01: Introduction

Network Oriented Software

Stefan Huber
`www.sthu.org`

Dept. for Information Technologies and Digitalisation
FH Salzburg

Summer 2023

Section 1

Course formalities

Outline of the course

The course network-oriented software engineering pursues two goals:

- Introduction to the Java programming language (and ecosystem)
- Distributed software systems

A rough outline:

I Introduction into the Java programming language

  1 Java as an interpreted, strongly typed, imperative programming language
  2 Object orientation in Java

II Concurrency in Java

  3 Threads and processes, synchronization

III Distributed software systems

  4 Network programming basics, sockets and client-server model
  5 RMI and middleware
  6 SOA, web services and REST

Prerequisites

We assume that you have working knowledge from previous courses in the following fields:

- Programming in a strongly-typed, imperative programming language, like C or C++.
- Object oriented programming paradigms, like C++ or C#.
- Basics in computer networks and the TCP/IP protocol suite.
- Basics in the construction of operating systems.

Course organization and grading

Blocking into lecture parts and lab parts:

Block 1:	2 VO	+	(2 LB)
Block 2:	2 VO	+	2 LB
Block 3:	2 VO	+	2 LB
Block 4:	2 VO	+	2 LB
Block 5:	2 VO	+	2 LB
Block 6:	2 VO	+	2 LB
Block 7:		+	2 LB
Block 8:	2 Ex.		

Lecture part:

- Stefan Huber

Lab parts:

- Josef Lettner
- Andreas Klinger

Grading:

- 50 % final exam
- 50 % lab performance

Passing and compensating

If your gradings sum up to at least 50 % then you passed.

Otherwise one of three cases applies:

- Only the exam is below 50 %.
  You repeat the exam.
- Only the lab is below 50 %.
  You get a compensation task. However, there is no compensation if you have less than 40 % at the lab.[1] Also, you cannot compensate a failed lab by improving at the exam.
- Each of them is individually below 50 %.
  Then both of the above cases apply.

The dates for exams and compensation are given by fhsys:

- The exam is on this precise date.
- The compensation task is given at this precise date.

[1] Because at some point you would need to compensate most of the entire course.

Crediting

Crediting of the course is possible under these conditions:

- You have passed one or more university-level courses that cover the topics of this course.
- You have to obtain 80 % of the points on an crediting exam interview.

Online course material and organization

Moodle courses and enrollment keys:

- ITBB4NWSIL: `tarsier2oyster`
- ITSB4NWSIL: `otter8gaur`
- WINB4NWSIL: `Nh(4Sw#`

Moodle will be used for material, quizzes and non-realtime interaction.

Teams teams and enrollment key:

- TEAM-ITS-NOS-2023-ss-WIN: `ew12u1o`
- TEAM-ITS-NOS-2023-ss-ITSB: `ajury7z`
- TEAM-ITS-NOS-2023-ss-ITSBB: `8si7zki`

Teams will be used for realtime interaction, like online lectures and chat.

Literature on Java

- Horstmann, Core Java Volume I and II. [HC18; Hor18]
  Mostly good quality and easy to read. Very verbose with background information. Also useful for beginners and often compares Java to C++.
- Herbert Schildt, Java complete reference [Sch18].
  Well written, more compact than Core Java. A good book to lookup stuff.
- And there is more: [EF18; Blo17; Har13]
- There are a couple of Java books in our library; mostly older[2] editions at the moment.

2 You do not require the latest editions of literature. Presumably the course will essentially rely on Java 9. Debian Buster ships Java 11.

Literature on Distributed Systems

- van Steen and Tanenbaum, Distributed Systems [vT23]
- Huber, lecture notes on Distributed Software Architectures [Hub19]
- Stevens, UNIX Network Programming [Ste]

More resources

Trust the specification! Question random code snippets on the web!

- Use the Java API Specification: `https://docs.oracle.com/en/java/javase/19/docs/api/index.html`.
- There is also The Java Language Specification: `https://docs.oracle.com/javase/specs/jls/se19/html/index.html`.

Web resources:

- `https://en.wikipedia.org/wiki/Java_(programming_language)`
- `https://www.w3schools.com/java/default.asp`

Section 2

Basics of the Java language and ecosystem

The rise of Java

Software industry in the mid 90s:

- The industry is dominated by C.
- OOP was known to academia for two decades. The famous book *Design Patterns* by the Gang of Four [Gam+94] was published in 1994.
- Software industry switches more and more to C++ and OOP is about to become the dominant programming methodology. Microsoft MFC is first released in 1992.
- The WWW was in its childhood but business shows massive excitement, eventually leading to the dot-com bubble between 1994 and 2000.
- Memory leaks and bad software design is omnipresent. Security is a mess.

The rise of Java

Software industry in the mid 90s:

- The industry is dominated by C.
- OOP was known to academia for two decades. The famous book *Design Patterns* by the Gang of Four [Gam+94] was published in 1994.
- Software industry switches more and more to C++ and OOP is about to become the dominant programming methodology. Microsoft MFC is first released in 1992.
- The WWW was in its childhood but business shows massive excitement, eventually leading to the dot-com bubble between 1994 and 2000.
- Memory leaks and bad software design is omnipresent. Security is a mess.

Java's first release was in 1996:

- Huge excitement. Buzzwords: clean, concise, easy, purely object-oriented, platform independent, secure, internet ready, ...
- Even mainstream media, like The New York Times and The Washington Post, mentioned it.

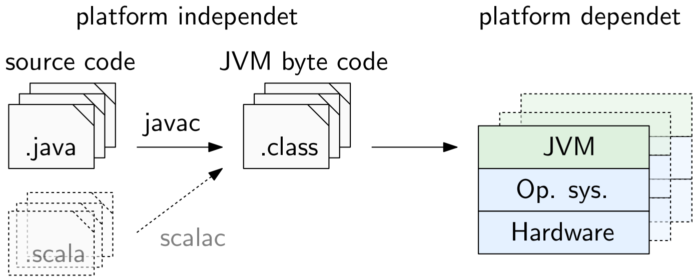
Java is platform independent

Java programs are interpreted by the Java Virtual Machine (JVM):

- Java originally targeted at embedded devices, like TV switch boxes. A variety of different platforms had to be supported.
- Java runs wherever you port the JVM to. Java is portable.

```
# Compile the Java code in HelloWorld.java to Java bytecode in HelloWorld.class
% javac HelloWorld.java
# Run HelloWorld.class by invoking the JVM
% java HelloWorld
Hello World
```

Java is platform independent



- The JVM is an abstract computing machine. It hides the details of the processor, like endianess, size of registers, details of the floating-point unit and so on. It is architecture neutral.
- The JVM comes at additional runtime costs. Just-In-Time (JIT) compilers reduce those costs significantly by translating the hotspots (often executed code snippets) on-the-fly and in-memory into native machine code.

Java is object oriented

- Java is purely object oriented.
- In Java, all functions are class members and they are called methods.
- Even `main()` is a method, as in C#, but unlike in C and C++.

```java
/**
 * The hello world class.
 */
public class HelloWorld {
    /** This is the program's entry point. */
    public static void main(String args[]) {
        // Purely object oriented: The System class contains an object out,
        // which provides a method println() to print-a-line.
        System.out.println("Hello world");
    }
}
```

A first step into the Java language

- Syntax close to the C-family, like C++, Objective-C or C#:
  Curly braces `{}` for blocks, whitespace is ignored, single-line comments *//* and multi-line comments */**/*, double quotes delimit strings.

```java
public class HelloWorld {
    /** This is the program's entry point. */
    public static void main(String args[]) {
        // Purely object oriented: The System class contains an object out,
        // which provides a method println() to print-a-line.
        System.out.println("Hello world");
    }
}
```

- The class `HelloWorld` must reside in the file `HelloWorld.java`.
  - Each class has its own file. Case sensitive names, also for the filenames.
  - When the JRE looks for the bytecode of the class `A` then it looks for the class file `A.class`.
- If a class contains a `public static void main()` method then the class can be executed as a program by the Java interpreter.

Java is an environment

- Java comes with a whole environment, the Java Runtime Environment (JRE).
- The JRE contains the JVM but also the Java standard library with thousands[3] of classes.
- For development you need a Java Development Kit (JDK).
  - There are different editions, we use the Standard Edition (SE).
  - Most current version is Java SE 19. Debian Buster ships version 11, which is fine.

```
# Debian Buster ships a Java 11 JRE (Java Runtime Environment):
% java -version
openjdk version "11.0.18" 2023-01-17
OpenJDK Runtime Environment (build 11.0.18+10-post-Debian-1deb11u1)
OpenJDK 64-Bit Server VM (build 11.0.18+10-post-Debian-1deb11u1, mixed mode, sharing)
```

3 Java 12 comes with 4433 classes, Java 5.0 with 3279 classes, Java 1.3.1 with 1840 classes.

Java ships a documentation generator

- Javadoc comments start with */*** and allow to generate source code documentation.
  - These are used to comment files, classes and its members and similar entities.
- Javadoc takes those comments and generates an HTML documentation of those entities.

```java
/** The obligatory hello world.
 *
 * This is a hello world demo that not only demonstrates the main() method,
 * println() and strings, but also javadoc comments.
 *
 * @author Stefan Huber <stefan.huber@fh-salzburg.ac.at>
 */

/**
 * The hello world class.
 */
public class HelloWorld {
```

Data types

Java is strongly typed:

- Every variable has a declared type.
- If a value is assigned to a variable, or an argument is passed to a function, the types are checked.

There are two categories of types:

- Primitive types
- Non-primitive types, e.g., class types, arrays, enums et cetera.

Primitive types

```java
/** The Java language knows exactly eight primitive types. */
class PrimitiveTypeDemo {
    public static void main(String args[]) {
        byte  a = -0x0a;       // A signed 8-bit integer. Hex literals as in C.
        short b = 0b1010;      // A signed 16-bit integer. Binary literals since Java 7.
        int   c = 1_000_000;   // A signed 32-bit integer. Underscores since Java 7.
        long  d = 42L;         // A signed 64-bit integer. Long literals by suffix L.

        float  e = 3.1416f;    // IEEE 754 single-precision floating-point number.
        double f = 1.256e-6;   // IEEE 754 double-precision floating-point number.

        char g     = '€';      // A character. (Actually, a UTF-16 code unit.)
        boolean h = true;      // There is only true or false, and no third!

        System.out.printf("%d %d %d %d %f %f %c %b\n", a, b, c, d, e, f, g, h);
    }
}
```

- `0x12`, `1.41f`, `3.14`, `'A'`, `false`, `"Hi"` and so on are called literals, and they have a type.
- Note that the C and C++ standards do not define the sizes of integer data types! This is why C99 introduced platform-independent types like `int32_t`.

Operators

The following operators work (almost) as in C:

- Arithmetic: `+ - * / %` and its assignment counterparts `+= -= *= /= %=`
- Incremental: `x++ ++x x-- --x`
- Relational: `== != <= >= < >` Produce a `bool`
- Logical: `! && ||` Take and produce `bool`
- Bitwise: `& | ^ ~ << >>` and its assignment counterparts Java knows a zero-padding `>>>`

```java
class OperatorsDemo {
    public static void main(String args[]) {
        assert 13 + 5 == 18;                   // Call java with -ea to enable assertions
        assert 5 / 13 == 0;                    // Integer division
        assert 1.0 / 2.0 == 0.5;               // Be careful: Floating-point equality!
        assert 1.0f + 0.00000001f == 1.0f;     // Numerical errors break exact equality

        //assert 3 >= 2 && 7;                  // Syntax error: 7 is not boolean
        //assert 3 == true;                    // Syntax error: cannot compare int and bool
        assert false;                          // We expect that to fail
    }
}
```

Operator precedence

If no parenthesis are used, like `(x && y)== z`, then the operator precedence defines which operators bind tighter. From highest to lowest:

Access and call	`[] . ()`	left to right
Unary	`! ~ ++ -- + - ()(cast)new`	right to left
Binary	`* / \%`	left to right
	`+ -`	left to right
	`<< >> >>>`	left to right
	`< <= > >= instanceof`	left to right
	`== !=`	left to right
	`&`	left to right
	`^`	left to right
	`|`	left to right
	`&&`	left to right
	`||`	left to right
Ternary	`?:`	right to left
Assignment	`= += -= *= /= &= |= ^= <<= >>= >>>=`	right to left

Math and conversion

The class `Math` contains a couple of mathematical functions:

```java
        System.out.println(Math.log(Math.E));            // Prints 1.0
        System.out.println(Math.atan(0.5 * Math.PI));   // Should be 1.0
        // abs, min, max, round, floor, ceil, sqrt, pow, exp, log, log10, ...
        // sin, cos, tan, asin, acos, atan, atan2, sinh, ...
```

A type cast performs an explicit type conversion: `int x = (int)0.5`

- Those conversions are done automatically, but the dotted lines may come at numerical loss:

$$\begin{tikzcd}
\texttt{byte} \arrow[r] & \texttt{short} \arrow[r] & \texttt{int} \arrow[r] \arrow[dr, dotted] \arrow[drr] & \texttt{long} \arrow[d, dotted] \arrow[dr, dotted] & \\
& \texttt{char} \arrow[ur] & & \texttt{float} \arrow[r] & \texttt{double}
\end{tikzcd}$$

- Numerical expressions with mixed number types cause type promotions:
  - `3 * 2.0` will carry out a `double`-multiplication as `3` is automatically promoted to `3.0`.
  - `int x = 3 * 2.0` is illegal in Java and would require a cast.

