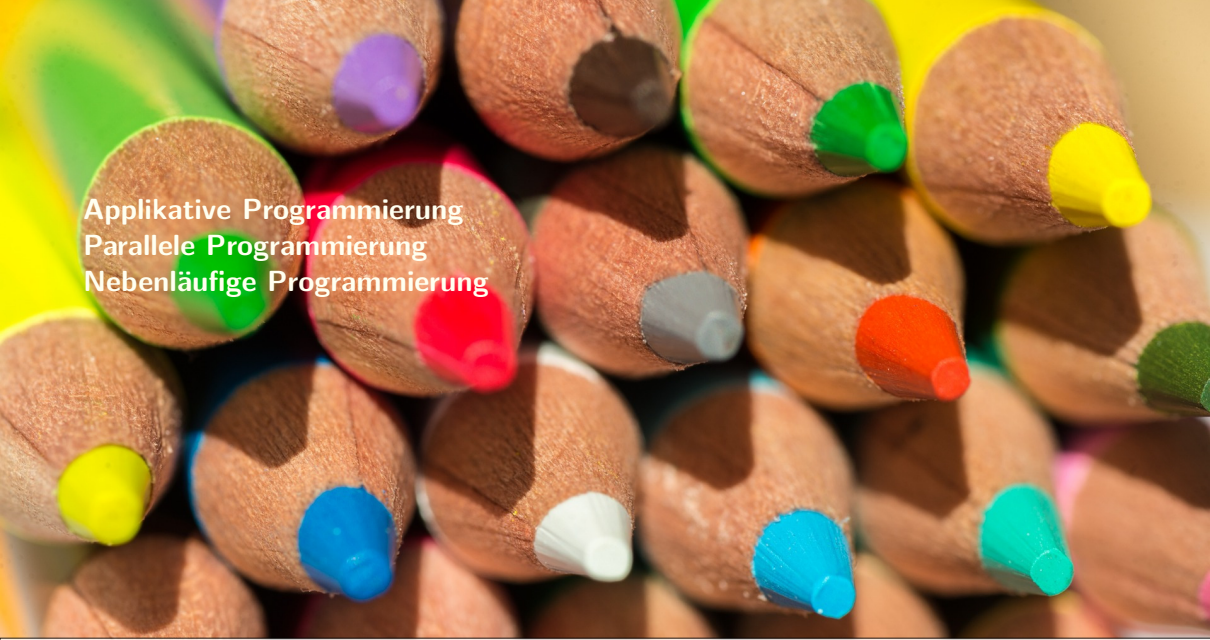

Applikative Programmierung
Parallele Programmierung
Nebenläufige Programmierung

Charakter der applikativen Programmierung

Variante der funktionalen Programmierung,
funktionale Formen mit Hilfsfunktionen parametrisiert und zusammengefügt,
verwendet vorgefertigte Teile mit wenig zusätzlichem Code

- → sehr produktiv
- → für komplexe Algorithmen geeignet
- → kurze, kompakte Programme mit wenigen Funktionen
- → Rekursion kaum sichtbar, nur im Hintergrund
- → beruht auf überschaubarer Menge zusammenpassender vorgefertigter funktionaler Formen
- → kann div. Programmiertechniken einfach unterstützen (Lazy-Evaluation, Parallelität, ...)
- → Programmstruktur kann sehr kreativ sein
- → hoher Abstraktionsgrad, strukturelle Abstraktion oder λ-Abstraktion
- → meist generisch und statisch typisiert mit Typinferenz
- → eher schwer lesbar

Beispiel zu Java-8-Streams mit Lambdas

```
public class ApplicativeCourse {
   public static void main(String[] args) {
      List<Stream<String>> ins = Stream.of(args).map(...).collect(...);
      Map<Integer, Stud> studs = ins.get(0).collect(...);
      List<Map<Integer, Integer>> lm = ins.stream().skip(1)
                                    .map(...).collect(Collectors.toList());
      System.out.println(studs.values().stream().map(s -> line(s, lm))
                              .collect(Collectors.joining("\n")));
   }
   private static String line(Stud s, List<Map<Integer, Integer>> lm) {
      List<Integer> ps = lm.stream().map(m -> m.getOrDefault(s.regNo, 0))
                                    .collect(Collectors.toList());
      return ... + ps.stream().map(i -> i.toString())
                     .collect(Collectors.joining("+"))
                 + " = " + ps.stream().reduce(0, Integer::sum);       } }
```

Lambda in Java

erzeugt Objekt mit einer Methode,
verwendbar wenn Kontext **funktionales Interface** als Typ festlegt,
funktionales Interface hat nur eine abstrakte Methode, entspricht Signatur des Lambdas

