

2018

Zusammenfassung Prog2

SEP 2018

PASCAL BRUNNER

1 Inhaltsverzeichnis

2	Error / Exception Handling	2
3	Tooling	5
4	Build Automation	9
4.1	Maven	9
5	Testing	12
6	Concurrency	16
7	Java Database Connectivity (JDBC).....	25
8	I/O Foundation and File IO	31
9	Java Networking	35
10	Graphical User Interfaces – GUI	45

2 Error / Exception Handling

Exception: Eine Exception ist ein Ereignis, das während der Ausführung eines Programms auftritt und den normalen Ablauf der Programmieranweisungen stört.

Sprachelemente:

- Throw
- Throws
- Try
- Catch
- Finally
- Assert

Defensives programmieren:

man geht davon aus, dass etwas schief gehen kann. Daher soll man sich Zeit nehmen, um gut und einfach zu programmieren. Hilfreich hierbei sind bspw. Design patterns. Verwenden von Unit Tests.

typische Fehlersituationen:

- Unkorrekte Implementation
 - o Verhält sich nicht, nach der gewünschten Spezifikation
 - Objektabfrage, welcher noch nicht erstellt wurde
 - o
 - Fehlerhafte Objektzugriff
 - o Zu grosser / kleiner Index
- Eine häufige Fehlerquelle ist eine zu hohe Komplexität des Programms

Ziel des Errorhandlings

- Problem so gut als möglich lösen
- Wenn möglich, dass Programm weiter ausführen mit einem möglichst kleinen Verlust
- Im schlimmsten Fall, soll das Programm terminieren ohne einen grossen Schaden

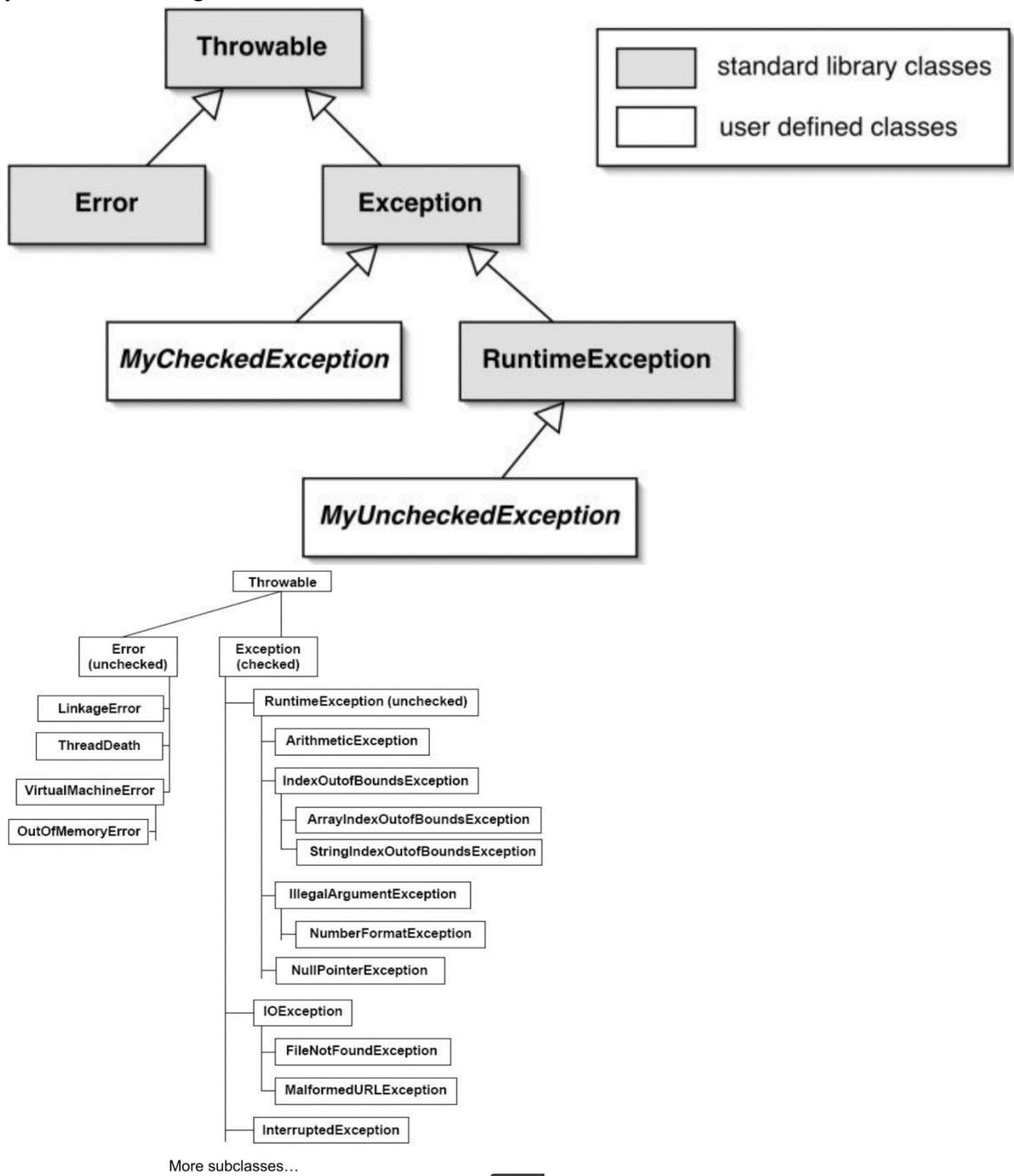
Beispiele

- Ungültiger Benutzerinput → Dem User eine (neue) korrekte Eingabe ermöglichen
- Zugriff auf inexistentes File → für einen neuen Filename fragen
- Drucker hat kein Papier mehr → unterbrechen und vom Benutzer zu verlangen, Papier nachfüllen
- Durch 0 dividieren → Programm terminieren
- Zugriff auf ein ungültiges Objekt → Fehlermeldung an den Benutzer
- Array index out of bounds → Programm terminieren, evtl. Array vergrössern
- Nicht genügend Speicher → Meldung an den Benutzer

Unchecked vs. Checked Exceptions

- Unchecked Exceptions:
 - o können jederzeit und überall geworfen werden
 - o Fehler welche ich nicht unbedingt erwarte
 - o Wiederherstellung des Programms wieder nicht erwartet
 - o Erbt immer von der RuntimeException
- Checked Exceptions:

- werden vom Compiler überprüft und müssen in der Methodensignatur angegeben werden.
 - Erbt immer von der Exceptionklasse
 - Einsetzen, wenn ich einen Fehler bereits kommen sehe
 - Einsetzen, wenn man noch etwas aufräumen muss und da Program weiterlaufen könnte
- Exceptionshandling soll so lokal als möglich passieren
- "throws" braucht es zwingend bei checked Exception. Kann jedoch auch für Unchecked Exceptions eingesetzt werden. Bei Unchecked Exceptions wird die Exception vom Kompilier jedoch **nicht** durchgesetzt



Handling von mehreren Exceptions

- ➔ Die Reihenfolge der Exceptions ist relevant! Zuerst generische, dann spezifische
 - Denn der erste Catch-Block greift

finally

- ➔ «*finally*» wird in jedem Fall ausgeführt, auch wenn das return-statement in einem try oder catch Block ausgeführt wurde

Bspw.

- Offene Dateien schliessen
- Netzwerkverbindungen
- Ressourcen freigeben

3 Tooling

Version Control System (VCS) – einige Terminologien:

- Repository
 - o Datenbank wo Dateien und metadaten (version history, branches, tags, ...) gespeichert sind
 - Working copy
 - o Ein Ort (jenachdem unterschiedlich) welches Kopien des aktuellen Files enthält. Es ist einzig ein Snapshot vom Repository zu einem bestimmten Zeitpunkt
 - Checkout
 - o Prozess für das Anlegen einer working copy vom repository
 - Commit
 - o Prozess damit Änderungen auf der Working copy auf das repository übernommen werden
- Bei Git wenn man eine lokale Kopie zieht, ist immer das komplette Projekt (inkl. sämtlichen Version) lokal abgespeichert
- o Commit: lokal speichern
 - o Push: auf Server speichern

Distributed Version Control Systems (DVCS)

- Jeder Computer hat eine locale Kopie des kompletten Repositories (meistens auch der working copy)
- Kein einzelnes autoratives Repository

Vorteile

- No single point of failure
- Offline-arbeiten möglich
- Schnell
- Gute Nachvollziehbarkeit
- Viele Möglichkeiten im Workflow
- Fördert Zusammenarbeit

Nachteil

- Gibt keine eigentliche letzte Version
- Keine Versionnummern
- Merging ist herausfordernd

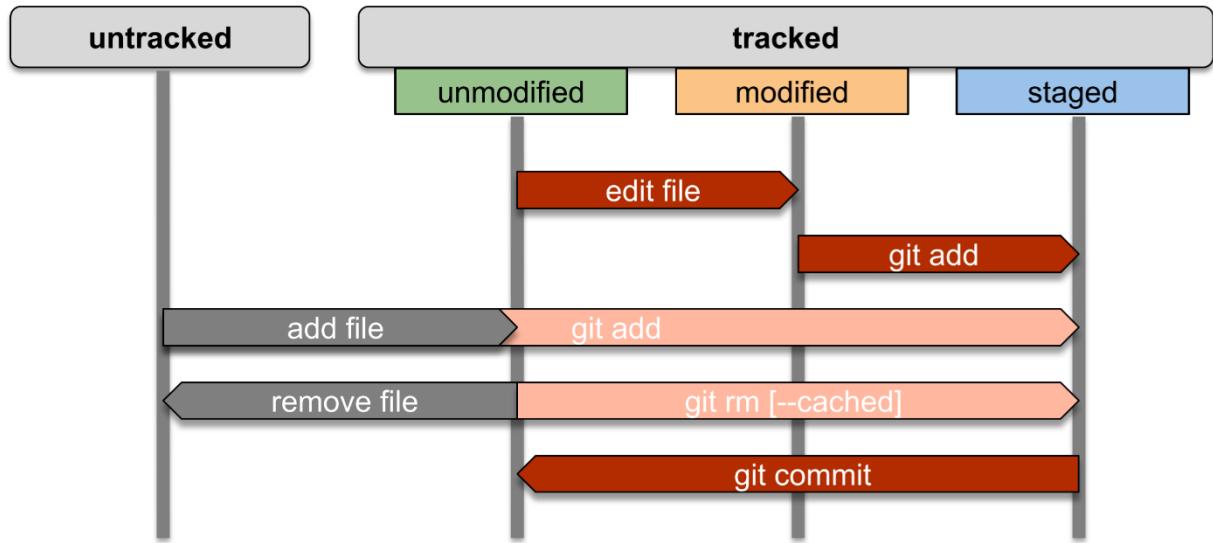
Die Versionnummer (revisionsID) sind normalerweise ein 40-Byte HashWert und sind dadurch eindeutig -> im Normalfall reichen 4-6 Zeichen um zu identifizieren

Jeder Branch (egal wie verzweigt) hat irgendwo einen Ursprung. Mit Merge geht man immer zum Ursprung

GIT

- GIT verfolgt den Inhalt und nicht die Datei.
- Es ist ein File System und handelt quasi wie ein stream of snapshots eines mini filesystems
- Hat Integrität, alles wird kontrolliert
- Löscht nichts, normalerweise wird nur etwas hinzugefügt.

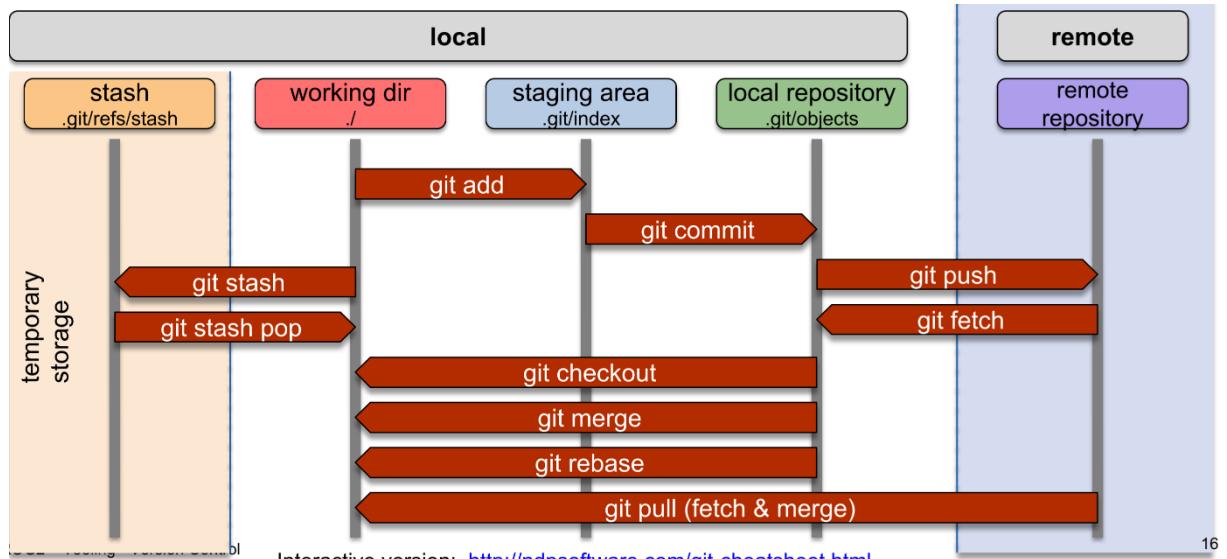
File Status Lifecycle



Alles was in der VCS beinhaltet werden soll, hat den Status tracked (untracked -> .gitignore)

Alles was staged ist, werden commited und gepushed. Denn man möchte nicht immer alles direkt commiten.