Strings

- A string is a sequence of Unicode characters.
- A string is actually an instance of the class `String`.
  - This class provides more than 50 methods.
  - We expect that you consult the API specification when required.
- Java has no operator overloading, unlike in C++.
  - However, the Java *language* explicitly defines + on strings as concatenation.
- Strings are immutable; they cannot be changed.

```java
class StringDemo {
    public static void main(String args[]) {
        String str = "Alan Turing";
        System.out.println(str.length());           // 11
        System.out.println(str.substring(2, 7));    // "an Tu"
        System.out.println(str + " rocks!");        // "Alan Turing rocks!"
    }
}
```

Equal versus identical

- The operator == on objects tests whether they are the same instances.
- This is different from testing whether they are equivalent w.r.t. their data or state.

```java
class StringEqualityDemo {
    public static void main(String args[]) {
        String str1 = "Hello";
        String str2 = str1 + "";

        System.out.println(str1 != str2);          // Not identical, different instances
        System.out.println(str1.equals(str2));     // But two equal instances
    }
}
```

Empty versus null

- When a variable is `null` then it refers to no object.
- Of course, an empty string is not the same as a null object.

```java
class EmptyNullDemo {
    public static void main(String args[]) {
        String notmuch = "";
        String notatall = null;

        System.out.println(notmuch.length());     // 0
        System.out.println(notatall == null);     // true
        //System.out.println(notatall.length()); // Runtime error: NullPointerException

        // Lazy evaluation: As notatall != null is already false the second
        // operand of && is not evaluated. Similar with ||.
        System.out.println(notatall != null && notatall.length() > 0);   // false
        System.out.println(notatall == null || notatall.length() == 0);  // true
    }
}
```

Variable scopes and control flow

- Control flow structures are essentially those of the C-family.
- The scope (life range) of a variable is the surrounding `{}`-block.

```
        System.out.println("Fibonacci series …");  // Btw., also code is in unicode
        int a = 1, b = 0;
        while (b < 100) {                     // Each {}-block defines a scope for variables
            System.out.print(b);
            System.out.print(" ");

            int c = a + b;                    // A variable lives only within its scope
            b = a;
            a = c;
        }                                     // c does not live outside this block anymore
        System.out.println();

        int n = 1;
        int fac = 5;
        for (int i = 1; i <= fac; ++i)
            n *= i;
        System.out.printf("%d factorial is %d\n", fac, n);
```

Control flow

```java
// Execute as long as condition is true
while (bool_cond) {
    // bool_cond must be of type bool:
    // if (7) {} is illegal in Java!
    if (bool_cond) {
        continue;    // Goto start of inner loop
    } else {
        break;       // Exit inner loop
    }
}

for (init_expr; bool_cond; post_expr) {
    // Equivalent to {
    //      init_expr;
    //      while (bool_cond) {
    //          { block }
    //          post_expr;
    //      }
    // }  The scope of init_expr is the loop
}
```

```java
do {    // Do not repeat when
        // condtion becomes false.
} while (bool_cond);

switch (choice) {
    // byte, char, int, …
    case constant_integer:
        break;
    case enum_constant:
        break;
    // Since Java 7
    case "constant_string":
        break;
    default:
        break;
}

// There is actually a "tame goto":
// A break to a labled block.
```

Arrays

Arrays in Java are similar to dynamic C++ arrays.

- The class `java.util.Arrays` contains many helper methods like string conversion and comparison.

```java
import java.util.*;                                  // We need java.util.Arrays

class ArraysDemo {
    public static void main(String args[]) {
        int[] a = new int[6];                        // Uninitialized
        for (int i=0; i < a.length; ++i)             // a.length gives the array size
            a[i] = i * i;
        System.out.println(Arrays.toString(a));

        int[] b = {0, 1, 4, 9, 16, 25};
        System.out.println(Arrays.equals(a, b));

        int sum = 0;
        for (int elem : b)                           // Java 5 has a foreach loop, as C++11
            sum += elem;
        System.out.println(sum);
    }
}
```

Multi-dimensional arrays

```java
import java.util.*;

class MultidimArrayDemo {
    public static void main(String args[]) {
        // Initialized with all zeros
        int[][] tictactoe = new int[3][3];
        System.out.println(Arrays.deepToString(tictactoe));

        // A ragged array: A multi-dim array is an array of arrays, unlike in C++.
        int[][] hashtable = {{ 1, 2, 3, 4},
                             { 10, 20},
                             { 100, 200, 300}};
        System.out.println(Arrays.deepToString(hashtable));
        System.out.println("hashtable[0][3] = " + hashtable[0][3]);
        for (int[] row : hashtable)
            System.out.println("Row = " + Arrays.toString(row));
    }
}
```

Java is robust and secure

Java has been designed to be robust and secure:

- Java is memory-managed so there are no use-after-free or double-free memory errors because a garbage collector frees memory.
- Java has no concept of a pointer so there is no invalid pointer dereference error.
- Array access is checked at runtime to eliminate out-of-bound access. This eliminates buffer overflow attacks.
- Java comes with security policies.

Personal opinion

The above reasons, plus Java's simplicity, plus its enterprise features[4], are probably main reasons why Java became very popular for business and finance software, and replaced PL/I and COBOL.

[4] Java EE, EJB, ...

Coding conventions

All Java code follows very similar coding conventions:

- CamelCase for identifiers, like variables, classes and so on.
- Classes and other types start with upper case, all others with lower case.[5]
- Hence, Java source files have CamelCase filenames, too.

Code Style

You shall follow these conventions!

[5] So-called interfaces, which are like C++ classes with pure virtual functions only, also start with upper case letters.

Preparations for the lab

For the lab assignment we need the following software:

- Oracle Java SE JDK
- Git
- Editor

Debian Linux: `apt install default-jdk git`

Windows:

- `https://git-scm.com/download/win`
- `https://www.oracle.com/java/technologies/javase-downloads.html`
  - If you are a Windows user, check that `javac` is in the `PATH` environment variable.[6]
- `https://notepad-plus-plus.org/` or any other editor
  - Later you may find the Eclipse IDE, IntelliJ IDEA, NetBeans or VSCode handy.

[6] See `https://docs.oracle.com/javase/9/install/installation-jdk-and-jre-microsoft-windows-platforms.htm`

Section 3

Lab: Formalities and grading

Coding assignments and Moodle quizzes

There will be coding assignments for homework

- Assignments are published on the Moodle course.
- Every student develops in their own git repository to send in his solutions.
- Solutions are presented and discussed in the next lab.

Major-minor code review:

- Major: Code solutions that are reviewed in more detail with feedback.
- Minor: A quick, superficial check of your submission.

10 minute Moodle quizzes take place at the beginning of each lab.

- Test quiz today to test the setup.

Grading

The lab performance is graded as follows:

- 40 % Moodle quizzes
  Top 5 of 6 are counted. That is, worst result of six is ignored. If you miss a lab then this quizz is ignored.
- 40 % major code review
- 20 % minor code review

Section 4

Lab 1: Java and git warm ups

Fast forward: Setting up your git config

Check that you have your name and e-mail address set in git. Those will be part of git commit messages later.

```
$ git config --get user.name
YOUR NAME AS FIRSTNAME LASTNAME
$ git config --get user.email
YOUR MAIL ADDRESS
```

Those can actually be set system-wide, globally (for your user) and per repository. If not set already and you are unsure then simply set:

```
# Of course, use your name and e-mail address below
$ git config --global user.name "firstname lastname"
$ git config --global user.email "user@fh-salzburg.ac.at"
```

Fast forward: Setting up your solutions repo

1. Open `https://its-git.fh-salzburg.ac.at/` in a browser.
2. Login with your FH-User and click on the button New project.
3. Choose project name of the form `nos-ss2023-firstname-secondname` and replace “firstname” and “secondname” accordingly.
4. Leave visibility level at Private.
5. Click on the button Create project.

6. Right of the SSH dropdown button the URL of this repo is shown and has the form `git@its-git.fh-salzburg.ac.at:username/nos-ss2023-firstname-secondname.git`. Copy it.
7. Open `https://etherpad.wikimedia.org/p/nos-ss2023` and paste the URL under the right heading.

8. On its-git switch on the left menu to Settings/Members, select the lab instructor as member, choose Maintainer as role, and click on Add to project.

Fast forward: Setting up your access

Create SSH keys if you do not have some already.

1. Open a terminal (like `git bash`) and enter `ssh-keygen`
   - When you asked about the path, press enter.
   - When you asked about the password, press enter (or enter a password).
2. When done enter `cat ~/.ssh/id_rsa.pub` and copy the content.
3. Switch to the browser with its-git, open the user menu on the right top and click on Settings.
4. Choose SSH Keys on the left menu.
5. Paste your key and click Add key.

Now we are ready to grab a copy of the repository:

- Copy your project URL again.
- Switch to the terminal and enter `git clone YOURPROJECTURL`.

Fast forward: A quick demo on uploading data

Open the terminal and switch to your repository.

Enter these commands

```
git status                # Gives some status info, like 'No commits yet'
echo "My repository" > README.md
git add README.md
git commit -m "Add README"
git push
```

Open a web browser with its-git and check that your commit is online.

Warning

You need to push your solutions until the deadline! Check on the web whether your solutions are online!

A quick intro into VCS

Version control systems (VCS):

- Also known as revision control or source (code) control.
- Solves two problems:
  - Keeping track of the evolution of files, in particular source code files.
    Kills zip files that mark versions of the source code.
  - Supports and enables collaboration in a controlled fashion.
    Kills careful (often manual) merges of source changes into a central file server directory. Kills file boundaries as responsibility boundaries for team members.

```
# Switch to some working directory first...
git init demo-01                         # Create repository demo-01
cd demo-01                               # Switch to directory demo-01
echo "test" > test.txt                   # Create file with some content
git add test.txt                         # Add file to repo(sitory)
git commit -v -m "My first commit"       # Commit file to repo
echo "another line" >> test.txt          # Modify the file
git status                               # Show status of the repo
git diff                                 # Show uncommited changes
```

A quick intro into DVCS

Two kinds of version control systems (VCS):

Centralized	The older, traditional approach. Basically like a central database or specialized file server that manages the commits and the history. Examples: CVS, subversion, TFS (TFVC), ...
Distributed	The modern approach. Every repository is full-fledged and autonomous, and therefore driven by the open source community. Examples: git, mercurial, BitKeeper, ...

Distributed VCS (DVCS), and in particular git, became the industry standard in the past decade.

Resources on git

- Git comes with extensive documentation, e.g., `git --help` and `git diff --help` print help and show the man pages of the respective commands, respectively.
- The Pro Git book is online: `https://git-scm.com/book/en/v2`
- A git cheat sheet: `https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf`
- A view videos: `https://git-scm.com/doc`

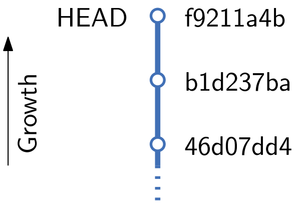
Windows users often like GitExtensions as GUI:

- `https://gitextensions.github.io/`

Local work with git

Initializing git

- You can turn any directory into a git repo by `git init`
- It creates a directory `.git` with the git object database and more



A commit is a change to files.

- A commit refers to a parent commit. The commits are therefore linked together.[7]
  - Often like a linked list.
  - But a merge commit has two parents.
- A commit contains additional meta information (author, date) and a commit message.
- A commit is identified by its SHA1 hash.
  - A unique prefix suffices to identify a commit.

[7] It is a so-called directed acyclic graph.

A few thoughts on a commit

- Commits should be logically coherent, like a single bug fix or a single functionality.
- Commits can be very small if the change makes sense by itself. Commits can be very large and touch many files, e.g., when refactoring names.
- Write meaningful commit messages. If the message tells many stories than you should probably split the commit up. Linus Torvalds on commit messages, see [Tor].

If you violate these principles then merging branches will become a hell later on.

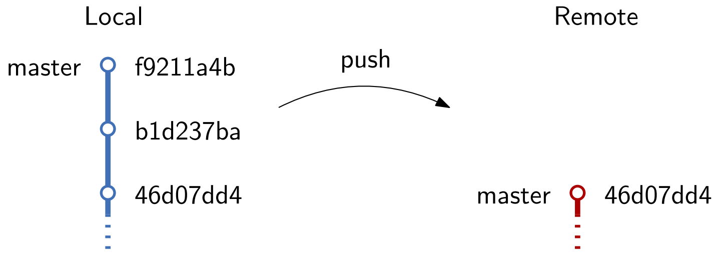
Exercise

1. Create a directory `git-demo-02` and add a `HelloWorld.java` that prints "Hello world".
2. Initialize a git repository in the directory, add the source file and commit it.
3. Change the source file in some way. Then play with these commands:
   - `git log`, `git status`, `git diff`
4. Add a file `README.txt` with some content. Commit using
   - `git commit -av`
5. Play again with these commands:
   - `git log`, `git status`, `git diff`
6. Change both files in some way. Commit again using
   - `git commit -v HelloWorld.java`
7. Play again with these commands:
   - `git log`, `git status`, `git diff`

Distributed work with git

In a typical collaborative workflow there is a central git repository.

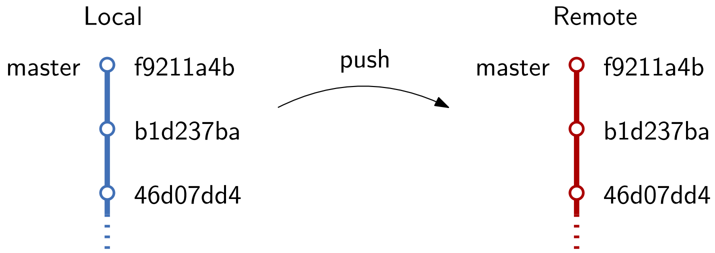
- The local repositories then synchronize with the central one.
- We start by cloning the remote repo with `git clone URL`.
- Then we pull changes from the remote side and push changes to the remote side by `git pull` and `git push`.



Distributed work with git

In a typical collaborative workflow there is a central git repository.

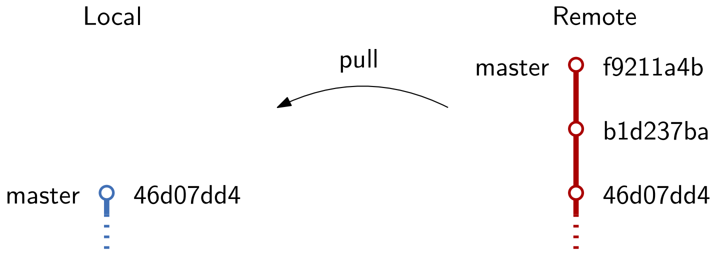
- The local repositories then synchronize with the central one.
- We start by cloning the remote repo with `git clone URL`.
- Then we pull changes from the remote side and push changes to the remote side by `git pull` and `git push`.



Distributed work with git

In a typical collaborative workflow there is a central git repository.

