBuffer.allocateDirect(int capacity)`
  - static method
  - allocated buffer is “more” tied with OS
    - usage of the buffer should be faster
    - depends on OS

# Buffer

- capacity
  - how many elements buffer contains
  - cannot be increased
- limit
  - index of the first element that will not be read or written
  - cannot be bigger than capacity
- position
  - index of the first element that will be written or read on a following operation
  - cannot be bigger than limit



# Buffer: methods

- flip()
  - sets the limit to the current position and
  - sets the position to 0
- clear()
  - sets the limit to the capacity and
  - sets the position to 0
- mark()
  - sets the mark to the current position
- reset()
  - sets the position to the mark
  - does not remove the mark
- rewind()
  - sets the position to 0 and removes the mark

# Copying between channels

- methods `transferTo()` and `transferFrom()`

```
public static void main(String[] args) throws
                                Exception {
    FileChannel
        in = new FileInputStream(args[0]).getChannel(),
        out = new FileOutputStream(args[1]).getChannel();

    in.transferTo(0, in.size(), out);

    // or:
    // out.transferFrom(in, 0, in.size());
}
```



# Using buffer

- views on buffers
- reading and writing primitive types
- methods on the ByteBuffer
  - asCharBuffer()
  - asDoubleBuffer()
  - asFloatBuffer()
  - asIntBuffer()
  - asLongBuffer()

```
ByteBuffer bb = ByteBuffer.allocate(1024);  
bb.asIntBuffer().put(1234);  
System.out.println(bb.getInt());
```

# Endian

- by default the `ByteBuffer` uses *big endian*
- can be changed to *little endian*
  - method `order(ByteOrder b)`
  - the class `ByteOrder` has two static attributes of the type `ByteOrder`
    - `BIG_ENDIAN`
    - `LITTLE_ENDIAN`

# Files mapped to the memory

- accessing a file like an array in memory
- method on a channel
  - MappedByteBuffer map()

```
public class LargeMappedFiles {
    static int length = 0x8FFFFFFF; // 128 Mb
    public static void main(String[] args) throws Exception {
        MappedByteBuffer out =
            new RandomAccessFile("test.dat", "rw").getChannel()
                .map(FileChannel.MapMode.READ_WRITE, 0, length);
        for(int i = 0; i < length; i++)
            out.put((byte) 'x');

        for(int i = length/2; i < length/2 + 6; i++)
            System.out.print((char)out.get(i));
    }
}
```

# File locking

```
FileOutputStream fos = new
FileOutputStream("file.txt");
FileLock fl = fos.getChannel().tryLock();
if (fl != null) {
    System.out.println("File locked.");
    Thread.sleep(100);
    fl.release();
    System.out.println("File unlocked");
}
fos.close();
```

- exact behavior depends on OS
- only a part of file can be locked
- lock() – waits until a file is locked
- tryLock() – does not wait

... back to Path/Files

# Opening files

- methods of Files
  - `BufferedReader newBufferedReader(Path p, Charset cs)`
  - `BufferedWriter newBufferedWriter(Path p, Charset cs, OpenOption... opts)`
  - `InputStream newInputStream(Path p, OpenOption... opts)`
  - `OutputStream newOutputStream(Path p, OpenOption... opts)`
  - `SeekableByteChannel newByteChannel(Path p, OpenOption... opts)`
  - `DirectoryStream<Path> newDirectoryStream(Path dir)`
  - ...

## Console

# Console

- access to the char console
  - not always available
- `System.console()`
  - obtaining the console
- `Console printf(String format, Object... args)`
  - as `printf()` in C
- `String readLine()`
  - returns a line (without the new line char at the end)
- `char[] readPassword()`
  - returns a line (without the new line char at the end)
  - typed characters are not shown
- `Reader reader()`
- `PrintWriter writer()`
  - returns reader/writer associated with the console



I/O

# Compression

# Overview

- package `java.util.zip`
- compression via filters
  - `FilterInputStream` and `FilterOutputStream`
- `CheckedInputStream`, `CheckedOutputStream`
  - provides check-sums of read/written data
- `InflaterInputStream`, `DeflaterOutputStream`
  - basic classes for compression and decompression
- `GZIPInputStream`, `GZIPOutputStream`
  - compression in the GZIP format
- `ZipInputStream`, `ZipOutputStream`
  - compression in the ZIP format

# GZIP

- compression of a single file
- compatible with the UNIX programs gzip and gunzip

```
BufferedInputStream in = new BufferedInputStream(  
    new FileInputStream(args[0]));  
BufferedOutputStream out = new  
    BufferedOutputStream(  
        new GZIPOutputStream(  
            new FileOutputStream("test.gz")));  
int c;  
while((c = in.read()) != -1)  
    out.write(c);  
in.close();  
out.close();
```

# ZIP

- compression of multiple files into a single archive
- compatible with ZIP programs
- creating an archive
  - `ZipOutputStream`
  - the method `putZipEntry(ZipEntry ze)`
    - next file to the archive
  - the class `ZipEntry`
    - name of the file
    - information about the file (size before/after compression, comment, check-sum,...)
- reading from an archive
  - `ZipInputStream`
    - the method `getNextEntry()`
  - `ZipFile`
    - the method `entries()` - returns `Enumeration`

# ZIP

```
ZipOutputStream zos = new ZipOutputStream(  
    new BufferedOutputStream(  
        new FileOutputStream("test.zip")));  
zos.setComment("Test ZIP");  
for(int i = 0; i < args.length; i++) {  
    System.out.println("Storing a file: " + args[i]);  
    BufferedInputStream in = new BufferedInputStream(  
        new FileInputStream(args[i]));  
    zos.putNextEntry(new ZipEntry(args[i]));  
    int c;  
    while((c = in.read()) != -1)  
        zos.write(c);  
    in.close();  
}  
zos.close();
```



Slides version J07.en.2020.01

This slides are licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).