

1. Betriebssysteme und Systemprogrammierung

1.1. Einführung

1.2. Computer-Hardware: Ein Kurz-Überblick

1.3. Instruktionsarchitektur (Instruction Set Architecture, ISA)

1.4. Virtuelle Maschinen

Teil 3



1.5. Java und die Java Virtual Machine

1.6. Zusammenfassung (Kapitel 1)

1.5. Java und die „Java Virtual Machine“

[1.5.1. Was ist die „Java Virtual Machine“ ?](#)

[1.5.2. Anfang und Ende einer Laufzeit-Instanz](#)

[1.5.3. Grundlegendes zu Multitasking und Multithreading](#)

[1.5.4. Erzeugung von Threads in Java](#)

[1.5.5. Die Struktur der Java Virtual Machine](#)

[1.5.6. Die Datentypen](#)

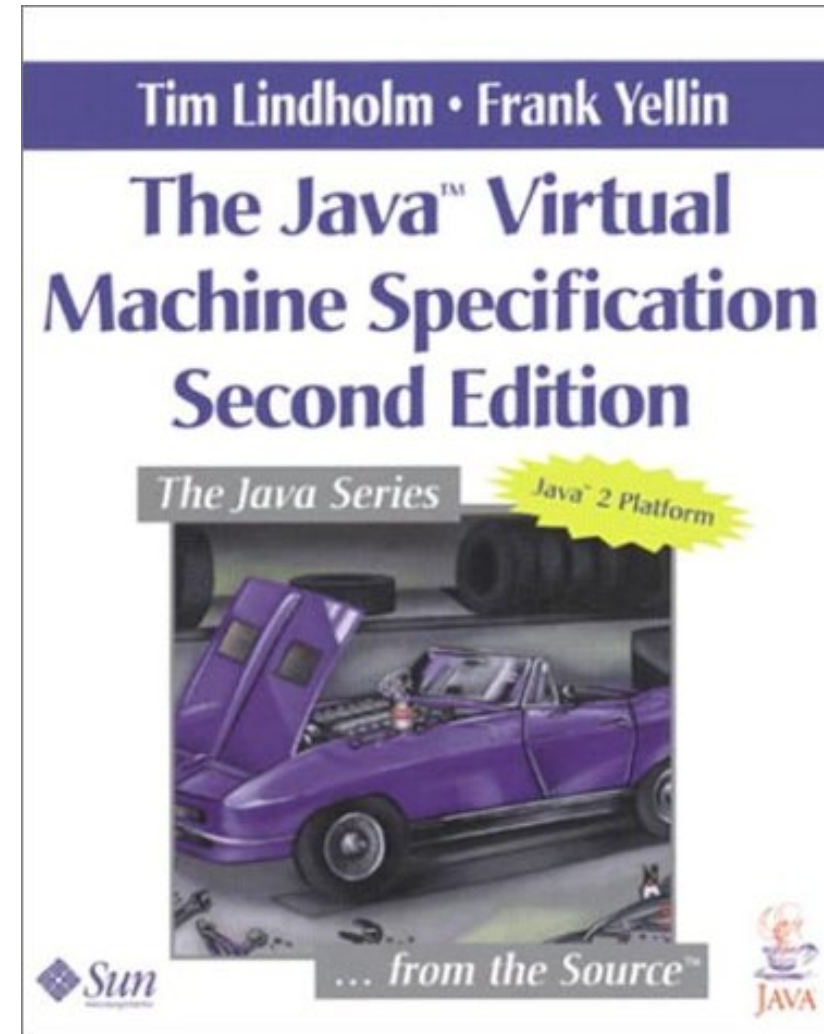
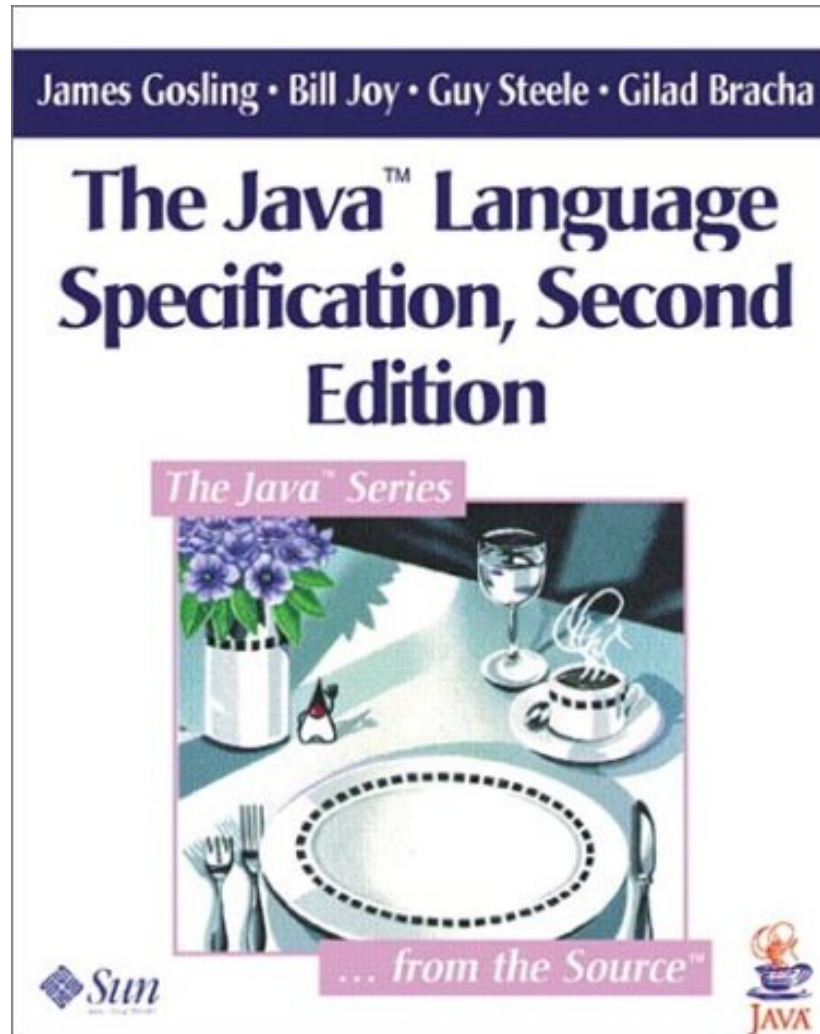
[1.5.7. Der Befehlssatz](#)

[1.5.8. Simulation der Java Virtual Machine](#)

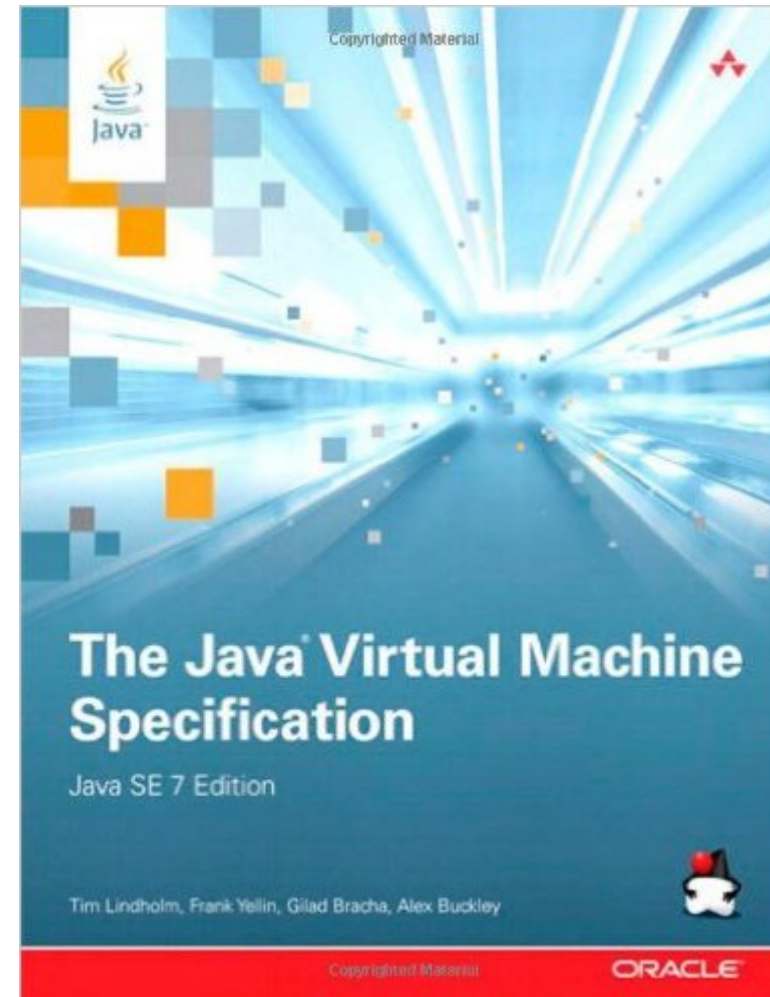
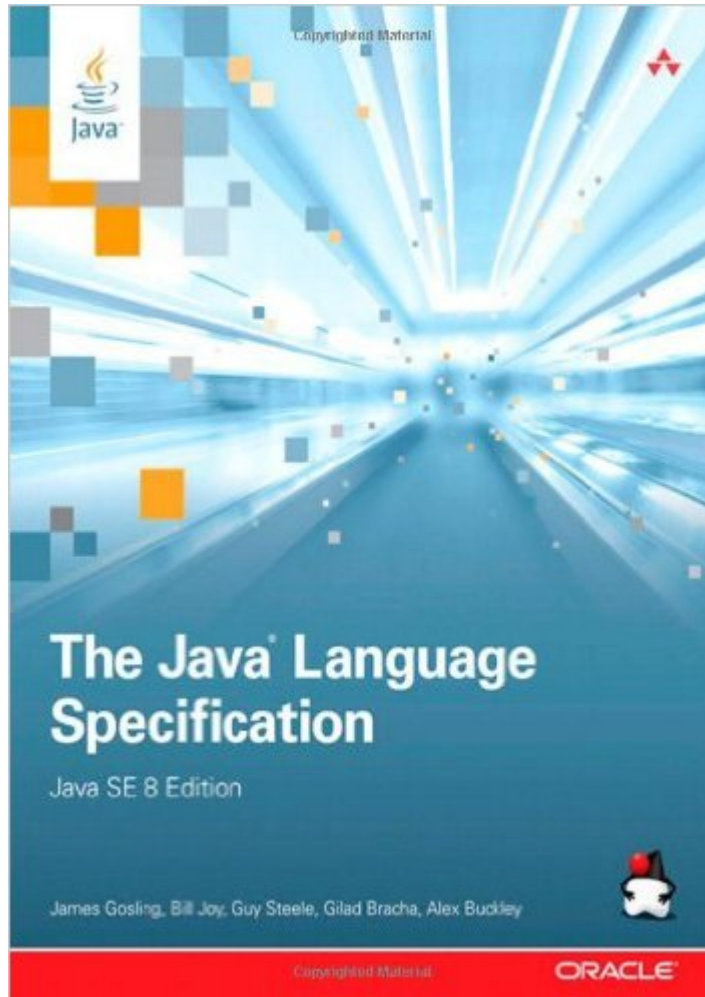
Literaturhinweis:

Neben den jeweils angegebenen Online-Quellen basieren wesentliche Teile dieses Kapitels auf:
Bill Venners, „Inside the JAVA Virtual Machine“, McGraw-Hill, 1998

Java: Die Sprache und die virtuelle Maschine



Java: Die Sprache und die virtuelle Maschine (Update)



The Java Virtual Machine Specification - Netscape

File Edit View Go Bookmarks Tools Window Help


http://java.sun.com/docs/books/vmspec/

Search

Mail AIM Home Radio My Netscape Search Shop Bookmarks Instant Message WebMail Calendar Radio People Yellow Pages

The Java Virtual Machine Specification

developers.sun.com

The Source for Developers
A Sun Developer Network Site

» Products & Technologies

» Technical Topics



Developers Home > Products & Technologies > Java Technology > J2SE > Community > Bookshelf >

Join a Sun Developer Network Community

Profile and Registration | Why Register?


Books & Authors

The Java Virtual Machine Specification

 Printable Page

Tim Lindholm • Frank Yellin


The Java™ Virtual Machine Specification
Second Edition








The Java™ Virtual Machine Specification, Second Edition is now available.

In *The Java Virtual Machine Specification, Second Edition*, Sun's designers of the Java virtual machine provide comprehensive coverage of the Java virtual machine class file format and instruction set. In addition, the book contains directions for compiling the virtual machine with numerous practical examples to clarify how it operates in practice. The book also demonstrates the Java virtual machine's powerful verification techniques. In all, the book provides sufficient detail to enable you to implement your own fully-compatible Java virtual machine, or on the other hand, to just really understand what makes the Java technology work.

 [Book description](#)

 [Preface](#)

 [Errata](#)

 Order this book through

[DigitalGuru](#)

[amazon.com](#)

You may print this book once. For the complete copyright notice, see [Copyright](#).


[View HTML](#)

[Download HTML \(tar.Z, ~492K\)](#)

[Download HTML \(tar.gz, ~325K\)](#)

[Download HTML \(zip, ~372K\)](#)

The Java Virtual Machine Specification [maintenance information](#).

 Done



Copyright © 2023 Prof. Dr.

Teil 3

5

The Java Language Specification - Netscape

File Edit View Go Bookmarks Tools Window Help

http://java.sun.com/docs/books/jls/ Search


Mail AIM Home Radio My Netscape Search Shop Bookmarks Instant Message WebMail Calendar Radio People Yellow Pages

The Java Language Specification

developers.sun.com >> search tips | Search: in Developers' Site

 The Source for Developers
A Sun Developer Network Site

» Products & Technologies
» Technical Topics




Developers Home > Products & Technologies > Java Technology > Reference > Documentation > Books >

Join a Sun Developer Network Community
Profile and Registration | Why Register?

Books

The Java Language Specification

 Printable Page



The Java Language Specification, Second Edition - Written by the inventors of the technology, this book is the definitive technical reference for the Java programming language. If you want to know the precise meaning of the language's constructs, this is the source for you.

The book provides complete, accurate, and detailed coverage of the syntax and semantics of the Java programming language. It describes all aspects of the language, including the semantics of all types, statements, and expressions, as well as threads and binary compatibility.

James Gosling is a Fellow and Vice President at Sun Microsystems, the creator of the Java programming language, and one of the computer industry's most noted programmers. He is the 1996 recipient of Software Development's "Programming Excellence Award." He previously developed NeWS, Sun's network-extensible window system, and was a principal in the Andrew project at Carnegie Mellon University, where he earned a Ph.D. in Computer Science.

Bill Joy is founder and Vice President of Research at Sun Microsystems, where he has led the company's technical strategy, working on both hardware and software architecture. He is well known as the creator of the Berkeley version of the UNIX operating system, for which he received a lifetime achievement award from the USENIX Association in 1993. He received the ACM Grace Murray Hopper Award in 1986. Joy has had a central role in shaping the Java programming language.

Guy L. Steele Jr. is a Distinguished Engineer at Sun Microsystems, where he is responsible for the specification of the Java programming language as well as research in parallel algorithms. He is well known as the co-creator of the Scheme programming language and for his reference books for the C programming language (with Samuel Harbison) and for the Common Lisp programming language. Steele received the ACM Grace Murray Hopper Award in 1988 and was named an ACM Fellow in 1994.