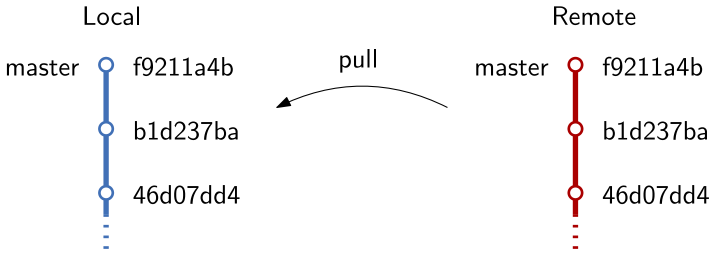
- The local repositories then synchronize with the central one.
- We start by cloning the remote repo with `git clone URL`.
- Then we pull changes from the remote side and push changes to the remote side by `git pull` and `git push`.



Distributed work with git

In a typical collaborative workflow there is a central git repository.

- The local repositories then synchronize with the central one.
- We start by cloning the remote repo with `git clone URL`.
- Then we pull changes from the remote side and push changes to the remote side by `git pull` and `git push`.



Your personal NOS git repository

Your repository must follow the following directory structure:

```
your-nos-git-repo/
   01/
     YourSolutionToAssignmentA.java
     YourSolutionToAssignmentB.java
     YourSolutionToAssignmentC.java
   02/
     YourSolutionToAssignmentA.java
   ...
```

That is, put all your solutions of unit 01 into a subdir 01. This is part of the grading.

You have to git-push your solution to the git server until the deadline!

References I

[Blo17] Joshua Bloch. *Effective Java*. 3rd ed. Addison-Wesley Professional, 2017, p. 412. ISBN: 978-0134685991.

[EF18] Benjamin J. Evans and David Flanagan. *Java in a Nutshell: A Desktop Quick Reference*. 7th ed. O'Reilly UK Ltd., 2018, p. 436. ISBN: 978-1492037255.

[Gam+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley, 1994, p. 395. ISBN: 0-201-63361-2.

[Har13] Elliotte Rusty Harold. *Java Network Programming: Developing Networked Applications*. 4th ed. O'Reilly and Associates, 2013, p. 500. ISBN: 978-1449357672.

[HC18] Cay S. Horstmann and Gary Cornell. *Core Java Volume I – Fundamentals*. 11th ed. Prentice Hall Press, 2018, p. 889. ISBN: 978-0135166307.

[Hor18] Cay S. Horstmann. *Core Java Volume II – Advanced Features*. 11th ed. Pearson Education, 2018, p. 1040. ISBN: 978-0135166314.

[Hub19] Stefan Huber. *Distributed Softate Architectures*. Lecture notes. 2019. URL: https://www.sthu.org/teaching/#lecture-notes.

References II

[Sch18] Herbert Schildt. *Java: The Complete Reference*. 11th ed. McGraw-Hill Education, 2018, p. 1248. ISBN: 978-1260440232.

[Ste] W. Richard Stevens. *UNIX Network Programming*. URL: http://www.unpbook.com/src.html.

[Tor] Linus Torvalds. *Contributing to subsurface*. URL: https://github.com/subsurface/subsurface/blob/master/README.md#contributing (visited on 02/27/2021).

[vT23] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. 4th ed. Jan. 2023. ISBN: 978-9081540636.

02: Object Orientation

Network Oriented Software

Stefan Huber
www.sthu.org

Dept. for Information Technologies and Digitalisation
FH Salzburg

Summer 2023

Section 1

Classes and Objects in Java

OOP in Java

We assume you know these four aspects of object-oriented programming (OOP):

- Abstraction
  Hiding details and complexities of inner workings, exposing simplicity
- Encapsulation
  Cohesion of data and code, data hiding, forming “has-a” relationships
- Inheritance
  Hierarchical subtyping, forming “is-a” relationships
- Polymorphism
  Type-depending behavior, overriding behavior in subtypes

Classes in Java

OOP languages organize code in classes.

- In Java, all code is in classes; it is purely OOP.
- Each `class A` is implemented in a source file `A.java`.[1]
- The keyword `class` followed by the class name defines a class.

```java
/** The class A is implemented in A.java */
class A {
}
```

[1] Well, there are also inner classes and static inner classes. And non-public classes can be legally defined without matching filename.

Class members

A class comprises

- Properties implemented as fields (member variables)
- Functionality implemented as methods (member functions)

```java
/** A natural person. */
class Person {
  /** The name of the person .*/
  private String name;

  /** Returns the name of the person. */
  public String getName() {
    return name;
  }
}
```

Each of them has one of four access specifiers that specify visibility:

private	can only be accessed within the class
protected	see later, (can be accessed from derived classes)
default	see later, (can be accessed from the same package)
public	can be accessed from everywhere

Object creation

An instance of a class is called object.

- In Java, the operator `new` creates class instances.

```java
Person p = new Person();
```

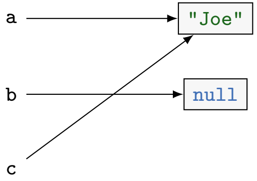
Variable versus object

The variable is *not* the object:

- We say that the variable binds to the object; it is an alias.
- Multiple variables may bind to the same object.

```java
Person a = new Person();
Person b = new Person();
Person c = a;                  // Binds to the same object as 'a'.
c.setName("Joe");              // The Person behind 'a' and 'c' is called "Joe" now
```

After the above code we have three variables bound to two objects.



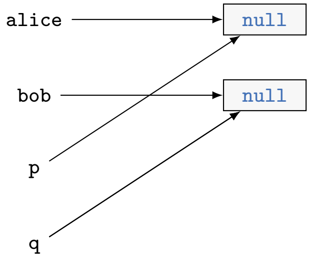
Parameter passing

Parameters behave like variables and they are passed by value.

- But think of the *variable per se* being passed by value, not the object it binds to!

```java
public void f(Person p, Person q) {
  // Both parameters now bind to the same object.
  p = q;
  // We change the object that p (and q) binds to.
  p.setName("Joe");
}

public void test() {
  Person alice = new Person();
  Person bob = new Person();
  f(alice, bob);
  // bob's name is now "Joe", but not alice's.
}
```



Remember

Java variables are more like C++ pointer variables, but without giving access to the actual addresses.

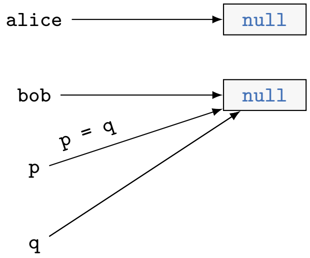
Parameter passing

Parameters behave like variables and they are passed by value.

- But think of the *variable per se* being passed by value, not the object it binds to!

```java
public void f(Person p, Person q) {
  // Both parameters now bind to the same object.
  p = q;
  // We change the object that p (and q) binds to.
  p.setName("Joe");
}

public void test() {
  Person alice = new Person();
  Person bob = new Person();
  f(alice, bob);
  // bob's name is now "Joe", but not alice's.
}
```



Remember

Java variables are more like C++ pointer variables, but without giving access to the actual addresses.

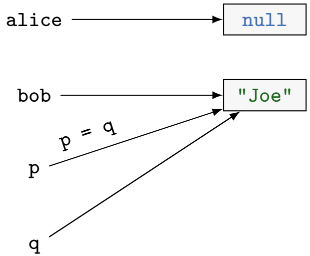
Parameter passing

Parameters behave like variables and they are passed by value.

- But think of the *variable per se* being passed by value, not the object it binds to!

```java
public void f(Person p, Person q) {
  // Both parameters now bind to the same object.
  p = q;
  // We change the object that p (and q) binds to.
  p.setName("Joe");
}

public void test() {
  Person alice = new Person();
  Person bob = new Person();
  f(alice, bob);
  // bob's name is now "Joe", but not alice's.
}
```



Remember

Java variables are more like C++ pointer variables, but without giving access to the actual addresses.

Inheritance

A class can extend (derive from, inherit from) one class, its superclass.

- In Java, the keyword `extends` is used.
- The derived class is also called subclass. The superclass is also called base class.

```java
class Student extends Person {
}
```

A class inherits all properties and functionality from its superclass.

- The access specifiers of the superclass are taken over.
- A class can access protected members of its superclass.

There is no multiple inheritance in Java.[2]

- Also, if a class does not extend any class then it implicitly extends the class `Object`.
- Conclusion: The class hierarchy in Java is a tree with `Object` at its root.

[2] But there are also interfaces, and they admit multiple inheritance.

Inheritance: an "is-a" relation

When `Student` extends `Person` then for the type system a `Student` *is a* `Person`.

- A `Person` variable can bind to a `Student` object.
- Hence, we can pass a `Student` argument for a `Person` parameter.
- The *Liskov substitution principle*[3] takes is-a verbatim.

```java
// A Student is a Person: A Person variable can bind to a Student object
Person p = new Student();
```

Static versus dynamic type[4]

- The variable `p` has the static type `Person`.
- But since it binds to a `Student` object it has the dynamic type `Student`.

3 This is the L of the SOLID principles of OOP.

4 Dynamic means "changing over time", i.e., an aspect at runtime. Static means "not changing over time", i.e., fixed at compile time.

Characteristics of objects and relationships between classes

Three characteristics define an object:

State	What properties does an object posses?
Behavior	What can an object do?
Identity	What distinguishes two objects from each other?

Three basic relationships exist between classes:

Inheritance	The is-a relationship between subtype and supertype.
Association	The has-a relationship between a class and its field.[5]
Dependency	The uses-a relationship when methods of one class modify or uses (instances of) other classes.

[5] Associations may be categorized into Aggregations and Compositions. In an aggregation the child can exist without the parent, i.e., there is an aggregation from lecture to students; destroying a lecture does not kill students. Aggregation is typical for shared ownership of the child. In a composition the child cannot exist without the parent, i.e., there is composition between a house and its rooms; there are no rooms without the house. Compositions means unique ownership.

Default field initialization

The following default initialization is applied:

Types	Literals
Booleans	`false`
Numerics	zero, e.g., `0`, `0l`, `0.0f`, `0.0`, `'\u0000'`
Objects	`null`

Code style

It is poor style to rely on default initialization:

- Explicitly initialize fields instead!

Constructors

The `new` operator for object creation calls a constructor:

- Like a method whose name is the class name and no return type.
- Primarily used to initialize the object's state.

```java
class Person {
  private String name;  // Initialized to null by default

  /** The default (no-argument) constructor of Person. */
  Person() {
    name = "";
  }

  /** Create a Person with given name. */
  Person(String name) {
    /* While 'name' is the parameter, 'this.name' is the field 'name' of the
    Person-instance 'this'. And 'this' is an implicit parameter to all methods. */
    this.name = name;
  }
}
```

More constructors

There can be more than one constructor:

- The no-argument constructor has zero parameters.
- Java knows method overloading. The overload resolution is based on argument types:

```java
// Since a String is passed as argument, the ctor Person(String) is deduced
Person p = new Person("Joe");
```

If no constructor is defined then there is an implicit no-argument constructor:

- It essentially does nothing.
- Hence, if there is at least one constructor then there is no implicit no-argument constructor.

Constructors calling constructors

- In the first statement of a constructor we can call another constructor of this class using `this`.
- This way, code duplication in constructors can be reduced. (C++ 11 knows that too.)

```java
class Person {
  private String name;
  private int age;

  Person() {
    this("");
  }
  Person(String name) {
    this(name, 0);
  }
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Constructor of the superclass

- In the first statement of a constructor, we can also call the constructor of a base class using `super`.

```java
/** A Student is a special Person. */
class Student extends Person {
  /** Matriculation number. */
  private int matnumber;

  /** Create a Student by its name and matriculation number. */
  Student(String name, int matnumber) {
    // Initialize the Person-part of Student by calling a Person-constructor.
    super(name);
    // Initialize non-Person-fields
    this.matnumber = matnumber;
  }
}
```

- If a class does not explicitly call a superclass constructor then the no-argument constructor of the superclass is implicitly called. If it does not have one – also no implicit one – the compiler reports an error.

Field initialization

In general, a field can be initialized in the following ways:

- At the field declaration.
- In an initialization block, but this is very uncommon.
- In a constructor.

```java
class Person {
  private String name = "";
  private int dayofbirth;

  { // Initialization block
    int ms = System.currentTimeMillis();
    dayofbirth = ms / 1000 / 60 / 60 / 24;
  }

  Person() {
  }

  Person(String name) {
    this.name = name;
  }

  Person(String name, int dateofbirth) {
    this.name = name;
    this.dateofbirth = dateofbirth;
  }
}
```

Object destruction

There are no destructors to free allocated memory. But sometimes we still need to release resources when an object is destructed.

- Java knows the finalize methods, which is called before the garbage collector destroys the object.
- But there is no guarantee when the garbage collector does so. This is typically undesirable.

Remember

It is better to be explicit on freeing up resources instead of using the finalize method. See try-with-resource as the better alternative later in the lecture.

Packages

In Java it is possible to group classes in packages.

- Package names (except the standard Java ones) start usually with a domain name in reverse, e.g. `at.ac.fhsalzburg.nos`. It is a naming convention in Java to use lower case characters for package names only.
- With the help of packages, the uniqueness of class names can be guaranteed, as classes with the same name in different packages can still be differentiated. It is similar to namespaces in C++.

```java
package com.example;

class A {
}
```

- Packages can be nested into each other, for example `java.util` is nested in `java`. For the compiler, there is no relationship between nested packages.

Packages and directories

The class `com.example.A` resides in the file `com/example/A.java`, so nested packages indicate nested directories.

- The `package` instruction says in which package classes are defined.

```java
package com.example;
public class Person {
...
```

Listing: com/example/Person.java

```java
package com.example;
public class Student {
...
```

Listing: com/example/Student.java

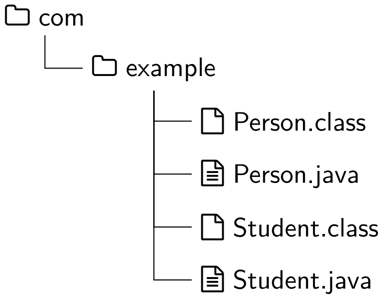


Figure: Directory structure.

Access modifiers for class elements

The table of access modifiers and their visiblity for fields and methods:

Modifier	Class	Package	Derived class	'Outside'
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

Remember

Default means "package protected", so declare your fields private to ensure encapsulation!

Also classes have access modifiers, and only public classes can be accessed outside the package.

