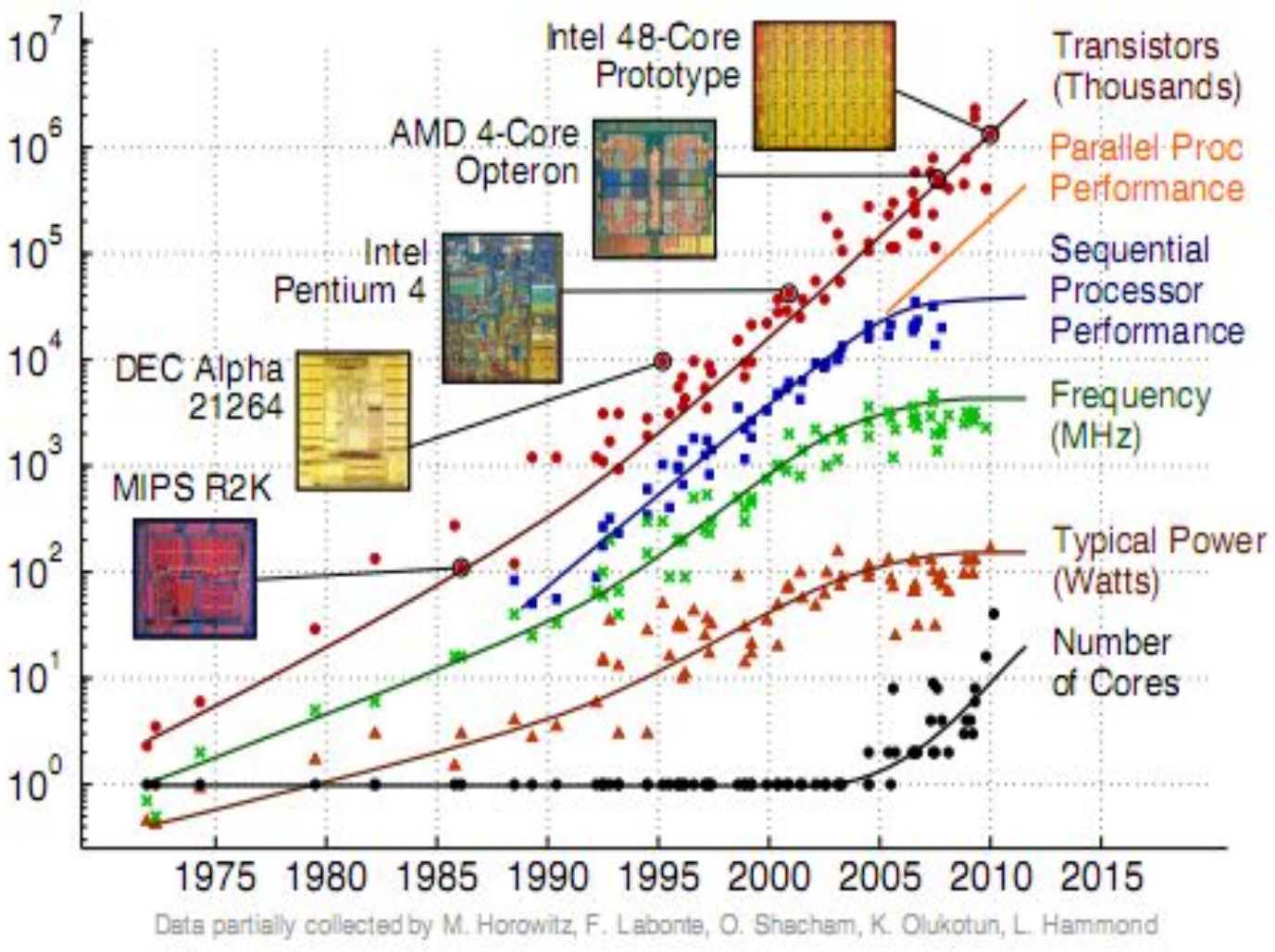
Threads

Inhalt

- Nebenläufigkeit
- Threads in JAVA
- SQL
- JDBC/JPA

The Free Lunch Is Over



Parallelität

- Verschiedene Programmteile laufen gleichzeitig
- Aufwendige Berechnungen im Hintergrund, trotzdem benutzbare GUI
- Verschiedene Netzwerkverbindungen gleichzeitig, auf alle soll reagiert werden
- Falls mehrere Prozessoren vorhanden sind: Jeder Programmteil bekommt volle Rechenleistung eines Prozessors, damit insgesamt schnellere Berechnung

Motivation

- Graphische Oberflächen benötigen einen unabhängigen Kontrollfluss
- Die Simulation realer Objekte wird erleichtert
- Mehrere Eingabequellen müssen effizient beantwortet werden
- Blockierende (oder langsame) Kommunikation soll nicht das gesamte Programm blockieren
- Effiziente und parallel arbeitende Webserver
  - Verteilte Systeme