Gilad Bracha is Computational Theologist at Sun Microsystems, and a researcher in the area of object-oriented programming languages. Prior to joining Sun, he worked on Strongtalk, the Anomorphic Smalltalk System. He holds a B.Sc. in Mathematics and Computer Science from Ben Gurion University in Israel and a Ph.D. in Computer Science from the University of Utah.

Done

1.5.1. Was ist die „Java Virtual Machine“ ?

The Java virtual machine is the cornerstone of the Java and Java 2 platforms. It is the component of the technology responsible for its **hardware- and operating system- independence**, the **small size of its compiled code**, and its **ability to protect users from malicious programs**.

The Java virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. It is reasonably common to implement a programming language using a virtual machine; the best-known virtual machine may be the P-Code machine of UCSD Pascal.

The first prototype implementation of the Java virtual machine, done at Sun Microsystems, Inc., emulated the Java virtual machine instruction set in software hosted by a handheld device that resembled a contemporary Personal Digital Assistant (PDA). Sun's current Java virtual machine implementations, components of its Java™ 2 SDK and Java™ 2 Runtime Environment products, emulate the Java virtual machine on Win32 and Solaris hosts in much more sophisticated ways. However, the **Java virtual machine does not assume any particular implementation technology, host hardware, or host operating system. It is not inherently interpreted, but can just as well be implemented by compiling its instruction set to that of a silicon CPU. It may also be implemented in microcode or directly in silicon.**

The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the class file format. A class file contains Java virtual machine instructions (or *bytecodes*) and a symbol table, as well as other ancillary information.

For the sake of **security**, the Java virtual machine imposes **strong format** and **structural constraints** on the code in a class file. However, any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine. Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java virtual machine as a delivery vehicle for their languages.

Quelle: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/Introduction.doc.html#4560>



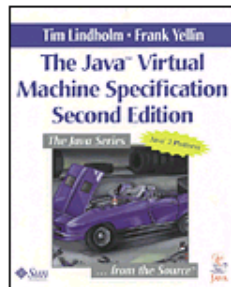
Products & APIs
Developer Connection
Docs & Training
Online Support
Community Discussion
Industry News
Solutions Marketplace
Case Studies

A-Z Index •

Search

THE SOURCE FOR JAVA™ TECHNOLOGY
java.sun.com

THE JAVA™ VIRTUAL MACHINE SPECIFICATION



The Java™ Virtual Machine Specification, Second Edition is now available.

In *The Java™ Virtual Machine Specification, Second Edition*, Sun's designers of the Java virtual machine provide comprehensive coverage of the Java virtual machine class file format and instruction set. In addition, the book contains directions for compiling the virtual machine with numerous practical examples to clarify how it operates in practice. The book also demonstrates the Java virtual machine's powerful verification techniques. In all, the book provides sufficient detail to enable you to implement your own fully-compatible Java virtual machine, or on the other hand, to just really understand what makes the Java technology work.

- ✻ [Book description](#)
- ✻ [Preface](#)
- ✻ [Errata](#)
- ✻ [Order this book through amazon.com](#)

You may print this book once. For the complete copyright notice, see [Copyright](#).

[View HTML](#)

[Download HTML](#) (tar.Z, ~492K)

[Download HTML](#) (tar.gz, ~325K)

[Download HTML](#) (zip, ~372K)

„Nur“ eine Blaupause für reale Implementierungen

The Java virtual machine specification has been written to fully document the design of the Java virtual machine. It is essential for compiler writers who wish to target the Java virtual machine and for programmers who want to implement a compatible Java virtual machine. **It is also a definitive source for anyone who wants to know exactly how the Java programming language is implemented.**

The Java virtual machine is an abstract machine. References to the *Java virtual machine* throughout this specification refer to this abstract machine rather than to Sun's or any other specific implementation. **This book serves as documentation for a concrete implementation of the Java virtual machine only as a blueprint documents a house. An implementation of the Java virtual machine (known as a runtime interpreter) must embody this specification, but is constrained by it only where absolutely necessary.**

The Java virtual machine specified here will support the Java programming language specified in *The Java™ Language Specification* (Addison-Wesley, 1996). It is compatible with the Java platform implemented by Sun's JDK releases 1.0.2 and 1.1 and the Java™ 2 platform implemented by Sun's Java™ 2 SDK, Standard Edition, v1.2 (formerly known as JDK release 1.2).

We intend that this specification should sufficiently document the Java virtual machine to make possible compatible clean-room implementations. **If you are considering constructing your own Java virtual machine implementation, feel free to contact us to obtain assistance to ensure the 100% compatibility of your implementation.**

**Auszug aus dem Vorwort zu: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“
<http://java.sun.com/docs/books/vmspec>**

1.5.2. Anfang und Ende einer Laufzeit-Instanz

Jede **Java-Anwendung** läuft

- **innerhalb einer Laufzeit-Instanz** einer
- **konkreten Implementierung**

der abstrakten Spezifikation der Java Virtual Machine.

Im Folgenden bezeichne der Begriff „**Java Virtual Machine**“

- die **abstrakte Spezifikation** der Java Virtual Machine
- eine **reale Implementierung** der Java Virtual Machine und auch
- eine **bestimmte (Laufzeit-) Instanz** der Java Virtual Machine.

Starten einer Instanz der Java Virtual Machine:

0. Hole und installiere eine Implementierung der **Java Virtual Machine**

1a. Starte eine Java Virtual Machine (Kommando hierzu: Implementierungsabhängig).

1b. Gib dieser Java Virtual Machine den **Namen der zuerst aufzurufenden Klasse**.

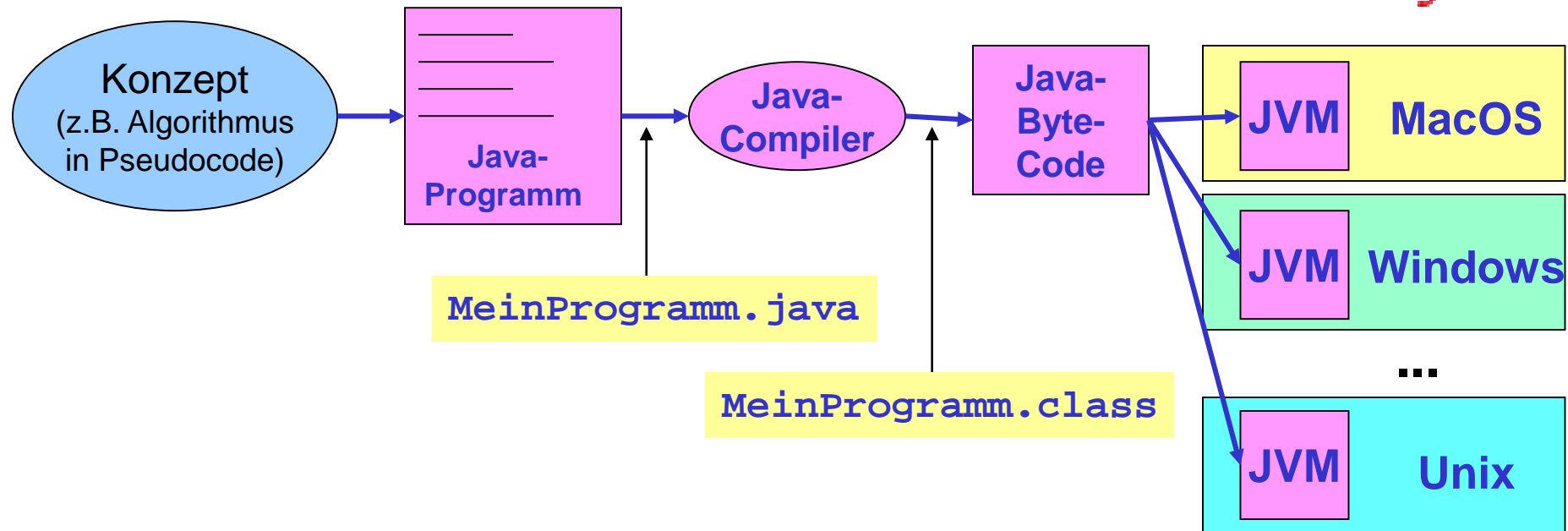
Beenden einer Instanz der Java Virtual Machine:

... später.

„Write once, run everywhere“

Der Java-Compiler erzeugt ein **Programm für eine „virtuelle Maschine“**:

„Java Virtual Machine“ (JVM)



Der **Maschinencode der „Java Virtual Machine“**

- ist auf einem realen Rechner **nicht unmittelbar lauffähig**,
- muss auf dem jeweiligen Rechner einem **Interpreter oder Compiler** übergeben werden.

Beispiel: „SagMal“

Als Beispiel betrachten wir nun eine ganz einfache Java-Anwendung, welche die beim Aufruf angegebenen Strings als Echo wiedergibt:

```
public class SagMal
{
    public static void main (String[] args)
    {
        int len = args.length;
        for (int i = 0; i < len; ++i)
        {
            System.out.println (args [i]);
        }
    }
}
```

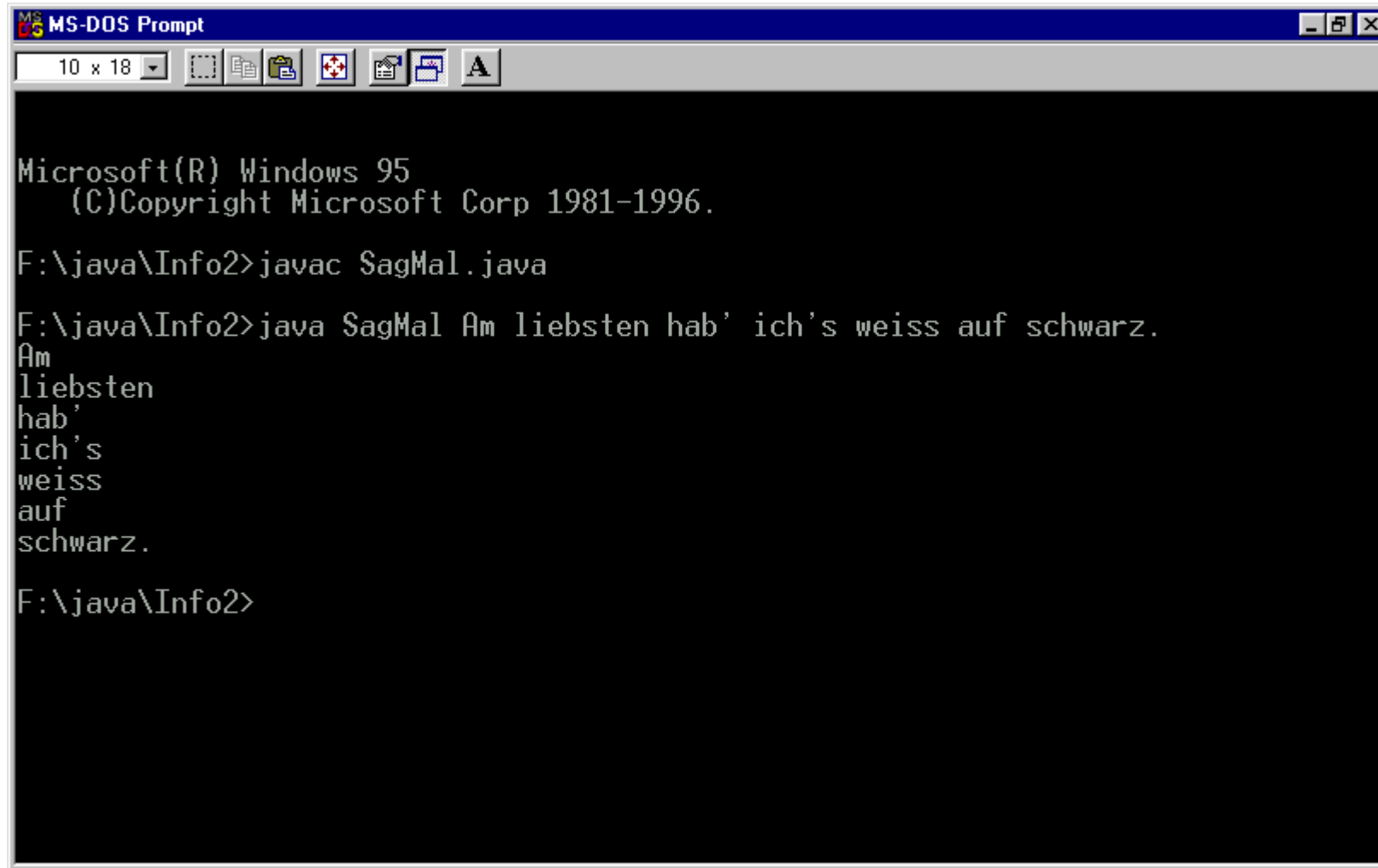
Startpunkt für den
Kontrollfluss („Thread“)

Dieses Programm muss zunächst in den Java-Byte-Code übersetzt werden.

Dadurch entsteht ein **Programm im Maschinencode der Java Virtual Machine.**

Zur Bearbeitung genau dieses Programms wird eine Java Virtual Machine instanziiert und ggf. mit Eingaben versorgt.

Erzeugung des Byte-Codes und Aufruf



The image shows a screenshot of an MS-DOS Prompt window. The title bar reads "MS-DOS Prompt". The window contains the following text:

```
Microsoft(R) Windows 95
(C)Copyright Microsoft Corp 1981-1996.

F:\java\Info2>javac SagMal.java

F:\java\Info2>java SagMal Am liebsten hab' ich's weiss auf schwarz.
Am
liebsten
hab'
ich's
weiss
auf
schwarz.

F:\java\Info2>
```

„Fäden“ und „Geisterfäden“ ?

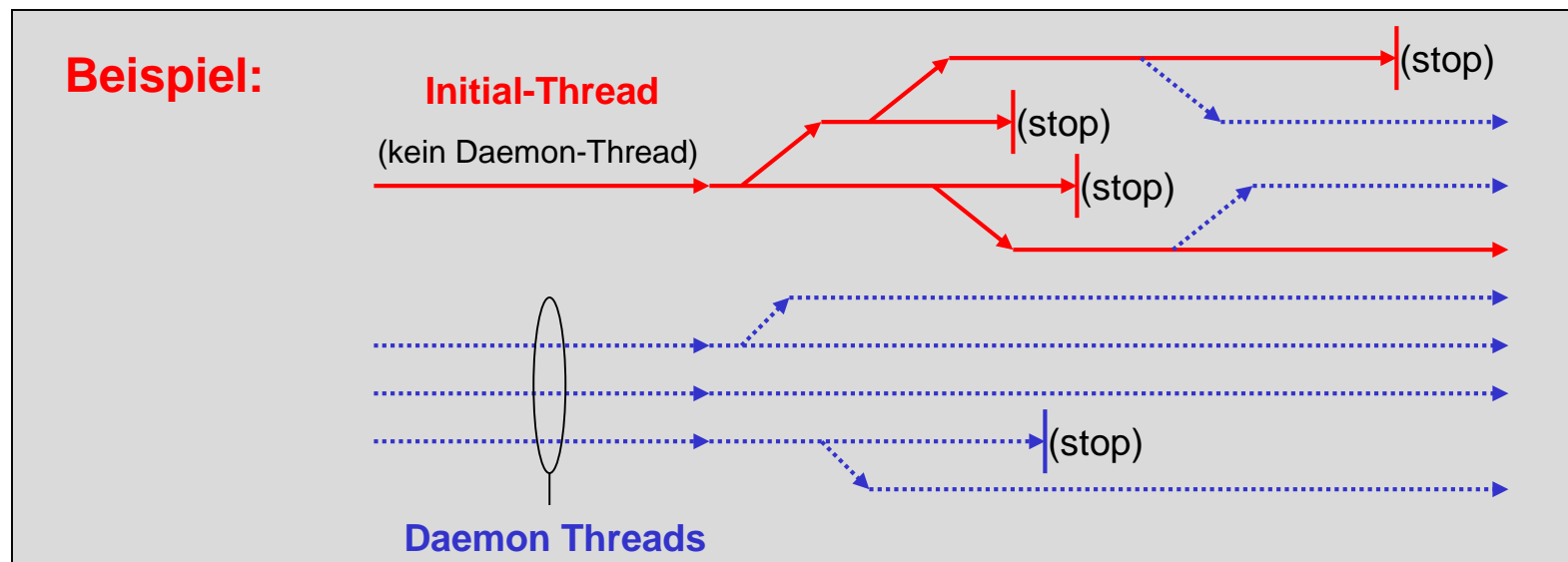
Der „rote Faden“ (= der Kontrollfluss), dem die Bearbeitung eines Programms von Befehl zu Befehl folgt, heißt im Englischen **„Thread“**.