```
public class Person {
}
```

Class importation

A class can use all classes that are in its own package as well all public classes from others.

- This can either be done by adding the full package name in front of every class name, or by using the import statement.
- This is like using namespaces in C++ and not like `#include` statements.

```java
import com.example.*;             // Import all classes from package com.example
import com.example.nos.Person;    // Import this specific class
```

Final fields

The value of fields defined as `final` cannot be change after initialization (cf. const in C++).

- Fields declared as final must be initialized upon object construction.
- A final field of a class type cannot re-bind to different object, but the object can be modified! (Different to C++!)
- Hence, useful for fields whose type is immutable (e.g. String) or primitive.

```java
class Person {
  final Person mother;
  final Person father;

  Person (Person mother, Person father) {
    this.mother = mother;
    this.father = father;
  }
}
```

Static fields

- A static field is like a “class field” rather than an “object field”.
- It exists exactly once, even if no or multiple instances have been created.
- Static variables are rather rare, static constants more common.

```java
class Physics {
  final static double VACUUM_PERMEABILITY = 1.256e-6; // H/m
  final static double SPEED_OF_LIGHT = 300e6;         // m/s
}

class Person {
  static int totalCount = 0;
  final int id;

  Person () {
    totalCount++;
    id = totalCount;
  }
}
```

Static methods

- Static methods do not operate on an object; they can be called by `classname.method()`
- They can access static fields and static methods.
- They cannot access `this` or object fields.
- The `main` method does not operate on an object, which is why it is a static method.

Class design guidelines

- Always keep fields private.
- Always initialize fields.
- Not each field always needs individual getter and setter methods.
- If a class has too many fields with a basic type then maybe the class should be split.
- If a class violates the *single-responsibility principle* then break it up.

Polymorphism

Polymorphism means “many forms”:

- A variable `Person p` can have many “forms”, e.g., the one of a `Student`.

```java
class Person {
  public String describe() {
    return "Name is " + getName();
  }
}
class Student extends Person {
  public String describe() {
    return "Name is " + getName() + " and stud ID is " + getStudID();
  }
}
```

- The key, however, is that the behavior is determined by the *dynamic type*, not the static type. This is called dynamic binding of the variable to the object: The right method is chosen at *runtime*.

```java
Person p = new Student();
// Dynamic type is Student: Student.describe() is called!
System.out.println(p.describe());
```

- Polymorphism in Java is done by method overriding.

Overriding methods

If we reimplement a method of a class in a subclass then we override it.

- That is, we replace the behavior of this method for this subclass. (How is this in C++?)
- The overridden method must have the same signature (method name and parameter list) and return type. The visibility cannot be more strict. (Why?)
- It is highly recommended to add the annotation `@Override` to the method definition: Then the compiler can check our intention. Common mistakes, like having the wrong parameter types or misspelling the method name, are less likely. We would accidentally *overload* rather than *override* the method.
- We can access the original version of the superclass by using `super` instead of `this`. (In C++?)

```java
class Student extends Person {
  @Override
  public String describe() {
    return super.describe() + " and stud ID is " + getStudID();
  }
}
```

Dynamic binding and method call resolution

When a method of an object is called:

- First the dynamic type of the object is determined.
- Then all methods of the given name for this type and its superclasses are determined. The overload resolution finds the best suiting method by matching argument types to the parameter types.
- It then calls the overridden method version of the most special type.
- In Java all methods are polymorphic (i.e., virtual in C++). However, by adding the keyword `final` to the method definition we can explicitly forbid overriding.

Attention

The `final` keyword has different meanings for different entities: Constant fields, non-overridable methods or non-inheritable classes.

Casting and type testing

Upcasting[6] is done automatically:

```java
Person p = new Student();
```

But downcasting requires an explicit cast:

```java
void f(Person p) {
  // I know I got a Student
  Student stud = (Student) p;
}
```

- This is done in C-style syntax, but behaves more like a `dynamic_cast` in C++. That is, it checks at runtime whether `Student` is a suitable static type for the variable binding to the object behind `p`.
- There is also an operator `instanceof` to check this.

```java
if (p instanceof Student) {
  Student stud = (Student) p;
}
```

6 Up in the inheritance hierarchy

Abstract classes

Sometimes a method cannot be meaningfully implement but is supposed to be overridden.

```java
class GeometricShape {
  public void rotate(double angle) {
    // I know how to rotate a Triangle, but an "abstract" GeometricShape?
  }
}
```

Java knows abstract classes for this use case:

- The class must be declared abstract to allow abstract methods.
- An abstract method has no implementation block.
- Hence, we cannot call this method. Hence, there cannot be an instance of this class! Hence, this class can only be a static type of a variable, not a dynamic type.
- Every non-abstract subclass *must* override this method.

```java
abstract class GeometricShape {
  public abstract void rotate(double angle);
}
```

The Template method pattern

The idea is that abstract classes define a “contract”.

- If `x instanceof GeoemtricShape` then `x` provides a `rotate()`, so we can use it.

What we see below is an application of the Template method design pattern.

```java
class GeometricShapeCollection {
  GeometricShape[] shapes;

  public void rotate(double angle) {
    // Rotate them all
    for (GeometricShape s : shapes)
      s.rotate();
  }
}
```

Interfaces

There is no multiple inheritance for classes, but for interfaces.

- An interfaces *is not* a class, but a set of requirements that a class has to fulfill. If at all, it would be similar to an abstract-only class, e.g., they cannot be instantiated.
- An interface may only declare methods (implicitly public abstract) and define constant fields (implicitly public static final).
- However, a class can implement multiple interfaces. And an interface can extend multiple interfaces.

```java
interface Comparable {
  int compareTo(Object o);
}

interface Hashable {
  int hashCode();
}
```

```java
class Person implements Comparable, Hashable {
  public int compareTo(Object o) {
    // TODO. You recall instanceof and casting?
  }
  public int hashCode() {
    // TODO
  }
};
```

Cloning: Obtaining a copy

Binding a second variable to the same object does not copy it. To obtain a copy we have to `clone()` it.

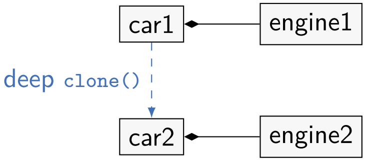
```java
Person p = new Person();
Person q = (Person) p.clone();
```

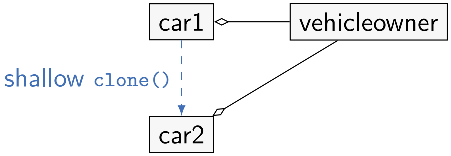
- Note that `clone()` of `Object` is protected. So you have to override it and make it public.
- Also, implement the `Cloneable` interface, otherwise `CloneNotSupportedException` is thrown by `Object`.

Shallow versus deep

Two types of clones:

Deep copy	If the field of the object is cloned, too. Typical for composition due to unique ownership.
Shallow copy	If the field is not cloned.





Section 2

Some further remarks

Parameter naming

There are several strategies to differentiate between instance fields and parameters

- Single-letter or non-meaningful or non-expressive variable names make code harder to read
- Fields and parameters can have the same name, but the latter could start with a prefix
- Parameters shadow fields having the same name. When accessing the parameter just use the name, when referring to the field use `this.fieldname`

03: Exceptions and Concurrency

Network Oriented Software

Stefan Huber
www.sthu.org

Dept. for Information Technologies and Digitalisation
FH Salzburg

Summer 2023

Section 1

Exceptions

Traditional error handling

Exception handling is the object-oriented way of error handling.

Traditional error handling strategies:

Traditional error handling

Exception handling is the object-oriented way of error handling.

Traditional error handling strategies:

- Return value of functions encodes error state.
- Error states are stored in a global variable, like libc's `errno`.

Issues with the traditional style:

- Global commitment how error numbers are to be interpreted. Hard to generalize, hard to extend, hard to manage, hardly possible to change.
- Additional information on the nature of the error is cumbersome to manage.
- Error handling logic grows, propagates and distributes through the entire code.

Exceptions basics

Exceptions enable us to decouple the recognition of the error from the error handling:

- Recognition: When an error situation happened, we throw an exception instance.
- Handling: At dedicated places we catch the exception instances and handle them.

Java takes care for passing the exception from the recognition to the handling:

```
    try {
      // The code in here may throw an exception
      if (error)
        throw new MyException();

      // Or in a nested method call an exception might be thrown
      myAlsoThrow();
    }
    catch (Exception e) {
      // The code to handle the exception.
    }
```

A minimal example

Examples:

- `IOExcecption` is thrown when an I/O operation failed, like reading data from a file.
- `FileNotFoundException` is thrown when we attempt to open a non-existing file to read from.

```java
/** Demonstrates catching exceptions. */
class ExceptionMinimal {
    public static void main(String args[]) {
        try {
            // Documentation says it may throw a FileNotFoundExcpetion
            java.io.InputStream is = new java.io.FileInputStream("doesnotexist.txt");
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("Got the exception:");
            e.printStackTrace();
        }
    }
}
```

Catching different exceptions

The general syntax for exception catching is as follows:

```java
try {
  // Code that may throw
}
catch (ExceptionType1 e) {
  // Handle all exceptions that are of (sub) type ExceptionType1.
}
catch (ExceptionType2 e) {
  // Handle all exceptions that are of (sub) type ExceptionType2.
}
// and more
finally {
  // This is executed, no matter whether exception has been thrown or not.
}
```

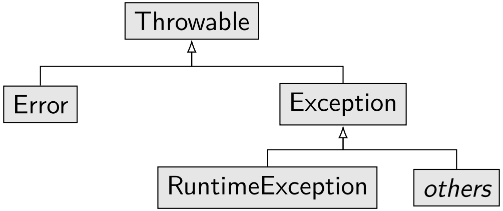
The first catch block that matches the type handles the exception.

- All (checked) exceptions derive from `Exception`, so catching type `Exception` catches them all.

Types of exceptions

Every exception derives from `Throwable`:

- `Error` captures internal errors of the JRE.
- `Exception` captures the errors that originate from your application.
  - `RuntimeException` captures programming errors, like `ArrayIndexOutOfBoundsException`.



Java knows two types of exceptions:

- Unchecked exceptions derive from `Error` and `RuntimeException`. We do not anticipate those.[1]
- Checked exceptions are all others. These can been seen as contract between provider and caller.

[1] *Where* would we catch them? "Everywhere"?

Checked exceptions

Java is very strict on catching or not catching *checked* exceptions:

- If a method throws (or passes on) a checked exception then it has to declare so!
- This is done by the keyword `throws` after the parameter list.

```java
/** Demonstrates not catching exceptions. */
class NoCatchDemo {
    // Without this throw clause we get this compiler error:
    //    NoCatchDemo.java:11: error: unreported exception IOException; must be
    //    caught or declared to be thrown
    public static void main(String args[]) throws java.io.IOException {
        System.out.println("Press any key...");

        // This method may throw an IOExcpetion but we do not catch it. Hence,
        // main() may throw an IOException.
        System.in.read();
    }
}
```

Section 2

Threads and Processes

Multi-tasking operating systems

Without an operating system, like on microcontrollers, the one and only program has

- exclusive access to the processor and
- exclusive access to the memory.

Simple view: A multi-tasking operating system can run multiple programs at a time.

Multi-tasking operating systems

Without an operating system, like on microcontrollers, the one and only program has

- exclusive access to the processor and
- exclusive access to the memory.

Simple view: A multi-tasking operating system can run multiple programs at a time.

To a computer scientist a multi-tasking OS provides the concept of concurrently executed processes.

A process is an illusion provided by a multi-tasking operating system:

- Like each process has its exclusive *virtual processor*.
- Like each process has its exclusive *virtual memory*.

Multi-tasking operating systems

Without an operating system, like on microcontrollers, the one and only program has

- exclusive access to the processor and
- exclusive access to the memory.

Simple view: A multi-tasking operating system can run multiple programs at a time.

To a computer scientist a multi-tasking OS provides the concept of concurrently executed processes.

A process is an illusion provided by a multi-tasking operating system:

- Like each process has its exclusive *virtual processor*.
- Like each process has its exclusive *virtual memory*.

Remark

Half of Computer Science is about abstractions and illusions!

Processes

The isolation of processes gives us this illusion, this abstraction.[2]

To provide the illusion of exclusive processor and memory an OS must:

- Isolate process memories by mapping the virtual memory pages to different physical page frames.
- Frequently schedule processes, i.e., assign a process to a processor for execution.
- Remember various states and resources per process in the *process control block*:
  - Process ID
  - Processor states: Instruction pointer, registers, virtual memory page mapping (page tables), …
  - Privilege information: User ID, Group ID, …
  - Resources: Open files, sockets, pipes, shared memory, …

[2] A *realtime* operating system additionally provides *temporal* isolation between processes.

Processes in Java

Java feels a bit different here:

- Java prohibits direct memory access, so there is no need for memory isolation mechanisms as in C.
- Only with Java 5 a convenient way to fork processes was introduced via `java.lang.ProcessBuilder`. Only with Java 9 the interface `ProcessHandle` was introduced to obtain, e.g., the PID of a process.

Many responsibilities of an OS moved into the JVM.

In Java concurrency is typically done by multi-threading rather than multi-processing.

Threads

A thread is a lightweight process:

- Threads are executed concurrently within one process.
- They share the same resources as a single process, e.g., the same process memory.

Threads (and processes) provide us with the means for concurrent programming.

- Concurrency does not imply parallelism!
- If your system possesses multiple processors or cores then threads *may* be executed in parallel.

Java supports multithreaded programming from the very first version.

Java threads

We require two ingredients to start a thread:

1. A class that implements the interface `java.lang.Runnable`, which possesses a method `run()`, which is like the thread's "`main()`".
2. A `java.lang.Thread` instance that encapsulates the thread and its properties.

Notice

Java decouples the task to be run (`Runnable`) from the mechanism of running it (`Thread`).

Java threads

```java
class ThreadMinimal {
    public static void main(String args[]) {
        // Create and start the worker thread
        java.lang.Thread th = new java.lang.Thread(new ThreadMinimalWorker());
        th.start();
    }
}

class ThreadMinimalWorker implements java.lang.Runnable {
    @Override
    public void run() {
        try {
            while (true) {
                System.out.println("99 bottles...");
                java.lang.Thread.sleep(500);
            }
        }
        catch (java.lang.InterruptedException e) {
        }
    }
}
```

Running multiple threads concurrently is more fun, see `ParallelJobDemo`.

Interrupting threads

The example before did not terminate after `main()` returned.

- The thread started is a so-called non-daemon thread. All non-daemon threads need to terminate in order for the process (JVM) to terminate.

We can kindly ask a thread to terminate by interrupting it.

- We call `th.interrupt()` for a `Thread th`.
- The interrupt status of `th` is set, which we can check by `Thread.isInterrupted()`.
- There is also a static method `Thread.interrupted()` which returns whether the current thread has been interrupted *and clears the flag*.
- However, if the thread is blocked (e.g, sleeping) we cannot check. This is when an `java.lang.InterruptedException` is thrown, which we catch.

Notice

Interrupting a thread is like sending a signal to it.

- But there is no guarantee that the thread is indeed terminating itself; it is the thread's choice.

ParallelJobInterruptDemo

Interrupting worker threads:

```
        // Interrupt all the threads and let them stop
        System.out.println("Interrupt the threads...");
        for (Thread th : workerThreads)
            th.interrupt();
        System.out.println("Good bye.");
```

Handling the interrupt in each worker thread:

```

        try {
            // Keep going as long as we did not receive the interrupt. If we
            // receive an interrupt, however, we most likely do so while
            // sleeping, so the exception handling will be executed.
            while (!Thread.currentThread().isInterrupted()) {
                cnt++;
                System.out.println("  Loop " + cnt + " th: " + threadid);
                Thread.sleep(DELAY_MS);
            }
        }
        catch (InterruptedException e) {
            System.out.println("  Thread " + threadid + " int'ed.");
```

Joining threads

In a next step we see that "Good bye" is printed before the threads terminated.

- If we would need to do some clean up, we would need to wait for the termination of the threads.
- The method `Thread.join()` blocks until the thread has terminated.