Geschichte der Verwendung von Nebenläufigkeit

- Die effiziente Ausnutzung von langsamer Peripherie führte zum Multitasking
- Die ersten interaktiven Mehrbenutzersysteme (multics, tops10, unix) basierten auf Multiprocessing
- Echtzeitsysteme benötigen zumindest eine einfache Form von Nebenläufigkeit (Interrupt-Technik)
- Die ersten OO-Sprachen (Simula, Smalltalk) enthielten Nebenläufigkeit

Geschichte der Verwendung von Nebenläufigkeit

- In den 70ern und 80ern wurden unterschiedliche Modelle von OO und non OO Nebenläufigkeit untersucht
- Seit den 70ern gibt es intensive Entwicklungen und Anwendungen im Bereich der Parallelverarbeitung
- Fast jeder Prozessor verfügt heute über mehrere Cores
- Heute ist Multiprocessing und Multithreading (praktisch) auf jedem System und in jeder Programmiersprache möglich (und nötig)

Sequentielle Programmausführung

- Ein sequentielles Java-Programm entspricht dem prozeduralen Paradigma
- Soweit man die Ergebnisse beobachten kann, ist die Ausführungsreihenfolge exakt durch das Programm vorgegeben
- Der Compiler, die Java-Laufzeitumgebung und der Prozessor können innerhalb dieser Grenzen Modifikationen vornehmen um die Effizienz zu steigern

Sequentielle Programmausführung

- Soweit es keinen Einfluss auf die Ergebnisse hat, kann die Reihenfolge von Befehlen verändert werden (reordering)
- Daten können zeitweise in Registern und Cache-Speichern gehalten werden. In dieser Zeit ist der Inhalt des Hauptspeichers nicht korrekt
- Es ist denkbar, dass ein Objekte niemals im Hauptspeicher erscheint (visibility)
- Im Ergebnis kann erst durch diese Maßnahmen die Leistungsfähigkeit moderner Computer erreicht werden

Nebenläufigkeit

- Unter Nebenläufigkeit versteht man die gleichzeitige Ausführung von Programmen oder Programmteilen
- In einem nebenläufigen Programm ist die Reihenfolge der Befehlsausführung nicht vollständig festgelegt
- Das Programm kann durch einen oder mehrere Prozessoren ausgeführt werden
- Nebenläufige Abläufe können über gemeinsamen Speicher verfügen

Ablaufreihenfolge bei Nebenläufigkeit

- Die genaue Ablaufreihenfolge ist nicht festgelegt, nur innerhalb eines Teil

Teil 1

```
foo = 0;
foo = foo + 1;
foo = foo + 2;
```

Teil 2

```
foo = 1;
foo = foo * 3;
foo = foo * 2;
```

Ablaufmöglichkeit

```
foo = 0;
foo = 1;
foo = foo + 1;
foo = foo * 3;
foo = foo + 2;
foo = foo * 2;

=> foo = 10
```

Ablaufmöglichkeit

```
foo = 0;
foo = foo + 1;
foo = foo + 2;

foo = 1;
foo = foo * 3;
foo = foo * 2;

=> foo = 6
```

- Ein nebenläufiges Programm kann bei verschiedenen Laufen verschiedene Ergebnisse haben

Probleme

- Bei nebenläufigen Programmen können verschiedene Probleme auftreten, die sonst nicht auftreten können
- Arbeiten Programmteile gleichzeitig auf den gleichen Variablen, können Programme Fehler enthalten, die nur im Zusammenspiel entstehen
  - sie können blocked bleiben (deadlock)
  - sie können auf nicht initialisierte Werte zugreifen
  - sie können unsaubere Daten sehen
  - Fehler treten nur mit gewisse Wahrscheinlichkeit auf

Probleme

Teil 1

```
foo = 0;
while(foo != 10) {
    foo = foo + 1;
}
```

Teil 2

```
foo = 0;
while(foo != 20) {
    foo = foo + 1;
}
```

Laufen die Teile gleichzeitig, können beide Threads stecken bleiben; oft bleibt nur einer oder keiner stecken

Probleme

```
class Foo {
    String name;
    public Foo() {}
    public void setName(name) {this.name = name;}
    public String getName() {return name;}
}
```

Teil 1

```
foo = new Foo();
foo.setName("Hallo Welt");
```

Teil 2

```
if(null != foo){
    System.out.println(foo.getName());
}
```

Probleme

```
public class Person {
    int alter; double gewicht;
    public Person(int alter, double gewicht) {
        this.alter = alter; this.gewicht = gewicht; }
    public Person kopie() {
        return(new Person(alter, gewicht)); }
    public update(int alter, double gewicht) {
        this.alter = alter; this.gewicht = gewicht; }
}
```

