

15. Aufzählungstypen und die switch-Anweisung

- Weitere Konzepte in Java
 - Aufzählungstypen
 - switch-Anweisung



Zusammenfassung

- Einführung in das Paket java.io
 - Verwaltung von Dateien
 - Klasse File
 - Wahlfreier Zugriff auf Dateien
 - RandomAccessFile
 - Sequentieller Zugriff mit Datenströmen
- Datenströme unterscheiden zwischen
 - Byte-Datenströme
 - Klassen InputStream und OutputStream
 - Character-Datenströme
 - Reader und Writer
 - Funktionalität zum Lesen und Schreiben von Textdateien

15.1 Aufzählungstypen

- Seit dem JDK 1.5 kann man in Java auch **Aufzählungstypen** definieren.
 - Es handelt sich dabei um Klassen mit einer fest vorgegebenen Menge von Objekten.
- Eine solche Klasse wird durch eine **enum-Deklaration** erzeugt, in der alle möglichen Objekte aufgelistet werden, wie in:

```
enum Farbe {Rot, Grün, Blau, Weiss, Schwarz}  
enum Ampel {Grün, Gelb, Rot, GelbRot}  
enum Wochentag {Mo, Di, Mi, Do, Fr, Sa, So}
```

- Wochentag ist nun eine Klasse mit genau 7 Objekten, **weitere können nicht erzeugt werden**.
Daher fehlt in der folgenden Zuweisung der gewohnte new-Operator:

```
Wochentag tag = Wochentag.Mi;
```

- Die Elemente der Klasse verhalten sich wie Objekte der Klasse und sind zugreifbar wie final static Datenfelder
- Objekte verschiedener enum-Typen sind nicht kompatibel, selbst wenn sie gleiche Name

```
Farbe.Grün == Ampel.Grün
```

- Der Vergleich **Farbe.Grün == Ampel.Grün** muss somit zu einem **Typfehler** führen.

Konstrukturen

- Enum-Klassen können benutzerdefinierte Felder und Methoden erhalten.
- Selbst **Konstrukturen sind möglich**.
 - Diese können benutzt werden, um die vorhandenen Objekte geeignet zu **initialisieren**.
 - **Eine Objekterzeugung mit dem new-Operator ist nicht möglich**.
- In dem folgenden Beispiel reichen wir die Klasse Wochentag um ein Feld `arbeitsTag` und eine gleichnamige Methode an.
 - Der Konstruktor wird bei der Definition der enum-Klasse eingesetzt, um in den vorhandenen Objekten das Datenfeld `arbeitsTag` geeignet zu initialisieren:

```
enum WochenTag {  
    Mo(true), Di(true), Mi(true), Do(true),  
    Fr(true), Sa(false), So(false);  
  
    private boolean arbeitsTag;  
  
    boolean istArbeitsTag() { return arbeitsTag;}  
  
    WochenTag(boolean mussArbeiten) { // Konstruktor  
        arbeitsTag = mussArbeiten;  
    }  
}
```

Oberklasse `java.lang.Enum`

- Alle enum-Klassen werden vom Java-Compiler in eine Unterklasse der **abstrakten Klasse `java.lang.Enum`** übertragen.
- Insbesondere folgende Methoden werden bereitstellt:
 - `boolean equals(Object other)`
 - `int compareTo(E o)`
 - Vergleich gemäß Definitionsreihenfolge der Objekte
 - `String toString()`
 - Zur Ausgabe der Objekte
 - `int ordinal()`
 - Liefert die Ordnungszahl des Enum-Objekts
 - `static <T extends Enum<T>> T valueOf(String)`
 - Zu einem Konstantennamen und einer Enum-Klasse T, wird das Enum-Objekt geliefert.