```java
        // Wait for the threads to have terminated. Note that join() is a
        // blocking operation, so it may throw InterruptedException itself.
        try {
            System.out.println("Join the threads...");
            for (Thread th : workerThreads)
                th.join();
        }
        catch (InterruptedException e) {
            System.out.println("Main interrupted");
        }
```

The program `ParallelJobJoinDemo` provides a complete program to showcase a pool of worker threads with a clean tear down.

Race conditions

In practice, threads often access shared data or communicate with each other:

- Non-atomic, concurrent access to shared data easily corrupts the data.
- We have to be very, very careful that correctness does not depend on the *lucky* timing of thread execution, which constitutes a race condition!

A typical race condition in `transfer()`:[3]

```java
    /** The balances of the two accounts */
    private int[] accounts = {0, 0};

    /** Transfer given amount of money from one account to the other.
     * The total balance stays invariant, i.e., sum of balances is constant. */
    public void transfer(int from, int to, int amount) {
        accounts[from] -= amount;
        accounts[to] += amount;
    }
```

[3] The complete example is in `BadBank.java`

Race condition step by step

Note that for `x += y` the `add` and `store` machine instructions of the JVM are performed one at a time. Consider the JVM essentially performing these machine instructions:

```
tmp = x + y
x = tmp
```

If the timing of thread execution is not lucky then two threads may show this execution order:

```
// accounts is {0, 0}
// thread 1 calls transfer(0, 1, 100);
tmp0 = accounts[0] - 100;   // tmp0 is -100





accounts[0] = tmp0;
tmp1 = accounts[1] + 100;   // tmp1 is -100
accounts[1] = tmp1;
// accounts = {-100, -100}
```

```
// accounts is {0, 0}
// thread 2 calls transfer(1, 0, 200);

tmp1 = accounts[1] - 200;   // tmp1 is -200
accounts[1] = tmp1;
tmp0 = accounts[0] + 200;   // tmp0 is  200
accounts[0] = tmp0;
// accounts = {200, -200}
//
//
//
//
```

Remember: Concurrent access to shared data requires synchronization mechanisms!

Synchronization

We need to make the method `transfer()` mutual exclusive:

- No two threads can execute this method at the same time!
- In other words, we need synchronized access to `transfer()`.

In Java, this can be conveniently done by adding the synchronized keyword to the method:[4]

```java
/** Transfer given amount of money from one account to the other.
 * The total balance stays invariant, i.e., sum of balances is constant. */
public synchronized void transfer(int from, int to, int amount) {
    accounts[from] -= amount;
    accounts[to] += amount;
}
```

4 The complete example is in `GoodBank.java`

Locks

Behind the scenes of `synchronized` we find the concept of locks.

- A lock can be acquired by only one thread at a time. If another did already then it is blocked until the lock is released again, so another thread can attempt to acquire it.
- Java provides the class `ReentrantLock`, which implements the interface `Lock`, both in the package `java.util.concurrent.locks`.

```
        // Thread is blocked until it acquires the Lock l
        l.lock();
        try {
            // The critical section
        }
        finally {
            // Important: Even if exception has been raised, we have to release
            // the lock again!
            l.unlock();
        }
```

Synchronized blocks

Synchronization leads to a serialization in execution:

- A synchronized method is like a tunnel where cars (threads) cannot pass in parallel, but one after the other.
- This serialization hits performance, so we would like to make the synchronized parts as tight as possible.

It would often be excessive to synchronize whole methods.

- Instead, we only would like to synchronize the critical sections.
- One way is to explicitly use `Lock` instances from before.
- Java provides synchronized blocks as a more convenient alternative.

```
        // Only this small critical section needs to be synchronized. Any
        // instance of Object can act as "lock" object here.
        synchronized (accounts) {
            accounts[from] -= amount;
            accounts[to] += amount;
        }
```

Synchronized blocks

A synchronized method locks the entire object `this`:

- It is like putting the entire method within a `synchronized(this)` block.

If you have multiple synchronized blocks that use *different* lock objects then those are not mutual exclusive to each other!

- One thread can enter the one block while another thread may enter the other!
- If you mix synchronized methods and synchronized blocks this can easily happen!
- Keep your lock mechanisms clean and *simple*!

Conditions

Assume now for our bank example that we would not like to leave negative account balances, so we wait until enough money is on the account.

```
synchronized (accounts) {
    if (accounts[from] < amount)
      // Wait until accounts[from] becomes >= acount. But no other thread can
      // enter this critical section, so we wait forever. If we move this out
      // of the critical section then we have the race condition all over again.
      magicWait();

    accounts[from] -= amount;
    accounts[to] += amount;
}
```

This is where the concept of a `Condition` jumps in:

- A `Lock.newCondition()` creates a condition on the lock.
- A condition is like a signal for which threads can wait for.
  In our case it could be the condition "account received money". Name the condition accordingly!
  When a thread sends money to an account then it has to notify the condition resp. all threads.

Responsible bank

See full code in `ResponsibleBank.java`

```java
    public void transfer(int from, int to, int amount) throws InterruptedException {
        accountLock.lock();
        try {
            // Wait until enough money is in the orignal account.
            while (accounts[from] < amount)
                // This thread gives up the lock and continues only after
                // having received the signal and re-aquired the lock again.
                receivedCondition[from].await();

            // Transfer the money.
            accounts[from] -= amount;
            accounts[to] += amount;

            // Tell all threads waiting for money on the destination account.
            receivedCondition[to].signalAll();
        }
        finally {
            accountLock.unlock();
        }
    }
```

Deadlocks

Despite the notorious difficulty in getting it done right, there is a another main issue that needs to be addressed in the course of concurrent programming: Deadlocks

- A deadlock is a circular dependency of threads in waiting for each other to give up locks, so no thread can proceed anymore.
- Thread T_1 waits for T_2 and vice versa. Both are blocked, so neither can release its lock.
- Thread T_1 waits for T_2, waits for T_3, ..., waits for T_n, waits for T_1.

Treating deadlocks is beyond the scope of this course.

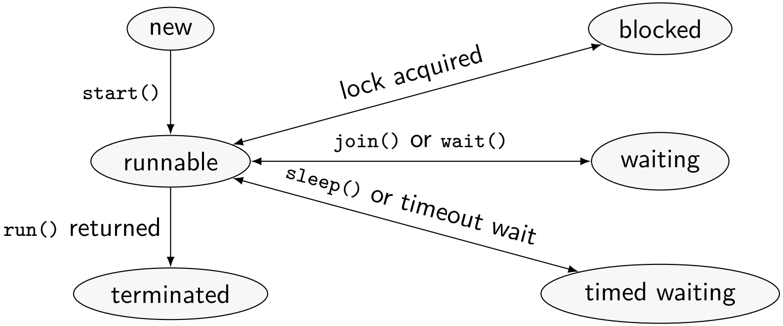
- Advise 1: Write clean and simple locking mechanisms for which you can *prove* that no circular locking dependencies (deadlocks) can occur!
- Advise 2: If possible, try to lock multiple ressources in the same order.

A livelock is similar to deadlocks, except that they are not blocked but instead busy only with locking and unlocking without being able to progress.

Thread states

The lifecycle of a thread is captured by a state machine:

- When a `Thread` instance is created it is not running, but in state *new*. After starting it it becomes *runnable* and can be scheduled for execution. After the `run()` method exits the thread has *terminated*.
- When it is in a *blocked* or *(timed) waiting* state it is temporarily inactive.



04: Network Programming & Client-Server-Architecture

Network Oriented Software

Stefan Huber
www.sthu.org

Dept. for Information Technologies and Digitalisation
FH Salzburg

Summer 2023

Section 1

Java I/O

Streams

The Java I/O classes are found in the package `java.io`[1].

- The API provided by `java.io` is the classic API.
- A more modern API for I/O is provided by `java.nio` since Java 7.[2]

The stream classes are for sequential streams of binary data to files, consoles or other "devices".

- `InputStream` is for reading and `OutputStream` is for writing.

Examples:

- `FileInputStream` and `FileOutputStream` are subclasses for file I/O.
- `ObjectOutputStream` and `ObjectIntputStream` can be used serialize and deserialize objects.

1 See `https://docs.oracle.com/javase/tutorial/essential/io/index.html`.

2 Actually, Java NIO for non-blocking I/O was introduced with Java 1.4 and added the selector, channel, et cetera classes. With Java 7, Java NIO.2 was introduced. It added the java.nio.file package, with improvements on file handling. However, also AsynchronousSocketChannel was introduced by Java.NIO2, which facilitates the Future generic class for asynchronous computation.

Readers and writers

Readers and Writers are built on top of streams to support character streams.

- It hides the details of dealing with encodings.
- It adds additional functionality, like buffering.

Examples:

- `InputStreamReader` reads from any `InputStream`, a `FileReader` reads from a file.
- `BufferedReader` adds buffers and wraps around other readers.
- Likewise there is `BufferedWriter` and `FileWriter`.
- `PrintWriter` supports all kind of print methods for formatted output.

Try with resource

Many resources require to be closed explicitly. Exception handling needs some care:

```java
BufferedReader br = new BufferedReader(new FileReader("/tmp/mfile.txt"));
try {
    // Work with resource br
} finally {
    if (br != null)
        br.close();
}
```

For this use case Java knows a simpler try-with-resource statement:

```java
try (InputStream is = new FileInputStream("/tmp/mfile.txt")) {
    // Work with resource is
}
```

Cleanup is done after the try block:

- This works for all resources implementing the interface `AutoCloseable`, which defines a `close()`.
- Like streams and readers.

Section 2

Network Programming

ISO/OSI Reference Model

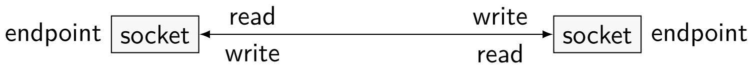
- The OSI model is a reference model where each of the 7 layers has a dedicated task.
- Each layer provides functionality to the layer above it and uses the service provided by the one below it.
- This is called Separation of Concerns. A concrete layer must be exchangeable for another implementation as long as it provides the same functions (it has to fulfill a "contract").

Layer	Name	Description
7	Application	High-level APIs, remote file access, resource sharing
6	Presentation	Data translation including encoding, compression, encryption
5	Session	Communication sessions with back-and-forth transmission
4	Transport	Sending data segments and datagrams
3	Network	Routing packets, addressing, traffic control
2	Data link	Data frames between nodes
1	Physical	Bit stream over physical medium

Sockets

Sockets in general are objects used to communicate with other programs.

- This can be on the same system (inter process communication, IPC) or via a network.
- A socket is a communication endpoint: the exchange happens between two sockets.
- They can be used bidirectionally, which means a socket can both send and receive data.

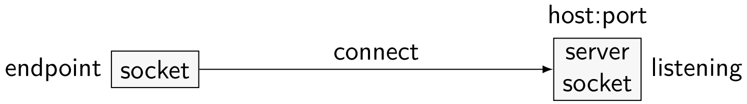


To connect a TCP socket to another TCP socket, it must know:

- The IP address of the other end, e.g., `127.0.0.1` or `::1` for localhost.
- The port to which it should connect; they can range from 0 to 65535, but ports below 1024 are well-known ports and privileged. Superuser rights are required to use them on UNIX systems.

Server sockets

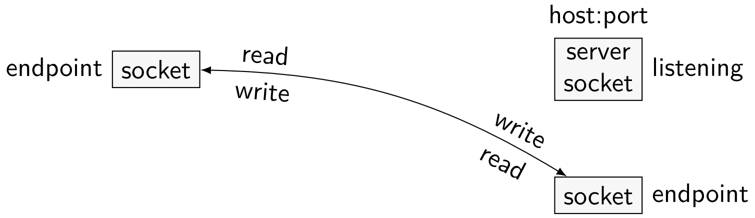
- A server socket is used to accept incoming TCP connections from clients.
- They are listening on a port for incoming communication requests.
- They do not need to specify an IP address, in which case they will listen on all network devices. But a specific IP address can be given to choose a specific network interface.



Server sockets are not created to send or receive data, but only to accept connections.

Server sockets

- A server socket is used to accept incoming TCP connections from clients.
- They are listening on a port for incoming communication requests.
- They do not need to specify an IP address, in which case they will listen on all network devices. But a specific IP address can be given to choose a specific network interface.



Server sockets are not created to send or receive data, but only to accept connections.

- Once a connection was accepted, a "regular" socket is created as communication endpoint. This new socket does not use the server socket's port but a different one; it is automatically assigned.

Socket communication sequence

The following table shows the typical sequence when making a connection and communicating with sockets.

Client	Server	Comment
	Opens a server socket Starts listening for incoming connections	setup of server
Creates a socket Connects to the server Sends/receives data Closes socket	 Accepts connection, gets a client socket Sends/receives data over client socket Closes client socket	for each connection
⋮	⋮	for each connection
	Closes server socket	tear-down of server

Sockets in Java: The client

- The `Socket` class in Java can be found in the `java.net` package.