Teil 1

```
Person foo = new Person(26, 77);
foo.update(27,85);
```

Teil 2

```
Person bar = foo.kopie();
```

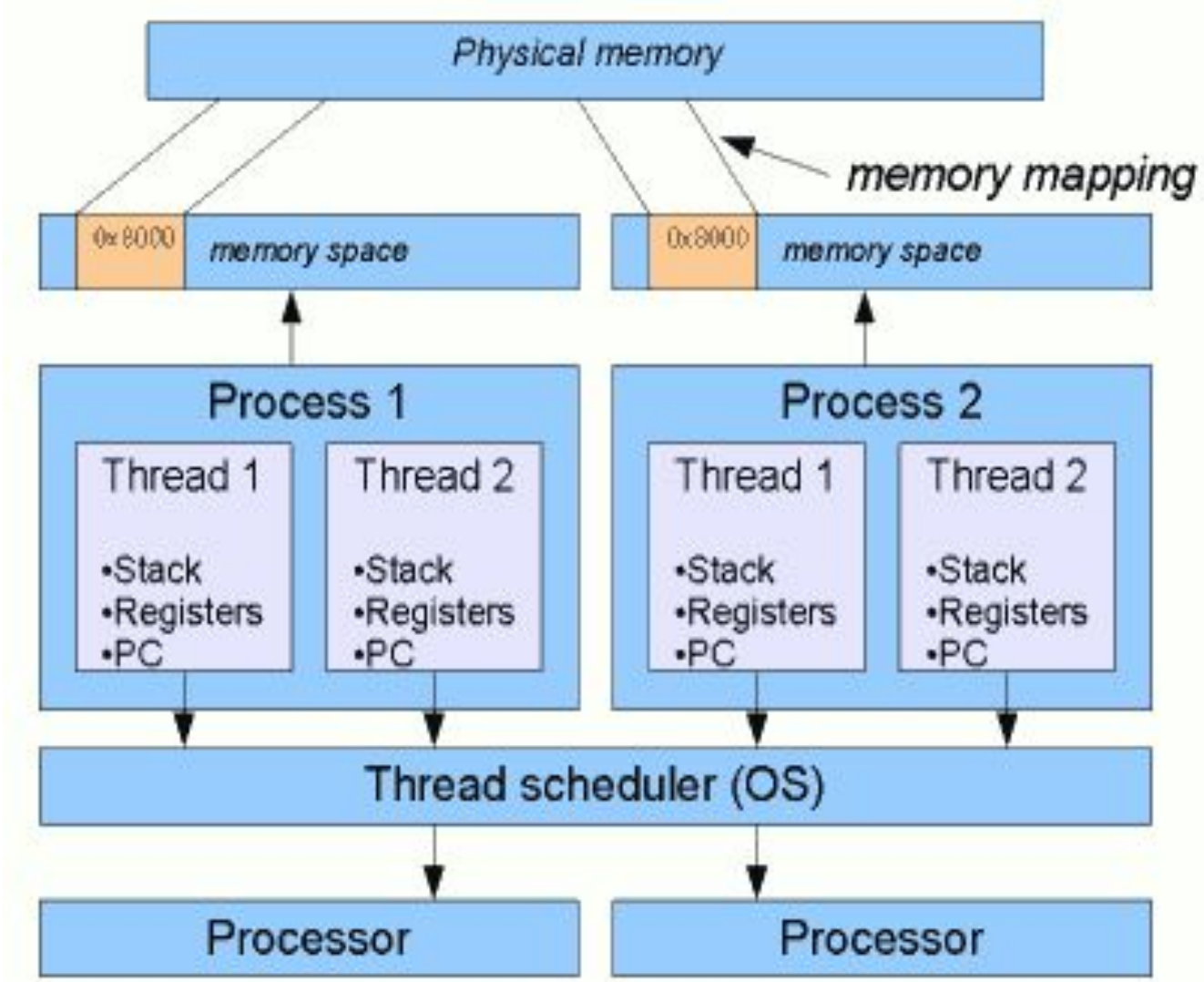
Prozesse

- Ein Prozess ist ein in der Ausführung begriffenes Programm
- Ein Programm beschreibt statisch Struktur und Ablauf
- Ein Prozess ist die dynamische Ausführung des Programm
- Prozesse werden von dem Betriebssystem verwaltet
- Interprozesskommunikation über das Betriebssystem ist langsam!

Threads

- Ein Thread ist eine Ausführungsreihenfolge innerhalb eines Prozesses
- Ein Prozess enthält mindestens 1 Thread
- Jeder Thread verfügt über einen separaten Stack (Rücksprung, lokale Variable)
- Jeder Thread verfügt über einen eigenen Programmzähler, d.h. eine eigene Kontrolle des Ablaufs

Threads


Physical memory
memory mapping
0x8000
memory space
0x8000
memory space
Process 1
Thread 1
•Stack
•Registers
•PC
Thread 2
•Stack
•Registers
•PC
Process 2
Thread 1
•Stack
•Registers
•PC
Thread 2
•Stack
•Registers
•PC
Thread scheduler (OS)
Processor
Processor

Threads und Objektorientierung

- In sequentiellen OO-Programmiersprachen ist zu jedem Zeitpunkt genau eine Methode aktiv
- In dem entgegengesetzten Actor-Konzept sind alle Objekte gleichzeitig aktiv
- Kommunikation erfolgt durch den Austausch von Nachrichten
- Thread-Modelle der prozeduralen Welt verlegen die Nebenläufigkeit auf die Ebene von Prozeduren
- Java: run() -> Thread gestartet werden