Wird eine Java-Anwendung gestartet, dann gibt es

- **innerhalb dieser Anwendung zunächst genau einen Kontrollfluss**, dessen Anfang durch die `main()` - Methode eindeutig festgelegt wird
- **innerhalb der Virtual Machine aber weitere Kontrollflüsse**
z.B. für Garbage Collection, die als **„daemon threads“** (daemon = Dämon, Geist) bezeichnet werden.

Threads der Anwendung (insbesondere der Initial-Thread) können **weitere Kontrollflüsse starten**, die - je nach Hardware - zeitlich verzahnt oder auch wirklich parallel bearbeitet werden können.

Die Anwendung kann derartige Threads als **„daemon threads“** markieren.



Das Ende einer Instanz der Java Virtual Machine

Offenbar gilt:

- Innerhalb **einer Virtual Machine** kann es **mehrere (Anwendungs-) Threads** geben.
- Jeder **Thread** existiert **nur innerhalb der zugehörigen Virtual Machine**.

Eine detaillierte **Behandlung von „Multithreading“** erfolgt **später**. Hier betrachten wir nur die Bedeutung der Threads für die Lebensdauer der zugehörigen Virtual Machine:

Eine Instanz der Java Virtual Machine wird beendet,

- wenn **alle Anwendungs-Threads beendet** sind oder
(von der Anwendung gestartete Daemon-Threads sind irrelevant)
- wenn die **Anwendung** die **`exit()` - Methode aufruft**
(sofern sie dies darf, vgl. security management).

1.5.3. Grundlegendes zu Multitasking und Multithreading

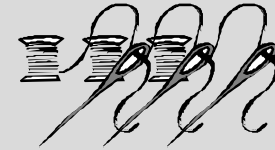
Multitasking

Die **Fähigkeit eines Betriebssystems**, mehrere Tasks, d.h. mehrere Anwendungsprogramme oder Teile von Anwendungsprogrammen, parallel ausführen zu können.

Quelle: Der Brockhaus – Computer und Informationstechnologie, 2003

Multithreading

Die **parallele Ausführung von zwei oder mehr Threads** (Programmfäden) eines Anwendungsprogramms. ...



Multithreading ist zu unterscheiden von Multitasking, bei dem mehrere Anwendungsprogramme parallel ausgeführt werden.

Quelle: Der Brockhaus – Computer und Informationstechnologie, 2003

Multitasking kann **ohne aktive Kooperation des Anwendungsprogrammierers** eingesetzt werden.

Multithreading setzt voraus, dass

- das **Anwendungsprogramm als Multithread-Anwendung programmiert** ist und
- das **Betriebssystem** dieses Verfahren **unterstützt**.



Multithread-Anwendung

Ein Programm, bei dem **parallel ausführbare Programmeinheiten** mit Threads **programmiert** sind, sodass sie bei der Ausführung tatsächlich parallel abgearbeitet werden (wenn das Betriebssystem dies unterstützt).

Quelle: Der Brockhaus – Computer und Informationstechnologie, 2003

Offenbar sind

- die **zeitliche Ablaufsteuerung** und
 - die **Synchronisation der Threads**
- nicht trivial.

Auch die Fehlerbereinigung (Debugging) macht häufig Probleme.

Hinzu kommt, dass sich **nicht jede Anwendung sinnvoll in Threads splitten** lässt.

In vielen Fällen lassen sich aber große (Laufzeit-) Gewinne erzielen.

Hyper-Threading Technology: Advanced Performance On Multithreaded Applications - Netscape

http://www.intel.com/technology/hyperthread/m

Hyper-Threading Technology: Advanced P...

intel.

US Home | Intel Worldwide

Where to Buy | Training & Events | Contact Us | About Intel

Search

Resource Centers | Products & Services | Solutions | Technologies & Trends | Support & Downloads

• Hardware Design

Design Components & Products

Design Solutions

Developer Programs

Tools & Downloads

Technologies & Initiatives

Events, Training & Publications

Hardware Design | Technologies and Initiatives | Hyper-Threading Technology

Hyper-Threading Technology

Hyper-Threading Technology enables multi-threaded software applications to execute threads in parallel. This level of threading technology has never been seen before in a general-purpose microprocessor. Internet, e-Business, and enterprise software applications continue to put higher demands on processors. To improve performance in the past, threading was enabled in the software by splitting instructions into multiple streams so that multiple processors could act upon them. Today with Hyper-Threading Technology, processor-level threading can be utilized which offers more efficient use of processor resources for greater parallelism and improved performance on today's multi-threaded software.

Hyper-Threading Technology provides thread-level-parallelism (TLP) on each processor resulting in increased utilization of processor execution resources. As a result, resource utilization yields higher processing throughput. Hyper-Threading Technology is a form of simultaneous multi-threading technology (SMT) where multiple threads of software applications can be run simultaneously on one processor. This is achieved by duplicating the architectural state on each processor, while sharing one set of processor execution resources. Hyper-Threading Technology also delivers faster response times for multi-tasking workload environments. By allowing the processor to use on-die resources that would otherwise have been idle, Hyper-Threading Technology provides a performance boost on multi-threading and multi-tasking operations for the Intel NetBurst® microarchitecture.

This technology is largely invisible to the platform. In fact, many

Training Materials

- Introduction to Hyper-Threading Technology
- Detecting Hyper-Threading Technology Enabled Systems
- Hyper-Threading Technology Demos for Server Operations

White Papers

- Hyper-Threading Technology on Intel® Xeon™ Processor Family for Servers
- Microsoft® Windows Based Servers and Intel® Hyper-Threading Technology
- Hyper-Threading Technology Architecture and Microarchitecture

Hyper-Threading Enabled Products

- Intel® Pentium® 4 processor with HT Technology Extreme

Ein Beispiel für Multithreading: DAC

Wir betrachten als Beispiel nun ein **Datenerfassungssystem** (Data-acquisition system, DAC) zur Überwachung eines physikalischen Prozesses. Dieses umfasst die folgenden Komponenten:

Collect

sammelt Daten von einem externen Sensor (mehrere Werte erfassen, dann weitermelden)

Log

archiviert die gelesenen Daten, z.B. auf Platte (mehrere Plattenzugriffe)

Stat

wertet die Daten statistisch aus und erkennt signifikante Abweichungen
(verbraucht nur CPU-Zeit)

Report

gibt das Ergebnis der Auswertung an einen Drucker, Roboter, ...
(mehrere Aktionen)

Eine natürliche (sequentielle) Implementierung (Pseudo-Code)

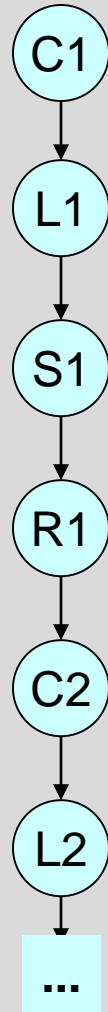
```
program single_THREAD_DAC
...
begin
  while true do
    begin
      collect_ad;           // Daten vom Sensor (AD-Wandler)
      log_d;                // Archivierung
      stat_p;               // statistische Bearbeitung
      report_pr             // drucke signifikante Änderung
    end {while}
  end {single_THREAD_DAC}
```

Beobachtungen:

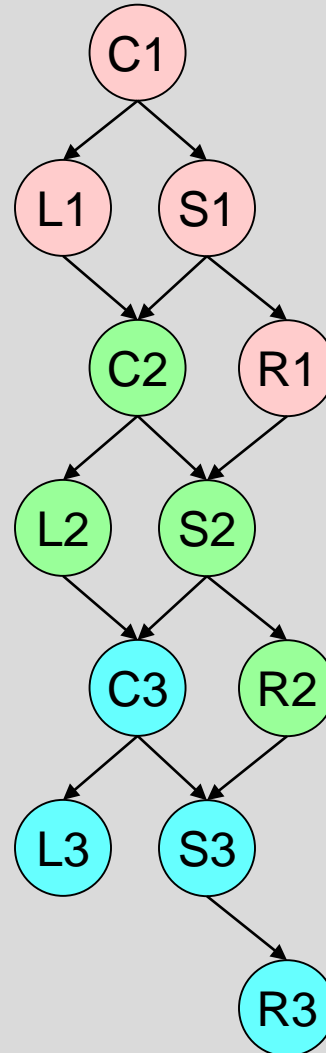
- a) **Stat** und **Log** können gleichzeitig ablaufen (völlig unabhängig voneinander)
- b) **Report** setzt das Ende von **Stat** voraus
- c) **Collect** setzt das Ende von **Stat** und **Log** voraus (Forderung kann ggf. entfallen)
- d) **Collect** und **Report** können gleichzeitig ablaufen
- e) **Stat** und **Log** setzen **Collect** voraus

Präzedenz-Graphen

Sequentiell:



Parallel:



Es ergibt sich der Eindruck, dass durch Multithreading eine **massive Beschleunigung** erzielbar ist.

Eine parallele Implementierung (Pseudo-Code)

```
process collect;  
begin  
  while true do  
    begin  
      wait_signal_from (log, stat);  
      collect_ad;  
      send_signal_to (log, stat);  
    end {while}  
  end {collect}
```

**Gestartet von Parent
durch 2 Signale.**

```
process stat;  
begin  
  while true do  
    begin  
      wait_signal_from (report, collect);  
      stat_p;  
      send_signal_to (report, collect);  
    end {while}  
  end {stat}
```

**„Report“-Signal zuerst
durch Parent, wartet
dann auf Collect.**

```
process log;  
begin  
  while true do  
    begin  
      wait_signal_from (collect);  
      log_d;  
      send_signal_to (collect);  
    end {while}  
  end {log}
```

**Wartet auf Ende des
1. Durchlaufs von Collect.**

```
process report;  
begin  
  while true do  
    begin  
      wait_signal_from (stat);  
      report_pr;  
      send_signal_to (stat);  
    end {while}  
  end {report}
```

**Wartet auf Ende des
1. Durchlaufs von Stat**

```
{parent process}  
begin {multithread_DAC}  
  initialize_environment;  
  send_signal_to (collect, collect, stat);  
  initiate collect, log, stat, report  
end {multithread_DAC}
```

**Signale an
Collect und Stat** ←

Input-Puffer und Output-Puffer für die Daten

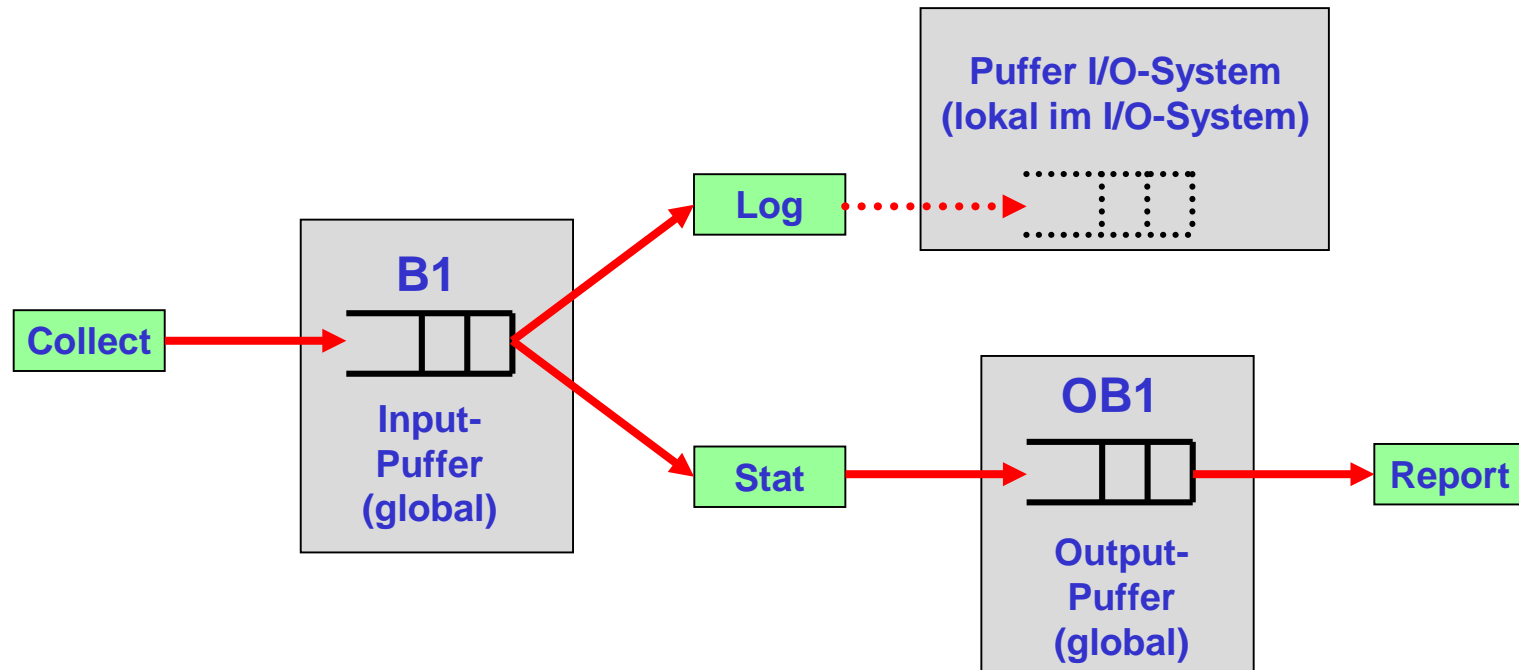
Es gebe 2 globale Pufferplätze:

B1: Input-Puffer

- Collect legt dort den Input ab
- Log und Stat holen von dort Daten
- Strikt read-only für Log und Stat