```java
        try {
            // Create a new socket and connect to a server
            // on the same system (localhost) on port 5000
            Socket s = new Socket("localhost", 5000);
```

- Since a number of different things can go wrong with network-related methods, sockets can throw an IOException. This exceptions must be handled when working with sockets.

```java
        } catch (IOException e) {
            System.out.println("There was a problem with the connection");
        }
```

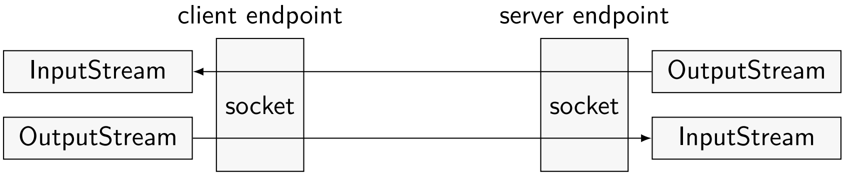
- When the socket is not in use any more, it must be closed.

```java
            // Close the Socket
            s.close();
```

  Hence, it is highly recommended that you use a try-with-resource statement, see `BetterSocketClient.java`.

Sockets in Java: Sending and receiving

An open socket can then send and receive data through streams.



- The `getOutputStream()` method returns its output stream.

```
// A PrinterWriter that wraps around the sockets output stream.
PrintWriter out = new PrintWriter(s.getOutputStream(), true);
// Send a message to the server.
out.println("Hello Server!");
```

- The `getInputStream()` method returns its input stream.

```
// A BufferedReader that wraps around the socket's input stream.
BufferedReader in = new BufferedReader(
                        new InputStreamReader(s.getInputStream()));
// Read a line from the InputStream
String str = in.readLine();
```

Sockets in Java: The server

The `ServerSocket` class is in the `java.net` package.

- Its constructor takes the port number on which it should listen.
- With the `accept()` method it waits for incoming connections. It returns a `Socket` object of the new connection to the client.
  - Via this socket it communicates to the client.

```java
        // Open server socket that listens on port 5000
        ServerSocket serv = new ServerSocket(5000);

        // Listens until it receives an incoming connection
        // returns an socket that handles the actual communication
        Socket s = serv.accept();
```

- Call `close()` for each socket when they are not used anymore. Use the try-with-resource statement.
  - The socket for the client connection, when done.
  - The server socket, when done.

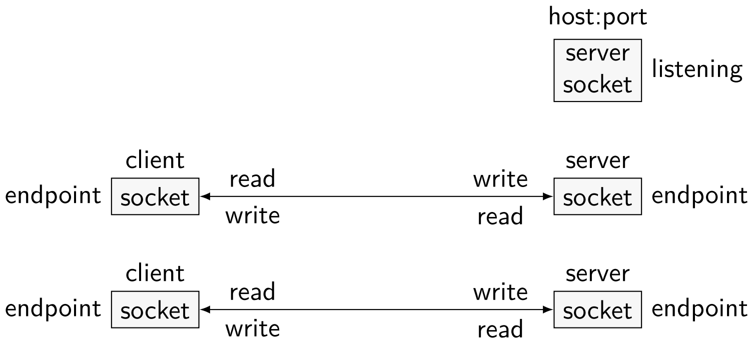
Section 3

Client-server architectures

Concurrency models

Usually a server has to handle more than one client connection. The two main models to achieve server-side concurrency are:

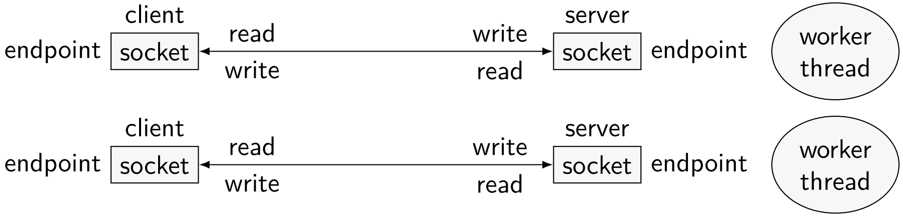
- Fork on request
- Event driven



Fork on request

Fork-on-request[3] is done in two steps:

- A main process/thread that listens for incoming connections, accepts them and starts (forks) a new worker process/thread for each one.
- The worker process/thread does the actual communication.



Use netcat to test your client or server.

3 The term "fork" refers to the POSIX function `fork()` to spawn a new process.

Fork on request: Server

The following example shows the handling of new connections on a server that uses the fork-on-request concurrency model.

```
        // Open a socket on port 5000. We use try-with-resource such that
        // socket is closed automatically.
        try (ServerSocket serv = new ServerSocket(5000)) {
            // TODO: Add logic for graceful tear down, including threads.
            while(true) {
                // Listen an accept new connections
                Socket incoming = serv.accept();
                // Start a new thread that handles the connection
                Thread t = new Thread(new EchoServerWorker(incoming));
                t.start();
                System.out.println("New thread " + t.getId());
            }
        } catch (IOException e) {
            System.out.println("There was an error");
        }
```

Fork on request: Worker

The following example shows how each thread can handle its connection and communication with the client it is serving.

```
            while(true) {
                // Read message from client
                String str = in.readLine();
                if(str == null)
                    break;

                // Send back "Echo:" + original message
                out.println("Echo: " + str);
                out.flush();
                System.out.println("Recv: " + str);

                if(str.trim().equals("BYE"))
                    break;
            }
```

See `ForkOnRequestEchoServer.java` and `EchoServerWorker.java` for all the details.

Preforking

Forking a process is a costly task:

- For instance, a new process memory has to be set up, but also all other management structures in the process control block.
- Starting a thread is much cheaper.

Preforking can cut the setup costs:

- A fixed pool of worker threads (or processes) is pre-initialized and waiting for work.
- Preforking with blocking I/O does not provide full concurrency. The number of concurrently serviced clients is restricted by the size of the pool.
  This can be suitable for certain applications, where a great number of connections cannot be handled anyway (e.g., a file server is often I/O-bound by disk).

Event-driven architecture

But also context switches between processes came at costs:

- When switching the virtual memory mapping a TLB flush has to be paid.
- Threads are cheaper, but there are still costs, like cache invalidation and the scheduling costs.

What if we want to handle many ten thousands or a million connections per second?

- We cannot afford the thread handling anymore!
- Can we be concurrent without threads or processes?
- Not with blocking I/O!

Conclusion

Massively parallel network communication cannot be built on top of blocking I/O.

Event-driven architecture

Corner points of an event-driven architecture:

- Non-blocking, asynchronous I/O calls are used.
- A single thread is used (or can be used). That is, concurrency is not founded on top of threads.

Non-blocking I/O

- We send I/O requests to the system and get notified when they are done. No waiting, no blocking. Hence, it is also known as asynchronous I/O.
- There is a single event loop that reacts on these notifications (events).

The concurrency logic moves from the OS and system libraries to the application, which results in more complex code, but at the benefit of performance.

Preforking is not required on an event-driven architecture. But it can be used to increase CPU utilization on a multi-core system.

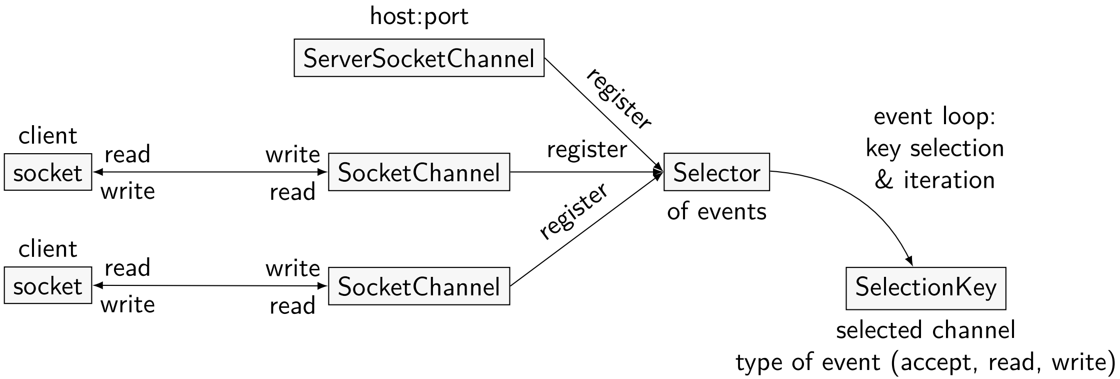
Non-blocking I/O in Java

The classic I/O API is stream-oriented and provides blocking I/O.

The modern I/O API in `java.nio` also provides non-blocking I/O:

- It is buffer-oriented. Buffers allow us to hand over or take over the data.
- Instead of streams, we have channels from which data is read and written to, through buffers.
  - A `SocketChannel` is a channel for a client sockets, a `ServerSocketChannel` for a server socket.
- Selectors are used to select those channels for which "I/O events" have arrived. For instance:
  - A client connection can be accepted.
  - Data can be read.
  - Data can be written.
- Only a so-called `SelectableChannel` can be registered at a `Selector`.
  - A `SelectableChannel` can be configured as non-blocking.
  - The socket channels inherit from `SelectableChannel`.

Channels and selectors (server)



A code example is provided as `EventDrivenServer` and `EventDrivenClient`.

05: Middleware

Network Oriented Software

Stefan Huber
www.sthu.org

Dept. for Information Technologies and Digitalisation
FH Salzburg

Summer 2023

Section 1

Autoboxing and reflection

Autoboxing and unboxing

We are sometimes forced to treat primitive-type variables as objects.

```java
int foo = 42;
// static int hashCode(Object o)
int foo_hash = java.util.Objects.hashCode(foo);
```

Java knows to every primitive type a corresponding class type:

`byte`	`java.lang.Byte`	`float`	`java.lang.Float`
`short`	`java.lang.Short`	`double`	`java.lang.Double`
`int`	`java.lang.Integer`	`boolean`	`java.lang.Boolean`
`long`	`java.lang.Long`	`char`	`java.lang.Character`

The `hashCode()` example works because Java is automatically converting `int` to `java.lang.Integer` in order to pass an `Object` to `hashCode()`.

- Autoboxing: Automatic conversion to class type.
- Unboxing: Automatic conversion back to primitive type.

Generics

Generics are in Java what templates are in C++, but less powerful.[1]

- Generics reduce type erasure and the need of casting from `Object`, but instead preserve types.
- Generics allow for generic programming.[2]

Without generics:

```
List list = new ArrayList();                 // ArrayList implements the interface List
list.add(42);                                // Autoboxing to java.lang.Integer
list.add("Hello");                           // Takes any Object
Integer head = (Integer) list.get(0);        // get() returns an Object: requires a
                                             // cast, which is error prone.
```

With Java 5 generics were added: We can pass the container element type as a type parameter:

```
List<Integer> list = new ArrayList<Integer>();
list.add(42);                                // Autoboxing to java.lang.Integer
list.add("Hello");                           // Compiler error: Not an Integer
int head = list.get(0);                      // Unboxing from java.lang.Integer
                                             // get() returns an Integer.
```

[1] They are computationally equally powerful as they are both Turing complete, cf. [Gri16]. In C++ this gives rise to *meta programming*. But C++ templates and Java generics differ in the sense that Java generics are rather syntactic sugar to hide casts. It is actually possibly to work around this and insert a `String` into a `list<Integer>` using some casts.

[2] There is a collections framework in Java with many different container types: https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/doc-files/coll-overview.html

Reflection

Reflection is in Java what RTTI is in C++, but more powerful.

- It allows to introspect objects and its features (fields, methods) at runtime, which will be useful for marshalling and RMI, see later.

The class `java.lang.Class` allows us to interrogate the type:[3]

```
        Class cl = Dummy.class;
        System.out.println("Dummy is a type called " + cl.getName());
        System.out.println("And int is " + int.class.getName());

        Dummy dummy = new Dummy();
        System.out.println("dummy is object of type " + dummy.getClass().getName());
        System.out.println("dummy is a Dummy? " + (dummy.getClass() == Dummy.class));
        System.out.println("dummy is also an Object? " + (dummy instanceof Object));
```

3 See more at `https://www.oracle.com/technical-resources/articles/java/javareflection.html`

Reflection

However, the reflection mechanism can even instantiate objects and call methods by name.

- We start by obtaining a `Class` instance by name using `Class.forName()`.
- This could be used to implement a plugin mechanism, for instance.
- Reflection is the technical foundation for distributed object architectures, see later.