Synchronisation

- Stellt sicher, dass Daten nicht gelesen werden, während sie geschrieben werden
- welche Programmteile nicht gleichzeitig ablaufen dürfen
- Kann prinzipiell selbst implementiert werden, Java stellt aber saubere Mechanismen dafür zur Verfügung: synchronized

Java

Es gibt verschiedene Möglichkeiten Nebenläufigkeit in Java zu implementieren

- `Thread`
- `Timer`
- `Runnable`

Java

- Nebenläufigkeit durch die Klasse `Thread`

- Klasse `Thread` erweitern
- Methode `void run()` implementieren
  - Inhalt bestimmt, was in dem Thread läuft
- Methode `void start()` startet den Thread nebenläufig

Beispiel

```
public class Runner {

    public void doit() {
        for(int i = 0; i < 7; i++) {
            System.out.println("Schleife " + i);
        }
    }
}
```

Beispiel

```
public class Main extends Thread {
    Runner runner;
    public Main(Runner run) { runner = run; }
    public void run() {
        runner.doit();
    }
    public static void main(String[] args) {
        Runner runner = new Runner();
        Main foo = new Main(runner);
        Main bar = new Main(runner);
        foo.start();
        bar.start();
    }
}
```

Ausgabe

Programm ergibt bei jedem Ablauf ein andere Ergebnis...

-

```
Schleife 0
Schleife 1
Schleife 2
Schleife 3
Schleife 4
Schleife 5
Schleife 6
Schleife 0
```

```
Schleife 0
Schleife 0
Schleife 1
Schleife 1
Schleife 2
Schleife 2
Schleife 3
Schleife 3
```

```
Schleife 0
Schleife 1
Schleife 0
Schleife 2
Schleife 1
Schleife 3
Schleife 2
Schleife 4
```

Synchronisation

- Synchronisation kann über `synchronized` implementiert werden
- synchronized-Methoden: Wird eine Methode als `synchronized` deklariert, so wird diese bei einem Objekt maximal einmal gleichzeitig ausgeführt
  - Bei verschiedenen Objekten kann sie immernoch gleichzeitig ausgeführt werden
- Synchronized-Blöcke: Enthalten in ihrem Aufruf ein Objekt.
  - Es wird gewartet, bis kein anderer Thread mehr in einem synchronized-Block mit diesem Objekt ist

Beispiel

```
public class Runner {

    public synchronized void doit() {
        for(int i = 0; i < 7; i++) {
            System.out.println("Schleife " + i);
        }
    }
}
```

Beispiel

```
public class Main extends Thread {
    Runner runner;
    public Main(Runner run) { runner = run; }
    public void run() {
        runner.doit();
    }

    public static void main(String[] args) {
        Runner runner = new Runner();
        Main foo = new Main(runner);
        Main bar = new Main(runner);
        foo.start();
        bar.start();
    }
}
```

Synchronisation

- Nur zwei Abläufe möglich:
  - foo arbeitet `runner.doit()` vollständig ab, danach bar
  - bar arbeitet `runner.doit()` vollständig ab, danach foo

- `doit()` als synchronized deklariert ist
- foo und bar das gleiche Runner-Objekt verwenden

Beispiel

```
public class Main extends Thread {

    Runner runner;

    public Main(Runner run) { runner = run; }

    public void run() {

        runner.doit();

    }

    public static void main(String[] args) {

        Runner runnerfoo = new Runner();

        Runner runnerbar = new Runner();

        Main foo = new Main(runnerfoo);

        Main bar = new Main(runnerbar);

        foo.start();

        bar.start();

    }

}
```

Beispiel

```
public class Runner {


    public void doit() {
        synchronized(this) {
            for(int i = 0; i < 7; i++) {
                System.out.println("Schleife " + i);
            }
        }
    }
}
```

Beispiel

```
public class Main extends Thread {

    Runner runner;

    public Main(Runner run) { runner = run; }

    public void run() {

        runner.doit();

    }


    public static void main(String[] args) {

        Runner runner = new Runner();

        Main foo = new Main(runner);

        Main bar = new Main(runner);

        foo.start();

        bar.start();

    }

}
```