Bsp: `map(s -> line(s, lm))`

→ Signatur von `map`: `Stream<String> map(Function<Stud, String> mapper)`
→ funktionales Interface `Function<T,R>` enthält abstrakte Methode `R apply(T t)`
→ `s -> line(s, lm)` enthält Methode der Signatur `String apply(Stud s)`
→ diese Methode führt `return line(s, lm);` aus, `lm` aus Kontext

Lambda ähnelt Objekt erzeugt durch `new LambdaX(lm)`: intern effizienter implement.

```
class LambdaX extends Function<Stud, String> {
   private final List<Map<Integer, Integer>> lm;
   public LambdaX(List<Map<Integer, Integer>> lm) { this.lm = lm; }
   public String apply(Stud s) { return line(s, lm); }
}
```

Syntaktische Varianten von Lambdas in Java

Syntax	auszuführender Code	
`s -> line(s, lm)`	`{return line(s, lm);}`	ein Parameter ohne Typspez.
`(Stud s) -> line(s, lm)`	`{return line(s, lm);}`	Parametertyp angegeben
`(x, y) -> x + y`	`{return x + y;}`	mehrere Parameter
`() -> 42`	`{return 42;}`	kein Parameter
`a -> System.out.print(a)`	`{System.out.print(a);}`	Anweisung statt Ergebnis
`p -> { ... }`	`{ ... }`	ganzer Methodenrumpf
`p -> { ...; return p; }`	`{ ...; return p; }`	Rumpf enthält `return`
`Integer::sum`	wie `sum` in `Integer`	vorhandene Methode
`System.out::print`	`{System.out.print(...)}`	Aufruf in Objekt
`Klasse::new`	wie Konstruktor von `Klasse`	Lambda erzeugt Objekt

Operationen auf Java-8-Streams

erzeugende Operation	→ {	modifizierende Operation	} →	abschließende Operation
`ps.stream()`		`map(...)`		`collect(...)`
`Stream.of(...)`		`filter(...)`		`reduce(...)`
`Stream.generate(...)`		`skip(...)`		`forEach(...)`
`...`		`...`		`...`

`map(x -> ...)` Lambda bildet Wert auf anderen Wert ab, Strom der Ergebniswerte entsteht

`collect(Collectors.toList())` Methoden für Collection-Erzeugung in `Collectors`

`collect(HashMap::new, (c,d)->{...}, HashMap::putAll)`
Collection erzeugen, Daten `d` in Collection `c` einfügen, Collections zusammenfassen

`reduce(0, (x, y) -> x + y)` Anfangswert 0, Lambda fasst Werte schrittweise zusammen

`forEach(p -> System.out.println(p))` jeden Wert wie angegeben bearbeiten

Lazy-Evaluation in Java-8-Streams

```
import java.util.stream.Stream;
public class Elem {                                                    0-0
   private static int num = 0;                                         1-1
   private int id;                                                     2-2
   private Elem() { System.out.print((id = num++)); }                  3-3
   public static Stream<Elem> stream() {                               4-4
      return Stream.generate(Elem::new);                               5-5
   }                                                                   6-6
   public String toString() { return "-" + id; }                       7-7
}                                                                      8-8
   ...                                                                 9-9
   Elem.stream().limit(10).forEach(System.out::println);
```

Charakter der parallelen Programmierung

Ziel: möglichst kurze Rechenzeit unter Ausnutzung paralleler Hardware,
breite Palette an Hardware und zur Zielerreichung einsetzbaren Werkzeugen

- → Zielerreichung durch *Speedup* ausgedrückt: $S_p = T_1/T_p$ (größer ist besser)
- → Speedup abhängig von Details der Aufgabenstellung, Daten und Hardware
- → Wissen über diese Details nötig
- → Summe der Rechenzeit (p Recheneinheiten) höher als sequentielle Zeit T_1 (daher $S_p < p$)
- → $S_p > 1$ nur wenn Parallelausführung Zusatzaufwand überkompensiert
- → hoher Aufwand auf vielen Ebenen (Hardware, Softwareentwicklung, Wartung, ...)
- → zahlt sich nur zur Verarbeitung großer Mengen einheitlich strukturierter Daten aus
- → wichtig ist Finden einer Vorgehensweise, die unabhängige Datenblöcke ermöglicht
- → fertige Bibliotheken für häufige Einsatzzwecke

Naive Parallelisierung

Beispiel: `stream()` ersetzt durch `parallelStream()`

→ spaltet Daten auf mehrere parellele Ströme auf und fasst am Ende zusammen

→ Ausführungszeit bei naivem Vorgehen verlängert, nicht verkürzt (Abhängig von Details)