```java
    public static void introspect(String clname) {
        try {
            System.out.println("Introspecting class " + clname);
            Class cl = Class.forName(clname);
```

```java
            // Get the no-argument constructor, instantiate an object, get the
            // toString() method and invoke it on the instance.
            Constructor noArgCtor = cl.getDeclaredConstructor();
            Object obj = noArgCtor.newInstance();
            Method toStringMethod = cl.getMethod("toString");
            String res = (String) toStringMethod.invoke(obj);
            System.out.println("(new " + clname + "()).toString() -> " + res);
```

Section 2

Distributed systems and middleware

Distributed systems

A computing system that is distributed on a computer network.

- Collection of *autonomous* components (nodes).
- Nodes cooperate to form a single *coherent* system.
- Nodes are distributed on and communicate via a computer network.

Characterization 1 (van Steen)

A *distributed system* is a collection of autonomous computing elements that appears to its users as a single coherent system.

Literature: Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. 4th ed. Jan. 2023. ISBN: 978-9081540636

Examples for distributed systems

WWW	Master-slave. The World Wide Web is the probably the largest distributed system. Web browsers and web servers as nodes that communicate via HTTP protocol over the Internet. For the user the WWW appears as a single system.
Bittorrent	A peer-to-peer file sharing protocol using so-called distributed hash tables to find peers.

Examples for distributed systems

WWW	Master-slave. The World Wide Web is the probably the largest distributed system. Web browsers and web servers as nodes that communicate via HTTP protocol over the Internet. For the user the WWW appears as a single system.
Bittorrent	A peer-to-peer file sharing protocol using so-called distributed hash tables to find peers.
GIMPS	Great Internet Mersenne Prime Search is an example for *distributed computing*.
NFS	The Network File System is a distributed file system based on Remote Procedure Calls.
Bitcoin	A cryptocurrency implementing a distributed ledger. Bitcoin is using an underlying blockchain and a distributed consensus algorithm (Proof of Work) to ensure integrity.

Examples for distributed systems

WWW	Master-slave. The World Wide Web is the probably the largest distributed system. Web browsers and web servers as nodes that communicate via HTTP protocol over the Internet. For the user the WWW appears as a single system.
Bittorrent	A peer-to-peer file sharing protocol using so-called distributed hash tables to find peers.
GIMPS	Great Internet Mersenne Prime Search is an example for *distributed computing*.
NFS	The Network File System is a distributed file system based on Remote Procedure Calls.
Bitcoin	A cryptocurrency implementing a distributed ledger. Bitcoin is using an underlying blockchain and a distributed consensus algorithm (Proof of Work) to ensure integrity.
Car	A car comprises a distributed system of several dozens ECUs, controlling the engine, brakes, doors, or HMI. A typical automotive network is CAN bus.
Automation	An industrial machine comprises controllers, drives, sensors, HMI with realtime communication over a classical fieldbus or, more modern, using distributed object systems, like OPC UA, over time-triggered Ethernet-based networks, like TSN.

Characteristics: Collection of autonomous nodes

- Nodes are in principle independent from each other, but cooperate.
- Nodes run concurrently.
- The collection may be very heterogeneous.
- Each node has its own notion of time. There may not be a global clock.
- Cooperation requires communication, e.g., message passing.

Characteristics: Single coherent system

- Wiktionary on *coherent*: Unified; sticking together; making up a whole.
- Level of coherence versus the level of distribution.
- Distribution transparency:
  - User does not need to know where something is processed, stored, et cetera.
  - Half of system engineering is abstraction! Abstraction means hiding details.

Characterization 2 (Lamport)

[A distributed system is] one in which the failure of a computer you didn't even know to exist can render your computer unusable.

Middleware

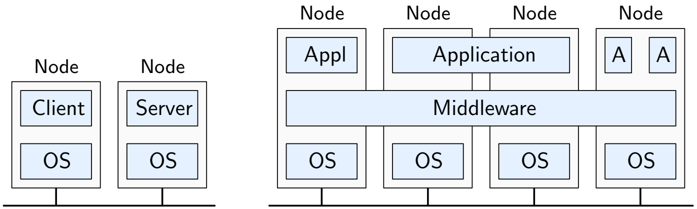
Distribution transparency

Distribution transparency allows for ignorance of the location of data or services.

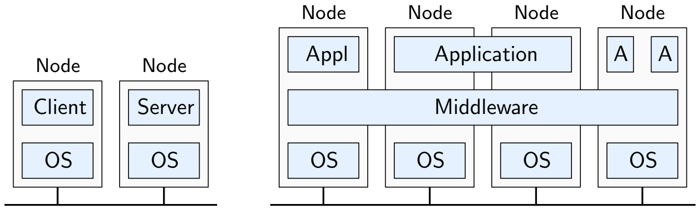
Middleware

Middleware is a cross-node layer on top of each OS to provide distribution transparency to distributed applications.

- It makes a distributed system appear as a single computer.
- Provides abstractions from the details of communication (on a data level).



Middleware services



Middleware : distributed system $\widehat{=}$ operating system : computer.

- Manages resources
- Provides services
  Naming, inter-application communication, failure tolerance, security ...

Types of transparencies

Distribution transparency may refer to different transparency types:[4]

Transparency	**Description**
Access	Hide differences in data representation and method of access
Location	Hide where object is located
Relocation	Hide that an object may be moved while used
Migration	Hide that an object is moving/mobile object
Replication	Hide that an object is replicated
Concurrency	Hide that an object is shared by independent users
Failure	Hide failure and recovery of an object

- Location: We are ignorant of the server that ships the page `www.google.com`.
- Relocation: We even do not notice if the server just changed.
- Migration: Relocation means moving objects *by* the system, but migration is an offer to the user, e.g., a mobile phone can move between base stations, even during a call.

[4] An object here may mean process or resource.

Section 3

Remote Procedure Call

Remote Procedure Call

Within an application, how do components communicate?

- Procedure calls.[5]

Within a distributed system, how do components communicate (so far)?

- Message exchange.
- Even for Inter-Process Communication on the same host we have a message-based communication.

5 Assuming an imperative programming paradigm.

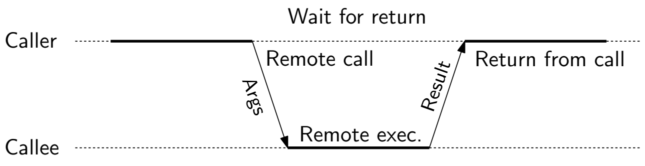
Remote Procedure Call

Within an application, how do components communicate?

- Procedure calls.[5]

Within a distributed system, how do components communicate (so far)?

- Message exchange.
- Even for Inter-Process Communication on the same host we have a message-based communication.



Simple yet powerful idea: Procedure calls within distributed applications.

- We can call a remote procedure as if it is local. No change of paradigm.
- Increased access transparency.

5 Assuming an imperative programming paradigm.

Marshalling

Openess

A distributed system shall be open; it shall be easy to use and integrate its components in other systems.

An open RPC mechanism has to deal with heterogeneous systems:

- Programming languages
- Operating system
- Processor instruction sets (x86, amd64, arm, powerpc, alpha, ...), register sizes, address bus sizes, byte order (little endian versus big endian), floating-point units, et cetera.

Marshalling

Marshalling and unmarshalling is the transformation of parameters and return values into a neutral data format forth and back.

The goal of marshalling is openess and access transparency.

XML-RPC

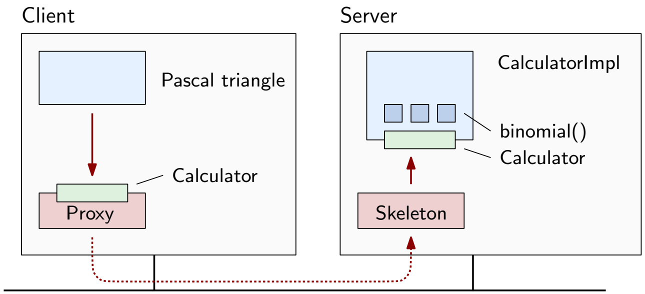
- A RPC protocol based on XML over HTTP.
- Developed by UserLand Software and Microsoft (1998).
- Later evolved to the SOAP protocol, used by W3C web services. A similar protocol is JSON-RPC.

```xml
<!-- A XML-RPC request -->
<?xml version="1.0"?>
<methodCall>
  <methodName>strlen</methodName>
  <params>
    <param> <value><string>hello</string></value> </param>
  </params>
</methodCall>
```

```xml
<!-- A XML-RPC response -->
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param> <value><int>5</int></value> </param>
  </params>
</methodResponse>
```

XML-RPC demo: Pascal's triangle

Output a Pascal's triangle, but computation of binomial coefficients is outsourced to an XML-RPC service.

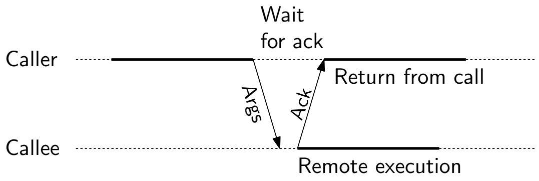


- Server: Define the interface `Calculator`, implement it on the server side, e.g., in `CalculatorImpl`, and launch an XML-RPC server that provides this as an XML-RPC service.
- Client: Tell the XML-RPC client library to connect to the server and provide a proxy object for the interface `Calculator`. The rest is behind the scenes.

Variations of RPC

Asynchronous call:

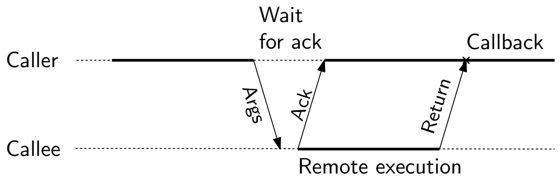
- If the remote procedure has no result or we do not care.
- Server immediately sends acknowledgment rather than after execution.
- **One-way RPC**: Do not even wait for ack at expense of reliability.



Variations of RPC

Deferred synchronous RPC:

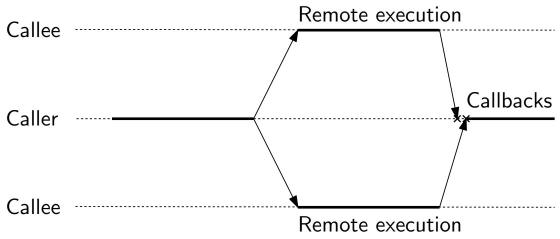
- Immediate ack, but returned values are sent later.
- Often returned values are signaled via a callback function. Alternative: Polling.
- For instance for parallel communication with server(s).



Variations of RPC

Multicast RPC:

- Sending RPC request to a group of servers.
- Return is signaled by callback by each server.
- Waiting for first callback versus waiting for all.
  - Multicast for fault tolerance: Wait for first callback.
  - Multicast for parallelization: Wait for all (and merge results?).



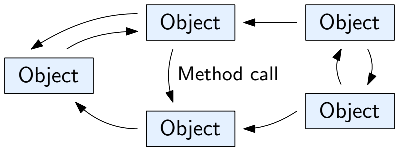
Section 4

Object-based architectures

Object-based architecture

Object-based architecture

An object-based architecture consists of a distributed set of objects that interact via method calls.



Encapsulation like in OOP:

- Objects encapsulate data, the object's state.
- Objects provide methods that operate on the state.

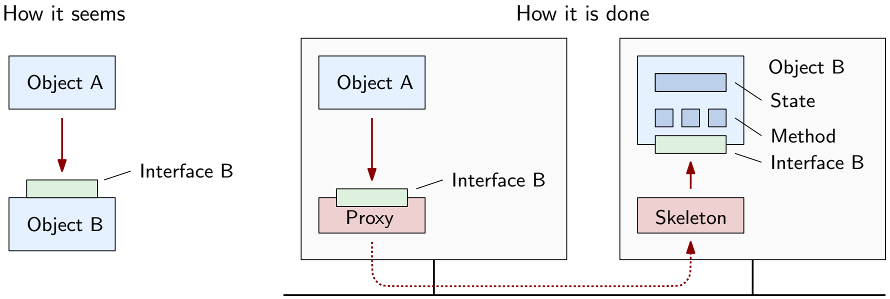
Method calls can take place over the network.

- But thanks to distribution transparency provided by a middleware, we (mostly) do not care.
- We speak of distributed objects.

Distributed objects

An interface defines the set of methods to access an object.

- The interface hides the actual implementation.



Assume Object A calls a method of an Object B on a different node.

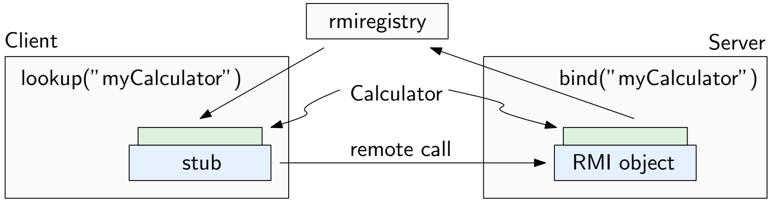
- For Object A all that matters is the interface of B.
- A proxy of Object B, which implements the same interface, is loaded at the location of Object A. The proxy then performs a remote procedure call.
- Proxy and skeleton provide the illusion of co-located objects.

RMI

RMI (Remote Method Invocation) is the foundation for distributed object architectures in Java.

- Remote objects are registered and looked up at the rmiregistry; it is a naming service.
- An interface extends `java.rmi.Remote`.

The calculator demo:



Other distributed object technologies

RMI in particular is a Java technology. But there are many others that are language-agnostic, including:

CORBA	The Common Object Request Broker Architecture originated in 1991 is a widely known, complex distributed object architecture.
Ice	The Internet Communication Engine is influenced by CORBA, but smaller and less complex. It originates in 2004 but its source code is available in GitHub since 2015.
COM	The Component Object Model was introduced by Microsoft in 1993 and is the basis for OLE, ActiveX, COM+ and so on. DCOM is basically COM over network.

They have in common to define an IDL (Interface definition language):

- In this language the interface is defined.
- An IDL compiler then generates skeleton and stub code for the target programming languages, like C++, Python, Java, et cetera.

Such architectures typically have a couple of management services:

- Naming and object directory services
- Load balancing and fail over mechanisms

References I

[Gri16] Radu Grigore. *Java Generics are Turing Complete.* 2016. arXiv: 1605.05274 [cs.PL].

[vT23] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems.* 4th ed. Jan. 2023. ISBN: 978-9081540636.

06: SOA, Web Services & REST

Network Oriented Software

Stefan Huber
www.sthu.org

Dept. for Information Technologies and Digitalisation
FH Salzburg

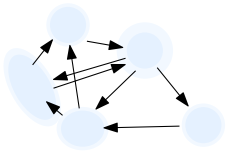
Summer 2023

Service-Oriented Architecture

Service-Oriented Architecture (SOA) separates an application into individual services.

A service:

- It is a self-contained entity; it can live and run by itself.
- A service can make use of other service.
  - It is still self-contained; the other service may actually be run by a different organization.

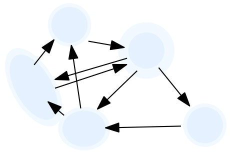


Service-Oriented Architecture

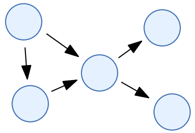
Service-Oriented Architecture (SOA) separates an application into individual services.

A service:

- It is a self-contained entity; it can live and run by itself.
- A service can make use of other service.
  - It is still self-contained; the other service may actually be run by a different organization.



bad: low cohesion, many depenencies, high coupling



good: high cohesion, strong interfaces, clear dependencies, loose coupling

- It is a black box to the consumer: strong encapsulation, high cohesion and loose coupling is emphasized. Interface definitions of services are a central aspect.

Examples

Example: Web shop selling goods

- Application logic: Selecting ordered item, registering, checking delivery channels (e.g., e-mail), checking payment.
- Payment could be a separate service, maybe run by a different organization.
- Delivery channel handling could be a separate service.

Example: Traveling website

- Multiple services of different airlines, hotels, car rental companies.
- An Expedia-like service uses the above services to provide a service to the user that does the aggregation, comparison, composition of a travel.

Misconceptions and fuzz

SOA has become a well-known and somewhat divisive acronym. If one asks two people to define SOA one is likely to receive two very different, possibly conflicting, answers.

– msdn.microsoft.com

[SOA is defined as] a loosely-coupled architecture designed to meet the business needs of the organization.

– msdn.microsoft.com

Wrong myths about SOA:[1]

- SOA would require web services.
- SOA would be new and revolutionary.

[1] https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx#_Introduction_to_SOA

Sharpening terms

Component:

- Unit of software that is independently replaceable and upgradable.

Library versus Services:

- Library is component that are linked into a program and called using in-memory function calls.
- Services are out-of-process components with communication like RPC or web service request.
  - Hence, they are independently deployable.
  - Also, we need more explicit component interfaces and breaking encapsulation is harder (and does not rely on discipline).

The Open Group SOA Working Group

The Open Group SOA Working Group provides the following definitions.[2]

Definition 1

- Service-Oriented Architecture (SOA) is an architectural style that supports service-orientation.
- Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

Definition 2

A service

- is a logical representation of a repeatable *business activity* that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports),
- is self-contained,
- *may be* composed of other services, and
- is a black box to consumers of the service.

[2] `https://web.archive.org/web/20160819141303/http://opengroup.org/soa/source-book/soa/soa.htm`

Advantages of SOA

SOA fosters and uses

- high cohesion, loose coupling
- open standards (e.g., XML)
- network distribution

This results in these advantages:

- Platform independence (openess)
- Simpler integration:
  - It reduces essentially a service specification.
  - Complexities are isolated and hidden behind the service specification, which makes integration easier.
- Parallel development of applications.
- Implementation details of a service do not impact other applications or services.

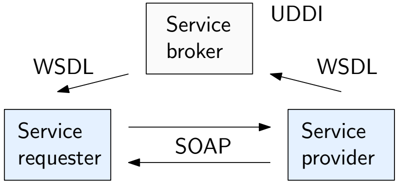
W3C Web Services

The W3C defines a web service as follows:[3]

Definition 3

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.

- It has an interface description in a machine-processable format (specifically WSDL).
- Other systems interact with the Web service [...] using SOAP-messages [...].



UDDI (Universal Description, Discovery and Integration):

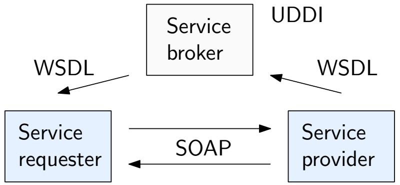
- A world-wide web service registry in the internet.
- In contrast, WS-Discovery is a multicast discovery protocol for a local network.

[3] https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice

Web Service operations

There are three kinds of interactions in a Web Service architecture:

- Publish:
  The provider publishes a service description via the service registry.
- Find:
  The requester retrieves a service description from the registry with its binding and location description.
- Bind:
  The requester initiates interaction with the service at runtime.



SOA versus microservices

- SOA services are relatively broad and on enterprise-level, like an entire payment processing service.
- Microservices are similar, but rather on application-level.
    - Lightweight communication, like resource-oriented HTTP API. “Smart endpoints and dumb pipes” like in the UNIX philosophy.
    - Naturally goes with DevOps and continuous development.

Resource-based Architectures

Issues with services in SOA:

- Service composition easily turned into a integration nightmare.
- E.g., SOAP client and server are still quite tightly coupled by a contract (WSDL).

REST provides a different approach:

- Model: A distributed system interpreted as a collection of resources.
- Resources can be created, read, updated, deleted (CRUD).
- "Browsing" resources rather than invoking RPCs.
- Stateless communication, self-contained messages, uniform interface.
- A client can start using a REST API with zero knowledge about the API, in contrast to SOAP.
  - There is an entry point and messages contain information (URIs) how to keep going.

REST is an architectural style, not a protocol.[4]

[4] It may not make sense, but one could theoretically implement a REST architecture over SOAP.

RESTful architecture

REST means REpresentational State Transfer

- Resources have states. A state has a representation (e.g., XML or JSON).
- REST is about the transfer of representations of states (of resources).

Basic principles:

- Client-server communication
- Stateless
  The messages sent to and from the service are fully self-described. There is no session state; the server forgets about the client after command execution.
- Cacheable
  There might be a layer between client and server that caches responses. Retrieving a state twice gives the same result (idempotence).
- Uniform interface
  There is a single uniform interface to all components. There is a single naming scheme (URIs) for resources. HATEOAS[5]: Only the initial URI but no further knowledge is necessary for access; further information is dynamically given through hypermedia.

[5] Hypermedia As The Engine Of Application State. `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`

REST via HTTP in vivo

The REST operation are mapped to HTTP operations.

- Hence, we can use `curl` to actually demonstrate REST on the command line.
- The media type is often JSON or XML.
- The response contains URIs that tell us how to proceed.