Synchronisation

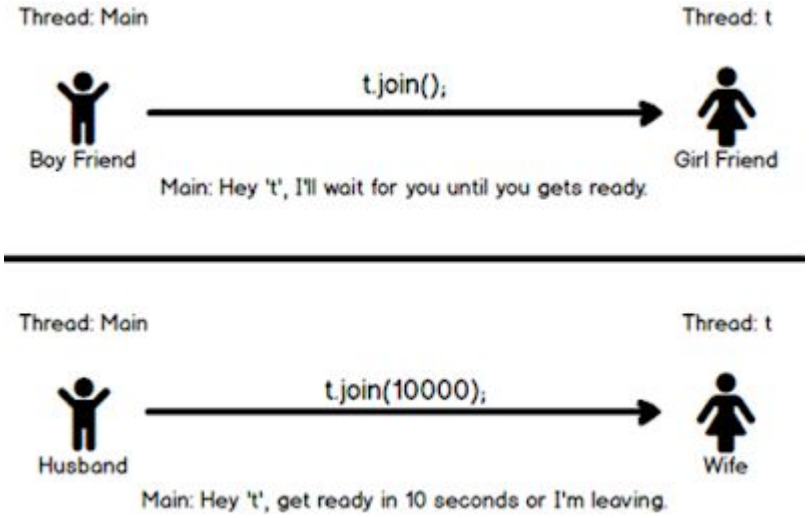
- Nur zwei Abläufe möglich:
  - foo arbeitet `runner.doit()` vollständig ab, danach bar
  - bar arbeitet `runner.doit()` vollständig ab, danach foo

- Die Schleife liegt in einem synchronized-Block
- In beiden Threads ist das Objekt zum Block das gleiche

Beenden von Threads

- Ein Thread endet, wenn die run-Methode beendet wird
- Ein Java-Prozess endet, wenn alle Threads beendet sind
- Durch den Aufruf System.exit() wird immer der Prozess sofort beendet
- Ansonsten kann man den Thread enden nur durch entsprechende Mitteilung
- Mittels der Methode join() kann man darauf warten, dass ein bereits ausgeführter Thread beendet ist

Beispiel


Thread: Main
Thread: t
t.join();
Boy Friend
Girl Friend
Main: Hey 't', I'll wait for you until you gets ready.
Thread: Main
Thread: t
t.join(10000);
Husband
Wife
Main: Hey 't', get ready in 10 seconds or I'm leaving.

Beispiel

```java
class MyRunnable implements Runnable{

    @Override
    public void run() {
        System.out.println("Thread started:::"+Thread.currentThread().getName());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread ended:::"+Thread.currentThread().getName());
    }

}
```

Beispiel

```java
public static void main(String[] args) {
    Thread t1 = new Thread(new MyRunnable(), "t1");
    Thread t2 = new Thread(new MyRunnable(), "t2");
    Thread t3 = new Thread(new MyRunnable(), "t3");

    t1.start();

    //start second thread after waiting for 2 seconds or if it's dead
    try {
        t1.join(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    t2.start();

    //start third thread only when first thread is dead
    try {
        t1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    t3.start();

    //let all threads finish execution before finishing main thread
    try {
        t1.join();
        t2.join();
        t3.join();
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    System.out.println("All threads are dead, exiting main thread");
}
```

Beispiel

```
public static void main(String[] args) {
    Thread t1 = new Thread(new MyRunnable(), "t1");
    Thread t2 = new Thread(new MyRunnable(), "t2");
    Thread t3 = new Thread(new MyRunnable(), "t3");

    t1.start();

    //start second thread after waiting for 2 seconds or if it's dead
    try {
        t1.join(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    t2.start();

    //start third thread only when first thread is dead
    try {
        t1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    t3.start();

    //let all threads finish execution before finishing main thread
    try {
        t1.join();
        t2.join();
        t3.join();
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    System.out.println("All threads are dead, exiting main thread");
}
```

```
> javac -classpath .:/run_dir/junit-4.12.jar:target/dependency/* -d . Ma
in.java
> java -classpath .:/run_dir/junit-4.12.jar:target/dependency/* Main
Thread Started::t1
Thread Started::t2
Thread ended::t1
Thread Started::t3
Thread ended::t2
Thread ended::t3
All dead
>
```

Exkurs: Threads

Implementiere ein Java-Programm, das die gerade und ungerade Zahlen von 1 bis 50 ausgibt.

Weak Memory Model

- der JVM-Bytecode entspricht nicht 100% dem erzeugten Machine-Code

```
//x eine lokale Variable

x++;
```

- wie viele Machine-Code Befehle sind generiert?
  - 4 - 6

- die JVM verwendet keine Register, sondern arbeitet mit Variablen, die auf einem Stack gespeichert sind

```
LLOAD_1      // push value from local variable #1
LCONST_1     // push value 1
LADD         // add 2 top-most values
LSTORE_1     // store value into local variable #1
```

Weak Memory Model

- die JVM benutzt ein schwaches Memory-Modell
- wie schon erwähnt dieses Modell erlaubt einige Reorderings beim write und read
- primär für Effizienz

```
x := a; y := 1; z := x;         //y := 1 might flush cache
x := a; r := x; y := 1; z := r; //r is a register, not memory
```

- statisch durch den Compiler, dynamisch durch den Prozessor
- das Modell definiert welche Reorderings tatsächlich erlaubt sind
- strong Memory-Modell: keine Reorderings erlaubt
- das Modell muss betrachten, ob die Operationen unabhängig voneinander sind

```
x := 1; r := x; cannot be reordered
r := x; x := 1; cannot be reordered
```