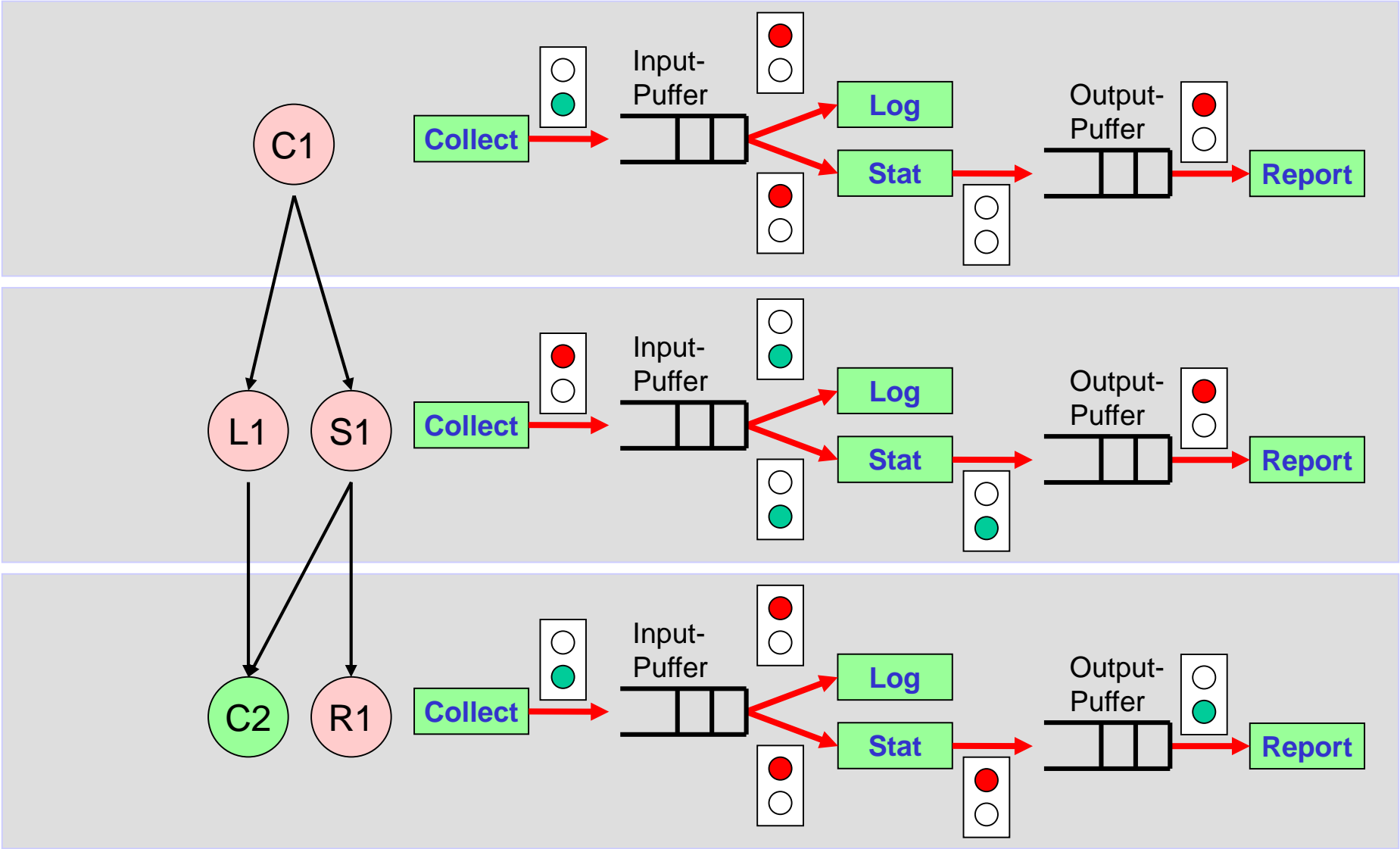
OB1: Output Puffer

- Stat legt dort Daten für Report ab





Der Plattenzugriff laufe über einen internen Puffer des Input/Output-Systems.
Somit kann Log den (globalen) Input-Puffer für Zugriff durch Collect freigeben,
sobald Log die Daten in diesen (lokalen) I/O-Puffer geschrieben hat.

Veranschaulichung des Ablaufs

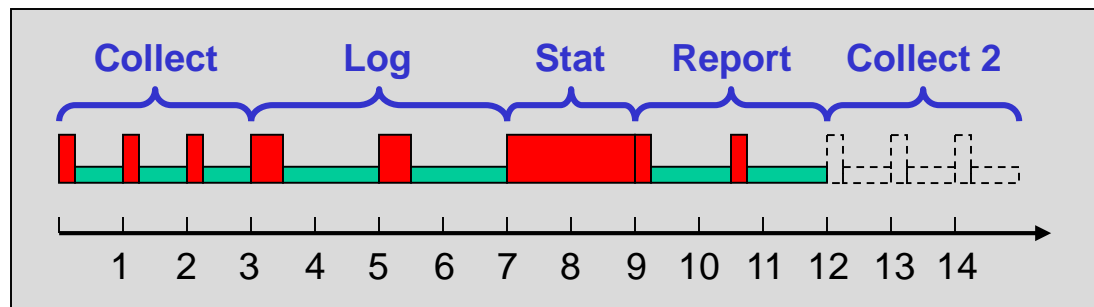
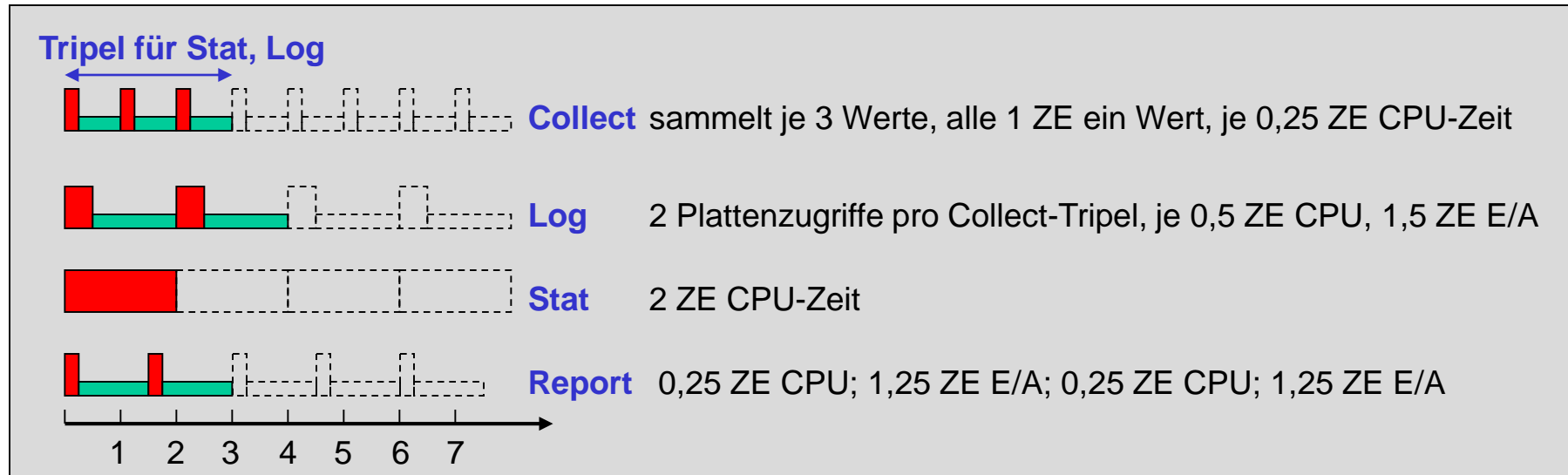


Zeitlicher Ablauf der Beispiel-Threads

Der zeitliche Ablauf soll nun graphisch dargestellt werden. Es bedeuten:

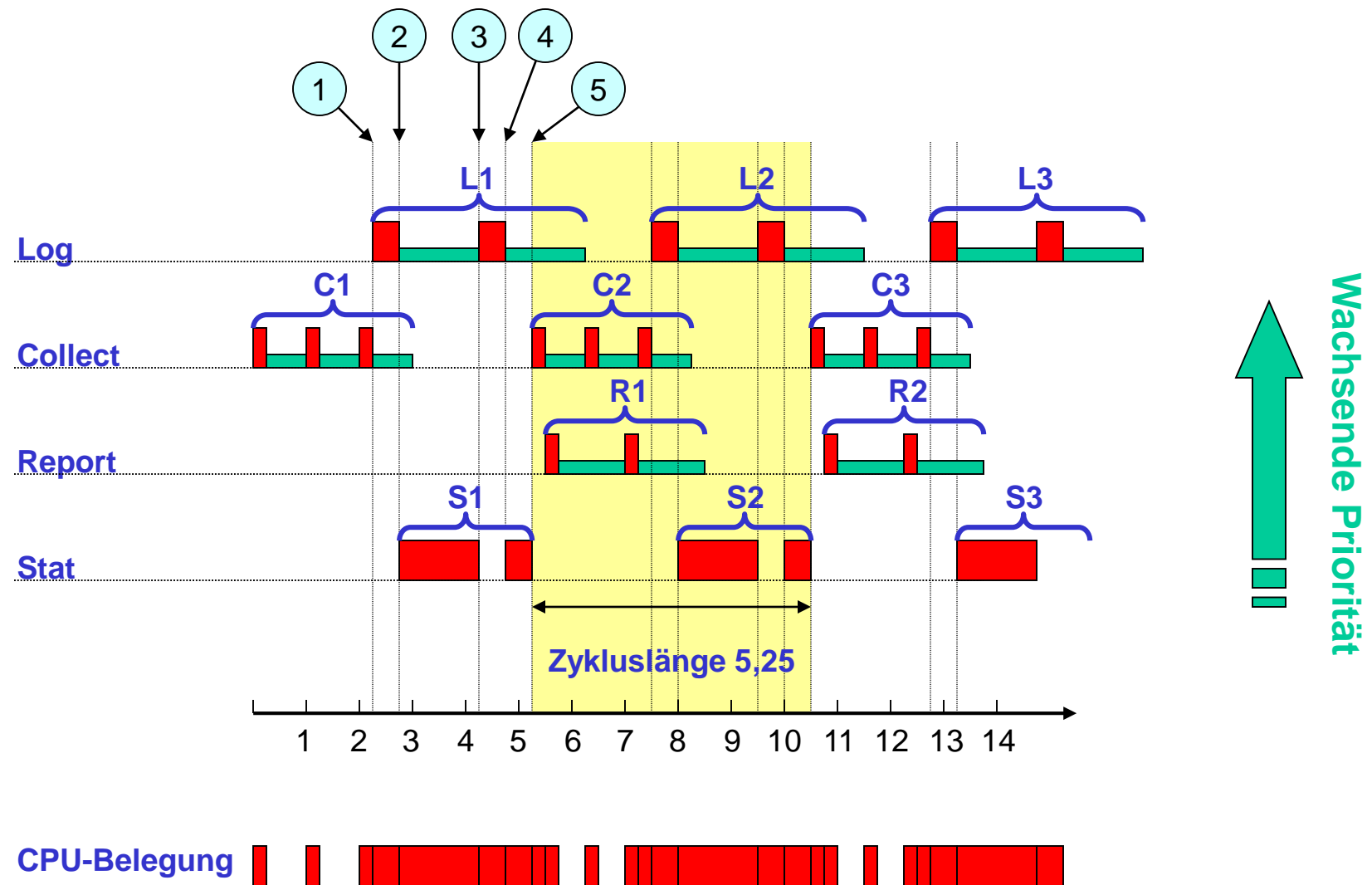
-  Thread belegt den Prozessor (verbraucht CPU-Zeit)
-  Ein-/Ausgabeoperation ohne Mitwirkung des Prozessors (Kein Verbrauch von CPU-Zeit)

Für eine genauere Untersuchung werden Zeitangaben (ZE = Zeiteinheit) benötigt.



**Sequentielle Bearbeitung;
Reine CPU-Zeit: 4,25 ZE**

Zeitlicher Ablauf der Beispiel-Threads (2)

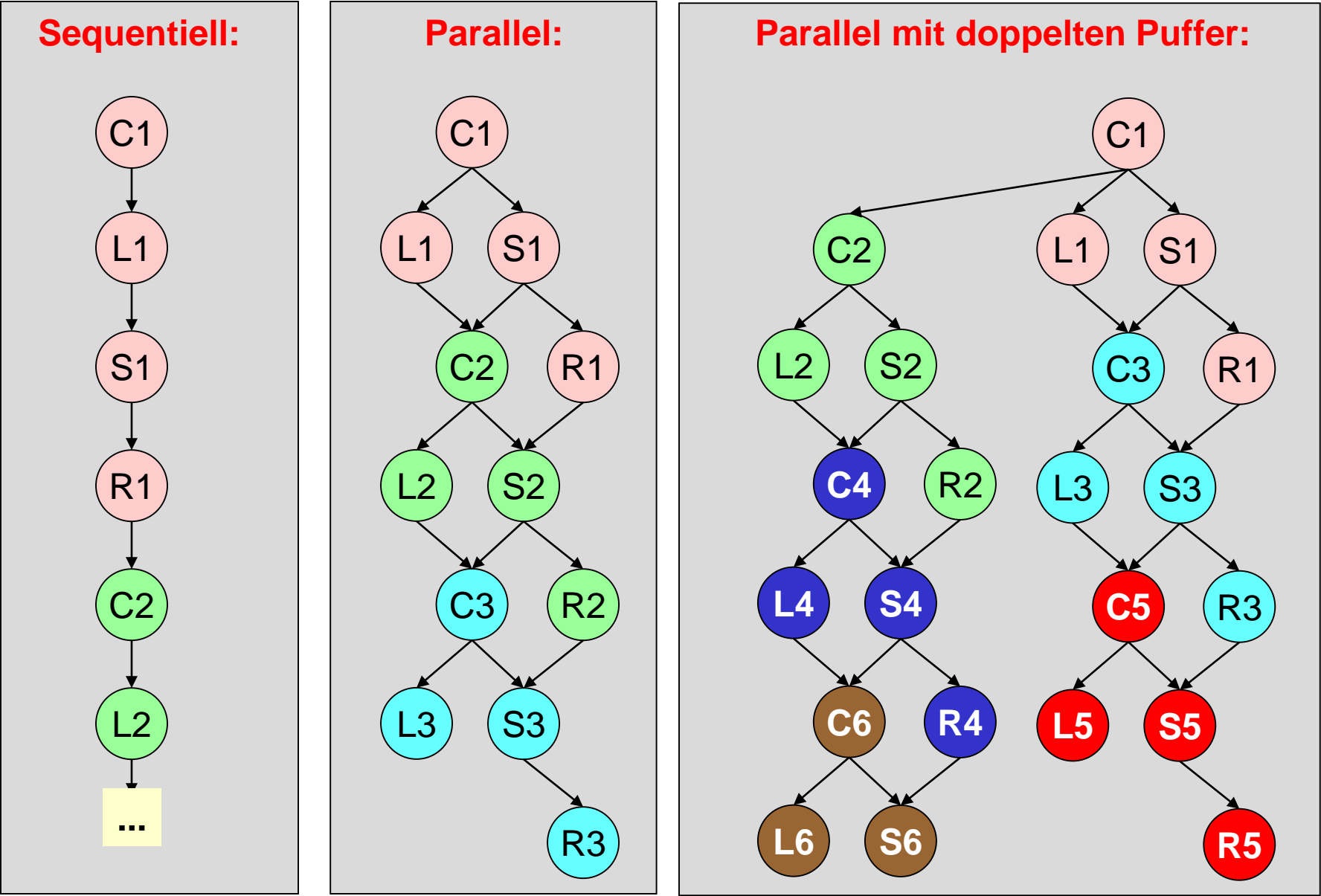


Zeitlicher Ablauf der Beispiel-Threads (3)

Grundzustand: **Nur Collect** ist ohne weitere Signale **lauffähig**.

- 1 Die ersten Werte stehen bereit. **Collect** sendet Signale an **Log** und **Stat**, **Log** beginnt aufgrund höherer Priorität (vom Programmierer vorgegeben!).
- 2 **Log** hat die Daten für den ersten Schreib-Zugriff bereitgestellt. Gibt die CPU frei. Nur **Stat** ist jetzt lauffähig.
- 3 **Stat** wird unterbrochen, da **Log** lauffähig wurde: „Preemptive Scheduling“.
- 4 **Log** gibt die CPU frei, die Bearbeitung von **Stat** wird fortgesetzt.
- 5 **Collect** erhält Signal von **Stat**, vorher schon von **Log** (im Diagramm nicht gezeigt).