Git Buckets



Working dir: Dateien an welchen der Benutzer gerade arbeitet

Staging area: Werden Änderungen für den nächsten commit gesammelt

Respository: Ort wo alle versionen der getracked dateien gespeichert werden

Wichtigsten Git commands

local commands	branch / merge	remote commands
<ul style="list-style-type: none"> • git config • git init • git add • git status • git commit • git log • git diff • git tag • git revert 	<ul style="list-style-type: none"> • git branch • git merge • git checkout • git rebase • git stash 	<ul style="list-style-type: none"> • git remote • git clone • git fetch • git pull • git push

Git init [path/to/project/root] → Erstellt neues Repository
Default for project root path is the current directory

Git add . → stashed alle updated Files

Git commit -m "<Comment>" → Files im Stash commitment

-a does not add untracked files

Git push → in Repo pushen

Git rm <file> → untrack and delete file (→ delete)

Git rm --cached <file> → untrack file but keep it in working directory (→ untrack)

Git reset HEAD -- <file> → unstage

Git mv <source> <destination> → Inhalt wird nicht angepasst, nur die Referenz

Git mv README README.md == mv README README.md / git rm README / git add README.md

Git log → shows entire history

-10 → last 10 commits

--since=2.weeks → filtert by dates

--html/ → specific path

-Sfunction_name → specific content

Git diff → unstaged changes

--cached → staged changes

HEAD^ → last commit

<VersionNr> → specific version

Git branch dev → creates a new branch

Git checkout dev → replaces files in working directory with files of dev (switch branch)

Git checkout -b dev → git branch dev && git checkout dev

Git clone <URL> → creates copy of repo

Git fetch [<remote>] [<branch>] → retrieves info from a remote that is not in the local repo

Git pull [<remote>] [<branch>] (=fetch + merge) → fetch info from remote repository and merge active branch

Merging

1. Switch to target branch – master → git checkout master

2. merge source branch – dev → git merge dev

Tries to automatically combine all changes → creates a new commit and moves current branch (master) to it

Merging failed

Git status shows the conflicting files

1. fix the conflict

2. remove the separators lines

3. git add <file>

4. git commit

4 Build Automation

Verschiedene Tasks welche im daily business durchgeführt werden müssen, werden automatisiert.

On-Demand: läuft ein Skript oder mittels Button drücken

Scheduled: zu einer bestimmten Zeit bspw. Nacht

Triggered: bei einem bestimmten Event bspw: commit/push

Hauptziel

Qualitätssteigerung

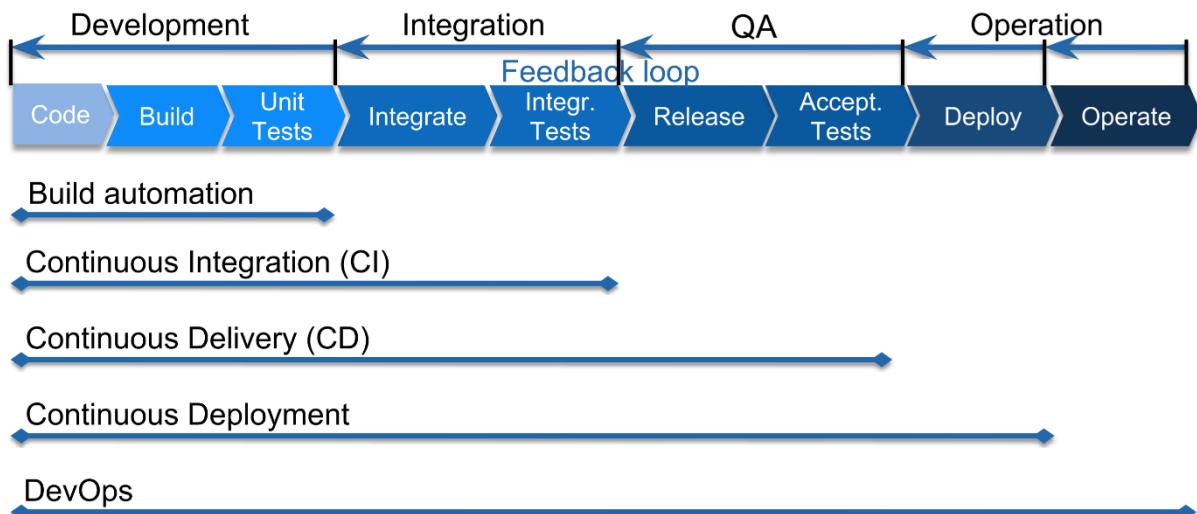
- Automatisiertes Testen
- Automatisiertes Code überprüfen
- History wird dokumentiert

Schnellere Einführung

- Reduziert der Prozess vom building to deployment
- Direktes Feedback
- Kürzere Zyklen

Verringert das Risiko

- Schnelle Fehlererkennung
- Klarheit über den aktuellen Softwarestatus



4.1 Maven

Konzepte

Der Lifecycle beschreibt einen klar definierten Ablauf von verschiedenen Phasen, welcher dafür da ist die Ziele zu erreichen

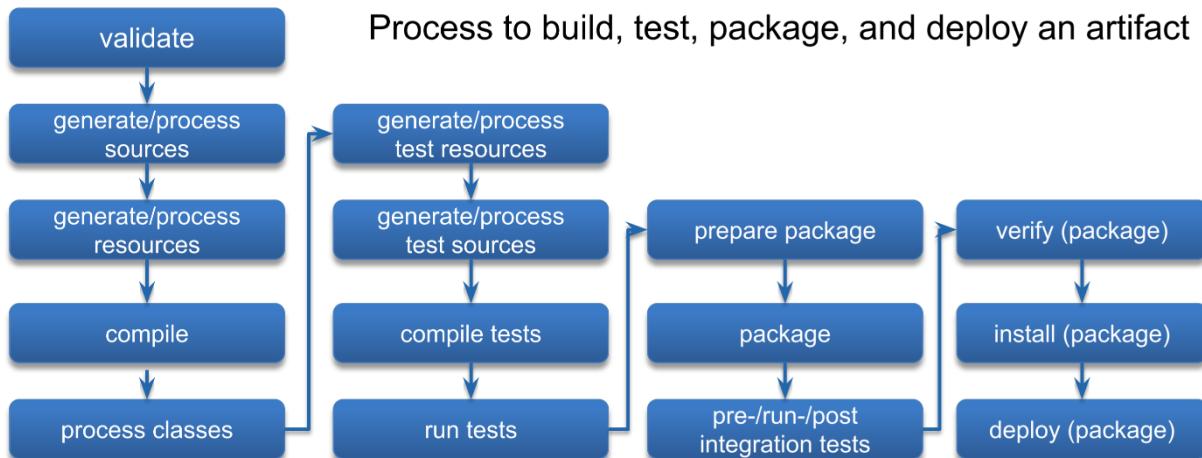
Maven hat 3 lifecycles:

- Default: bildet typw. Projekt Artifakten
- Clean: Räumt auf
- Site: bildet Dokumentation und Reporte

Das **Ziel** ist an eine bestimmte Phase des Lifecycles gebunden

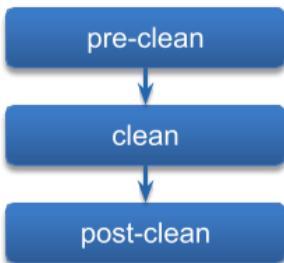
Plugins hängen an einer bestimmten Phase und definieren die Ziele

Default (build) Lifecycle

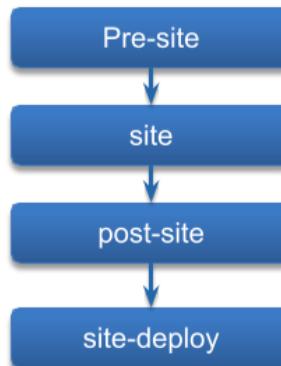


Clean and Site Lifecycle

Clean generated and temporary files



Build documentation and project reports



Mvn install → generiert und kompiliert, Test, Package, Integration-test, install

Mvn clean → clean lifecycle to remove generated artifacts

Mvn clean compile → clean lifecycle and when successfully finished invoke build until the compile phase

Mvn compile install (same as mvn install)

Mvn clean install → clean old builds and invoke install (see on top)

Mvn jetty:run → activates the jetty plugin and invokes the run phase to start the jetty web server

Mvn archetype:generate → generate basic project structure using archetype plugin

Projektstruktur

<code> \${basedir}</code>	Root folder of the artifact
└── pom.xml	Project Object Model description
└── src	Folder containing all sources
└── it	Integration tests
└── main	
└── config	Config files
└── db	Database files (e.g. SQL-Scripts)
└── java	Java Source files to be compiled
└── resources	Non-compiled source files
└── scripts	Script files for devs and admins
└── webapp	Web assets like JSP files, CSS, images
└── site	Files used for the project site
└── test	
└── java	unit test source code
└── resources	Non-compiled test source files
└── target	holds the generated artifacts
└── classes	Compiled class files
└── example-1.0-SNAPSHOT.jar	packaged project jar
└── test-classes	Compiled unit tests

POM

Project Object Model (XML File) enthält Modul Metadaten und Projektinformationen

Name und Version

Projekttyp

Source Code Locations

Dependencies (Abhängigkeiten)

Plugins

Dependency hinzufügen

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

5 Testing

Man testet um Fehler zu erkennen und nicht um die Korrektheit zu zeigen

Black-box Testing

Auch als Data-Driven oder input/output driven testen bekannt.

Man betrachtet nur bei einem Input den entsprechenden Output und kümmert sich nicht um das Interne Verfahren

White-box Testing

Auch als logic-driven testen bekannt.

Kümmert sich um die eigentliche Implementation

Principle 1 – Specification of Input and Output

Es braucht eine klare, verständliche und sehr genaue Beschreibung des erwarteten Outputs

Principle 2 – Separate Creation and Testing

Ein Programmierer sollte nicht seinen eigenen Code testen

Principle 3 – Completeness of Tests

Testfälle müssen sowohl unerwartete und ungültige, wie auch für erwartete und gültige Eingaben abdecken

Principle 4 – Testing is an investment

Testing ist ein wertvolles Investment für Zeit, Geld und Ideen

Principle 5 – Error Clusters

Die Wahrscheinlichkeit, dass in einem Abschnitt eines Programms mehr Fehler auftreten, ist proportional zur Anzahl der Fehler, die in diesem Abschnitt bereits gefunden wurden.

Unit Testing

- Auf Modulebene
- Isoliert
- Um das Verhalten einer Unit extern zu prüfen
- Schafft eine Art künstliche Umgebung
- Definiert Input und Output
- Überprüft den Output auf einen vordefinierten Wert

Limites

- Externes Testen garantiert nicht das korrekte Verhalten
- Gewisse Dinge sind schwierig isoliert zu betrachten (Datenbank, Konfig, Netzwerk)
- Wie kann man eine Klasse Testen das von anderen Komponenten abhängt

Mock Testing

- Simuliert Teile eines Verhaltens
 - o Methoden und Returnwerte für spezifizierte Parameter
 - o Anzahl Aufrufe

- Werden «Expectations» genannt
- System unter Test kann gut isoliert getestet werden, da Komponenten mit Mocks (Scheinobjekte) reproduziert werden können

```

/* Test an "Order" object that requests items from a "Warehouse" object (using jMock):
 - if the Warehouse has enough on stock, the Order gets filled and the Warehouse is decreased
 - if the Warehouse is low on stock, the Order isn't filled and the Warehouse remains unchanged */

public class OrderInteractionTester extends MockObjectTestCase {
    private static String TALISKER = "Talisker";

    public void testFillingRemovesInventoryIfInStock() {
        // configuration
        Order order = new Order(TALISKER, 50);
        Mock warehouseMock = new Mock(Warehouse.class);
        // expectations
        warehouseMock.expects(once()).method("hasInventory").with(eq(TALISKER), eq(50)).will(returnValue(true));
        warehouseMock.expects(once()).method("remove").with(eq(TALISKER), eq(50)).after("hasInventory");
        // exercise
        order.fill((Warehouse)warehouseMock.proxy());
        // verify
        warehouseMock.verify();      //verify expected behavior
        assertTrue(order.isFilled()); //verify state
    }
}

```

Pattern

1. Mock für das Interface oder Klasse erstellen
2. Das genaue Verhalten des Mocks spezifizieren
3. Mock in Standard unit testing verwenden bspw. JUnit -> wird als normales Objekt betrachtet
4. Verhalten verifizieren

Vorteile

Outside-in Design Methode möglich

Isoliertes Testen möglich

Einfaches testen auch für ressourcen gebundene APIs wie bspw. JDBC

Nachteile

Implementierung wird genau gespiegelt, was das Testen anfällig macht

Mocks können sehr komplex werden jeder Test definiert ein neues Objekt

Mockito Framework

- Behaviore driven
 - Whitebox testing
 - Verifiziert Korrektheit eines Verhaltens
- Mock-Objekte
 - Für Mock-Testing
 - Entferne «dependencies» und verhalten wir durch Mocks getestet
 - Track behavior of System under Test

Mockito Workflow

1. Create the mock object

```
Import static org.mockito.Mockito.*  
  
//create a Mock  
  
Song mock = mock(Song.class);
```

2. Specify the mock object

```
// Specify return value by wrapping call with when() method  
  
when(mock.getTitle()).thenReturn("Frozen");
```

3. Use the mock object

```
// execute tests  
  
box.addSong(mock);  
  
box.playTitle("Frozen");  
  
assertEquals("Frozen", box.getActualSong().getTitle());
```

4. Verify how the mock object is used

```
// verify behavior  
  
verify(mock, times(2)).getTitle();  
  
verify(mock).start();
```

- Mit *verify* kann man verifzieren ob eine Methode (mit angegebenen Parameter) tatsächliche aufgerufen wird. Falls nicht, dann erscheint eine Fehlermeldung
- Mit *times* kann man überprüfe wie häufig eine Methode aufgerufen wurde bspw oben: sollte 2 mal aufgerufen werden
- Mit *description* kann eine benutzerdefinierte Fehlermeldung ausgegeben werden, falls der Test failed
- Mit *timeout(ms)* kann man weitere Spezifikation machen
 - Verify(mockedList, timeout(100).times(2)).add("once"); → innert 100ms 2x aufrufen
- Mit *verifyZeroInteractions* kann man testen, dass keine Interaktionen stattgefunden haben
- Mit *inorder* kann die korrekte Reihenfolge gewährleistet werden
 - VerificationInOrderFailure -> geworfene Fehlermeldung
- Mit *thenThrow(new Exception())* kann man eine neue Exception werfen