Zugriff auf die Objekte

- Die statische Methode **values()** liefert für jede Aufzählungsklasse ein Array mit allen Objekten der Klasse, diese kann man dann z.B. mit einer for-Schleife aufzählen:

```
for(WochenTag t : WochenTag.values())  
    if(t.istArbeitsTag()) System.out.println(t);
```

Mo
Di
Mi
Do
Fr

- Eine weitere nützliche Methode **ordinal()** liefert die Nummer jedes Enum-Objektes. Daher hätten wir **istArbeitsTag()** auch berechnen können:

```
boolean istArbeitsTag() {  
    return this.ordinal() <= 4;  
}
```

- Die Ordinalzahl entspricht dem Index im Array

Noch ein Beispiel

```
enum Farbe {
    weiss, schwarz, rot, gold, gruen, blau;
    static Farbe[][] fahnen = { {weiss, blau}, {schwarz, rot, gold},
                                {rot, weiss}, {rot}
                                };
}

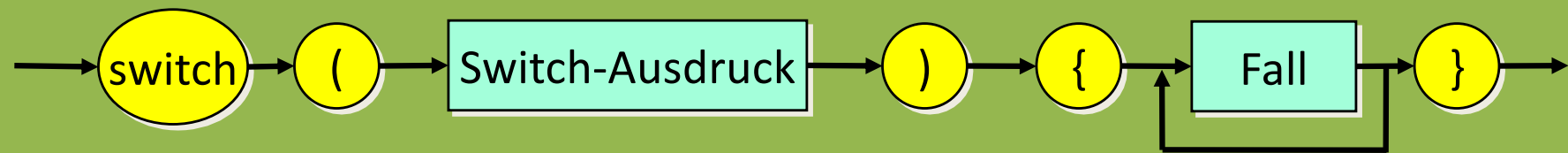
class EnumFarbenTest{
    public static void main(String[] args){
        for(Farbe[] f: Farbe.fahnen) {
            for (Farbe g: f)
                System.out.print(g + " ");
            System.out.println();
        }
    }
}
```

```
weiss blau
schwarz rot gold
rot weiss
rot
```

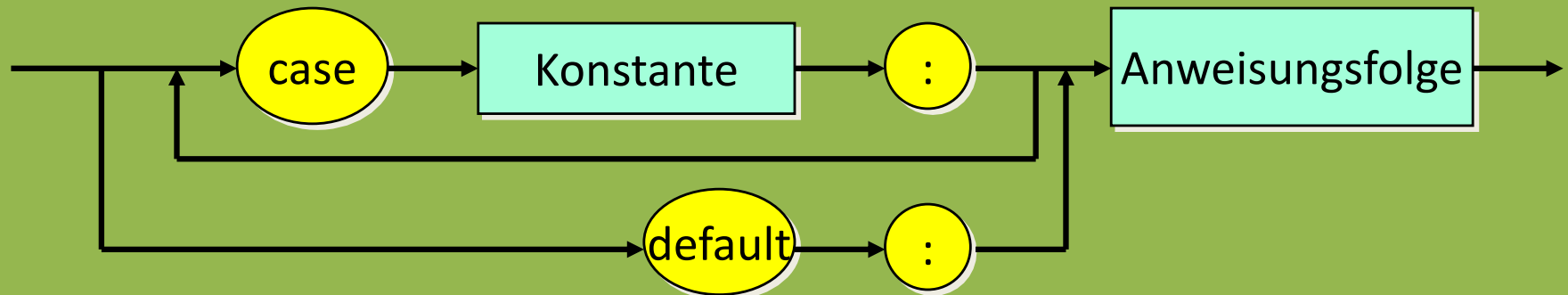
15.2 Die switch-Anweisung

- Wenn in **Abhängigkeit von dem Wert eines enum-Objekts** eine entsprechende Anweisung ausgeführt werden soll, kann man diese oft mit einer **switch-Anweisung** zusammenfassen.