Weitere Verbesserung: Doppelte Puffer



Erläuterungen (Doppelte Puffer)

Collect

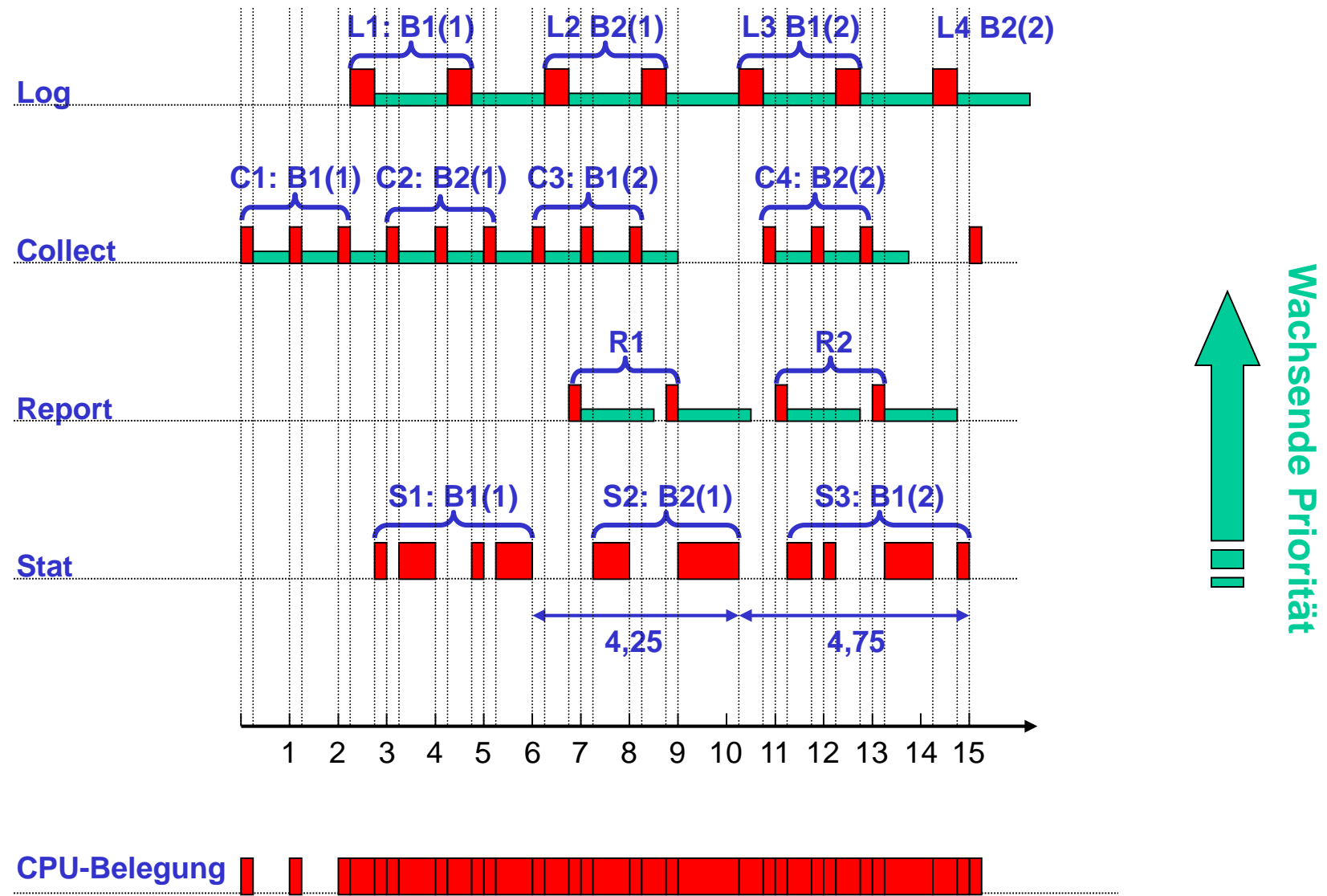
- schreibt in der ersten „Runde“ in den (globalen) Input-Puffer B1,
- kann in der zweiten „Runde“ schon in einen zweiten (globalen) Input-Puffer B2 schreiben, während **Log** und **Stat** die Daten der ersten „Runde“ in B1 bearbeiten,
- greift in der dritten Runde erneut auf B1 zu, benötigt hierfür aber die Freigabe durch **Log** und **Stat**.

Log und Stat

- greifen jeweils auf den Puffer B1 bzw. B2 zu, der
 - unmittelbar vorher von **Collect** mit Daten gefüllt wurde und
 - aktuell für **Collect** gesperrt ist,
- geben den Puffer B1 bzw. B2 so schnell wie möglich wieder für Nutzung durch **Collect** frei, können dies zur Vermeidung von Inkonsistenz aber erst nach vollständiger Bearbeitung der von Collect vorher dort abgelegten Daten tun.

Auch der (globale) Output-Puffer werde als doppelter Puffer betrieben.

Zeitlicher Ablauf der Beispiel-Threads (Doppelte Puffer)



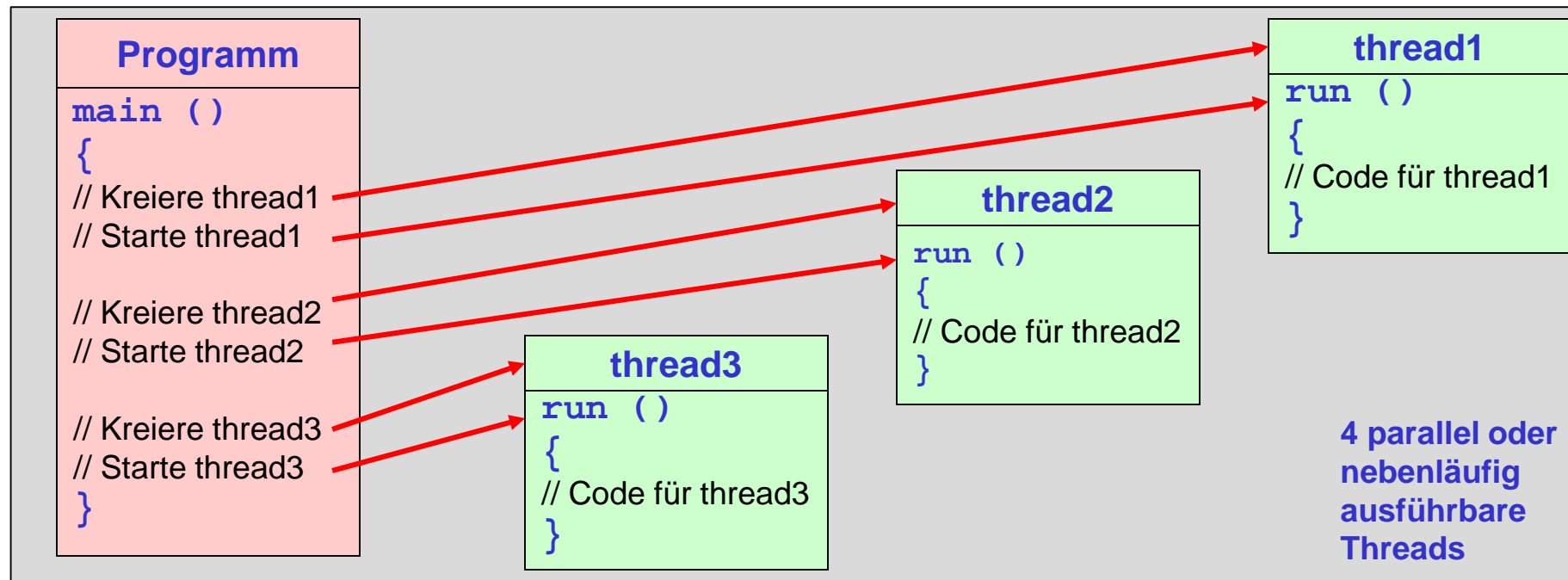
1.5.4. Erzeugung von Threads in Java

Ein laufendes Java-Programm hat **immer mindestens einen Thread**:

- **Programm:** Der Kontrollfluss beginnt mit dem **Anfang von `main()`**
- **Applet:** Der Kontrollfluss beginnt mit dem **Browser**.

Weitere Threads können* **als Objekte** der Klasse `java.lang.Thread` erzeugt werden. Die Codeausführung beginnt dort stets mit der Methode `run ()`.

Die Methode `run ()` ist **public**; sie akzeptiert keine Parameter und liefert keine Werte.



* Alternativ: Implementierung für das Interface „Runnable“:

```
public class MeineRunnableClass extends IrgendeineKlasse implements Runnable { ... }
```

Beispiel: MeineThreads

Hier ein Beispiel, bei dem **zwei Threads der neu definierten Klasse MeineThreads** (Unterklasse von Thread) **erzeugt und gestartet** werden.

```
public class MeineThreads extends Thread {
    int i;

    public static void main (String args []) {

        MeineThreads meinErsterThread = new MeineThreads ();
        meinErsterThread.start ();

        MeineThreads meinZweiterThread = new MeineThreads ();
        meinZweiterThread.start ();

    }

    public void run () {

        while (true) {
            System.out.println (Thread.currentThread().getName());
            i = 0;
            while (i<1000000000)
                i += 1;
        }
    }
}
```

„meinErsterThread“
als neuen Thread
deklarieren und
erzeugen.

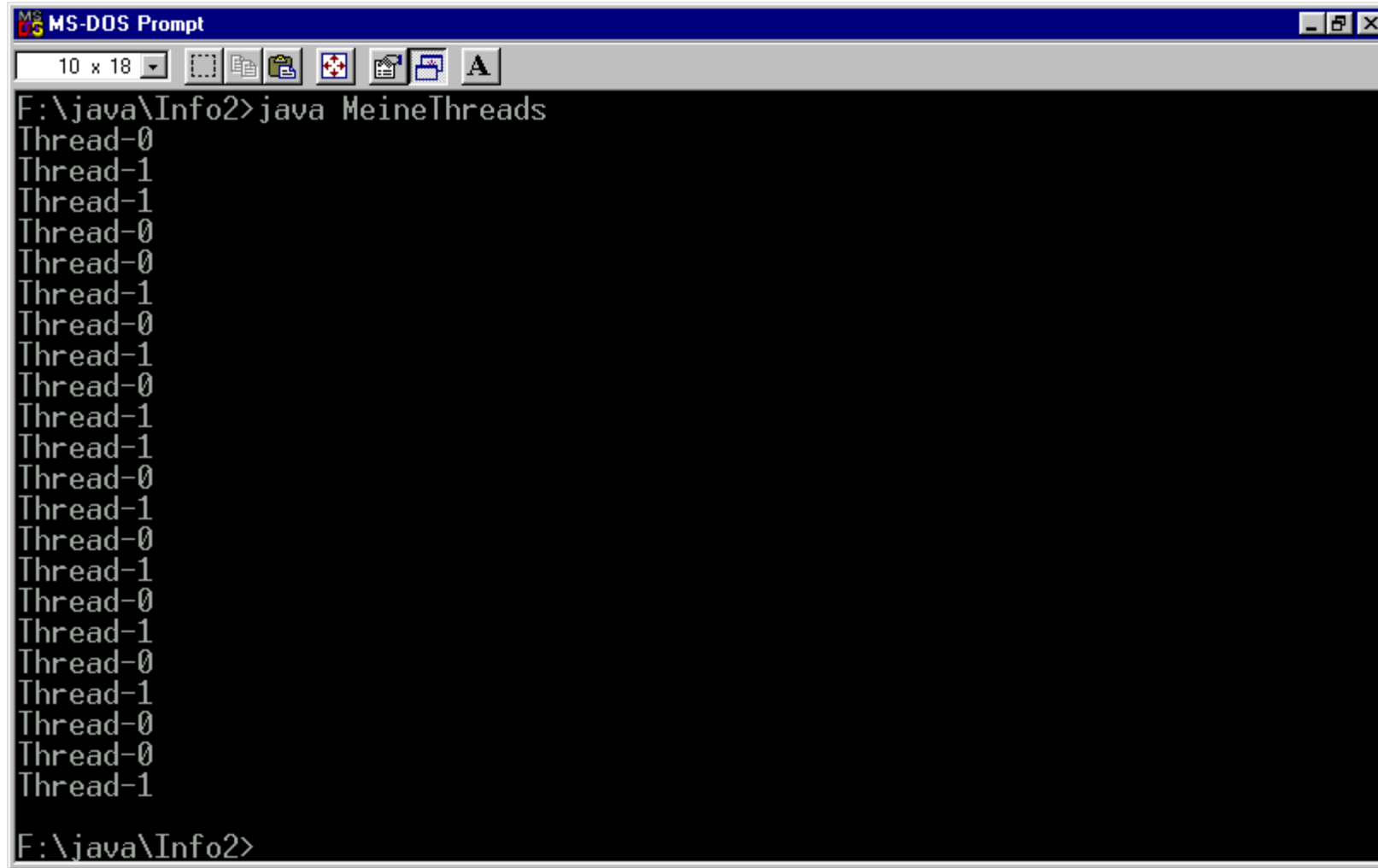
Erzeugten Thread
starten.

Weiteren Thread
deklarieren,
erzeugen und
starten.

Nach dem Starten
bearbeiten beide
Threads die
Methode run.

Beispiel: MeineThreads (2)

Die nebenläufige Bearbeitung der Threads brachte auf dem in der Vorlesung eingesetzten Laptop folgenden Output.



```
MS-DOS Prompt
10 x 18
F:\java\Info2>java MeineThreads
Thread-0
Thread-1
Thread-1
Thread-0
Thread-0
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-0
Thread-0
Thread-1
F:\java\Info2>
```

Man erkennt deutlich den Einfluss der Einplanung durch das Betriebssystem.

1.5.5. Die Struktur der Java Virtual Machine

Die Spezifikation der Java Virtual Machine bestimmt die

- **abstrakte innere Struktur** der
- **abstrakten Java Virtual Machine.**

Eine konkrete Implementierung ist hiermit in keiner Weise festgeschrieben.