Stubbing

Kann dem Mock (thenReturn()) mitteilen, was ausgegeben werden soll, wenn die Methode aufgerufen wird

Pattern

```
when(mock.someMethod()).thenReturn(object);
```

Default Values

Tbd – Default Values & Return Smart Nulls

Spy

Wird gebraucht für «echte Objekt» -> keine Stubs

Sollte gebraucht werden, wenn nicht das ganze Objekt gestubbed werden sollte

```
List list = new LinkedList();
//create a spy on the real object
List spy = spy(list);

//stub the size() method
when(spy.size()).thenReturn(100);

//add() is not stubbed. So it will use the real method
spy.add("one"); spy.add("two");

assertEquals("one", spy.get(0));
assertEquals(100, spy.size());
```

Annotations

Mocks und Spies können auch mit einer Annotation initialisiert werden

→ @Mock oder @Spy

Danach muss man dann MockitoAnnotations.initMocks() verwenden

```
public class MockAnnotations {
    //Create a Mock with an annotation
    @Mock Person mock;

    //Create a Spy with an annotation instead of
    //LinkedList spy = spy(new LinkedList())
    @Spy LinkedList spy;

    @Before
    public void setUp() throws Exception {
        //initialize the annotations in the class
        MockitoAnnotations.initMocks(this);
        when(mock.getLongName()).thenReturn("HansPeter");
    }

    @Test
    public void testMethod() {
        spy.add("Hello ZHAW");
        mock.getLongName();
    }
}
```

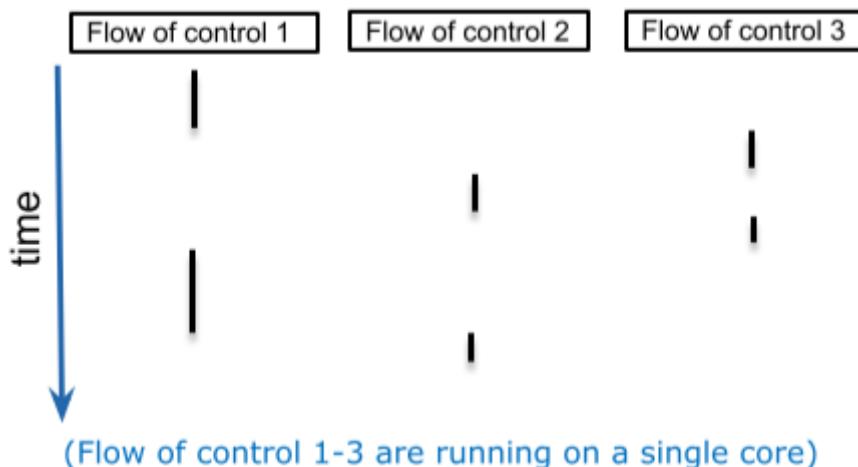
6 Concurrency

True Concurrency

Auf einem Computer mit mehreren CPU Kernen, kann jeder Kern einen Flow einzeln behandeln

Interleaving Concurrency

Es steht nur einen CPU Kern zur Verfügung, der CPU wechselt nun in unglaublicher Geschwindigkeit die einzelnen Flows auf einem Kern nacheinander, so dass der Benutzer das Gefühl hat, dass alles parallel läuft



Scheduling

Dieser ist für den Wechsel zwischen den Flows zuständig

Welcher Flow erhält den Kern? Mögliche Strategien:

- Nicht präventiv (kooperativ)
 - o Prozess gibt Kern freiwillig frei
 - o First come first served
 - o Shortest process next -> Batch processing (Stappelverwaltung)
- Präventiv (verdrängend)
 - o Scheduler kann einen Prozess unterbrechen (Zeit basiert)
 - o Round Robin, Multi-Level & Multi-Level-Feedback (Prioritäten basiert) → interaktives System
- Real Time
 - o Sehr enge Zeitvorgaben
 - o Rate Monotonic, Deadline scheduling -> Echtzeit-Systeme

Terms

Java lässt multi processing nicht zu

- Programm
 - o Sequenz von Anweisungen, geschrieben, um einen bestimmten Task zu erfüllen
 - o Wenn ausgeführt, braucht es einen oder mehrere Prozesse
- Process
 - o Ausführung eines Programms (oder Teil eines Programms)

- Zuweisung des Hauptspeichers und CPU Register, braucht CPU Zeit, Zuordnung Peripheriegeräte
- Thread («Faden»)
 - Ein Prozess besteht aus einem oder mehreren Threads

A program with multiple flows of control can be implemented in several ways:

- Program consisting of multiple processes (each with a single thread)
 - Program consisting of one process with multiple threads
 - Program consisting of multiple processes with multiple threads

 - Multiple processes (multi-tasking)
 - Läuft je auf einem separaten Speicherort
 - Es ist nicht möglich von einem Prozess in den Speicher eines anderen Prozesses einzugreifen
 - Inter Process Communication (IPC) verwendet einen speziell geteilten Speicherort oder Mechanismen wie pipes oder sockets
 - Prozesswechsel ist sehr teuer → Der ganze Prozess muss gespeichert und neugestartet werden
 - Multiple threads within a process (multi-threading)
 - Läuft auf einem allgemeinen Speicherort
 - Daten sind innerhalb der Threads aufrufbar
 - Threadwechsel ist billig → Prozessstatus wird nicht geändert
 - Es wechselt im Wesentlichen nur der program pointer zur neuen Position im Code
- Prozesse / Threads können unabhängig oder abhängig voneinander laufen
- Bspw. verwenden gemeinsame Ressourcen (Dateien, Variablen etc.)

Concurrency in Java

- Java ist ein single process system (ein Prozess pro JVM)
- Java Virtual Machine (JVM) unterstützt Multi-Threading
- Die Thread-Klass enkapsuliert die Funktionalität der Thread-Implementation
- JVM Runtime Model
 - Während der Initialisierung startet die Java-VM einen non-daemon Thread, der die statische Methode main(); aufruft; weitere Threads müssen vom Programmierer gestartet werden.
(Anmerkung: Einige Subsysteme wie RMI, AWT, etc. starten ihre eigenen Threads.
 - Die JVM läuft solange bis alle (non-deamon) Threads terminiert sind

Es gibt zwei Arten einen Thread in Java zu implementieren:

1. Extend Thread-Klass und run() überschreiben
 - a. Public class xyz **extends Thread**
 - b. Instanz eines Threads erstellen *Thread myThread = new Mythread();*
 - c. Thread starten *myThread.start();*
 - d. start() erstellt eine neue Thread-Laufzeit, führt den Code im Thread durch den Aufruf der run()-Methode aus und returned sofort.
 - e. Überschreiben der run()-Methode in der Klasse welche Thread extended
Public void run(){ //code to run in thread }
2. Das Runnable-Interface in einer separaten oder anonymen Klasse implementieren

- a. Class MyRunnable **implements Runnable** extends ...
- b. Überschreiben der Run-Methode public void run(){ //code to run in thread}

Methoden eines Threads:

- Start () → Startet den Thread
- Join() → aktueller Thread wartet bis der angegebene Thread beendet
- setName(String str) → setzt den Namen eines Threads
- String getName() → returned der Name des Threads
- setPriority(int prio) → setz Priorität eines Threads (vordefinierte Werte: MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY)
- [int] getPriority() → returned die Priorität des Threads
- [boolean] isAlive() → returned true wenn der Prozess noch läuft
- [Klassenmethode] sleep(int ms) → aktueller Thread schlafst für die Anzahl ms (Gibt frei für einen anderen Thread, welcher bereit ist)
- Static Thread currentThread() → returned eine Referenz zum aktuell ausgeführten Thread
- Static void yield() → Veranlasst das aktuell ausgeführte Thread-Objekt, vorübergehend anzuhalten und anderen Threads die Ausführung zu ermöglichen.

Folgende Methoden sollen **nicht** verwendet werden, da diese unsicher sind:

- stop() → stoppt einen laufenden Thread
- suspend() → suspendiert einen laufenden Thread
- resume() → continues einen suspendierten Thread

Status eines Threads:

- new → Thread wurde erstellt mit new(), wurde aber noch nicht gestartet
- running → Thread wurde einem CPU Kern zugewiesen
- runnable → Thread ist bereit einem CPU zugewiesen zu werden, welcher aktuell noch von einem anderen Thread besetzt ist
- blocked → Thread kann nicht weiterfahren, da es auf einen Event wartet (sleep, input, ...)
- terminated → run() wurde beendet

Wie terminiert ein Thread?

- Unbehandelte Exception während run()
- Thread.stop() → unsicher und nicht empfohlen
- System.exit() → normalerweise schlecht

Empfohlenes Terminieren:

1. run() soll terminieren
2. eine Flag-Variablen verwenden um einen run-loop zu beenden
 - a. Damit man sicherstellen kann, dass der Thread den korrekten Wert nimmt, deklariert man die Variable als volatile

```
volatile boolean doContinue = true;
public void run() {
    while(doContinue) { // run-loop
        // do work
    }
}
```

Thread synchronisation ist zwingend, wenn:

- Mehrere Threads auf die gleiche Ressource greifen
- Kooperation / Koordination von Threads

Dabei unterscheidet man zwischen:

- Mutual Exclusion (wechselseitiger Ausschluss) von geteilten Ressourcen → max. ein Thread greift auf die Ressource zu
- Condition Synchronization (Zustandsynchronisation) für Kooperation → Der Thread wartet auf eine bestimmte Bedingung (bspw. Element verfügbar)
- ➔ Oftmals kann man nicht verhindern, dass mehrere Threads auf die gleiche Ressource zugreifen, aus diesem Grund wird oftmals auf Mutual Exclusion gesetzt

Mutual Exclusion

- Sicherstellen, dass die «critical sections» nicht gleichzeitig mehrfach verwendet werden
- Verwenden Sie eine geeignete Synchronisation, um den gegenseitigen Ausschluss im kritischen Bereich sicherzustellen.
 - Mit *synchronized* kann man den kritischen Teil markieren^

Synchronized Method

```
class Account {
    private int saldo = 0;
    ...
    public synchronized void changeSaldo(int delta) {
        this.saldo += delta;
        System.out.println(delta);           critical section
    }
}
```

The whole **method** `changeSaldo()` is executed
mutually excluded

Remark: Current object (`this`) is implicitly used as Monitor-Object

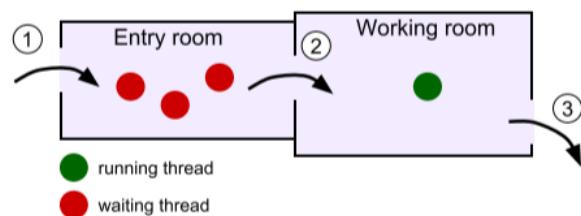
Synchronized Block

```
class Account {
    private int saldo = 0;
    ...
    public void changeSaldo(int delta) {
        synchronized (this) {           Monitor-Object
            this.saldo += delta;     (see later)
            System.out.println(delta);
        }
    }
}
```

Code in the **marked block** `{..}` is executed
mutually excluded

Wie funktioniert «synchronized»?

- Jedes Objekt oder Klasse hat genau einen Monitor
 - Monitor = sperren für exklusives Zugriffsrecht
- Analogie: Warteraum beim Arzt → Maximal ein Thread kann den Monitor besitzen, die anderen Threads müssen im «Warteraum» warten
- 1. Thread muss Monitor erwerben=> Eingangsraum eintreten → synchronized wurde aufgerufen
- 2. Monitor wurde erfolgreich übergeben, wird gesperrt und tritt in working room ein → synchronized (Code-) Block wird aufgerufen
- 3. Verlässt Monitor → verlässt working room → verlässt synchronized (Code-) Block



- Start des synchronized (Code-) Block: Monitor erwerben
 - o Falls bereits besetzt → im Eingangsraum warten
 - o Andernfalls → in Eingangsraum gehen
- Während dem synchronized (Code-) Block: Mutual exclusion garantiert
 - o Der Thread besitzt den Monitor
 - o Kein anderer Thread kann in den Arbeitsraum eintreten
- Ende des synchronized (Code-) Block: verlässt den Monitor
 - o Falls andere im Eingangsraum sind, wird einer der Monitor beanspruchen

Monitors sind rekursiv

```
class myClass {
    public synchronized void methodA() {
        ...
    }
    public synchronized void methodB(){
        ...
        this.methodA();
    }
}
```

- Thread blockt nicht, weil der Monitor bereits im Besitz ist

Wichtig – trotzdem sollte versucht werden von einem synchronized (Code-)Block nicht einen anderen synchronized-Aufruf getätigigt werden → möglicher **Deadlock**

```
class FooBar {
    public synchronized void foo() { ... }           ← Object-Lock (this)
    public synchronized static void bar() { ... }     ← Class-Lock (FooBar)

    public void fooBlock() {
        synchronized (this) { ... }                   ← Object-Lock (this)
    }
    public void barBlock() {
        synchronized (FooBar.class) { ... }          ← Class-Lock (FooBar)
    }
}
```

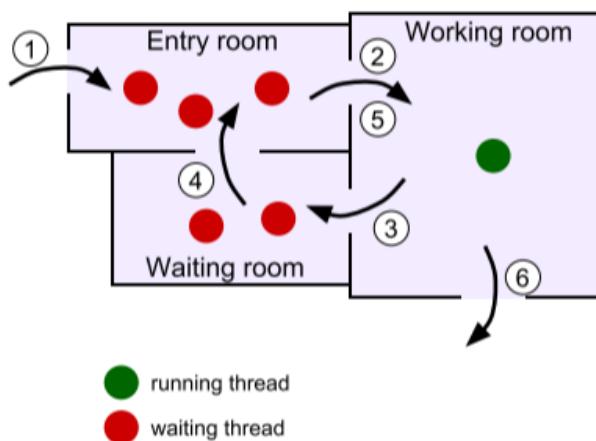
- Objekt-Locks
 - o Verwendet die angegebene Objektinstanz als Monitor (bzw. implizit bei synchronisierten Methoden)
- Class-Locks
 - o Verwendet die angegebene Klasse als Monitor (bzw. die aktuelle Klasse als static Methode)