Switch-Anweisung:



Fall:



Aufbau und Bedeutung der switch-Anweisung

- Der Typ des **switch-Ausdrucks** muss **char**, **byte**, **short**, **int** , **String** oder ein **enum-Typ** sein.
- Für alle **unterstützten** Ergebniswerte dieses Ausdrucks kann jeweils ein **Fall** formuliert werden.
 - Dieser besteht immer aus einer Konstanten und einer Folge von Anweisungen.
 - Schließlich ist noch ein **Default-Fall** erlaubt.
- Verhalten der switch-Anweisung
 - Wenn der **switch-Ausdruck** gleich dem Wert eines der **Fälle ist**, so wird die **zugehörige Anweisungsfolge** und die **aller folgenden Fälle (!)** ausgeführt.
 - Ansonsten wird, falls vorhanden, der **Default-Fall** und alle **folgenden Fälle (!)** ausgeführt.

Default-Fall sollte letzter Fall sein!

Verwendung von break und return

- Die **switch-Anweisung** ist gewöhnungsbedürftig und führt häufig zu unbeabsichtigten Fehlern.
- Die Regel, dass nicht nur die Anweisung eines Falles, sondern auch die Anweisungen aller auf einen Treffer folgenden Fälle ausgeführt werden, führt häufig zu Fehlern.
 - Tatsächlich ist dies ein Relikt von der Programmiersprache C.
- Wenn man möchte, dass immer nur genau ein Fall ausgeführt wird, so muss man jeden **Fall** mit einer
 - **return-Anweisung** oder
 - **break-Anweisung** enden lassen. (Es wird der durch die **switch-Anweisung** gebildete Block verlassen)

Beispiel: switch mit int

- Im folgenden Beispiel werden **Fälle** mit einer **return-Anweisung** beendet
- Wir definieren eine Methode, die die Anzahl der Tage eines Monats bestimmt.

```
static int tageProMonat(int j, int m){  
    switch (m){  
        case 1: case 3: case 5: case 7:  
        case 8: case 10: case 12: return 31;  
        case 2: if (schaltJahr(j)) return 29; else return 28;  
        default: return 30;  
    }  
}
```

Beispiel: switch mit String

- Im folgenden Beispiel testen wir ein switch mit einem String-Ausdruck.

```
static int getMonatAlsZahl(String monat) {  
    int monatAlsZahl = 0;  
    if (monat == null) return monatAlsZahl;  
    switch (monat.toLowerCase()) {  
        case "januar":    monatAlsZahl = 1; break;  
        case "februar":   monatAlsZahl = 2; break;  
        case "märz":      monatAlsZahl = 3; break;  
        case "april":     monatAlsZahl = 4; break;  
        case "mai":       monatAlsZahl = 5; break;  
        case "juni":      monatAlsZahl = 6; break;  
        case "juli":      monatAlsZahl = 7; break;  
        case "august":    monatAlsZahl = 8; break;  
        case "september": monatAlsZahl = 9; break;  
        case "oktober":   monatAlsZahl = 10; break;  
        case "november":  monatAlsZahl = 11; break;  
        case "dezember":  monatAlsZahl = 12; break;  
        default:         monatAlsZahl = 0; break;  
    }  
    return monatAlsZahl;  
}
```

Beispiel: switch mit enum

- Im folgenden Beispiel testen wir ein switch mit einem enum-Ausdruck.

```
enum Season{WINTER, SPRING, SUMMER, FALL};

public class SeasonTest {

    public static String getGermanName(Season season) {
        String str = null;
        switch (season) {
            case WINTER: str = "Winter"; break;
            case SPRING: str = "Frühling"; break;
            case SUMMER: str = "Sommer"; break;
            case FALL: str = "Herbst"; break;
        }
        return str;
    }

    public static void main(String[] args) {
        for(Season s : Season.values())
            System.out.println(s + " = " + getGermanName(s));
    }
}
```

16 Innere Klassen

- Klassen und Interfaces können zur besseren Strukturierung von Programmen verschachtelt werden.
 - Eine **innere Klasse** wird innerhalb einer anderen Klasse definiert.
 - Eine nicht-innere Klasse wird auch als **Top-Level Klasse** bezeichnet.

```
public class TopLevelKlasse {  
    ...  
    public class InnereKlasse {  
        ...  
    }  
    ...  
}
```

- Innere Klassen sind nützliche Konzepte
 - Sie bieten insbesondere die Grundlagen für die funktionalen Konzepte in Java.