Die **Kreativität des Implementierers soll nicht unnötig eingeschränkt** werden. Dies gilt insbesondere für

- die Realisierung der **Speicherverwaltung** inkl. „Garbage Collection“
- die **Umsetzung der Befehle** der Java Virtual Machine **in realen Maschinen-Code.**

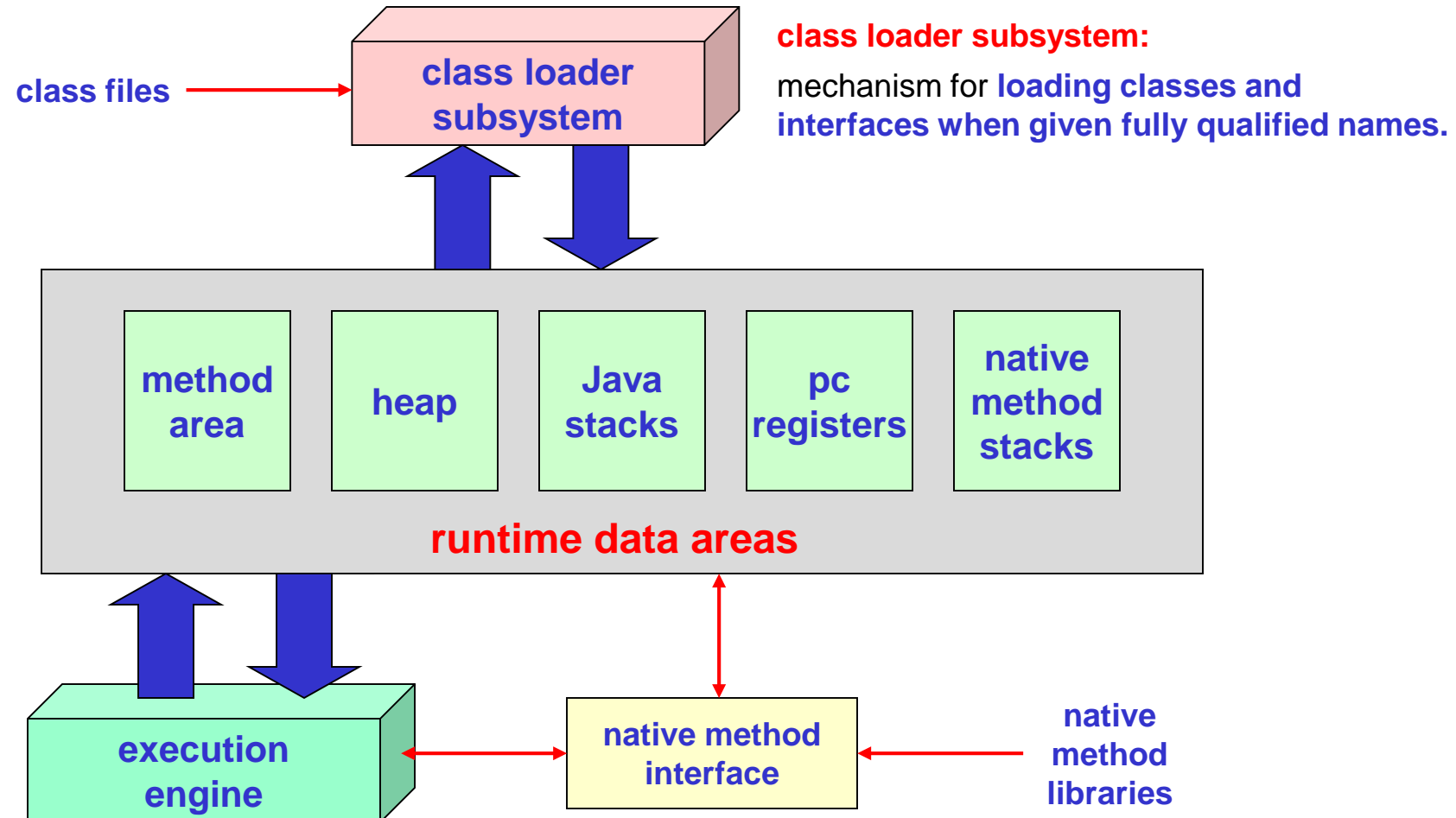
[2.8.5.1. Die interne Struktur im Überblick](#)

[2.8.5.2. Gemeinsam genutzte Datenfelder](#)

[2.8.5.3. Exklusiv genutzte Datenfelder der Threads](#)

1.5.5.1. Die interne Struktur im Überblick

Die nachfolgende Graphik gibt einen Überblick über die interne Struktur der Java Virtual Machine. Der für die Programmbearbeitung benötigte **Speicherplatz** wird in sog. „runtime data areas“ strukturiert bzw. organisiert.

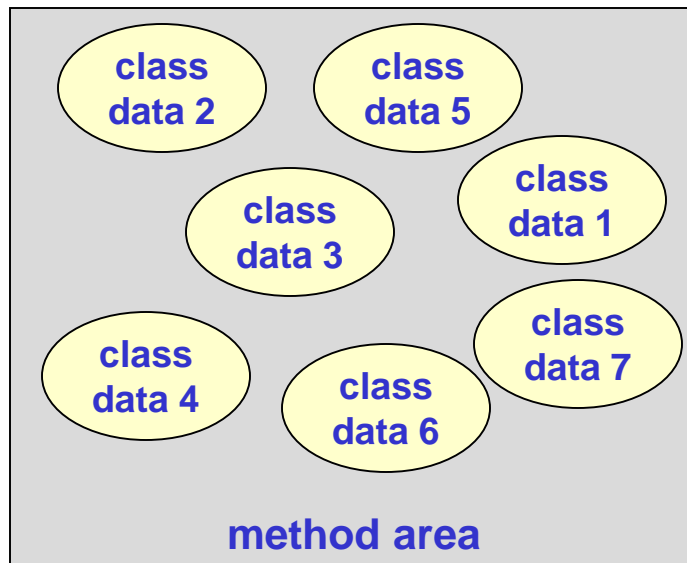


1.5.5.2. Gemeinsam genutzte Datenfelder

Alle Threads der virtuellen Maschine **nutzen gemeinsam „Method Area“ und „Heap“**:

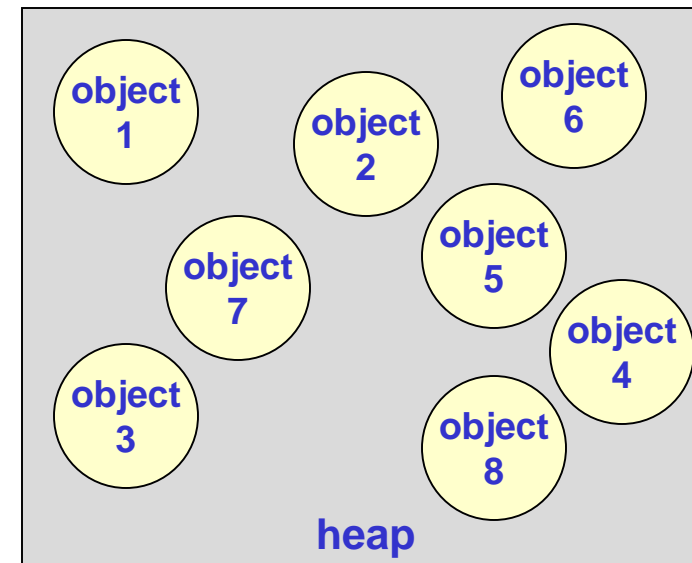
Das Method Area

- wird **beim Start** der Virtual Machine **kreiert**,
- wird **von allen Threads** der jeweiligen Java Virtual Machine **gemeinsam genutzt**,
- ist **für Threads in anderen Virtual Machines nicht zugänglich**,
- beinhaltet u.a. **pro Klasse** die zugehörigen **Klassenvariablen, Programmcode** der Methoden und Konstruktoren.



Der Heap (Halde, Haufen)

- wird **beim Start** der Virtual Machine **kreiert**,
- wird **von allen Threads** der jeweiligen Java Virtual Machine **gemeinsam genutzt**,
- ist **für Threads in anderen Virtual Machines nicht zugänglich**,
- beinhaltet u.a. **pro Objekt** die zugehörigen **Instanzvariablen** und eine **Referenz auf die Daten der zugehörigen Klasse** im Methodenfeld,
- beinhaltet **nicht: Programmcode der Methoden**.



Method area

The Java virtual machine has a *method area* that is shared among all Java virtual machine threads. The method area is analogous to the storage area for compiled code of a conventional language or analogous to the "text" segment in a UNIX process.

It stores per-class structures such as the runtime constant pool, field and method data, and the code for methods and constructors, including the special methods ... used in class and instance initialization and interface type initialization.

The method area is created on virtual machine start-up. Although the method area is logically part of the heap, **simple implementations may choose not to either garbage collect or compact it.** This version of the Java virtual machine specification does not mandate the location of the method area or the policies used to manage compiled code.

The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary. **The memory for the method area does not need to be contiguous.**

A Java virtual machine implementation may provide the programmer or the user **control over the initial size of the method area**, as well as, in the case of a varying-size method area, **control over the maximum and minimum method area size.**

The following exceptional condition is associated with the method area:

- If memory in the method area cannot be made available to satisfy an allocation request, the Java virtual machine throws an **OutOfMemoryError**.

**Quelle: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/Overview.doc.html#15730>**

Heap

The Java virtual machine has a *heap* that is shared among all Java virtual machine threads. The heap is the runtime data area from which **memory for all class instances and arrays** is allocated.

The heap is created on virtual machine start-up. Heap storage for objects is reclaimed by an automatic storage management system (known as a *garbage collector*); objects are never explicitly deallocated. **The Java virtual machine assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements.** The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.

A Java virtual machine implementation may provide the programmer or the user **control over the initial size of the heap**, as well as, if the heap can be dynamically expanded or contracted, **control over the maximum and minimum heap size.**

The following exceptional condition is associated with the heap:

- If a computation requires more heap than can be made available by the automatic storage management system, the Java virtual machine throws an **OutOfMemoryError**.

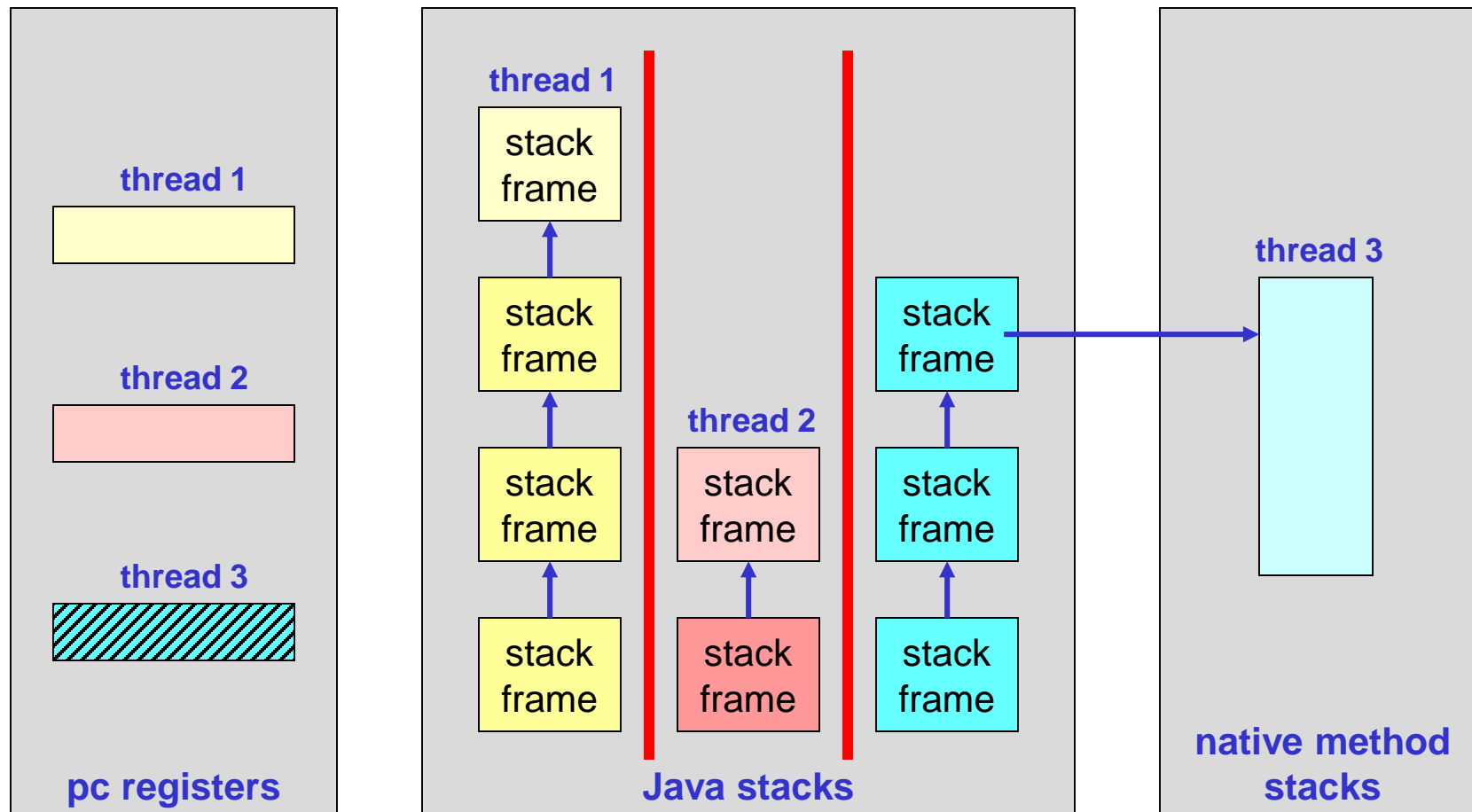
**Quelle: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/Overview.doc.html#15730>**

1.5.5.3. Exklusiv genutzte Datenfelder der Threads

Beim Kreieren eines Threads wird diesem zugewiesen:

- **ein eigener Programmzähler** (gültig nur bei Bearbeitung von Java-Methoden),
- **ein privater „Java Virtual Machine Stack“ für die zugehörigen „Stackframes“**

Bei jedem Methoden-Aufruf werden u.a. die lokalen Variablen und der Operanden-Stack „gerettet“. Bei erfolgreicher Rückkehr aus der Methode werden der alte Zustand restauriert und der Programmzähler aktualisiert.



Frames

A **frame** is used to **store data** and **partial results**, as well as to perform **dynamic linking**, **return values for methods**, and **dispatch exceptions**.

A **new frame** is created **each time a method is invoked**. A frame is **destroyed when its method invocation completes**, whether that completion is normal or abrupt (it throws an uncaught exception). Frames are allocated from the Java virtual machine stack of the thread creating the frame. Each frame has its **own array of local variables**, its **own operand stack** and a **reference** to the runtime **constant pool** of the **class of the current method**.

The sizes of the local variable array and the operand stack are determined at compile time and are supplied along with the code for the method associated with the frame

Thus the size of the frame data structure depends only on the implementation of the Java virtual machine, and the memory for these structures can be allocated simultaneously on method invocation.

Only one frame, the frame for the executing method, is active at any point in a given thread of control. This frame is referred to as the **current frame**, and its method is known as the **current method**. The class in which the current method is defined is the **current class**. **Operations on local variables and the operand stack are typically with reference to the current frame.**

A frame ceases to be current if its method invokes another method or if its method completes. When a method is invoked, a new frame is created and becomes current when control transfers to the new method. On method return, the current frame passes back the result of its method invocation, if any, to the previous frame. The current frame is then discarded as the previous frame becomes the current one.

Note that a **frame created by a thread is local to that thread and cannot be referenced by any other thread.**

Quelle: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“

<http://java.sun.com/docs/books/vmspec/2nd-edition/html/Overview.doc.html#15730>

Local Variables

Each frame contains an array of variables known as its **local variables**. The length of the local variable array of a frame is determined at compile time and supplied in the binary representation of a class or interface along with the code for the method associated with the frame

A single local variable can hold a value of type boolean, byte, char, short, int, float, reference, or returnAddress. A pair of local variables can hold a value of type **long** or **double**.

Local variables are addressed by indexing. The index of the first local variable is zero. An integer is be considered to be an index into the local variable array if and only if that integer is between zero and one less than the size of the local variable array.

A value of type long or type double occupies two consecutive local variables. Such a value may only be addressed using the lesser index. For example, a value of type double stored in the local variable array at index n actually occupies the local variables with indices n and $n + 1$; however, the local variable at index $n + 1$ cannot be loaded from. It can be stored into. However, doing so invalidates the contents of local variable n .

The Java virtual machine does not require n to be even. In intuitive terms, values of types double and long need not be 64-bit aligned in the local variables array. Implementors are free to decide the appropriate way to represent such values using the two local variables reserved for the value.