Weak Memory Model

- Mögliche Reorderings
  - Read-read
  - Write-read
  - Read-write
  - Write-write
- die meisten schwachen Memory-Modelle garantieren sequentielle Konsistenz: Wenn eine “No race Conditions” Bedingung erfüllt ist, dann ist das beobachtbare Verhalten des Programms wie bei einem starken Memory-Modell
- “No data race” kann eine sehr starke Einschränkung sein und zu unnötiger Synchronisation führen kann
- Der Begriff "beobachtbares Verhalten" hängt von der Programmiersprache ab
- Wir brauchen eine feine Kontrolle - `volatile`

Weak Memory Model

```
public class C {
  private volatile long l = 5;
  long incRet() { return l++; } //called from two threads
}
```

- Alle Lese- und Schreibzugriffe auf l sind atomar
- Alle Schreibzugriffe auf l sind für alle Threads sofort sichtbar
- In Bezug auf das Memory-Modell: keine Lese- und Schreibzugriffe auf l werden vor jedem write neu geordnet
- Bezogen auf den Speicher: l wird aus dem globalen Speicher gelesen und geschrieben, nicht aus den Thread-Caches
- führt keine Synchronisierung ein, beseitigt aber Möglichkeiten zur Optimierung und macht Zugriff teurer
- Grobe Richtlinie für die Verwendung von `volatile`
  - wenn ein Feld keine “Race Conditions” haben soll, sollte man `volatile` nicht verwenden
  - Wenn ein Feld “Race Conditions” haben wird und man diese nicht entfernen will bzw. kann, sollte man `volatile` verwenden

Java Standard Library

- Die Standardbibliothek von Java bietet weitere Datenstrukturen für komplexe, aber effiziente Bearbeitung von Daten
- Thread-Safe Collection sind weniger effiziente, aber intern “Race Conditions”-freie Versionen von Sammlungen
- Atomare Klassen kapseln Daten mit effizientem, atomarem Zugriff

Java Standard Library

- Atomare Klassen sind für Daten und Referenzen verfügbar
- Feste Operationen, die ohne synchronisierte Blöcke atomar sind
- Sind effizienter, aber weniger klarer Kontrollfluss

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;}
}
```

```
public class SynchronizedCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {c.incrementAndGet();}
    public void decrement() {c.decrementAndGet();}
    public int value() {return c.get();}
}
```

Thread Pool

- Ein ExecutorService verwaltet eine Reihe von Threads und akzeptiert Runnable-Instanzen