Producer-Consumer Problem

- Threads müssen evtl. zusammenarbeiten

Type 1 – Producer und Consumer Thread teilt nur einzelnes Datenobjekt bzw. schickt nur ein Signal (keine Daten) → Bsp. Uhr (Sekunden, Minuten, Tick):

- Der Producer-Thread erzeugt ein Datenobjekt für den Consumer
 - Consumer-Thread ruft Datenobjekt ab, wenn es bereit ist
 - Consumer-Thread muss möglicherweise warten, bis das Datenobjekt bereit ist
 - Der Producer-Thread muss möglicherweise warten, bis das Datenobjekt verarbeitet wurde (andernfalls könnten Datenobjekte verloren gehen, wenn der Verbraucher-Thread zu langsam ist)
- Um dies zu lösen ist ein zusätzlicher Warteraum im Monitoring-Konzept notwendig



1. Try to acquire monitor: Enter entry room
→ synchronized is called
2. Monitor successfully acquired:
Got the lock, enter working room
→ enter synchronized block
3. Release monitor and wait for event:
Go to waiting room, release the lock
→ memorize position in synchronized block
4. Get event and try reacquiring monitor:
Re-enter entry room
5. Monitor successfully reacquired:
Got the lock, enter working room
→ continue at position in synchronized block
6. Release monitor: Leave working room
→ leave synchronized block

3. `public final void wait() throws InterruptedException`

- Der aufgerufene Thread ist “suspended” und geht in den Warteraum
 - Der Monitor ist wieder frei → Der nexte Thread aus dem Entry Room kann in den working room
4. `Public final void notify()`
- «weckt» einen zufälligen Thread aus dem Waiting room
 - Dieser geht in den Entry Room
 - Zum in den Working Room zugelangen, muss er den Monitor erhalten
 - Der aufgerufene Thread läuft weiter
 - Besitzt immer noch den Monitor
 - Mit `public final void notifyAll()` werden alle Threads aus dem Waiting Room geweckt und gehen in den Entry Room -> Danach identisch mit `notify()`

Bedingtes Warten in Java

- `Wait()` / `notify()` / `notifyAll()` → dürfen nur aufgerufen werden, wenn man den Monitor besitzt → aktueller Thread befindet sich im synchronized (Code-)Block
- Andernfalls wird eine `IllegalMonitorStateException` geworfen

Public final wait(long timeout) throws InterruptedException

- Erwacht Thread nach dem timeout in Millisekunden
- Wait(0) == wait() → endloses Warten

Thread 1: Consumer	Thread 2 : Producer
<pre>while(running){ // run-loop wait until item is provided; ← block shared object; retrieve & consume item; release shared object; → (1) wake-up producer; → (2) }</pre>	<pre>while(running){ // run-loop produce new item; wait until previous item is consumed; block shared object; provide item; release shared object; wake-up consumer; }</pre>

Type 2 – Producer und Consumer Thread teilt eine Warteschlange mit Datenobjekte:

- Producer füllt die Warteschlange mit produzierten Daten
- Consumer ruft die Daten aus der Warteschlange ab
- Consumer muss nur warten, wenn die Warteschlange leer ist
- Producer muss nur warten, wenn die Warteschlange voll ist

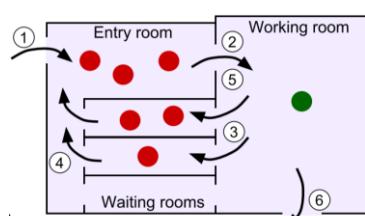
Thread 1: Producer	Thread 2 : Consumer
<pre>while(running){ // run-loop produce item; wait if queue is full; block shared object; provide item to queue; release shared object; wake-up consumer; }</pre>	<pre>while(running){ // run-loop wait if queue is empty; block shared object; fetch item from queue; release shared object; wake-up producer; }</pre>

Vorteil von Encapsulating bei Monitor Synchronisation

- Synchronisation ist an einer Stelle konzentriert
- Synchronisation kann von einem Spezialisten durchgeführt werden
- Verbleibende Programm muss sich nicht um die Synchronisation kümmern

Lock & Conditions

- Effizientes Warten einer spezifischen Bedingung
- Lock-Object werden für die mutual exclusion verwendet
- Condition-Objects werden für die Synchronisations-Bedingungen verwendet
 - o Jede Bedingung gehört genau zu einem Lock-Objekt
 - o Selektives Warten auf eine spezifische Bedingung



1. try to acquire lock: myLock.lock()
2. lock acquired: lock() successful
3. wait for cond X: condX.await()
4. thread in WR called condX.signal()
5. lock reacquired: continue after await()
6. release lock: myLock.unlock()

```

public class CondSyncQueue<E> {
    private Lock mutex = new ReentrantLock();
    private Condition notEmpty = mutex.newCondition();
    private Condition notFull = mutex.newCondition();
    private LinkedList<E> myList = new LinkedList<E>();
    private int capacity = 5;

    public void add (E item)
        throws InterruptedException {
        mutex.lock();           // enter critical section
        try { // condition 1: queue not full
            while(myList.size() >= capacity) {
                notFull.await();
            }
            myList.addLast(item);
            notEmpty.signal();
        } finally {
            mutex.unlock();      // exit critical section
        }
    }
}

```

Similar to Monitor with wait() & notify():

- **Lock** object replaces synchronized block or method
- **Condition** objects provide Wait-Sets (waiting rooms) for selective waiting (await) and wakeup (signal).

```

public E remove() throws InterruptedException {
    E item = null;
    mutex.lock();           // enter critical section
    try { // condition 2: queue not empty
        while (myList.isEmpty()) {
            notEmpty.await();
        }
        item = myList.removeFirst();
        notFull.signal();
    } finally {             // why finally?
        mutex.unlock();    // exit critical section
    }
    return item;
}

```

Class SynchronousQueue<E>

- Threads treffen sich bei der Warteschlange
- Producer wartet auf Consumer
- Consumer wartet auf Producer
- Hat einen Fairness-Parameter
 - o Mehrere Producer und mehrere Consumer warten nach dem FIFO-Prinzip

ReadWriteLock

- Write lock → Exklusives Zugriffsrecht (lesen und schreiben) auf eine geteilte Ressource
- Read lock → geteilter Zugriff (nur lesen) auf eine geteilte Ressource

		Already used locks	
		r-lock	w-lock
Requested locks	r-lock	access	block
	w-lock	block	block

```

class Account {
    private final ReadWriteLock rwLock = new ReentrantReadWriteLock();
    private final Lock readLock = rwLock.readLock(); // shared lock
    private final Lock writeLock = rwLock.writeLock(); // exclusive lock
    private int saldo = 0;
    ...

    // write & read access -> exclusive lock
    public void changeSaldo(int delta) {
        writeLock.lock();
        try {
            this.saldo += delta;
        } finally {
            writeLock.unlock();
        }
    }

    // read only access -> shared lock
    public void getSaldo(int delta) {
        readLock.lock();
        try {
            return this.saldo;
        } finally {
            readLock.unlock();
        }
    }
}

```

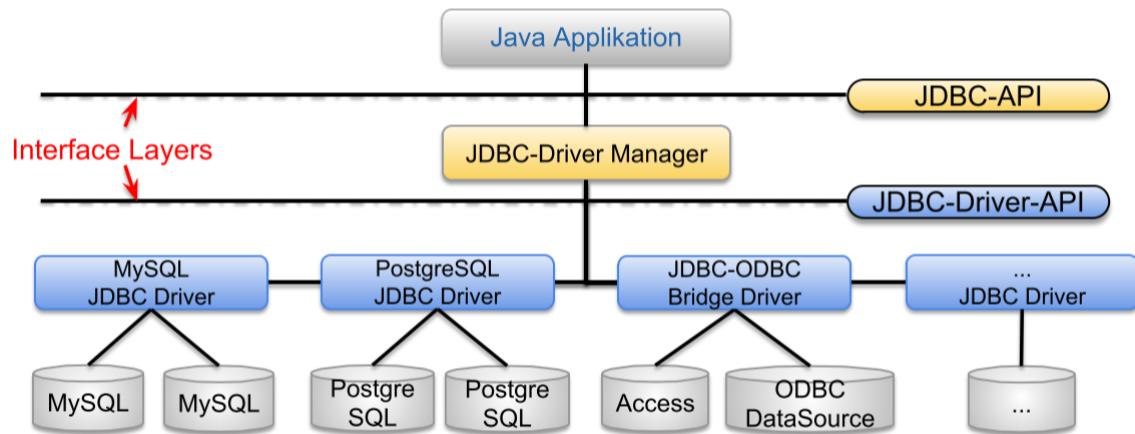
Deadlocks

Ein Deadlock kann nur unter den folgenden vier Bedingungen (alle zusammen) auftreten:

- Mutual Exclusion
 - o Jese Resource ist nur einmal verfügbar
 - Hold and Wait Bedingung
 - o Bereits blockierte Threads, sollten nochmals blockiert werden
 - Kein «Vorkaufsrecht» (Preemption)
 - o Eine blockierte Resource kann nicht durch das Betriebssystem weggenommen werden
 - Zirkuläres Warten → Das ist was man vorbeugen kann!
 - o Eine Anzahl von Prozesse sind im Wartezustand für eine Ressource, welche bereits von einem anderen Ausführer in der Kette blockiert ist
- ➔ Wann immer möglich sollte auf diese Bedingungen verzichtet werden.

7 Java Database Connectivity (JDBC)

- Standardisiertes Interface für relationale Datenbanken mit SQL-Statements für Java
- Das Programm basiert auf interlinked Objects
- Die Datenbank hat Daten-Tuples und keine Objekte
 - o Link zwischen Objekten und Daten ist nicht direkt sichtbar
 - o Das Mapping (Objekt von/zu Daten-Tuple) muss durch den Entwickler realisiert werden
- Object-Relational-Mapping (ORM) implementiert das Mapping zwischen Objekten und den Datenbank-Tabellen
 - o Alle ORM-Frameworks verwenden JDBC als ein Interface-Layer zum relationalen Datenbank-Management-System
 - o Obwohl man einen solchen Layer hat, macht es Sinn die Business Logik von den Daten zu trennen → Data Access Layer
 - o Bspw: Java Persistence API (JPA) oder Hibernate



JDBC-Driver Types

- Type 1
 - o Verwendet eine JDBC-ODBC-Bridge (Teil von JRE) zum auf die ODBC-DataSourcen zuzugreifen.
 - o Benötigt einen nativen ODBC-Driver (bspw. MS-Access)
 - o ODBC = Open Database Connectivity from Microsoft
- Type 2
 - o Braucht einen Plattformspezifischen Driver auf Seite des Clients
 - o Zusätzlich sind Programm-Libraries für die einzelnen Plattformen notwendig (bspw. PostgreSQL für Windows)
- Type 3
 - o Serverside Middleware kommt zum Einsatz -> Keine Installation auf der Clientseite notwendig
 - o Protokoll ist von der Middleware abhängig (bspw. ObjectWeb)
 - o Middleware sendet die Anfragen zur Datenbank
- Type 4
 - o Implementiert das ganze DB-Interface in Java mit einem JDBC-Driver
 - o Werden keine zusätzlichen Komponenten benötigt
 - o Plattformunabhängig

JDBC verwenden

Always the same basic steps:

1. Connect to SQL database
2. Execute SQL statements
3. Evaluate / Process results
4. Commit or Rollback DB modifications
5. Close Connection to database

DB0023_10 - Java Database Connectivity

12

→ Das jar-File des Drivers muss im Projekt-Classpath liegen

Verbindung zur Datenbank herstellen

```
import java.sql.*; // required to access JDBC classes
String url = "jdbc:postgresql://localhost:5432/mydb"; // DB URL
String user = "mustepet"; // DB username
String passwd = "secret"; // DB password
Connection con = DriverManager.getConnection(url, user, passwd);

• URL defines the DB to access
• Form jdbc:<subprotocol>:<dbSource>
  <subprotocol> name of the DB-Driver (+ access type if required)
  <dbsource> driver-specific path, to identify the DataSource
```

SQL-Query ausführen

- Um ein SQL-Query auszuführen ist ein Statement oder PreparedStatement notwendig!
- Das komplette Resultat der Abfrage, wird an eine Instanz der Klasse ResultSet geliefert

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM mytable WHERE x=500");
while (rs.next()) { // read the ResultSet
    System.out.print("Column 1 contains " + rs.getString(1));
}
rs.close();
st.close();
```

PreparedStatement optimiert das Laufzeit-Verhalten und beugt zudem vor SQL-Injection vor, in dem es automatisch Benutzerinhalt löscht

ResultSet gibt Zugriff auf die Resultat Tabelle. Mittels next() kann man auf die nächste Reihe zugreifen. Next() liefert true, falls das resultat mehrere Linien enthält. Ungebrauchte ResultSet sollte mit close() freigegeben werden

Statement ausführen

- executeQuery() → Daten einfügen (SELECT) → returned ein ResultSet
- executeUpdate() → für das modifizieren von Daten (INSERT, UPDATE, DELETE, CREATE, DROP, ALTER,...) → returned ein int-value (Anzahl betroffene Tupels)
- execute() → ausführen von gespeicherten Prozesse (bspw. Scripts)

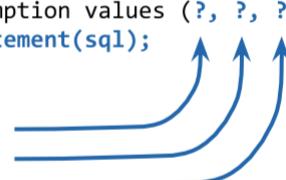
Beispiel Abfrage Daten

```
String dbURL = "jdbc:postgresql://localhost:5432/coffeeDB";
String user  = "mustepet", passw = "secret";
Connection con;
try {
    con = DriverManager.getConnection(dbURL, user, passw);
    Statement st = con.createStatement();
    String query = "SELECT programmer, sum(cups) as totalcups from coffeeConsumption " +
                   "GROUP BY programmer ORDER BY totalcups DESC";
    ResultSet result = st.executeQuery(query);
    while (result.next()) {
        String name = result.getString("programmer"); // using column name
        int cups = result.getInt(2);                  // using column index
        System.out.println("Name: " + name + " Total cups: " + cups);
    }
    result.close(); st.close();
} finally {
    con.close();
}
```

Beispiel Daten einfügen