Ursachen für längere Laufzeit (Standard-Prozesor mit mehreren Kernen):

hoher Verwaltungsaufwand	z. B. Daten aufspalten, am Ende zusammenfassen
Umschalten zw. Aufgaben	einzelne Aufgaben dürfen nicht zu klein sein
gemeinsamer Speicher	Aufgaben verwenden unterschiedliche Daten (Race-Condition)
Datenabhängigkeiten	erfordern Synchronisation (Hintereinanderausführung)
Liveness-Probleme	Starvation, Deadlock, Livelock, . . . ; extreme Verzögerung möglich
Scheduling nötig	Effizienz von Strategie abhängig; nicht immer kontrollierbar
Zerlegung schwierig	sequentielle Teile oder Phasen bleiben
Engpässe	bei ungleichmäßiger Systemauslastung, z. B. Ein-/Ausgabe

Formen der Parallelität

- Hardware:
  - Prozessorkern führt mehrere Befehle gleichzeitig aus — VLIW, superscalar (implizit)
  - MIMD — Multiple-Instruction-Multiple-Data, mehrere unabhängige Befehlsströme
    - Prozessor mit mehreren Kernen — Speicheranbindung wird leicht zu Engpass
    - UMA-Multiprozessor — Uniform-Memory-Access, aufwändige Speicheranbindung
    - NUMA-Multiprozessor — Non-Uniform-Memory-Access, Speicherzugriffe ungleich
    - Rechnernetzwerk — häufig verteilte, nicht parallele Programmierung
  - SIMD — Single-Instruction-Multiple-Data; Vektoreinheiten, GPUs
- Software:
  - Prozess — Ausführungseinheit des Betriebssystems; bestmögliche Abgrenzung durch MMU
  - Thread — Ausführungseinheit innerhalb eines Prozesses; gemeinsamer Speicher
- Interprozesskommunikation:
  - Dateien — lesen und schreiben; Synchronisation durch Öffnen und Schließen (bzw. Locks)
  - Pipelines, Sockets — ähneln Dateien, potentiell unendlich; Sync. durch Datenverfügbarkeit
  - Shared-Memory — von mehreren Prozessen zugreifbar; Sync. explizit; systemabhängig

Beispiel (Primzahlberechnung)

```
public class Par {   public static long MAX = ...;
    private static long[] prims = { 2L, 3L, 5L, 7L, 11L, 13L };
    public static void main(String[] args) { long low = 16L, high = 256L;
        do {int size = prims.length; long[] nPrims = inRange(low, high);
            prims = Arrays.copyOf(prims, size + nPrims.length);
            System.arraycopy(nPrims,0,prims,size,nPrims.length);
            low = high; high = high * high; if (high > MAX) high = MAX;
        } while (low < MAX);
        System.out.println(prims.length);    }
    private static long[] inRange(long low, long high) {
        return LongStream.range(low, high).parallel()
                         .filter(Par::isPr).toArray();    }
    private static boolean isPr(long n) { long sqrt = (long)Math.sqrt(n);
        return Arrays.stream(prims).takeWhile(v -> v <= sqrt)
                                   .allMatch(v -> n % v != 0);    } }
```

Charakter der nebenläufigen Programmierung

viele gleichzeitig/überlappt ablaufende Handlungsstränge, um Programm zu vereinfachen, Ausführung der Handlungsstränge häufig durch Warten auf Ereignisse unterbrochen

→ vielfältige Ziele und Anwendungsgebiete, z. B. Webserver, Telefonanlage, Simulation
→ Bewältigung vieler Handlungsstränge (hohe Last) meist wichtiger als kurze Antwortzeiten
→ Anzahl der Handlungsstränge an Bedarf, nicht an Hardwarefähigkeiten ausgerichtet
→ in Java Handlungsstränge meist als *Threads* implementiert
→ Zugriffe auf gemeinsame Daten kaum vermeidbar, Synchronisation wichtig
→ Vermeidung von Liveness-Problemen notwendig, aber schwierig
→ zahlreiche Synchronisationsmechanismen, in Java hauptsächlich *Monitore*

Beispiel (Simulation Bakterienwachstum)

```
public class BactSim implements Runnable { ...
    private static final State[][] field = new State[...][...];
    public static void main(String[] args)
        { ...; new Thread(new BactSim(...)).start(); }
    private BactSim(int x, int y) { ... = field[x][y].occupy(); }
    public void run() {
        while (...) { try { Thread.sleep(...); }
                      catch (InterruptedException e) { break; }
                      if (...) new Thread(new BactSim(...)).start(); }
        field[x][y].gone();
}   }
class State { private int food = ...;
    public synchronized int occupy() { ... }
    public synchronized void gone() { ... }
}
```