```
ExecutorService service = Executors.newCachedThreadPool(0,3);
//starts with 0 threads
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
//up to 3 threads running
```

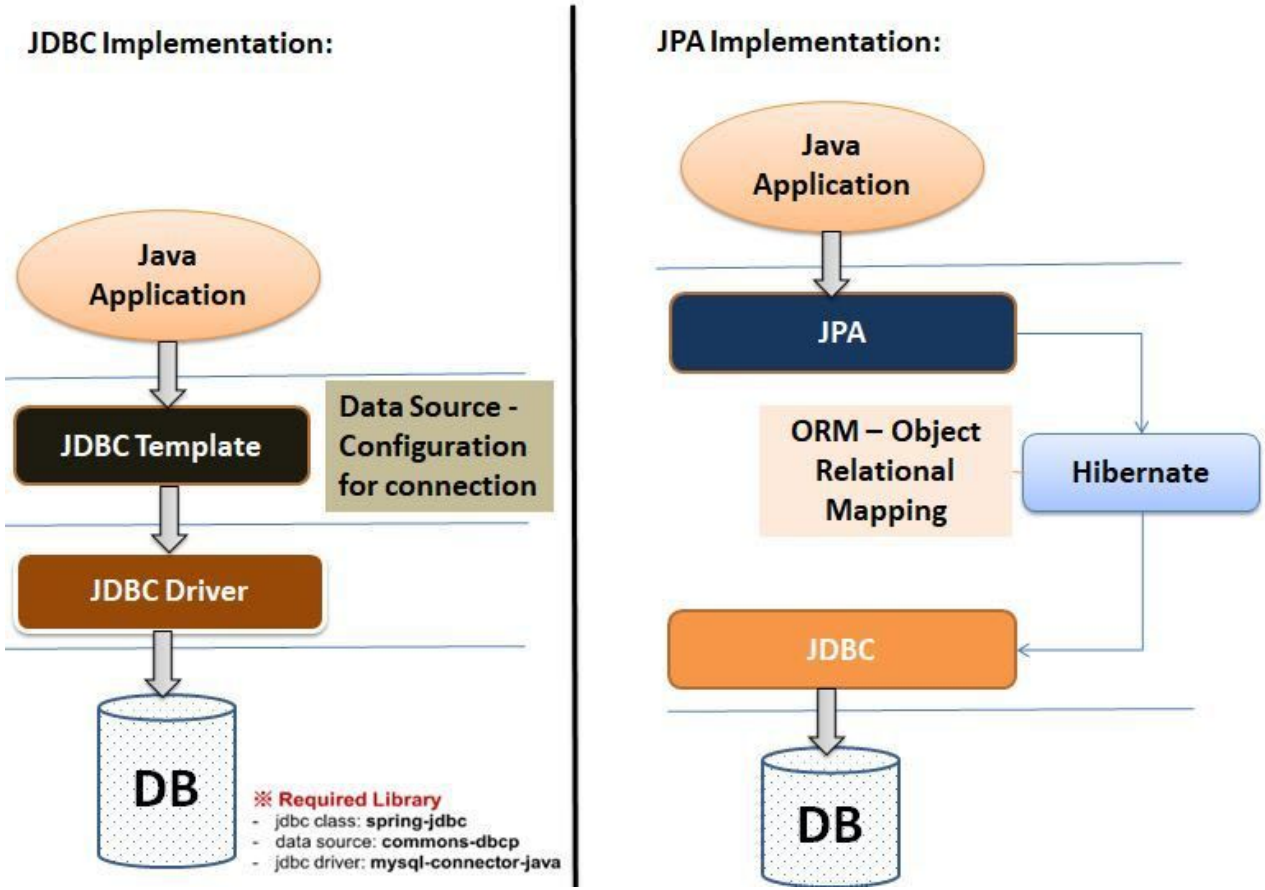
```
//has exactly 2 threads
ExecutorService service = Executors.newFixedThreadPool(2);
Future<Int> f = service.submit(() -> { /* do */ return 1;});
...
Int = f.get(); //essentially a join
```

Database

- database
- table
- row
- column

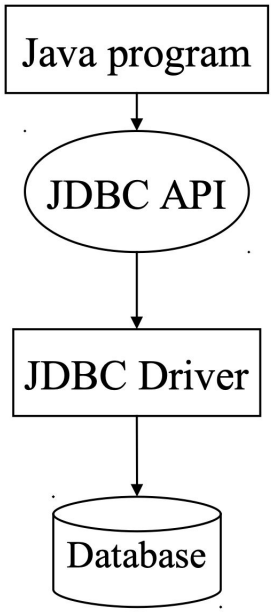
SQL

- JDBC = Java Database Connectivity
- JPA = Java Persistence API



JDBC

- JDBC = Java Database Connectivity
- Sammlung von Klassen und Interfaces zur Arbeit mit Datenbanken auf Basis von SQL
- Die Klassen befinden sich im Package java.sql
- Um mit einer Datenbank zu kommunizieren bedarf es eines Jdbc Drivers, der die Klasse driver.class implementiert
- Der jdbc-Driver enthält die wesentliche Funktionalität zur Kommunikation mit einer Datenbank



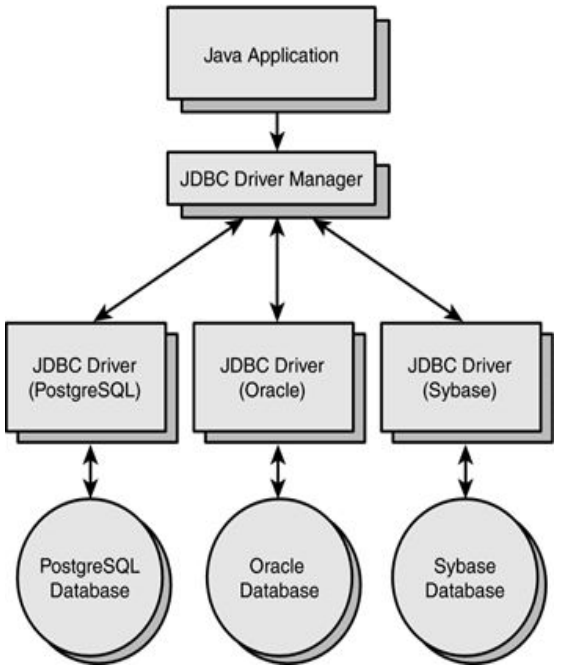
JDBC

```
Connection connection = DriverManager.getConnection(URL,
USER, PASS);

Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery("SELECT …");

  while (rs.next()) {
              int id = rs.getInt("id");
    …
}
```



Quellen, Driver, Driver Manager

- Datenbank

- jdbc:<Sub-Protocol>:<Datasource-Name>
- jdbc:mysql://localhost:3306/myDataBase

- Driver
  - Objekt, das die Verbindung mit einer DB vermittelt
  - implementiert JDBC Driver Schnittstelle für eine bestimmte DB

- Driver Manager
  - parsiert eine URL und lädt einen entsprechenden Driver
  - gibt ein Connection Objekt zurück

Connection

- repräsentiert eine Session mit einer DB
- man SQL Anweisungen über eine Connection ausführen
- mehrere Connection können parallel laufen
- benötigt um
  - SQL Anweisungen ausführen zu können
  - Meta-Information über die Verbindung
  - Commit, Rollback Änderungen

JDBC Statement

- erzeugt ein Objekt aus der Connection, das man für SQL Anweisungen verwenden kann
- createStatement()
- prepareStatement(String sql)
- ResultSet executeQuery(String sql)
  - Anweisungen, die ein einziges ResultSet zurückgeben
  - ResultSet = eine Tabelle, die ein Ergebnis repräsentiert
- int executeUpdate(String sql)
  - INSERT, UPDATE, DELETE

JDBC - Beispiel

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.27</version>
    </dependency>
</dependencies>
```

```
String DB_URL = "jdbc:mysql://localhost:3306/test";
String USER = "root";
String PASS = "";
```