```
Object[][] inserts = { {"Peter", "Espresso", 7},
                      {"Simon", "Cappuccino", 3},
                      {"Andy", "Espresso", 8} };
Connection con;
try {
    con = DriverManager.getConnection(dbURL, user, passw);
    String sql = "insert into coffeeConsumption values (?, ?, ?)";
    PreparedStatement st = con.prepareStatement(sql);
    int count = 0;
    for(Object[] insert : inserts) {
        st.setString(1, (String)insert[0]);
        st.setString(2, (String)insert[1]);
        st.setInt(3, (int)insert[2]);
        count += st.executeUpdate();
    }
    st.close();
} finally {
    con.close();
}
```

RQG2 – IO – Java Database Connectivity



Beispiel Daten updaten / löschen

```

// UPDATE
String sql = "UPDATE coffeeConsumption SET cups = ? " +
    "WHERE programmer = ? AND type = ?";
PreparedStatement st = con.prepareStatement(updateSql);
st.setInt(1, 4);
st.setString(2, "Simon");
st.setString(3, "Cappuccino");
int count = st.executeUpdate();
System.out.println("Records updated: " + count);

// DELETE
String sql = "DELETE FROM coffeeConsumption WHERE programmer = ?";
PreparedStatement st = con.prepareStatement(deleteSql);
st.setString(1, "Simon");
int count = st.executeUpdate();
System.out.println("Records deleted: " + count);

```

Transaction (commit & rollback)

- Eine Transaction ist ein SQL-Statement
 - o welche in einem Schritt («atomar») ausgeführt wird → con.commit()
 - o oder den Status welches es bereits zuvor hatte → con.rollback()
- Transaktionen sind ACID (Atomic, Consistent, Independet, Durable)
- Wenn auto-commit eingeschaltet ist (ist standardmäßig), dann wird jeder Statement-Ausführung automatisch committed
 - o Con.setAutoCommit(boolean)

```

Connection con;
try {
    con = DriverManager.getConnection(dbURL, user, passw);
    con.setAutoCommit(false); // disable auto-commit
    Statement st = con.createStatement();
    st.executeUpdate("... SQL update statement 1 ... ");
    st.executeUpdate("... SQL update statement 2 ... ");
    st.executeUpdate("... SQL update statement 3 ... ");
    con.commit(); // transaction (3 statements) is committed atomically
} catch (SQLException e) {
    con.rollback(); // an error occurred -> rollback
    System.out.println("Error: " + e.getMessage());
} finally {
    con.close(); // in any case close the connection
}

```

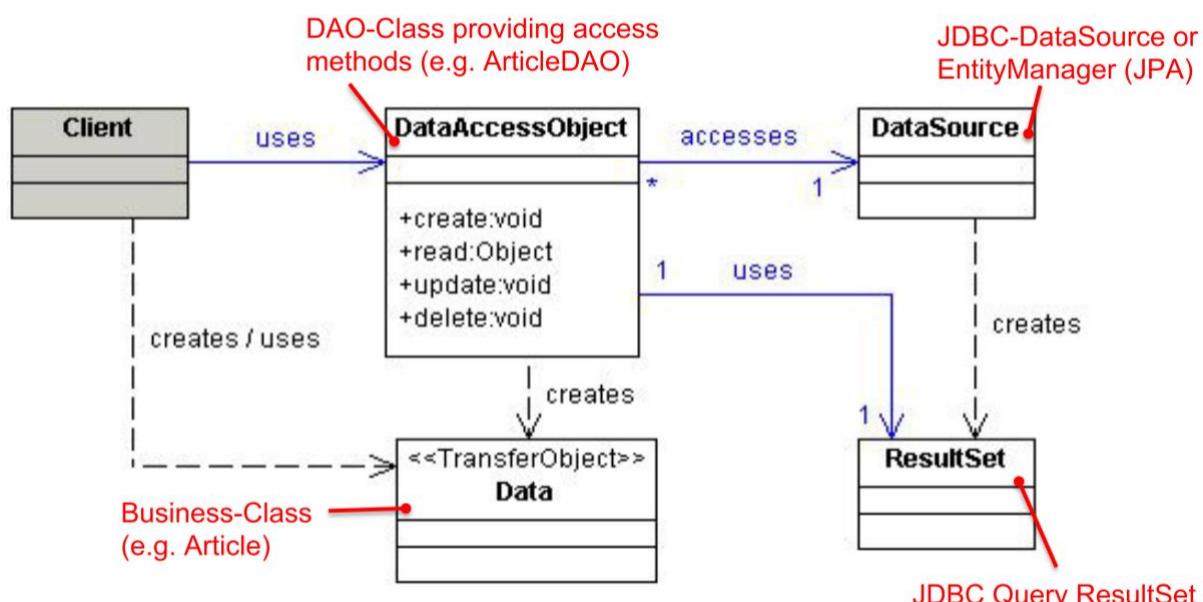
Multi-Tier Architecture

- Damit man eine saubere Trennung über die verschiedenen Bereiche hat, ist die Applikation in verschiedenen Tiers (Layers) gegliedert
 - o Jedes Tier hat eine klar definierte Funktion
 - o Der Zugriff auf untere Layers ist möglich, um die Implementierung zu vereinfachen
- Data Access Layer
 - o Fassade für den Zugriff auf den Data Store Tier
 - o Implementierung mit dem Data-Access-Object-Pattern

- Die Business Logik ist von der Datenhaltung getrennt und kennt dessen Technologie nicht um Daten zu persistieren
 - o Dies ermöglicht das Ersetzen von Daten ohne die Business Logic und Objekte zu ändern

Data Access Object Pattern

- Data Access Object (DAO) liefert ein Pattern für das Trennen zwischen dem Business Logic Tier (business objects) und dem persistieren von Daten (data store) → separation of concerns
- Business-Objekte greifen via DAO auf die Daten zu
- Im Idealfall muss bei einem Datenbankwechsel o.ä. nur das DAO geändert werden, nicht aber die Business Objekte
- Implementieren meist mind. CRUD-Methoden
 - o Create → insert(object)
 - o Retrieve → findAll(), findById(id), findBy...()
 - o Update → update(object)
 - o Delete → delete(id)



```

public class Article {                                Data Object
    private long id;
    private String name;

    public Article (long id, String name) { ... }

    public void setId(long id) { this.id = id; }
    public long getId() { return id; }
    public String getName() { return name; }
}

// Interface to be implemented by all ArticleDAOs
public interface ArticleDAO {
    public void insert(Article item);
    public void update(Article item);
    public void delete(Article item);
    public Article findById(int id);
    public Collection<Article> findAll();
    public Collection<Article> findByName(String name);
}

```

```
public class ArticleJdbcDAO implements ArticleDAO
// ... Constructors setting DB URL and credentials
public void insert(Article item) {
    Connection con;
    try {
        con = DriverManager.getConnection(dbURL, dbUser, dbPassw);
        String sql = "INSERT INTO article VALUES (?)";
        PreparedStatement st = con.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
        st.setString(1, item.getName());
        st.executeUpdate();
        ResultSet newKeys = st.getGeneratedKeys(); } ← Clean way to acquire id's created by the DB
        if (newKeys.next()) { → (supported by most Drivers since JDBC4)
            item.setId(newKeys.getLong("id"));
        } else {
            throw new SQLException("Creating article failed, no generated key obtained.");
        }
    } catch (SQLException e) {
        throw new RuntimeException("DB operation failed: insert(article): " + item, e);
    } finally {
        con.close();
    }
}
```

PROG2 – IO – Java Database Connectivity

8 I/O Foundation and File IO

Java Streams

- Input / Output wird in Java durch Streams gelöst
 - Ein Stream ist mit einem physikalischen Gerät verbunden (Tastature, Konsole, Hard Disk, Network Interface)
 - Jeder Stream verhält sich gleich
 - o Unabhängig vom Gerät
 - o Die gleiche I/O Klasse kann für jedes Gerät verwendet werden
 - In Java gibt es zwei Arten von Streams
 - o Byte Streams
 - Byte orientiert (8 Byte)
 - Generischen bspw. binäre Daten
 - o Character Streams
 - Character orientiert
 - Unicode
- Die unterste Ebene ist immer Byte-orientiert (Char-Stream ist nur eine Abstraktion, welche es einfacher für den Benutzer macht)

Byte Streams

Ist durch zwei abstrakte Klassen definiert

- InputStream
 - o Liest die Daten einer Quelle ins Programm
- OutputStream
 - o Schreibt Daten vom Programm an einen bestimmten Ort (destination)
- Beide haben die gleichen Methoden
 - o read()
 - o write()
 - o etc.
 - o Die meisten von diesen Methoden sind als abstrakte Methoden definiert

Character Streams

Ist durch zwei abstrakte Klassen definiert

- Reader
- Writer
- Beide haben Schlüssel-Methoden
 - o read()
 - o write()
- designed für Unicode
 - o ca. 128'000 Zeichen inkl. emoji character

Predefined Streams

- Jedes Java programm importiert java.lang package
 - o Beinhaltet die Klasse System
- System hat drei vordefinierte (predefined) Streams Variablen
 - o System.in
 - Typ: InputStream (Byte-orientiert) → referenziert auf die Tastatur (Default)

- System.out
 - Typ: PrintStream (Byte-orientiert) → referenziert auf Konsole (Default)
- System.err
 - Typ: PrintStream → referenziert auf Konsole (Default)

Konsolen Input lesen

- Die Verwendung von character-oriented Stream ist einfacher zu initialisieren und zu unterhalten
- Verwenden von BufferedReader → unterstützt buffered input (bspw. String)

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

Adds buffer (read as a String)

Converts bytes to characters.

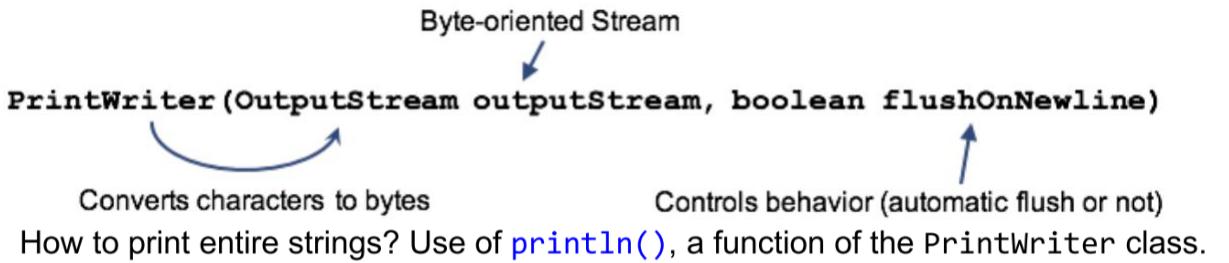
```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead {
    public static void main(String args[]) throws IOException {
        char c;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");
        do {
            c = (char) br.read();
            /* returns character read, as an integer (32bit) in the range 0 to 65535
             * (0x00-0xffff, 16bit), or -1 (0xffff-ffff) if the end of the stream has been reached */
            System.out.println(c);
        } while(c != 'q');
    }
}
```

How to read entire strings? Use `readLine()`, a function of the BufferedReader class

```
// Use a BufferedReader to read an entire string from the console.
import java.io.*;
class BRRead {
    public static void main(String args[]) throws IOException {
        String str;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while(!str.equals("stop"));
    }
}
```

Konsolen Output schreiben

- System.out ist vom Typ PrintStream → System.out.println()
 - Byte-orientiert
- Für Text-Output verwendet man besser einen Character orientierten Stream



How to print entire strings? Use of `println()`, a function of the PrintWriter class.

```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
    public static void main(String args[]) {
        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("This is a string"); // Print entire string to System.out
        int i = -7;
        pw.println(i); // Convert to string and print to System.out
        double d = 4.5e-7;
        pw.println(d); // Convert to string and print to System.out
    }
}
```

File IO

- Um Files zu lesen oder zu schreiben verwendet man

- o FileInputStream
 - o FileOutputStream

```
import java.io.*;
class ShowFile {
    public static void main(String args[]) throws IOException{
        FileInputStream fin; //Represents file, read access, byte stream
        try {
            fin = new FileInputStream(args[0]); //try to open File
            System.out.println("File Found and Opened");
            // ... other File IO operations come here
        }
        catch(FileNotFoundException e) { System.out.println("File Not Found"); return; }
        catch(ArrayIndexOutOfBoundsException e) { System.out.println("Usage: ShowFile File"); return; }
        catch(IOException e) { System.out.println("IO Error: " + e.getMessage()); }
        finally {
            try {
                if (fin != null) fin.close();
            } catch(IOException e) {
                System.out.println("Error while trying to close File")
            }
        }
        return;
    }
}
```

Klasse Pfad

```
import java.nio.*; import java.io.*;
class PathDemo {
    public static void main(String args[]) throws IOException{
        String pth="/home/joe/foo.txt";                                // Unix, Linux, Solaris Syntax
        Path path = Paths.get(pth); // Paths class provides Factory methods to create a Path object
        //Some examples using Path methods
        System.out.format("toString: %s%n", path.toString());           // /home/joe/foo.txt
        System.out.format("getFileName: %s%n", path.getFileName());       // foo.txt
        System.out.format("getName(0): %s%n", path.getName(0));          // home
        System.out.format("getNameCount: %d%n", path.getNameCount());     // 3
        System.out.format("subpath(0,2): %s%n", path.subpath(0,2));       // home/joe
        System.out.format("getParent: %s%n", path.getParent());          // /home
        System.out.format("getRoot: %s%n", path.getRoot());              // /
        //Using Path in methods of class File (File will be introduced later)
        if(Files.isReadable(path) && !Files.isExecutable(path))
            System.out.println("File is a readable and not executable");
    }
}
```

BRG2 – IO – Foundation & File IO

2

Advanced File Writing