The Java virtual machine uses local variables to **pass parameters on method invocation.** **On class method invocation any parameters are passed in consecutive local variables starting from local variable 0. On instance method invocation, local variable 0 is always used to pass a reference to the object on which the instance method is being invoked** (this in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable 1.

**Quelle: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/Overview.doc.html#15730>**

Operand Stacks

Each frame contains a last-in-first-out (LIFO) stack known as its *operand stack*. The **maximum depth of the operand stack of a frame is determined at compile time** and is supplied along with the code for the method associated with the frame.

Where it is clear by context, we will sometimes refer to the operand stack of the current frame as simply the operand stack.

The operand stack is empty when the frame that contains it is created. The Java virtual machine supplies instructions to load constants or values from local variables or fields onto the operand stack. Other Java virtual machine instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack. The operand stack is also used to prepare parameters to be passed to methods and to receive method results.

For example, the *iadd* instruction adds two `int` values together. It requires that the `int` values to be added be the top two values of the operand stack, pushed there by previous instructions. Both of the `int` values are popped from the operand stack. They are added, and their sum is pushed back onto the operand stack.

Subcomputations may be nested on the operand stack, resulting in values that can be used by the encompassing computation.

Each entry on the operand stack can hold a value of any Java virtual machine type, including a value of type long or type double.

Values from the operand stack must be operated upon in ways appropriate to their types. It is not possible, for example, to push two `int` values and subsequently treat them as a `long` or to push two `float` values and subsequently add them with an *iadd* instruction. A small number of Java virtual machine instructions (the *dup* instructions and *swap*) operate on runtime data areas as raw values without regard to their specific types; these instructions are defined in such a way that they cannot be used to modify or break up individual values. **These restrictions on operand stack manipulation are enforced through class file verification.**

At any point in time an operand stack has an associated depth, where a value of type long or double contributes two units to the depth and a value of any other type contributes one unit.

**Quelle: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/Overview.doc.html#15730>**

1.5.6. Die Datentypen

Wie die Programmiersprache Java, so arbeitet auch die Java Virtual Machine mit zwei Arten von Datentypen: **Grundtypen und Referenzen**.

Grundtypen (Primitive Types)

- **Ganze Zahlen (Integer Types)**

byte (8 Bit Zweierkomplement)

int (32 Bit Zweierkomplement)

short (16 Bit Zweierkomplement)

long (64 Bit Zweierkomplement)

[char] (16 Bit Zahlen, Darstellung von Unicode)

- **Gleitpunktzahlen (Floating-point Types)**

float (32 Bit, ähnlich IEEE 754)

double (64 Bit, ähnlich IEEE 754)

optional zusätzlich:

„float-extended-exponent value set“ und „double-extended-exponent value set“; unter gewissen Umständen einsetzbar zur Darstellung von Werten der Typen **float** und **double**.

- **Rücksprungadressen (Return Address)**

Rücksprungadresse bei (methodeninternen) „Mini-Unterprogrammen“

- **Boolesche Werte (Boolean)**

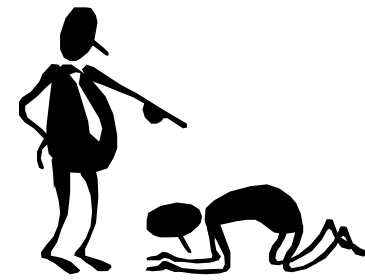
Referenztypen (Reference Types)

- **Class Type, Interface Type, Array Type**

Referenz auf eine Instanz des jeweiligen Typs.

1.5.7. Der Befehlssatz

Der **Entwurf des Befehlssatzes** für den Java-Byte-Code wurde von der gleichen **Prinzipien** geleitet wie der Entwurf der Programmiersprache Java:



1. Plattformunabhängigkeit:

Der Befehlssatz folgt i.w. einer **stack-basierten Philosophie**.

Dies erleichtert die Implementierung auf Rechnern mit nur wenigen Registern, z.B. der Familie Intel 80X86.

2. Netzweite Mobilität:

Um auch über **Kanäle mit geringer Bandbreite** gute Übertragungseigenschaften zu erzielen, wurde eine effiziente Codierung angestrebt. So wird z.B. die Anzahl der Befehle so klein gehalten, dass **für den Op-Code nur ein Byte** benötigt wird. Dies schränkt die Zahl der Optimierungsmöglichkeiten ein.

3. Sicherheit:

Die sog. „**Byte Code Verification**“ ist ein zentraler Bestandteil des Sicherheitskonzeptes.

Durch starke Typisierung und **weitgehenden Verzicht auf überladene Operatoren*** wird eine derartige Codeprüfung vor Beginn der Laufzeit erleichtert.

Anmerkung:

Dem **Implementierer** der Virtual Maschine bleiben **viele Freiheitsgrade**, wann, ob, wie aufwendig und nach welchen Kriterien der Byte Code geprüft wird.

*in der nachfolgenden Darstellung: b: byte; s: short; i: int; l: long; c: char; f: float, d: double, a: reference

Beispiel 1: `iconst_<i>`

Operation

`Push int constant // Integer-Konstante auf den Stack legen`

Format

`iconst_<i>`

Forms

iconst_m1 = 2 (0x2) *iconst_0* = 3 (0x3) *iconst_1* = 4 (0x4) *iconst_2* = 5 (0x5) *iconst_3* = 6 (0x6)
iconst_4 = 7 (0x7) *iconst_5* = 8 (0x8)

Operand Stack

... ..., *<i>*

Description

Push the `int` constant *<i>* (-1, 0, 1, 2, 3, 4 or 5) onto the operand stack.

Notes

Each of this family of instructions is equivalent to *bipush <i>* for the respective value of *<i>*, except that the operand *<i>* is implicit.

Nach: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“, <http://java.sun.com/docs/books/vmspec>

Beispiel 2: `iload_<n>`

Operation

Load int from local variable // Integer-Wert in lokaler Variable auf den Stack legen

Format

`iload_<n>`

Forms

`iload_0` = 26 (0x1a) `iload_1` = 27 (0x1b) `iload_2` = 28 (0x1c) `iload_3` = 29 (0x1d)

Operand Stack

... ..., *value*

Description

The `<n>` must be an index into the local variable array of the current frame (§3.6).

The local variable at `<n>` must contain an `int`. The *value* of the local variable at `<n>` is pushed onto the operand stack.

Notes

Each of the `iload_<n>` instructions is the same as `iload` with an *index* of `<n>`, except that the operand `<n>` is implicit.

Nach: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“, <http://java.sun.com/docs/books/vmspec>

Beispiel 3: istore_<n>

Operation

Store int into local variable // Integer-Wert aus dem Stack holen; in lokale Variable legen

Format

istore_<n>

Forms

istore_0 = 59 (0x3b) *istore_1* = 60 (0x3c) *istore_2* = 61 (0x3d) *istore_3* = 62 (0x3e)

Operand Stack

..., *value* ⇒

Description

The <*n*> must be an index into the local variable array of the current frame (§3.6).

The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *value*.

Notes

Each of the *istore_<n>* instructions is the same as *istore* with an *index* of <*n*>, except that the operand <*n*> is implicit.

Nach: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“, <http://java.sun.com/docs/books/vmspec>

Beispiel 4: iinc

Operation

Increment local variable by constant // Wert der lokalen Variablen um Konstante erhöhen

Format

| |
|--------------|
| iinc |
| index |
| const |

Forms

iinc = 132 (0x84)

Operand Stack

No change

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The *const* is an immediate signed byte. The local variable at *index* must contain an *int*. The value *const* is first sign-extended to an *int*, and then the local variable at *index* is incremented by that amount.

Notes

The *iinc* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index and to increment it by a two-byte immediate value.

Beispiel 5: imul

Operation

Multiply int // Die beiden obersten Werte auf dem Stack multiplizieren, Ergebnis auf den Stack

Format

imul

Forms

imul = 104 (0x68)

Operand Stack

..., *value1*, *value2* ..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, **execution of an *imul* instruction never throws a runtime exception.**

Nach: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“, <http://java.sun.com/docs/books/vmspec>

Beispiel 6: goto

Operation

Branch always // Unbedingter Sprung, Offset des nächsten Befehls in den nächsten 2 Bytes.

Format

| |
|--------------------|
| goto |
| branchbyte1 |
| branchbyte2 |

Forms

goto = 167 (0xa7)

Operand Stack

No change

Description

The unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is $(branchbyte1 \ll 8) \mid branchbyte2$. Execution proceeds at that offset from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

Nach: T. Linholm, F. Yellin, „The Java Virtual Machine Specification“, <http://java.sun.com/docs/books/vmspec>

Op-Codes der Java Virtual Machine im Überblick

| | | |
|----|--------|-------------|
| 0 | (0x00) | nop |
| 1 | (0x01) | aconst_null |
| 2 | (0x02) | iconst_m1 |
| 3 | (0x03) | iconst_0 |
| 4 | (0x04) | iconst_1 |
| 5 | (0x05) | iconst_2 |
| 6 | (0x06) | iconst_3 |
| 7 | (0x07) | iconst_4 |
| 8 | (0x08) | iconst_5 |
| 9 | (0x09) | lconst_0 |
| 10 | (0x0a) | lconst_1 |
| 11 | (0x0b) | fconst_0 |
| 12 | (0x0c) | fconst_1 |
| 13 | (0x0d) | fconst_2 |
| 14 | (0x0e) | dconst_0 |
| 15 | (0x0f) | dconst_1 |

| | | |
|----|--------|---------|
| 16 | (0x10) | bipush |
| 17 | (0x11) | sipush |
| 18 | (0x12) | ldc |
| 19 | (0x13) | ldc_w |
| 20 | (0x14) | ldc2_w |
| 21 | (0x15) | iload |
| 22 | (0x16) | lload |
| 23 | (0x17) | fload |
| 24 | (0x18) | dload |
| 25 | (0x19) | aload |
| 26 | (0x1a) | iload_0 |
| 27 | (0x1b) | iload_1 |
| 28 | (0x1c) | iload_2 |
| 29 | (0x1d) | iload_3 |
| 30 | (0x1e) | lload_0 |
| 31 | (0x1f) | lload_1 |

| | | |
|----|--------|---------|
| 32 | (0x20) | lload_2 |
| 33 | (0x21) | lload_3 |
| 34 | (0x22) | fload_0 |
| 35 | (0x23) | fload_1 |
| 36 | (0x24) | fload_2 |
| 37 | (0x25) | fload_3 |
| 38 | (0x26) | dload_0 |
| 39 | (0x27) | dload_1 |
| 40 | (0x28) | dload_2 |
| 41 | (0x29) | dload_3 |
| 42 | (0x2a) | aload_0 |
| 43 | (0x2b) | aload_1 |
| 44 | (0x2c) | aload_2 |
| 45 | (0x2d) | aload_3 |
| 46 | (0x2e) | iaload |
| 47 | (0x2f) | laload |

Quelle: <http://java.sun.com/docs/books/vmspec/html/Mnemonics.doc.html>

Op-Codes der Java Virtual Machine im Überblick (2)

| | | |
|----|--------|----------|
| 48 | (0x30) | faload |
| 49 | (0x31) | daload |
| 50 | (0x32) | aaload |
| 51 | (0x33) | baload |
| 52 | (0x34) | caload |
| 53 | (0x35) | saload |
| 54 | (0x36) | istore |
| 55 | (0x37) | lstore |
| 56 | (0x38) | fstore |
| 57 | (0x39) | dstore |
| 58 | (0x3a) | astore |
| 59 | (0x3b) | istore_0 |
| 60 | (0x3c) | istore_1 |
| 61 | (0x3d) | istore_2 |
| 62 | (0x3e) | istore_3 |
| 63 | (0x3f) | lstore_0 |

| | | |
|----|--------|----------|
| 64 | (0x40) | lstore_1 |
| 65 | (0x41) | lstore_2 |
| 66 | (0x42) | lstore_3 |
| 67 | (0x43) | fstore_0 |
| 68 | (0x44) | fstore_1 |
| 69 | (0x45) | fstore_2 |
| 70 | (0x46) | fstore_3 |
| 71 | (0x47) | dstore_0 |
| 72 | (0x48) | dstore_1 |
| 73 | (0x49) | dstore_2 |
| 74 | (0x4a) | dstore_3 |
| 75 | (0x4b) | astore_0 |
| 76 | (0x4c) | astore_1 |
| 77 | (0x4d) | astore_2 |
| 78 | (0x4e) | astore_3 |
| 79 | (0x4f) | iastore |

| | | |
|----|--------|---------|
| 80 | (0x50) | lastore |
| 81 | (0x51) | fastore |
| 82 | (0x52) | dastore |
| 83 | (0x53) | aastore |
| 84 | (0x54) | bastore |
| 85 | (0x55) | castore |
| 86 | (0x56) | sastore |
| 87 | (0x57) | pop |
| 88 | (0x58) | pop2 |
| 89 | (0x59) | dup |
| 90 | (0x5a) | dup_x1 |
| 91 | (0x5b) | dup_x2 |
| 92 | (0x5c) | dup2 |
| 93 | (0x5d) | dup2_x1 |
| 94 | (0x5e) | dup2_x2 |
| 95 | (0x5f) | swap |

Quelle: <http://java.sun.com/docs/books/vmspec/html/Mnemonics.doc.html>

Op-Codes der Java Virtual Machine im Überblick (3)

| | | |
|-----|--------|------|
| 96 | (0x60) | iadd |
| 97 | (0x61) | ladd |
| 98 | (0x62) | fadd |
| 99 | (0x63) | dadd |
| 100 | (0x64) | isub |
| 101 | (0x65) | lsub |
| 102 | (0x66) | fsub |
| 103 | (0x67) | dsub |
| 104 | (0x68) | imul |
| 105 | (0x69) | lmul |
| 106 | (0x6a) | fmul |
| 107 | (0x6b) | dmul |
| 108 | (0x6c) | idiv |
| 109 | (0x6d) | ldiv |
| 110 | (0x6e) | fdiv |
| 111 | (0x6f) | ddiv |

| | | |
|-----|--------|-------|
| 112 | (0x70) | irem |
| 113 | (0x71) | lrem |
| 114 | (0x72) | frem |
| 115 | (0x73) | drem |
| 116 | (0x74) | ineg |
| 117 | (0x75) | lneg |
| 118 | (0x76) | fneg |
| 119 | (0x77) | dneg |
| 120 | (0x78) | ishl |
| 121 | (0x79) | lshl |
| 122 | (0x7a) | ishr |
| 123 | (0x7b) | lshr |
| 124 | (0x7c) | iushr |
| 125 | (0x7d) | lushr |
| 126 | (0x7e) | iand |
| 127 | (0x7f) | land |

| | | |
|-----|--------|------|
| 128 | (0x80) | ior |
| 129 | (0x81) | lor |
| 130 | (0x82) | ixor |
| 131 | (0x83) | lxor |
| 132 | (0x84) | iinc |
| 133 | (0x85) | i2l |
| 134 | (0x86) | i2f |
| 135 | (0x87) | i2d |
| 136 | (0x88) | l2i |
| 137 | (0x89) | l2f |
| 138 | (0x8a) | l2d |
| 139 | (0x8b) | f2i |
| 140 | (0x8c) | f2l |
| 141 | (0x8d) | f2d |
| 142 | (0x8e) | d2i |
| 143 | (0x8f) | d2l |

Quelle: <http://java.sun.com/docs/books/vmspec/html/Mnemonics.doc.html>

Op-Codes der Java Virtual Machine im Überblick (4)