```
curl "https://api.github.com"
{
  "current_user_url": "https://api.github.com/user",
[...]

curl "https://api.github.com/users/torvalds"
{
  "login": "torvalds",
  "id": 1024025,
[...]
```

Mapping to HTTP methods

CRUD	HTTP Method	Usage
Create	POST	Create a subordinate resource *in a collection*
Read	GET	Retrieve data from the server
Update	PUT	Replace an existing resource (or, more rarely, place a new one)
Delete	DELETE	Remove a resource on the server

Note that the operations GET, PUT, DELETE are idempotent by RFC 7231.[6]

- The intended effect of multiple identical requests with that method is the same as the effect for a single such request. It is not about status code.
- Idempotent operations can be simply repeated after communication failure.
- RFC 7231 does not really define GET as being idempotent, but rather “safe”, meaning read-only. Hence, safe methods are trivially idempotent, too.
- POST is not idempotent!

6 `https://tools.ietf.org/html/rfc7231`

Parameters and example requests

Parameters are passed in either of two ways:

- Path parameter: `http://api.example.com/customer/9876`
- Query parameter: `http://api.example.com/customer?id=4563` (or as part of the HTTP header, if it's a POST request)

Requests with different types to the same path can call different methods:

- A POST request to `http://api.example.com/customers` would create a new customer in the collection resource `customer`, e.g., `http://api.example.com/customer/5123`.
- A GET request to the same URI and path could retrieve data about the `customers` collection.
- A GET request to `http://api.example.com/customer/5123` will retrieve data about the specific customer.
- A PUT to that URI would replace (or create) the customer 5123.
- A DELETE to that URI would remove that customer.

Spring Boot

Spring Boot is a Java framework to create web services.

- Part of the larger Spring application framework.
- Embedding into application servers like Apache Tomcat:
  - Tomcat is a Java web server that runs Java code on the server side. That is, Tomcat is a Java servlet engine, i.e., it runs server-side applications using the Java servlet API.
  - Those web applications can be packages as WAR files, however, we do not need to with Spring Boot.
- Convention over configuration to enable rapid application development.
  - Start configuration (POM files) for the Maven build system
  - Automatic configuration of Spring
  - No code generation and no required XML configuration
- Very popular in the Java ecosystem:
  - Wide range of tools.
  - Integration provided by many IDEs.

SpringBootApplication

The following code snippets, which are taken out of the provided RestfulWebServer and WebServiceClient code examples, use the Spring Boot framework.

The `@SpringBootApplication` annotation is used to mark a class with a `main()` method as the application.

- It is a shorthand[7] for other annotations that enable auto configuration of the framework and a component scan.
- `SpringApplication.run()` starts the web application.
- The framework takes care of the rest, such as starting the application server and offering the services.

```java
@SpringBootApplication
public class RestfulWebServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(RestfulWebServiceApplication.class, args);
    }
}
```

7 https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/using-boot-using-springbootapplication-annotation.html

RestController and Requests

The `@RestController` annotation marks a class as a service when the classpath is scanned by the framework.

```java
@RestController
public class MessageController {
```

Within the controller, `@RequestMapping` tells which method handle which HTTP request.

- It is a routing information from requests to Java methods.
- A request here is specified by the HTTP method and the REST resource path.
- Test it with `curl http://localhost:8080/hello-nos` or `http http://localhost:8080/hello-nos`

```java
    @RequestMapping(method = RequestMethod.GET, path = "/hello-nos")
    public String helloNos() {
        return "Hello NOS";
    }
```

Requests and parameters

Another annotation to achieve this is `@GetMapping`.

- Similar annotations are available for all request types.[8]

Method parameters can be marked as query parameters with `@RequestParam`.

- Test with `curl "http://localhost:8080/hello-name/?name=Alice"`

```
    @GetMapping("/hello-name")
    public String helloName(@RequestParam String name) {
        return "Hello " + name;
    }
```

Path parameters are are marked with `@PathVariable`.

- That must also be part of the mapping path using curly braces.

```
    @GetMapping(path = "/hello-name-path/{name}")
    public String helloWorldPathVariable(@PathVariable String name) {
        return "Hallo " + name;
    }
```

[8] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html

Default values and optional parameters

A default value can be provided for a parameter.

```
    @GetMapping("/hello-name-default")
    public String helloNameDefault(@RequestParam(defaultValue = "NoName") String name) {
        return "Hello " + name;
    }
```

A parameter can be marked as optional.

- If such a parameter is not passed to the resource, it is null.

```
    @GetMapping("/hello-name-optional")
    public String helloNameOptional(@RequestParam(required = false) String name) {
        return "Hello " + name;
    }
```

Parameter aliases and object return values

If a parameter should have a different name in the URI than in the method, it can be given an alias.

```java
    @GetMapping("/hello-name-id")
    public String helloNameId(@RequestParam("id") String name) {
        return "Hello " + name;
    }
```

A method may return an object, which is translated to JSON.

```java
    @GetMapping("/hello-name-id")
    public String helloNameId(@RequestParam("id") String name) {
        return "Hello " + name;
    }
```

Other request types

```java
    @PostMapping("/persons")
    public MessageBean createPerson(@RequestBody String name) {
        //Some Creation operation
        return new MessageBean("Person " + name + " created");
```

```java
    @PutMapping("/persons/{id}")
    public MessageBean updatePerson(@RequestBody String name, @PathVariable Long id) {
        //Some Update operation
        return new MessageBean("ID: " + id + ", Person: " + name + " updated");
```

```java
    @DeleteMapping("/persons/{id}")
    public MessageBean deletePerson(@PathVariable Long id) {
        //Some Delete operation
        return new MessageBean("ID: " + id + " deleted");
```

Test it on the command line:[9]

```
curl -X POST -H 'Content-Type: application/json' -d 'Harry' http://localhost:8080/
    ↪ persons
http POST localhost:8080/persons "name=Harry"       # Sends JSON request data
```

[9] In real code, of course, passes a JSON structure that encodes a persons data. And a proper RESTful web service would return information about its ID and a URI to retrieve the person.

Getting started

Literature:

- There are many good tutorials, primarily the one from the Spring website: `https://spring.io/guides/gs/spring-boot/`.
- Craig Walls. *Spring Boot in Action*. 1st ed. Manning Publications, Jan. 2016, p. 264. ISBN: 978-1617292545

Starting with an application:

- Use the Spring Initializr here: `https://spring.io/guides/gs/spring-boot/#scratch`.
- Use the integration into your favorite IDE, if available.
- Clone an existing project and adapt it, e.g., the one from the lecture notes or the one from the spring.io tutorial.

REST versus RPC

Very often, any “web API with URLs” is called REST. That is a misconception:

- If your API feels like RPC then it is not REST. REST is not just HTTP, nice URLs or CRUD.

REST is about resources. RPC is about procedure calls.

- The URI refers to resources. If it contains a verb, like `example.com/addCar`, it feels like a procedure. You probably are thinking the wrong way.

A RESTful API is about HATEOAS:

- Loose coupling by using the API with zero prior knowledge.
- See `https://spring.io/guides/tutorials/rest/` on how Spring Boot supports link creation.

References I

[Wal16] Craig Walls. *Spring Boot in Action*. 1st ed. Manning Publications, Jan. 2016, p. 264. ISBN: 978-1617292545.

Client of a web service

The `org.apache.http` package contains methods that let an Java application consume web services.[10][11]

- This example uses the `CloseableHttpClient` class to communicate with the server.
- Also supports state management (sesions, cookies), authentication, caching, multi-threaded request execution, HTML forms, and more.
- Use `curl -sv` or `http -v` to see the HTTP protocol working.

There is a class for each request type, e.g. `HttpGet`.

- Its constructor takes the URI and path to the resource that should be called.
- The method `execute()` of `CloseableHttpClient` sends the request, and the response can be stored in an `CloseableHttpResponse` object.

```
        HttpGet get = new HttpGet(serviceProvider + "//persons?name=Harry");

        // Sends the request and stores the response
        CloseableHttpResponse response = httpClient.execute(get);
```

10 `https://hc.apache.org/httpcomponents-client-ga/tutorial/html/index.html`

11 Since Java 11, there is also a similar `java.net.http.HttpClient`. A much more basic class `java.net.HttpURLConnection` exists since Java 1.1.

Parameters and responses

Parameters for a POST request can be passed by setting the content entity.[12]

```java
post.setHeader("Content-type", "application/json");

CloseableHttpResponse response = httpClient.execute(post);
```

The service response contains, among other things, the message from the server.

- Here it is parsed as a JSON object.

```java
if (entity != null) {
    String result = EntityUtils.toString(entity);
    JSONObject json = new JSONObject(result);
    return json.getString("message");
}
```

12 https://hc.apache.org/httpcomponents-client-ga/tutorial/html/fundamentals.html#d5e95