```
import java.io.*;
public class AdvFileOutputStreamDemo {
    public static void main(String[] args) {
        BufferedWriter bw = null;                                // Declare a BufferedWriter
        try {
            File f = new File("/tmp/test/hello/newFile.txt"); // Create a File representation of the given file
            File d = f.getParentFile();                      // Create a File representation for the parent directory
            if(!f.exists()){                                // If the directory and file does not exist, create it
                d.mkdirs();                                // Creates directory (incl. all missing parent dirs)
                f.createNewFile();                          // Creates an empty file on the filesystem
            }
            FileWriter fileWriter = new FileWriter(f); // Create a standard FileWriter, character-oriented
            bw = new BufferedWriter(fileWriter); // Wrap FileWriter with a buffer for better performance
            bw.write("My first line!");
            bw.newLine();                                // Add a new-line character (system dependent)
            bw.write("My second line ");
            bw.flush();                                 // Flush the buffer, to enforce data to the medium (e.g. Hard Disk)
            bw.write("keeps going on...");               // Add additional content to the buffer
        } catch (IOException e) { e.printStackTrace();
        } finally { try { if (bw != null){ bw.close(); } } catch (IOException ex) { ex.printStackTrace(); } }
    }
}
```

9 Java Networking

Client-Server Paradigma

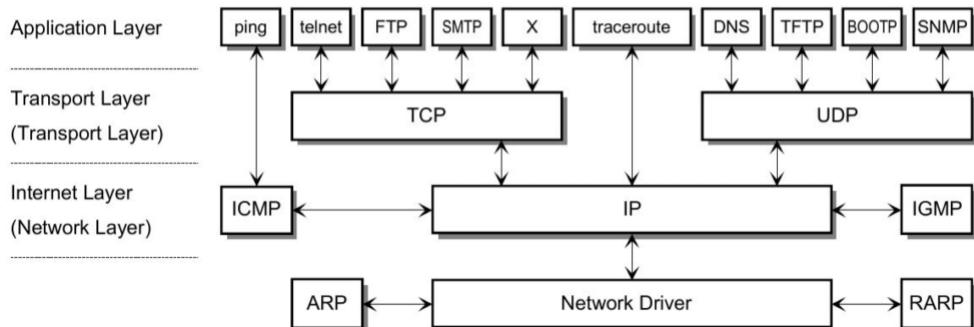
- Beziehung zwischen kooperierende Programinstanzen als Teil einer verteilten Software
- Ein Server stellt eine Funktion oder Service einem oder mehreren Clients zur Verfügung
- Clients setzen Requests an den Server ab.

Peer to Peer Paradigma

- Beziehung zwischen kooperierende Programminstanzen als Teil einer verteilten Software
- Keine Unterscheidung, alle Netzwerkteilnehmer sind Peers

Kommunikation zwischen Endpunkten

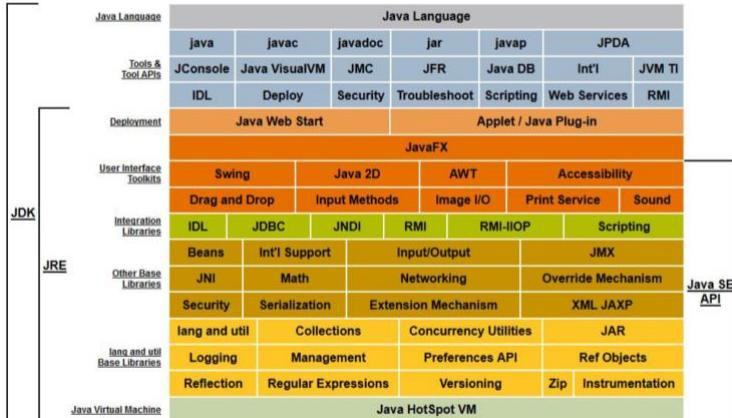
- Peers, Clients und Server sind Endpunkte
 - Endpunkte kommunizieren über das Netzwerk
 - Das meist genutzte Netzwerk ist das Internet
 - o Paket orientiert
 - o TCP/UDP/IP
 - o Layer 3+4 im OSI Model
 - Jeder Endpunkt ist eindeutig identifizierbar durch eine eindeutige Adresse
 $<\text{IP-Address}>:\langle\text{Transport Protocol}\rangle:\langle\text{Port Number}\rangle$
192.168.1.1:TCP:80
- The application-level protocol is not part of TCP/IP addressing (but implicitly represented by the port number)



- IP Adresse 32Bit = 4 Octets = 4 Bytes
- DNS verwandelt eine einfach zu merkende Adresse (Domain Names «String») in eine maschinenlesbare Adresse (Domain Numbers «IP-Adresse»)
- Die Ports sind positive Nummern zwischen 0 – 65535 (2^{16-1}) → 16 Bit
- Ports sind in zwei Ranges eingeteilt
 - o Vorbelegte / «well-known» → 0 – 1024 (bspw. 20 FTP oder 80 http)
 - o 1025 – 65535 sind frei zugebrauchen

Java Networking

Die Java SDK stellt eine Networking SDK mittels `import java.net.*` zur Verfügung.



IP-Addresses in Java

```
// --- Begin Code Snippet --- //
import java.util.Enumeration;
import java.net.*;

try {
    Enumeration<NetworkInterface> interfaceList = NetworkInterface.getNetworkInterfaces();
} catch (SocketException se) {
    System.out.println("Error getting network interfaces:" + se.getMessage());
}

//... code from previous slide ...
while (interfaceList.hasMoreElements()) {
    NetworkInterface iface = interfaceList.nextElement();
    System.out.println("Interface " + iface.getName() + ":");
    Enumeration<InetAddress> addrList = iface.getInetAddresses();           // get list of *all* IPs
    if (!addrList.hasMoreElements()) {
        System.out.println("\t(No addresses for this interface)");
    }
    while (addrList.hasMoreElements()) {
        InetAddress address = addrList.nextElement();                         // extract a single IP from the list of IPs
        //identify address type
        String ip=((address instanceof Inet4Address ? "(v4)" : (address instanceof Inet6Address ? "(v6)" : "(?)")));
        System.out.print("\tAddress "+ ip );
    }
}
```

- Klasse InetAddress wird als generelle Abstraktion verwendet
 - o Equal wird verwendet für
 - File (file and directory)
 - Path (file system path)
 - NetworkInterface (network interfaces)
 - o Superklasse für IPv4 und IPv6
 - o Eine Instanz von InetAddress beinhaltet
 - Eine IP-Adresse (Unicast, multicast, link-local, site-local, global)
 - Korrespondiert zum Hostname
 - o InetAddress definiert eine Nummer einer statischen Funktion

```

import java.net.InetAddress; import java.net.UnknownHostException;
public class SimpleNameLookup {
    public static void main(String[] args) {
        // Get name(s)/address(es) of hosts given on command line
        for (String host : args) {
            try {
                System.out.println(host + ":");
                InetAddress[] addressList = InetAddress.getAllByName(host);
                // Name-to-IP, many IPs to one name is possible
                for (InetAddress address : addressList) {
                    System.out.println("\t" + address.getHostName() + "/" + address.getHostAddress());
                }
            } catch (UnknownHostException e) {
                System.out.println("\t Unable to find address for " + host);
            }
        }
    }
}

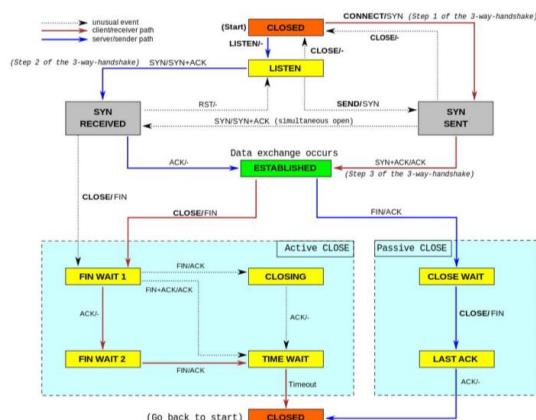
```

Socket

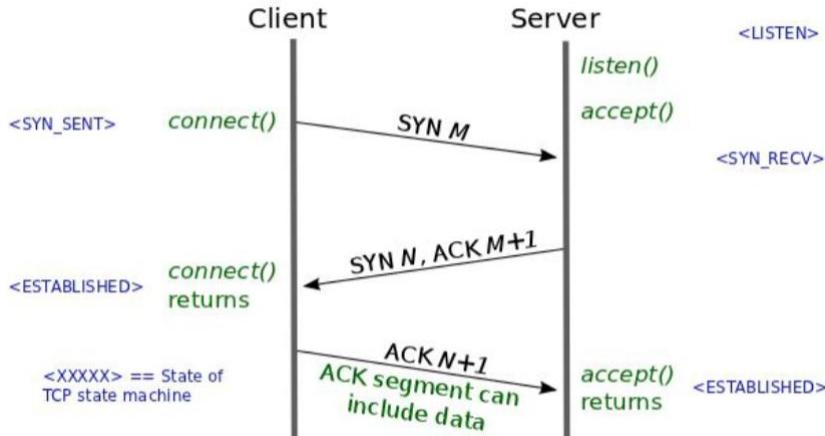
- Socket ist eine Abstraktion durch welche die Applikation Daten sendet und empfängt
- Socket erlaubt es einer Applikation mit dem Networkstack eines Betriebssystems zu verbinden
- Es gibt zwei Arten von Socket Exit
 - o Stream Socket
 - Stellt einen zuverlässigen, verbindungsorientierten Byte-Stream Service zur Verfügung
 - Basiert auf TCP
 - o Datagram Socket
 - Stellt einen unzuverlässigen, verbindungslosen, Datagram Service zur Verfügung
 - Basiert auf UDP

Stream (TCP) Sockets

- TCP stellt einen zuverlässigen, byte- und verbindungsorientierten Übertragungs-Service zur Verfügung
- Verbindung ist ein zwei-Weg (bi-direktional) Kanal, welcher durch <IP>:<TCP>:<PORT> identifiziert wird
- Eine TCP Verbindung muss aufgebaut und abgebaut werden
- Java stellt zwei Klassen für TCP-basierte stream sockets zur Verfügung
 - o Socket
 - o ServerSocket



- TCP Verbindungsaufbau – Client Side
 - o Client erstellt eine socket-Instanz
 - o Client verbindet zum Server
 - o Client anerkennt die Verbindung



```

import java.net.*; import java.io.*;

public class TCPClient {
    public static void main(String[] args) throws IOException {
        if ((args.length < 2) || (args.length > 3)) // Test for correct # of args
            throw new IllegalArgumentException("Parameter(s): <Server name> [<Port>]");
        String server = args[0]; // Server name
        int servPort = Integer.parseInt(args[1]); //Server port
        Socket soc = new Socket(); // create a client socket
        InetSocketAddress endpoint;
        try{
            endpoint = new InetSocketAddress(InetAddress.getByName(server), servPort);
            soc.connect(endpoint); // connect to endpoint (server)
            System.out.println("Connected to server...");
        } catch(IOException e) {
            System.err.println("Exception during connection setup");
        } finally {
            soc.close(); // Close the socket and its streams
            System.out.println("Disconnected from server...");
        }
    }
}
  
```

- TCP Verbindungsaufbau – Server Side
 - o Server erstellt eine Socketinstanz
 - o Server ändert den Socketzustand zu Listen
 - o Server akzeptiert die Verbindungsanfrage
- TCP Verbindungsaufbau und Abbau – Server Side
 - o `servSoc.accept()` blockiert solange die Client-Verbindung bis die Anfrage angekommen ist
 - o Für jeden Client wird ein neuer Socket erstellt
 - Jeder Socket gehört genau zu einer Verbindung

```

// --- Start Code Snippet --- //
public static void main(String[] args) throws IOException {
    if (args.length != 1) // test for correct # of args
        throw new IllegalArgumentException("Parameter(s): <Port>");
    int servPort = Integer.parseInt(args[0]); // server port
    // Create a server socket to accept client connection requests
    ServerSocket servSoc = new ServerSocket(servPort); // listening on server port
    while (true) { // Run forever, accepting connections
        Socket clntSock = servSoc.accept(); // get client connection
        SocketAddress clientAddress = clntSock.getRemoteSocketAddress();
        System.out.println("Handling client at " + clientAddress);
        clntSock.close(); // Server-side socket close
    }
}
// --- End of Code Snippet --- //

```

- Senden und Empfangen von Daten via TCP Verbindung – Client Side

```

// --- Start Code Snippet --- //

Socket soc = new Socket(server, servPort); //create client, connect to host:port
System.out.println("Connected to server...sending echo string");

InputStream in = soc.getInputStream(); //stream for receiving, byte-oriented
OutputStream out = soc.getOutputStream(); //stream for sending, byte-oriented

out.write(dataOut); //Send bytes to server
dataIn=in.read(); //Receive bytes

// --- End of Code Snippet --- //

```

- Senden und Empfangen von Daten via TCP Verbindung – TCPEchoServer

```

// --- Start Code Snippet --- //
public static void main(String[] args) throws IOException {
    if (args.length != 1) // Test for correct # of args
        throw new IllegalArgumentException("Parameter(s): <Port>");
    int servPort = Integer.parseInt(args[0]); // Server port
    int recvMsgSize; // Size of received message
    byte[] receiveBuf = new byte[30]; // Receive buffer
    // Create a server socket to accept client connection requests
    ServerSocket servSoc = new ServerSocket(servPort);
    while (true) { // Run forever, accepting and servicing connections
        Socket clntSock = servSoc.accept(); // Get client connection
        InputStream in = clntSock.getInputStream();
        OutputStream out = clntSock.getOutputStream();
        while ((recvMsgSize = in.read(receiveBuf)) != -1) // Read "some" number of bytes from the receive buffer
            out.write(receiveBuf, 0, recvMsgSize);
        clntSock.close(); // Server-side socket close
    }
}
// --- End of Code Snippet --- //

```

Datagram (UDP) Sockets

- UDP stellt einen unzuverlässigen, verbindungslosen, datagram-orientierten Stream Service zur Verfügung
- Eine UDP Verbindung ist ein abstrakter zwei-Weg (bi-direktional) Kanal, welcher durch <IP>:<UDP>:<PORT> identifiziert wird
- UDP definierte keine klare Verbindungsvorstellung

- «Send and pray» Paradigma
- Die Klasse DatagramPacket ist eine Abstraktion eines UDP-Datagrams

```
DatagramPacket(byte[] buf, int offset,
               int length, InetAddress address, int port)
```
- UDP Aufbau und Abbau – Client Side