144 (0x90) d2f
145 (0x91) i2b
146 (0x92) i2c
147 (0x93) i2s
148 (0x94) lcmp
149 (0x95) fcmpl
150 (0x96) fcmpg
151 (0x97) dcmpl
152 (0x98) dcmpg
153 (0x99) ifeq
154 (0x9a) ifne
155 (0x9b) iflt
156 (0x9c) ifge
157 (0x9d) ifgt
158 (0x9e) ifle
159 (0x9f) if_icmpeq

160 (0xa0) if_icmpne
161 (0xa1) if_icmplt
162 (0xa2) if_icmpge
163 (0xa3) if_icmpgt
164 (0xa4) if_icmple
165 (0xa5) if_acmpeq
166 (0xa6) if_acmpne
167 (0xa7) goto
168 (0xa8) jsr
169 (0xa9) ret
170 (0xaa) tableswitch
171 (0xab) lookupswitch
172 (0xac) ireturn
173 (0xad) lreturn
174 (0xae) freturn
175 (0xaf) dreturn

176 (0xb0) areturn
177 (0xb1) return
178 (0xb2) getstatic
179 (0xb3) putstatic
180 (0xb4) getfield
181 (0xb5) putfield
182 (0xb6) invokevirtual
183 (0xb7) invokespecial
184 (0xb8) invokestatic
185 (0xb9) invokeinterface
186 (0xba) xxxunusedxxx
187 (0xbb) new
188 (0xbc) newarray
189 (0xbd) anewarray
190 (0xbe) arraylength
191 (0xbf) athrow

Quelle: <http://java.sun.com/docs/books/vmspec/html/Mnemonics.doc.html>

Op-Codes der Java Virtual Machine im Überblick (5)

192 (0xc0) checkcast
193 (0xc1) instanceof
194 (0xc2) monitorenter
195 (0xc3) monitorexit
196 (0xc4) wide
197 (0xc5) multianewarray
198 (0xc6) ifnull
199 (0xc7) ifnonnull
200 (0xc8) goto_w
201 (0xc9) jsr_w

Quick opcodes:

203 (0xcb) ldc_quick
204 (0xcc) ldc_w_quick
205 (0xcd) ldc2_w_quick
206 (0xce) getfield_quick
207 (0xcf) putfield_quick
208 (0xd0) getfield2_quick
209 (0xd1) putfield2_quick
210 (0xd2) getstatic_quick
211 (0xd3) putstatic_quick

212 (0xd4) getstatic2_quick
213 (0xd5) putstatic2_quick
214 (0xd6) invokevirtual_quick
215 (0xd7) invokenonvirtual_quick
216 (0xd8) invokesuper_quick
217 (0xd9) invokestatic_quick
218 (0xda) invokeinterface_quick
219 (0xdb) invokevirtualobject_quick
221 (0xdd) new_quick
222 (0xde) anewarray_quick
223 (0xdf) multianewarray_quick
224 (0xe0) checkcast_quick
225 (0xe1) instanceof_quick
226 (0xe2) invokevirtual_quick_w
227 (0xe3) getfield_quick_w
228 (0xe4) putfield_quick_w

Reserved opcodes:

202 (0xca) breakpoint
254 (0xfe) impdep1
255 (0xff) impdep2

Quelle: <http://java.sun.com/docs/books/vmspec/html/Mnemonics.doc.html>

1.5.8. Simulation der Java Virtual Machine

Abschließend betrachten wir noch eine Simulation der Java Virtual Machine, welche die **internen Abläufe noch ein wenig veranschaulicht**.

Es handelt sich dabei um eines von mehreren „**Simulation Applets**“, die auf der CD-ROM zum Buch „**Inside the Java Virtual Machine**“, McGraw-Hill, 1998, von Bill Venners, enthalten sind. Diese Applets (= Java Mini-Applikationen) können mit aktuellen Versionen der gängigen Internet-Browser abgespielt werden.

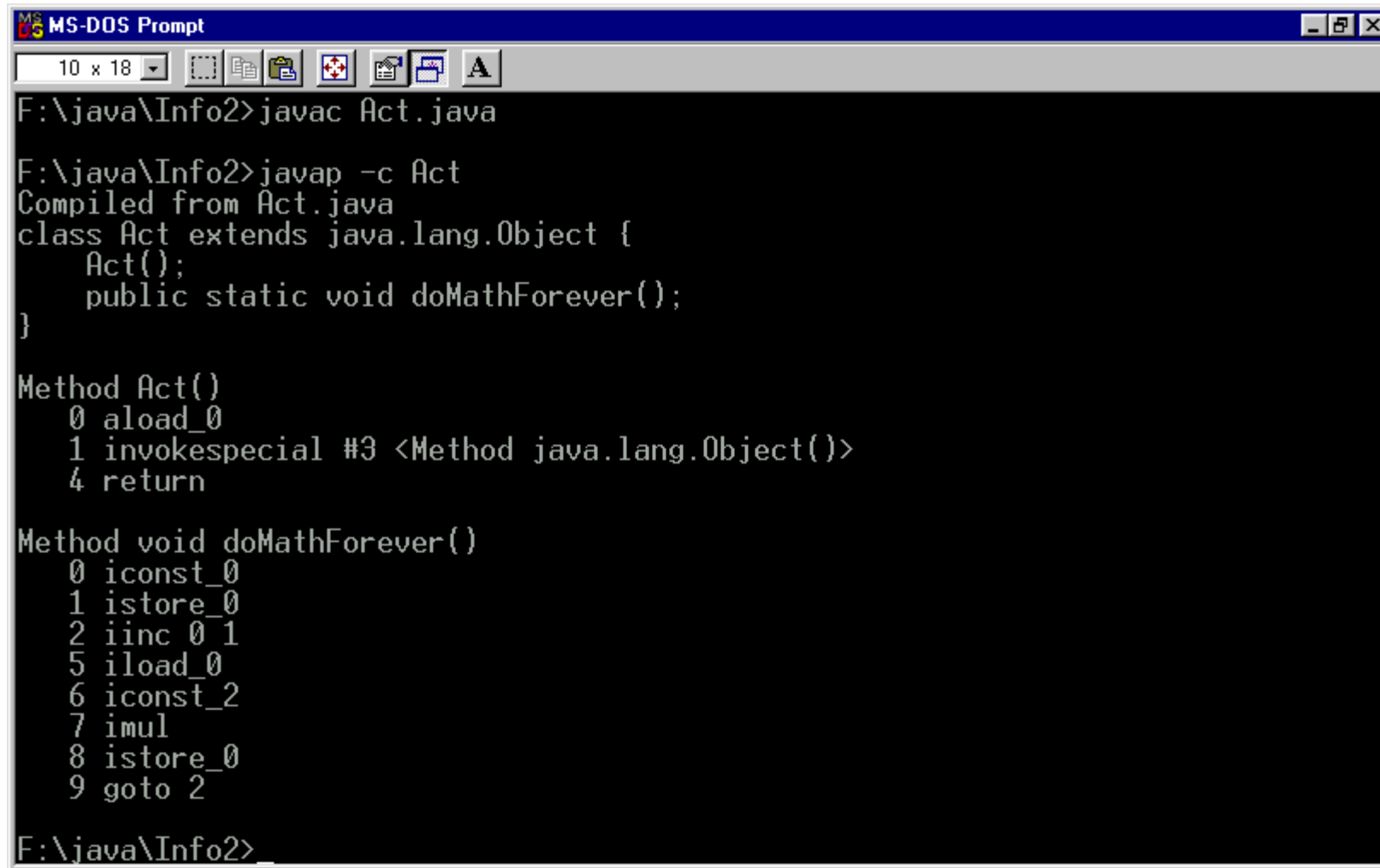
Bei dem auf den nachfolgenden Seiten betrachteten Beispiel handelt es sich um den Java-Byte-Code, der **vom Java-Compiler javac** zu folgendem Programm **generiert** wurde:

```
class Act
{

    public static void doMathForever()
    {
        int i = 0;
        for (;;)
        {
            i += 1;
            i *= 2;
        }
    }
}
```


Der Java-Byte-Code zu „doMathForever ()“

Der mit **javac** hierzu generierte Code der Java Virtual Maschine kann mit **javap** in mnemotechnischer Form angezeigt werden:



```
MS-DOS Prompt
10 x 18
F:\java\Info2>javac Act.java

F:\java\Info2>javap -c Act
Compiled from Act.java
class Act extends java.lang.Object {
    Act();
    public static void doMathForever();
}

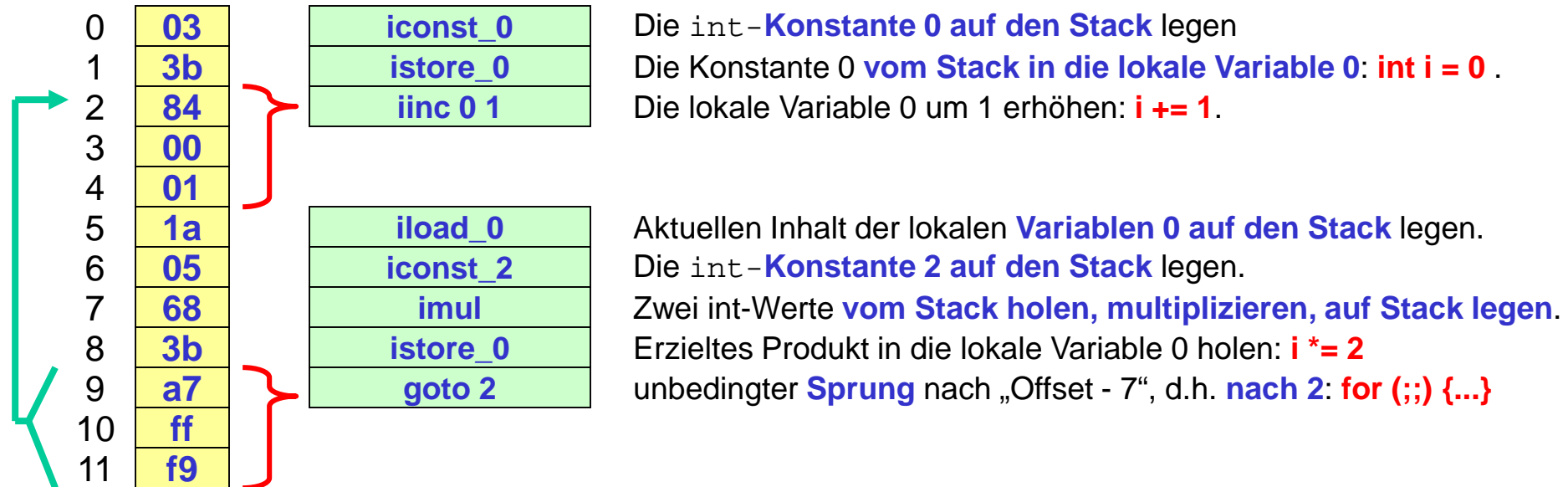
Method Act()
  0 aload_0
  1 invokespecial #3 <Method java.lang.Object()>
  4 return

Method void doMathForever()
  0 iconst_0
  1 istore_0
  2 iinc 0 1
  5 iload_0
  6 iconst_2
  7 imul
  8 istore_0
  9 goto 2

F:\java\Info2>
```

Der Java-Byte-Code zu „doMathForever ()“ (2)

033b8400011a05683ba7fff9



Hier sind einige **typische Charakteristika des Java-Byte-Codes** erkennbar:

- Der **Operanden-Stack** nimmt eine **zentrale Stellung** ein.
- **Keine Register** für inhaltsunabhängige Speicherung;
statt dessen: **typisierte lokale Variablen**.
- **Operationen** mit Variablen-Inhalten nur mit **Umweg über den Stack** (Ausnahme: `iinc`)



Eternal Math

A Simulation of the Java Virtual Machine

The *Eternal Math* applet, included below, simulates a Java Virtual Machine executing a few bytecode instructions. This applet accompanies Chapter 5, "The Java Virtual Machine," of *Inside the Java Virtual Machine*.

ETERNAL MATH

Local Variables

| index | hex value | value |
|-------|-----------|-------|
| 0 | | |

optop 0

Operand Stack

| offset | hex value | value |
|--------|-----------|-------|
| 0 | | |
| 1 | | |

pc 0

The Method

| offset | bytecode | mnemonic |
|--------|----------|----------|
| 0 | 03 | iconst_0 |
| 1 | 3b | istore_0 |
| 2 | 84 | iinc 0 1 |
| 3 | 00 | |
| 4 | 01 | |
| 5 | 1a | iload_0 |
| 6 | 05 | iconst_2 |
| 7 | 68 | imul |
| 8 | 3b | istore_0 |
| 9 | a7 | goto 2 |
| 10 | ff | |
| 11 | f9 | |

Step Run

Reset Stop

iconst_0 will push int 0 onto the stack.

The instructions in the simulation were generated by the `javac` compiler given the following code:

Simulation von „Eternal Math“

„Dieses Simulations-Applet wird auf dem Web-Server des Instituts für Informatik IV“ verfügbar gemacht. Dort sind auch weitere Simulations-Applets sowie Copyright-Info zu finden.

ETERNAL MATH

Local Variables

| index | hex value | value |
|-------|-----------|-------|
| 0 | | |

optop 0

Operand Stack

| offset | hex value | value |
|-----------|-----------|-------|
| optop > 0 | | |
| 1 | | |

pc 0

The Method

| | offset | bytecode | mnemonic |
|------|--------|----------|----------|
| pc > | 0 | 03 | iconst_0 |
| | 1 | 3b | istore_0 |
| | 2 | 84 | iinc 0 1 |
| | 3 | 00 | |
| | 4 | 01 | |
| | 5 | 1a | iload_0 |
| | 6 | 05 | iconst_2 |
| | 7 | 68 | imul |
| | 8 | 3b | istore_0 |
| | 9 | a7 | goto 2 |
| | 10 | ff | |
| | 11 | f9 | |

Step Run

Reset Stop

iconst_0 will push int 0 onto the stack.

Achtung: Der Stack wächst von oben nach unten.

1.5.9. Zusammenfassung (Kapitel 1)

- Die Mehrzahl der heute eingesetzten Computer basiert auf dem Konzept des Von-Neumann-Rechners. Hierbei liegen Programm und Daten in demselben Speicher.
- Geschickter Datentransport hat fundamentale Bedeutung für die Leistungsfähigkeit eines Computers.
- Sprungbefehle - bei höheren Programmiersprachen verpönt - sind zentrale Bestandteile von Programmen in Maschinensprache.
- Programme mit n-Adressbefehlen können in Programme für m-Adressmaschinen umgewandelt werden ($m, n \in \mathbb{IN} \cup \{0\}$).
- Unterprogrammtechnik kann sehr flexibel mit Hilfe eines Stacks für die Rücksprungadressen realisiert werden.
- Sind in einem Assembler-Programm Vorwärts-Sprünge zulässig, dann benötigt der Assembler für die Umsetzung in Maschinen-Code mindestens 2 Durchgänge.
- Makros sind Kurzschreibweisen für Befehlssequenzen.
- Die Entwicklung der Prozessoren ist untrennbar mit der Entwicklung der Speichertechnik verknüpft.
- Pipelining, Super-Pipelining und Superskalar-Technik sind in (fast) allen modernen Prozessoren zu finden. Flexible Parallelverarbeitung wird zunehmend auch für Standardanwendungen eingesetzt.
- Die Kunst des Prozessor-Designs besteht weniger im „Erfinden“ neuartiger Techniken als in der geschickten Mischung existierender Techniken.
- Die Java Virtual Machine ist ein abstrakter Computer, dessen Befehlssatz stark stack-orientiert ist. Der Programm-Code zu den Methoden existiert pro Klasse nur einmal.