Varianten von inneren Klassen

- Unterscheidung zwischen
 - Geschachtelte Top-Level-Klassen
 - Klassen und Interfaces, die innerhalb anderer Klassen definiert sind, aber sich **trotzdem wie Top-Level-Klassen** verhalten.
 - Elementklassen
 - innere Klassen, die **in anderen Klassen** definiert sind.
 - Lokale Klassen
 - Klassen, die **innerhalb einer Methode oder eines Java-Blocks** definiert werden.
 - Anonyme Klassen
 - Lokale Klassen **ohne Namen**

Geschachtelte Top-Level-Klassen

- Geschachtelte Top-Level-Klassen bzw. geschachtelte Top-Level-Interfaces verhalten sich wie andere Klassen bzw. Interfaces in einem Paket.
 - Einziger Unterschied
 - Ihrem Namen muss immer der **Name der umgebenden Klasse als Präfix** vorangestellt werden.
 - Motivation
 - Bessere Strukturierung durch geschachtelte Klassen
- Geschachtelte Klassen bzw. Interfaces werden mit **static** deklariert.

```
public class A{  
    ...  
    public static class B {  
        ...  
    }  
    ...  
}
```

```
// Verwendung der Klassen  
  
A a = new A();  
  
A.B ab = new A.B();
```


Elementklassen


- Im Gegensatz zu den geschachtelten Top-Level-Klassen handelt es sich bei Elementklassen um **echte innere** Klassen.
 - Die Definition der Klassen erfolgt **ohne das Schlüsselwort static**.
- Eine Instanz einer Elementklasse ("inner") existiert in einer Instanz der umgebenden Klasse ("outer")
- Eine Instanz einer Elementklasse hat vollen **Zugriff auf alle Datenfelder und Methoden** der umgebenden Instanz
 - Innere Klassen können dadurch den **(privaten)** Zugriffsschutz aufweichen.
 - Diese Lösung ist aber oft besser als die Klassen nebeneinander zu implementieren und den Zugriffsschutz von private auf package-protected abzusenken

Class-Dateien

- Die Elementklassen tragen den Namen der übergeordneten Klasse(n) und den eigenen, wobei diese mit \$ unterteilt werden.

```
public class A{  
    public class B {  
        public class C {  
        }  
    }  
}
```

javac A.java



Erzeugte class-Dateien

- A.class
- A\$B.class
- A\$B\$C.class

Objekte von Elementklassen

- Objekte von Elementklassen sind **immer von einem Objekt der umgebenden Klasse abhängig**.
 - Insbesondere der Konstruktor der Elementklasse kann nur mit Hilfe eines Objekts der äußeren Klasse aufgerufen werden.
 - Damit kann ein Objekt der Elementklasse auch auf die (privaten) Datenfelder und Methoden der äußeren Klasse zugreifen.

```
public class A{  
    private int i = 42;  
    public class B {  
        public int j;  
        B() {  
            j = i; // ok  
        }  
    }  
}
```

```
// Anwendung  
  
A a = new A();  
  
B b = a.new B();  
  
System.out.println(b.j);  
// Zugriff auf b.i funktioniert nicht
```

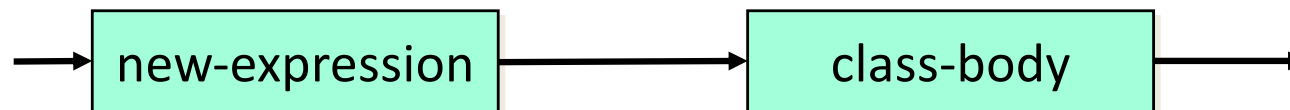
- Es gilt folgende Einschränkung:
 - Elementklassen haben **keine statischen Datenfelder und Methoden**.