```
// --- Start Code Snippet --- //
public class UDPEchoClientTimeout {
    public static void main(String[] args) throws IOException {
        DatagramSocket socket = new DatagramSocket();                                //UDP datagram socket
        socket.close();                                                               //close socket
    }
} // --- End of Code Snippet --- //
```
- Senden und Empfangen von Daten via UDP – Client Side

```
// --- Start Code Snippet --- //

DatagramPacket sendPkt = new DatagramPacket(bytes, bytes.length, servAddr, Port);
DatagramPacket recPkt = new DatagramPacket(new byte[bytes.length], bytes.length);
socket.send(sendPkt);      // Send the echo
socket.receive(recPkt);   // Attempt (!) echo reply reception

// --- End of Code Snippet --- //

// --- Begin Code Snippet --- //
socket.setSoTimeout(TIMEOUT);                                         // maximum receive blocking time
(milliseconds)
do {
    try {
        socket.receive(receivePacket);                                     // reception attempt, wait max TIMEOUT ms
        if (!receivePacket.getAddress().equals(serverAddress)) {          // check source
            throw new IOException("Received packet from an unknown source");
        }
        receivedResponse = true;
    } catch (InterruptedIOException e) {                                    // did not receive anything
        tries += 1;
    }
} while(!receivedResponse && (tries < MAXTRIES));                  // try MAXTRIES times

// --- End of Code Snippet --- //
```
- UDP Aufbau und Abbau – Server Side

```
// --- Begin Code Snippet --- //
public class UDPEchoServerTimeout {
    public static void main(String[] args) throws IOException {
        DatagramSocket socket = new DatagramSocket(300, 127.0.0.1);      // datagram socket
        socket.close();                                                       // close socket
    }
} // --- End of Code Snippet --- //
```
- Senden und Empfangen via UDP – Server Side

```
// --- Begin Code Snippet --- //

DatagramSocket socket = new DatagramSocket(servPort);
DatagramPacket packet = new DatagramPacket(new byte[MAX],
MAX);

while (true) {
    socket.receive(packet); // receive packet from client
    socket.send(packet);   // return same packet to client
    packet.setLength(MAX); // reset length
                           // avoid buffer to shrink
}

// --- End of Code Snippet --- //
```

TCP Server – Handling Multiple Clients with Threads

- Erstellt für jeden Client einen neuen Thread
- Thread implementiert ClientHandler, dieser implementiert das Runnable Interface

```
public class TCPEchoServerThread {  
    public static void main(String[] args) throws IOException {  
        int echoServPort = Integer.parseInt(args[0]); // Server port  
        ServerSocket servSock = new ServerSocket(echoServPort); // Server socket, accepts clients  
        // run forever, accept clients, create a dedicated thread per connection (=client)  
        while (true) {  
            Socket clntSock = servSock.accept(); // Block; waiting for connection  
            Thread clientThread = new Thread(new ClientHandler(clntSock)); // create a Client thread  
            clientThread.start(); //prepares client thread environment, calls run() of ClientHandler  
        }  
    }  
}
```

TCP Server- Generic ClientHandler

```
import java.io.*; import java.net.*;  
public class ClientHandler implements Runnable {  
    private Socket clntSock;  
  
    public ClientHandler(Socket clntSock) {  
        this.clntSock = clntSock; // client socket associated with this connection  
    }  
    public void run() {  
        handleClient(); // forward handling of the client connection for the new thread  
    }  
    public void handleClient() {  
        try {  
            int bytesHandled = echoProtocol(); // delegate handling packets to the echoProtocol method  
            System.out.println("Handled " + bytesHandled + " bytes");  
        } catch (IOException ex) {  
            System.err.println("Error in handling client: " + ex.getMessage());  
        } finally {  
            try {  
                clntSock.close(); // always close socket (connection) when ending  
            } catch (IOException e) {} // ignore close exception  
        }  
    }  
}
```

Java Logging

- Java.util.logger package
 - o Loggers sammeln Meldungen und erstellen Log-Records
 - Hat einen unique name

```
import java.util.logging.*  
  
public class LoggerObtain1 {  
    private static final Logger logger = Logger.getLogger("MyLogger"); // Returns a logger singleton instance  
    ...  
}  
public class LoggerObtain2 {  
    private static final Logger logger = Logger.getLogger("MyLogger"); // Returns the same instance  
    ...  
}  
public class LoggerObtain3 {  
    private static final Logger logger = Logger.getLogger("MyOtherLogger"); // Returns a different instance  
    ...  
}
```

- o Levels definieren den Schweregrad des Log-Records

- Seven pre-defined System-levels exist as static variables of class Level *)
 - SEVERE (1000), WARNING (900), INFO (800), CONFIG (700), FINE (500), FINER (400), FINEST (300)
 - user-defined levels are possible

```
import java.util.logging.*

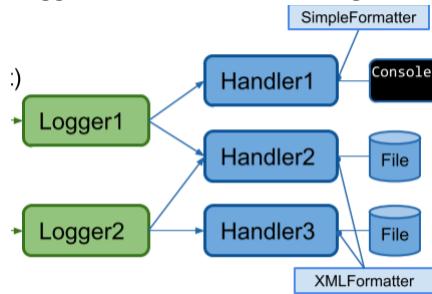
public class UniverseDemo {
    private static final Logger logger = Logger.getLogger("UniverseLogger"); // Obtain a suitable Logger instance
    public static void main (String args[]) {
        logger.finer("Starting up ..."); // debugging message using finer level convenience method*)
        try {
            logger.fine("Try to compute size of universe"); // debugging message using the fine level method
            Number size = Compute.sizeOfUniverse();
            // info level status message using placeholder and parameters to build the record
            logger.log(Level.INFO, "Success! Found {0} elements.", size); // content of size is inserted at {0}
        } catch(Exception ex) {
            // error message submitting the exception to output in log handler
            logger.log(Level.WARNING, "Failed to compute size of universe", ex);
        }
        logger.finer("Shutting down..."); // another debugging message using the finer level method*)
    }
}
```

*) in this specific use-case the convenience methods entering(String class, String method, Object[] params) and exiting(String class, String method, Object result) could be used to write standardized (finer-level) debug messages entering/exiting a method.

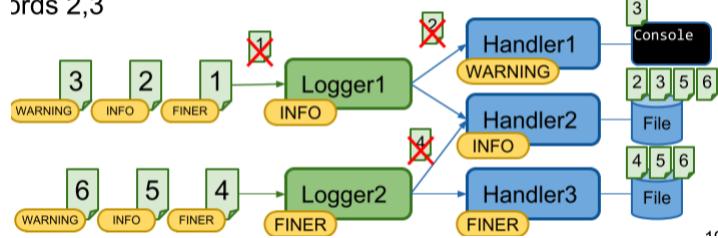
17

- Handlers schreiben die Logs in einen definierten Kanal

- Loggers senden Ihre Meldung an 1 bis n Handler



- Handler können auch nach Schweregrad eingeteilt werden



Handling Multiple Clients with a Thread Pool

- Konfigurierbarer Parameter wie viel Threads erstellt werden
- Wenn Thread fertig ist, landet er wieder
- Der ThreadPool erstellt einen ServerSocket
- Jeder Thread läuft mit einer Referenz auf dieser ServerSocket

```

public class TCPEchoServerPool {
    private static final logger = Logger.getLogger(TCPEchoServer.class.getName());
    public static void main(String[] args) throws IOException {
        if (args.length != 2) throw new IllegalArgumentException("Parameter(s): <Port> <Threads>");
        int echoServPort = Integer.parseInt(args[0]); int threadPoolSize = Integer.parseInt(args[1]);
        // Create a shared(!) server socket to accept client connection requests
        final ServerSocket servSock = new ServerSocket(echoServPort);
        for (int i = 0; i < threadPoolSize; i++) { // Spawn a fixed number of threads to service clients
            Thread thread = new Thread() {
                public void run() {
                    while (true) {
                        try {
                            Socket clntSock = servSock.accept();           // Wait for a connection
                            new ClientHandler(clntSock).handleClient(); // Create handler and handle data
                        } catch (IOException ex) {
                            logger.log(Level.WARNING, "Client accept failed", ex);
                        }
                    }
                }
            };
            thread.start();
            logger.info("Created and started Thread = " + thread.getName());
        }
    }
}

```

Thread Pool

Server Thread

Keep Alive

Wenn längere Zeit keine Daten mehr gesendet werden, muss man dafür schauen, dass die Verbindung nicht in den Close Zustand wechselt. Aus diesem Grund stellt TCP eine sogenannte «Keep Alive Message» zur Verfügung. → *void setKeepAlive(), boolean getKeepAlive()*

```

// Create socket that is connected to server on specified port
Socket soc = new Socket(server, servPort);
// Enable TCP Keep Alive messages
soc.setKeepAlive(true);

```

TCP No Delay

TCP Daten müssen eine gewisse Grösse haben, damit diese auch versendet werden. Wenn man nun sehr kleine Packete verschicken möchte ohne auf eine «Füllung» der Daten zu warten. So kann man TCP No Delay verwenden. → *boolean getTCPNoDelay(), void setTCPNoDelay()*

```

// Create socket that is connected to server on specified port
Socket soc = new Socket(server, servPort);
// disable buffering by the TCP stack
soc.setTCPNoDelay(true);

```

Urgent Data

TCP implementiert ein ein Konzept von «urgent Data», welche es ermöglicht die Reihenfolge der Daten zu Umgehen. «out of band» (OOB) → *void sendUrgent(), boolean getOOBInline(), setOOBInline()* → Sollte mit grosser Vorsicht verwendet werden

Linger

```

InputStream in = clntSock.getInputStream();
OutputStream out = clntSock.getOutputStream();
clntSock.setSoLinger(true, 20);      // enable lingering for maximally 20 seconds
byte[] echoBuffer = new byte[32];     // Receive Buffer
recvMsgSize = in.read(echoBuffer);   // read and ...
out.write(echoBuffer, 0, recvMsgSize); // ... write in return
clntSock.close();                   // close socket, blocks until all pending data is
                                    // sent/confirmed by receiver
                                    // or the 20 seconds timeout expires

```

URL

- Java stellt `java.net.URL` zur Verfügung

```
import java.net.*;
public class URLDemo {
    public static void main(String[] args) throws MalformedURLException {
        URL url = new URL("https://www.zhaw.ch/de/engineering/studium/bachelorstudium/informatik/");
        System.out.println("Protocol: " + url.getProtocol());
        System.out.println("Host: " + url.getHost());
        System.out.println("Port: " + url.getPort());
        System.out.println("File: " + url.getFile());
        System.out.println("Ext: " + url.toExternalForm());
    }
}
```

undefined !

- A URL is based on four components:
 - Application Protocol to use (e.g. https, http, ftp, webdav, ...) following ":"//"
 - Host name or IP address of the host to connect to
 - Port number, separated by ":" from host (only required if not default port of protocol)
 - Resource reference, separated by "/" from host:port
- Example: `http://tmb.nginet.de:80/?page_id=634`

java.net

URL-Verbindung

- Mittels `URL.openConnection()` kann man Verbindung zu einem remote Server herstellen

```
import java.net.*; import java.sql.Timestamp;
public class URLConnectionDemo {
    public static void main(String args[]) throws MalformedURLException{
        URL url = new URL("https://www.zhaw.ch/de/engineering/studium/bachelorstudium/informatik/");
        try {
            URLConnection urlCon = url.openConnection(); // open connection to remote host and get properties
            System.out.println("Date: " + new Timestamp(urlCon.getDate()).toLocalDateTime());
            System.out.println("Content-Type: " + urlCon.getContentType());
            System.out.println("Expire: " + new Timestamp(urlCon.getExpiration()).toLocalDateTime());
            System.out.println("Content-Length: " + urlCon.getContentLength());
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

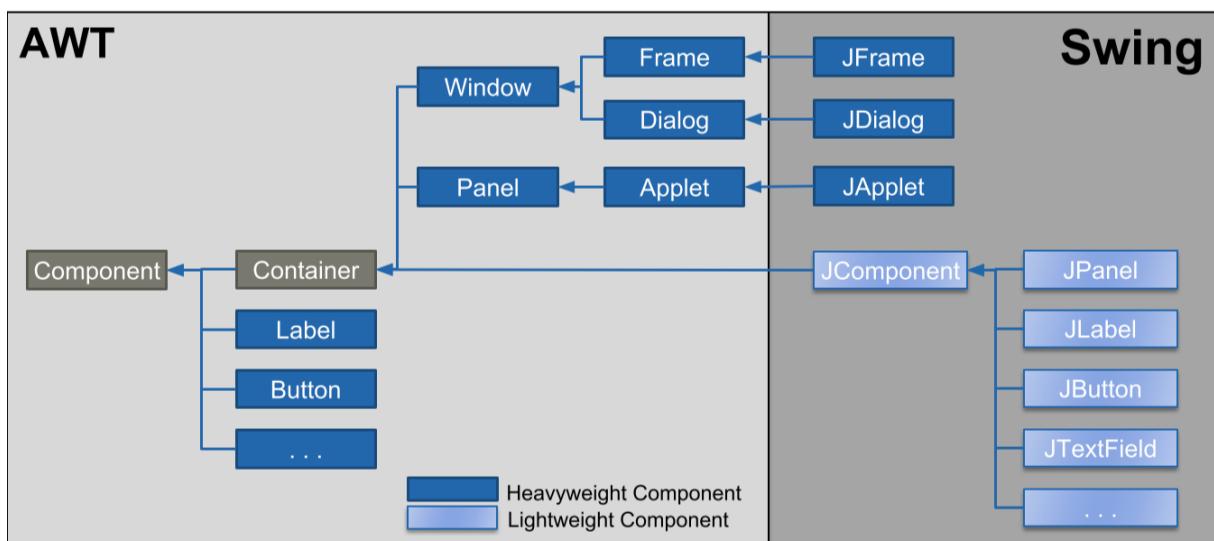
OG2 – IO – Java Networking Advanced

```
$ java URLConnectionDemo
Date: 2018-08-21T09:11:30
Content-Type: text/html; charset=utf-8
Expire: 2018-08-21T09:11:30
Content-Length: 196464
```