JDBC - Beispiel

```java
package org.example;

import java.sql.*;

public class Main {
    public static void main(String[] args) throws SQLException {

        Connection conn = DriverManager.getConnection( url: "jdbc:mysql://localhost:3306/test", user: "root", password: "");

        Statement insert = conn.createStatement();

        String insert_string = "INSERT INTO Student(id, name, email) VALUES (10, 'Bob', 'bob@email.com')";
        insert.executeUpdate(insert_string);

        String insert_string_fancy = "INSERT INTO Student(id, name, email) VALUES (?, ?, ?)";

        PreparedStatement insert_fancy = conn.prepareStatement(insert_string_fancy);
        insert_fancy.setInt( parameterIndex: 1, x: 20);
        insert_fancy.setString( parameterIndex: 2, x: "Fob");
        insert_fancy.setString( parameterIndex: 3, x: "fob@email.com");

        insert_fancy.executeUpdate();

        Statement select = conn.createStatement();
        ResultSet result = select.executeQuery( sql: " SELECT * FROM Student");

        while (result.next()) {
            Student student = new Student();

            student.setId(result.getInt( columnLabel: "id"));
            student.setName(result.getString( columnLabel: "name"));
            student.setEmail(result.getString( columnLabel: "email"));

            System.out.println(student);
        }


    }
}
```

```java
package org.example;

public class Student {
    3 usages
    private int id;
    3 usages
    private String name;
    3 usages
    private String email;

    1 usage
    public Student() {}

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", email='" + email + '\'' +
                '}';
    }

    public int getId() { return id; }

    1 usage
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }

    1 usage
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    1 usage
    public void setEmail(String email) { this.email = email; }
}
```

JPA

- um Java Objekts persistieren zu können, braucht man theoretisch keinen SQL Code

- Object/Relation Mapping (ORM). JPA verwaltet Objekte und deren Beziehungen

- @Entity, @Id

```
@Entity
public class Student {

   @Id
    private int id;
    private String Name;
    private String Uni;
    // getters, setters, ctors
}
```

Entity

- eine persistente Abstraktion
- Java Klasse, die eine Tabelle in der DB repräsentiert
- Instanz eine Reihe in der Tabelle
- default Konstruktor sollte vorhanden sein (public, protected)
- Attribute dürfen nicht public sein
- @Entity Annotation muss ergänzt werden
- @Id Primärschlüssel
- EntityManager definiert Methoden um den Kontext anpassen zu können: create, remove, find
- Kontext = eine Menge von Objekten, die in einer DB existieren

Application Start

- javax.persistence.Persistence
- javax.persistence.EntityManagerFactory

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("MyPU");

EntityManager em = emf.createEntityManager();
```

- Container-Managed Transactions
- Application-Managed Transactions

```
EntityManager em = emFactory.createEntityManager();
EntityTransaction transaction = em.getTransaction();
transaction.begin();
…
em.getTransaction().commit();
```

Beziehungen

- Beziehungen sind gespeichert und erzeugt wenn Entities aus der DB eingelesen sind
- können unidirektional oder bidirektional sein
- one-to-one, many-to-many, one-to-many, many-to-one
- Cascading

JPA


```
@Entity
public class Employee {

  @OneToOne
  private Office office;


  public Office getOffice() {
     return office;
  }

  public void setOffice(Office office) {
     this.office = office;
  }

...
}
```

```
@Entity
public class Office {

  @OneToOne(mappedBy="office")
  private Employee owner;

  public Employee getEmployee() {
   return owner;
  }

  public void setEmployee(Employee employee) {
       owner = employee;
  }
...
}
```

JPA

```
@Entity
public class Employee {

  @ManyToOne
  private Department dept;

  public Department getDepartment() {
     return dept;
  }

  public void setDepartment(Department dept) {
     this.dept = dept;
  }

...
}
```

```
@Entity
public class Department {

  @OneToMany(mappedBy="dept")
  private Collection<Employee> employees = new
ArrayList<>();


  public Collection<Employee> getEmployees() {
   return employees;
  }

  public void setEmployees(Collection<Employee>
employees) {
       this.employees = employees;
  }
...
}
```

JPA - pom.xml

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.27</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.0.Final</version>
    </dependency>
</dependencies>
```

JPA - META-INF\persistence.xml

```
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

<persistence-unit name="TestDB">
       <class>Entities.Student</class>
   <properties>
       <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5Dialect"/>
       <property name="hibernate.hbm2ddl.auto" value="update"/>
       <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>
       <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/test"/>
       <property name="javax.persistence.jdbc.user" value="root"/>
       <property name="javax.persistence.jdbc.password" value=""/>
   </properties>
</persistence-unit>
</persistence>
```