→ Anzahl aktiver Threads ändert sich mit der Zeit

→ Warten simuliert Zufallsabhängigkeit

→ verfügbare Ressourcen (`food`) steuern Ablauf

→ `synchronized` garantiert gegenseitigen Ausschluss (mutual exclusion)

Monitor-Konzept in Java

Synchronized-Block: `synchronized(obj) {...}` Block als *kritischer Abschnitt*

→ Synchronisationsobjekt (`obj`) zu jedem Zeitpunkt für höchstens einen Thread „locked“

→ wenn `obj` vorher nicht „locked“, während Ausführung für ausführenden Thread „locked“

→ normale Ausführung wenn `obj` schon auf ausführenden Thread „locked“ (Rekursion)

→ sonst vor Ausführung warten bis Synchronisationsobjekt frei (nicht mehr „locked“)

Synchronized-Methode: `synchronized R m(T t) {...}` Rumpf als kritischer Abschnitt

→ wie Synchronized-Block, wobei als Synchronisationsobjekt implizit `this` verwendet

Warten auf Bedingung: `while (...) try { wait(); }` in kritischem Abschnitt
`catch (InterruptedException e) {...}`

→ `wait()` versetzt Thread in Wartezustand für Synchronisationsobjekt, „locked“ freigegeben

→ Bedingung nach Aufwecken neuerlich geprüft, weil Aufwecken jederzeit möglich

explizites Aufwecken: `notifyAll()` in kritischem Abschnitt

→ alle Threads im Wartezustand für Synchronisatonsobjekt werden aufgeweckt

→ weitere Ausführung erst wenn Synchronisationsobjekt wieder frei (neuerlich „locked“)

Beispiel (Buffer)

```
class Buffer {
   private final static int SIZE = ...;
   private String[] buffer = new String[SIZE];
   private int in, out;
   public synchronized void put(String v) {
      while (buffer[in] != null) try {wait();}
                                 catch (InterruptedException e) {return;}
      notifyAll();  buffer[in] = v;  in = (in + 1) % SIZE;
   }
   public synchronized String get() {
      while (buffer[out] == null) try {wait();}
                            catch (InterruptedException e) {return null;}
      notifyAll();  String res = buffer[out];
      buffer[out] = null;  out = (out + 1) % SIZE;  return res;
}  }
```

Beispiel (Producer-Consumer)

```
public class ProducerConsumer {
   public static void main(String[] args) {
      Buffer buffer = new Buffer();
      for (...) new Thread(new Producer(..., buffer)).start();
      for (...) new Thread(new Consumer(..., buffer)).start();
   }
}
class Producer implements Runnable {  private Buffer buffer; ...
   public Producer(..., Buffer buffer) { this.buffer = buffer; ... }
   public void run() { ...; for(...) buffer.put(...); ... }
}
class Consumer implements Runnable {  private Buffer buffer; ...
   public Consumer(..., Buffer buffer) { this.buffer = buffer; ... }
   public void run() {...; for(...) ... buffer.get() ... }
}
```

Liveness-Probleme

Starvation wichtige Programmteile bekommen nicht genug Ressourcen (etwa Rechenzeit)

- → Auswirkung: langsamer Programmfortschritt, kann zum Stillstand führen
- → Ursache: unwichtige Programmteile binden Ressourcen (schlechte Ressourcenverteilung)
- → Erkennen des Problems: ausgiebig Testen
- → Problembeseitigung: gezielte Steuerung, z. B. Zwischenschalten passender Puffer

Deadlock mehrere Threads warten gegenseitig auf exklusive Objektzugriffe, die sie halten

- → Auswirkung: gegenseitige dauerhafte Blockade, kein Programmfortschritt
- → Ursache: mehrere Threads halten und brauchen exklusive Zugriffe auf gleiche Objekte
- → Erkennen des Problems: ausgiebig Testen, Programmanalyse (statisch oder dynamisch)
- → Problembeseitigung: Timeout, exklusiver Zugriff nur in vorbestimmter Reihenfolge

Livelock ähnelt Deadlock, statt Warten wird nachgefragt, ob exklusiver Zugriff möglich

- → Auswirkung und Ursache wie bei Deadlock
- → Erkennen des Problems: ausgiebig Testen, Programmanalyse kaum zielführend
- → Problembeseitigung: Timeout, nicht aktiv warten, nicht „schrittweise Anfragen“