3

10 Graphical User Interfaces – GUI

- Swing ist eine Bibliothek für das Erstellen von GUI in Java
 - o Abstract Window Toolkit (AWT) Bibliothek
 - o Plattform unabhängig
 - o Objektorientiert
 - o Building-Block-Prinzip
- Heavyweight Components
 - o Alle AWT
 - o Minimales portables Set an Elemente
 - o Schwierig zum Erweitern
- Lightweight Components
 - o Meisten Swing
 - o Gezeichnet in Java
 - o Betriebssystem unabhängig
 - o erweiterbar



Frame erstellen

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ImageViewer {

    private JFrame frame;

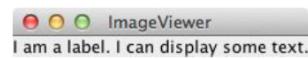
    public ImageViewer() {
        makeFrame();
    }

    public static void main(String[] args) {
        ImageViewer i = new ImageViewer();
    }

    private void makeFrame() {
        frame = new JFrame("ImageViewer");
        Container contentPane = frame.getContentPane();

        JLabel label = new JLabel("I am a label. " +
                               "I can display some text.");
        contentPane.add(label);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```



Adding menus

- **JMenuBar**
 - Displayed below the title
 - Contains the menus
- **JMenu**
 - Contains the menu items
 - e.g. **File**
- **JMenuItem**
 - Individual item
 - e.g. **Open**

PROG2 – GUI – Foundations

```
public ImageViewer() {  
    makeFrame();  
    makeMenuBar(frame);  
}  
  
private void makeMenuBar(JFrame frame)  
{  
    JMenuBar menubar = new JMenuBar();  
    frame.setJMenuBar(menubar);  
    frame.setSize(frame.getWidth(),  
                 frame.getHeight()+20);  
  
    // create the File menu  
    JMenu fileMenu = new JMenu("File");  
    menubar.add(fileMenu);  
  
    JMenuItem openItem = new JMenuItem("Open");  
    fileMenu.add(openItem);  
  
    JMenuItem quitItem = new JMenuItem("Quit");  
    fileMenu.add(quitItem);  
}
```

Event handling

- Korrespondieren zu den Interaktionen mit den Komponenten
- Komponenten sind einem Event Type zugehörig
- Objekte können informiert werden, wenn ein Event eintrifft
 - → Listeners
- ActionEvent
 - event.setActionCommand()
 - event.getActionCommand()
- ActionListener
 - Interface wird für jedes Listening Objekt implementiert
 - component.addActionListener(ActionListener)
 - Link zwischen Komponent und Listener
 - listener.actionPerformed(ActionEvent)
 - Wird aufgerufen, wenn Event eintritt

Centralized ActionListener applied to ImageViewer

```
public class ImageViewer implements ActionListener {
    private void makeMenuBar(JFrame frame) {
        ...
        openItem.addActionListener(this);
        quitItem.addActionListener(this);
        ...
    }

    public void actionPerformed(ActionEvent event) {
        switch (event.getActionCommand()) {
            case "Open" : System.out.println("Open has been clicked");
                break;
            case "Quit" : System.exit(0);
                break;
            default :     System.err.println("Unknown event: " + e.getActionCommand());
        }
    }
}
```

```
public interface ActionListener {
    public void actionPerformed(ActionEvent ev);
}
```

Button hinzufügen

```
private JTextField textField;

private void makeFrame() {
    frame.setLayout(new FlowLayout());
    // add button
    JButton okButton = new JButton("Draw another");
    contentPane.add(okButton);
    okButton.addActionListener(this);
    // add text field
    textField = new JTextField("flag");
    contentPane.add(textField);
}

public void actionPerformed(ActionEvent event) {
    //... switch (event.getActionCommand())
    case "Draw another" :
        if (textField.getText().equals("flag")) {
            System.out.println("Lets draw a Swiss flag");
        }
    }
}
```

Anonymous inner class

- Regeln einer Inner-Class
- Nur eine Instanz → keinen Klassennamen notwendig
- Spezieller Syntax

```
openItem.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            openFile();
        }
    });

```

Class definition

Anonymous object creation

Actual parameter: single instance of the anonymous class

```
//Example: Anonymous ActionListener
JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        openFile();
    }
});
```

Lambda Ausdruck

```
openItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        openFile();
    }
});
```



```
openItem.addActionListener(() -> openFile());
```

```
okButton.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("flag")) {
            drawFlagOnCanvas();
        }
    }
});
```



```
okButton.addActionListener( e -> {
    if (e.getActionCommand().equals("flag"))
        drawFlagOnCanvas();
});
```

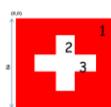
JComponent

- Wenn man custom Componenten erstellen möchte, kann man dies mit JComponent machen

```
import java.awt.*;
import javax.swing.*;
class SwissFlag extends JComponent {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        int size = Math.min(getWidth(), getHeight());
        int part = size/40;
        int xPos = 2 * part;
        int yPos = 2 * part;

        g.setColor(Color.RED);
        g.fillRect(xPos,yPos,32*part,32*part); //1
        g.setColor(Color.WHITE);
        g.fillRect(xPos + 13*part, yPos + 6*part,
                   6*part, 20*part); //2
        g.fillRect(xPos + 6*part,yPos + 13*part,
                   20*part, 6*part); //3
    }
}
```



28

- Der Nullpunkt (0,0) ist oben links
 - o X-Achse geht nach rechts
 - o Y-Achse geht zum Boden

Static, when setting up the GUI

```
public void main() {
    frame = new JFrame("SwissFlag");
    frame.setLayout(new FlowLayout());
    contentPane = frame.getContentPane();

    SwissFlag flag = new SwissFlag();
    flag.setPreferredWidth(
        new Dimension(contentPane.getWidth()-10,
                     contentPane.getHeight()-10)
    );
    contentPane.add(flag);
    frame.setDefaultCloseOperation(
        JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
}
```

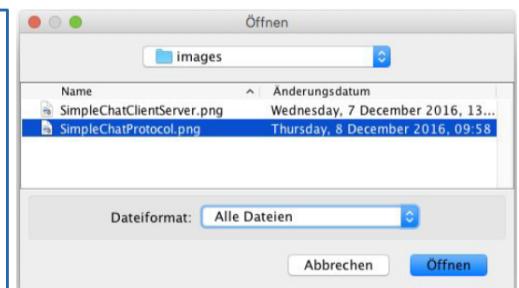
Dynamic on an ActionEvent

```
public void actionPerformed(ActionEvent e)
{
    //...
    drawFlagOnCanvas();
    //...
}

public void drawFlagOnCanvas() {
    SwissFlag flag = new SwissFlag();
    flag.setPreferredWidth(new Dimension(20, 20));
    frame.getContentPane().add(flag);
    frame.pack();
}
```

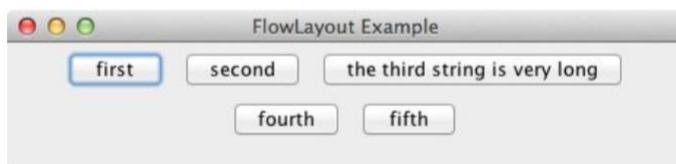
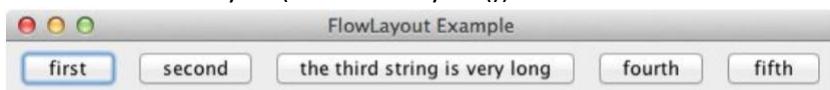
JFileChooser

```
import javax.swing.JFileChooser;
//...
private void openFile() {
    File selectedFile = null;
    JFileChooser chooser = new JFileChooser();
    int result = chooser.showOpenDialog(null);
    if (result == JFileChooser.APPROVE_OPTION) {
        selectedFile = chooser.getSelectedFile();
        //...
    }
}
```



FlowLayout

- Komponenten sind in einem direktonalen Flow angeordnet, es folgt ein Umbruch, falls der Rand erreicht wird
- contentPane.setLayout(new FlowLayout())



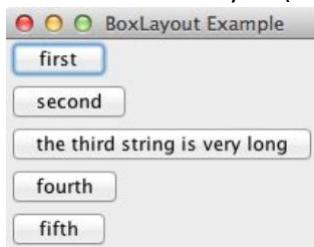
GridLayout

- Komponenten werden in Rechtecken angeordnet
- Rechtecke werden der Grösse entsprechend angepasst
- contentPane.setLayout(new GridLayout(0,2)); // row, col



Box Layout

- Komponenten werden vertical (Y-Achse) und horizontal (X-Achse) angeordnet
 - o Kein Umbruch
 - o Kein Resizing
- `contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.Y_AXIS));`



Border Layout

- Komponenten werden in fünf Teilen eingeteilt
 - o NORTH, SOUTH, EAST, WEST, CENTER
 - NORTH, SOUTH werden horizontal skaliert
 - EAST, WEST werden vertikal skaliert
 - CENTER skaliert in beiden Richtungen
- `contentPane.setLayout(new BorderLayout());`
- `contentPane.add(someComponent, BorderLayout.NORTH);`



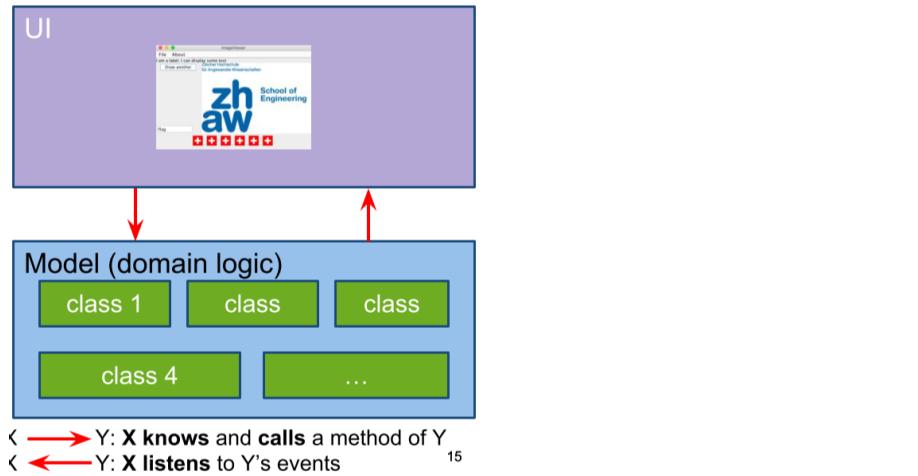
Struts and Glue

- unsichtbare Komponente, welche als Weisraum verwendet werden
- Strut
 - o Fixe Grösse
 - o `Component createHorizontalStrut(int width)`
 - o `Component createVerticalStur(int height)`
- Glue
 - o Füllt verfügbaren Platz
 - o `Component createHorizontalGlue()`
 - o `Component createVerticalGlue()`

Model-View-Controller Pattern

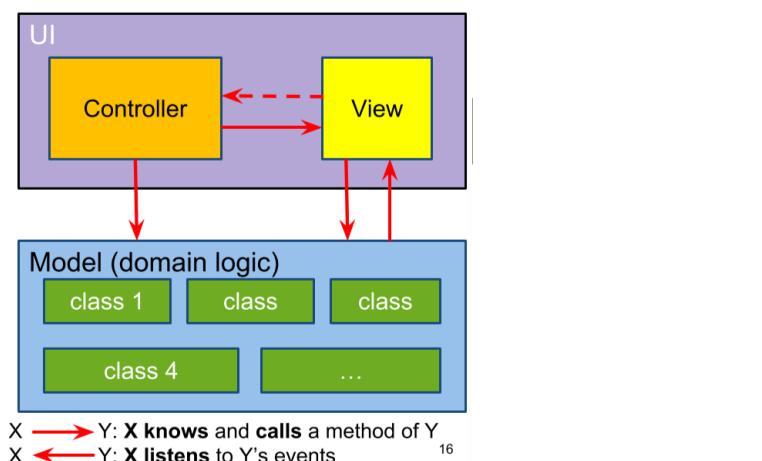
Als erster Schritt wird das Model und UI getrennt

- User Interface (UI)
 - o GUI Komponenten & Logik
 - o Zuständig für die Interaktion zwischen User und Applikation
- Model
 - o Beinhaltet Daten und Implementation der Logik
 - o Nicht sichtbar für den User
 - o Unabhängig von UI



Zweiter Schritt: Trennung View & Controller

- View
 - o Beinhaltet GUI Komponenten (Buttons, etc)
 - o Kennt das Model
 - o Hört auf Änderungen in den Daten (Model)
- Controller
 - o Hört auf Events von der View (User Input)
 - o Kennt das Model
 - o Lädt View neu



- ➔ UI (View & Controller) hat direkten Zugang zum Model
- ➔ Model ruft die UI nie direkt auf

- View (Controller) ist als Listener
- UI bekommt Meldungen wenn Änderungen passiert sind und updated sich selber

JTable

```

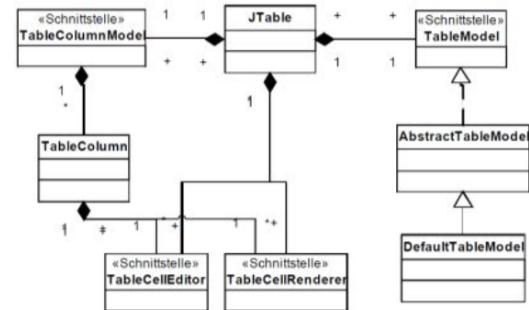
public interface TableModel {
    public int getRowCount();
    public int getColumnCount();
    public String getColumnName(int columnIndex);
    public Class getColumnClass(int columnIndex);
    public boolean isCellEditable(int rowIndex, int columnIndex);
    public Object getValueAt(int rowIndex, int columnIndex);
    public void setValueAt(Object aValue, int rowIndex, int columnIndex);
    public void addTableModelListener(TableModelListener l);
    public void removeTableModelListener(TableModelListener l);
}

public class TableModelEvent
    extends java.util.EventObject {
    public int getFirstRow() { ... };
    public int getLastRow() { ... };
    public int getColumn() { ... };
    public int getType() { ... }
}

public interface TableModelListener
    extends java.util.EventListener {
    public void tableChanged(TableModelEvent e);
}

```

what changed?



PROG2 – GUI – Toolbox