Bsp.: Iterator als Elementklasse

```
public class LinkedList<E> implements Iterable<E> {  
    private ListElement<E> head, tail;  
    ...  
  
    public Iterator<E> iterator() {  
        return new ListenIterator();  
    }  
  
    /* Elementklasse lokal innerhalb von LinkedList */  
    private class ListenIterator implements Iterator<E> {  
        private ListElement<E> nextElem = head; // Zugriff auf head erlaubt !  
  
        public E next() {  
            ListElement<E> tmp = nextElem;  
            nextElem = nextElem.next;  
            return tmp.data;  
        }  
  
        public boolean hasNext(){  
            return nextElem != tail;  
        }  
    }  
}
```

Anonyme Klassen

- Anonyme Klasse werden wie lokale Klassen innerhalb von Anweisungsblöcken definiert.
 - Anonyme Klassen haben jedoch **keinen Namen!**
- Sie entstehen immer gleichzeitig zusammen mit **einem** Objekt, aber
 - sie haben **keinen Konstruktor**
 - sie haben die gleichen Beschränkungen wie lokale Klassen.
- Syntax von anonymen Klassen



- Für die Anonyme Klasse kann man **keine extends- oder implements-Klauseln** angeben.

Bsp.: Iterator als anonyme Klasse

```
/**
 * Ein iterator mit einer anonymen Klasse innerhalb der Klasse
 * LinkedList.
 */
public Iterator<E> iterator() {
    return new Iterator<E>() {
        ListElement nextPos = head;                // Aktuelle Position

        public boolean hasNext() {
            return nextPos != tail;
        }

        public E next() {
            ListElement tmp = nextPos;
            nextPos = nextPos.next;
            return tmp.data;
        }
    };
}
```

Dies ist möglich, obwohl Iterator ein Interface ist.

// Semikolon der return-Anweisung nicht vergessen.

Diskussion

- Durch die Verwendung von Generics und anonymen Klassen haben wir eine allgemein einsetzbare Lösung zum Filtern von Daten auf Iteratoren entwickelt.
- Jedoch lässt sich darüber streiten, ob dies wirklich eine **elegante Lösung** ist.
 - Wir müssen eine (anonyme) Klasse bauen, obwohl wir nur eine kleine Funktion (test) implementieren wollen.
 - Wir müssen die Typen aufeinander abstimmen, so dass auch alles zueinander passt.
- Dieses Probleme werden durch die Verwendung von den neuen funktionalen Konzepten von Java („Lambdas“) größtenteils behoben.

17. Parallele Programmierung in Java mit Threads

- Überblick
 - Klassische Verwendung von Threads
 - Callable und Future
 - Synchronisation



Motivation

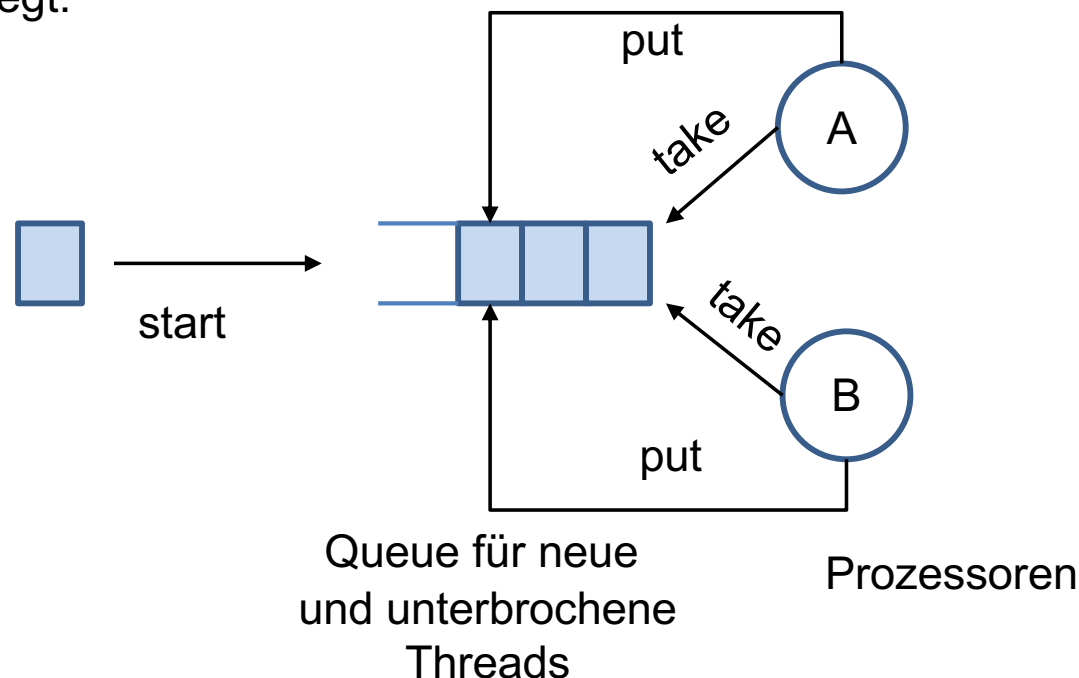
- Komplexe Programmsysteme müssen gleichzeitig verschiedene Benutzer bedienen (Pseudo-Parallelität).
 - Verschiedene Operationen laufen gleichzeitig miteinander (auch wenn nicht notwendigerweise mehrere CPUs vorhanden sind).
 - Beispiele
 - Betriebssysteme
 - Datenbanksysteme
- Parallele Rechnersysteme
 - Eine Aufgabe soll parallel programmiert werden, um alle Kerne eines Rechners zu nutzen und somit die Laufzeit eines Programms zu verringern.
 - Parallelisieren von klassischen Algorithmen wie Sortieren, Suchen, ...

Ausführungsthreads

- Eine moderne CPU besitzt mehrere Kerne
 - Durch Simultaneous Multi-Threading (SMT) kann jeder Kern mehrere Instruktionen parallel ausführen
 - Typische Desktop CPUs haben 4-8 Kerne, die jeweils 2 SMTs ausführen können
 - D.h., 8-16 Hardware-Threads
- Oft benötigen Anwendungen mehr Ausführungsthreads
 - Außerdem werden mehrere Anwendungen gleichzeitig ausgeführt
- Das Betriebssystem und die Programmiersprache unterstützen durch Software-Threads wesentlich mehr Ausführungsthreads
 - Größenordnung mehrere Tausend bis Zigtausend

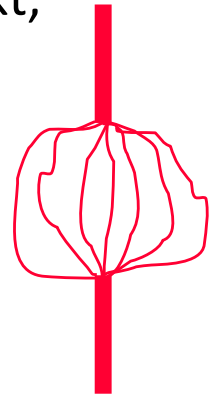
Realisierung von Software-Threads durch Betriebssystem/ Java Virtuelle Maschine

- Threads werden nach dem Start in einer Queue abgelegt.
- Prozessoren nehmen sich die Threads aus der Queue und verarbeiten diese bis ein Ereignis, wie z. B. das Zeitlimit ist überschritten, eintritt.
 - Falls Thread noch nicht fertig ist, wird dieser unterbrochen und in die Queue abgelegt.



17.1 Threads

- Bisher
 - Ein Java-Programm hat zu jedem Zeitpunkt genau eine aktive Methode.
- Threads erlauben die gleichzeitige Verarbeitung von mehreren aktiven Methoden in einem Programm.
 - Der Programmablauf verläuft gemeinsam bis zu einem gewissen Punkt, dann **teilt** sich die Ausführung in mehrere **Threads**.
 - Zu einem späteren Zeitpunkt kann die Ausführung dann wieder **gemeinsam** fortgesetzt werden.
- Wichtige Klassen und Interfaces in Java
 - **funktionales Interface Runnable**
 - Klasse Thread



Interface Runnable

- Funktionale Schnittstelle bietet die Methode `void run()`.
 - In der Methode wird die gewünschte Funktionalität implementiert, die über ein Objekt der Klasse Thread parallel ablaufen soll.
- Zwei Möglichkeiten
 - Implementierung über eine eigene Klasse

```
class TicTacToe implements Runnable{  
    String was;  
    TicTacToe(String s){ was = s; }  
    public void run() {  
        System.out.println(was);  
    }  
}
```

- Üblicher: Implementierung als anonyme Klasse oder Lambda

```
Runnable tic = () -> System.out.println("Tic");
```

Klasse Thread

- Die Klasse Thread wird benutzt, um ein Runnable-Objekt parallel auszuführen.

- Erzeugung eines Objekts der Klasse mit einem Runnable-Objekt

```
Thread t = new Thread(tic);
```

- Starten des Threads mit der Methode start()

```
t.start();
```

- Danach gibt es zwei aktive Methoden (Ausführungsfäden, Threads)
 - Hauptprogramm wird fortgeführt.
 - Dieser Ausführungsfaden endet mit der Methode main.
 - Methode start führt die Methode run des Objekts tic aus.
 - Dieser Ausführungsfaden endet mit der Methode run.
 - Beide laufen voneinander unabhängig.

Beispiel

- Betrachten wir folgendes Programm:

```
public static void main(String args[]) {  
    System.out.println("Thread Beispiel");  
    Runnable tic = () -> System.out.println("Tic ");  
    Runnable tac = () -> System.out.println("Tac ");  
    Runnable toe = () -> System.out.println("Toe ");  
    new Thread(tic).start();  
    new Thread(tac).start();  
    new Thread(toe).start();  
    System.out.println("Hauptprogramm")  
}
```

1. Programmstart
z.B.:

Thread Beispiel
Hauptprogramm
Tac
Tic
Toe

- Die Ausgabe ist **nichtdeterministisch**.

- Das gleiche Programm kann ganz verschiedene Ausgaben produzieren

2. Programmstart
z.B.:

Thread Beispiel
Tic
Toe
Tac
Hauptprogramm

Weitere Methoden der Klasse Thread

- `static void sleep(long millis)`
 - Beim Aufruf dieser Methode wird die aufrufende Methode für *millis* Millisekunden schlafen gelegt.
- `void setPriority(int newPriority)`
 - Hiermit kann die Priorität eines Threads auf *newPriority* gesetzt werden.
 - Ein Thread mit hoher Priorität kommt öfters auf einem Prozessor zur Ausführung als ein Thread mit niedriger Priorität.
- `void join()`
 - An dieser Stelle wird gewartet bis der zuvor gestartete Thread endet.

Beispiel

```
public static void main(String args[]) {  
    System.out.println("Thread Beispiel");  
    Runnable tic = () -> System.out.println("Tic ");  
    Runnable tac = () -> System.out.println("Tac ");  
    Runnable toe = () -> System.out.println("Toe ");  
    Thread[] t = {new Thread(tic), new Thread(tac), new Thread(toe)};  
    for (i=0; i < 2; i+=1)  
        t[i].start();  
    for (i=0; i < 2; i+=1)  
        try {  
            t[i].join();  
        }  
        catch (InterruptedException ie ){ /* ToDo */ }  
    System.out.println("Hauptprogramm");  
}
```

Thread Beispiel
Tic
Toe
Tac
Hauptprogramm

- Die Ausgabe ist immer noch nichtdeterministisch.
 - Jedoch ist garantiert, dass stets am Schluss "Hauptprogramm" ausgegeben wird.

Thread Beispiel
Hauptprogramm
Tac
Tic
Toe