Lektion 22

Geschachtelte Klassen (anonyme, lokale)
Threads
Visitor Pattern

Nebenläufige Programmierung mit Threads

Es gibt verschiedene Kategorien von Geschachtelten Klassen (Nested Classes):

static nested classes
inner classes
local classes
anonymous classes
lambda expressions

Wir schauen uns noch ein Beispiel mit einer anonymen Klasse an.

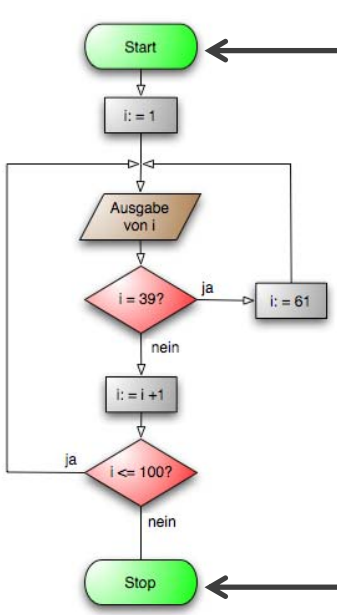
static nested classes
inner classes
local classes
anonymous classes
lambda expressions

Typische Beispiele für anonyme Klassen sind Klassen, von denen nur ein Objekt benötigt wird, u.a.:

- parallel auszuführender Code
- Befehle bei grafischen Oberflächen

Klassischer Programmablauf

Bei einem herkömmlichen Ablauf behält das Programm vom Start bis zum Ende die Kontrolle. Allenfalls Exceptions können den Ablauf unterbrechen.



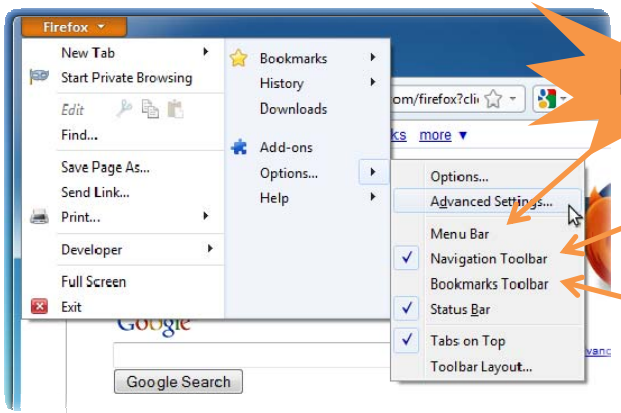
Fixer Startpunkt

```
public static void main(String[] args)
{
  …
}
```

Fixer Endpunkt

Ereignisgesteuerter Programmablauf

Oft kann diese umfassende Kontrolle nicht gewährleistet werden.
Das Programm muss primär auf Ereignisse reagieren.

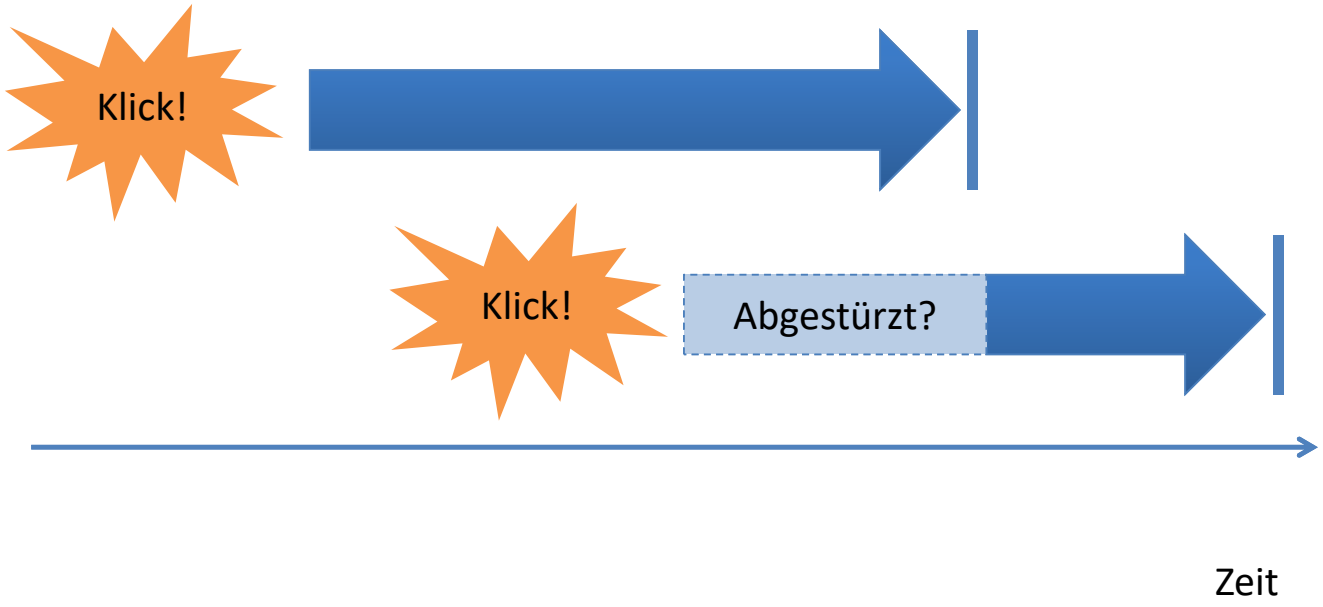


Klick!

```
public void onMenuBarClicked()
{
  …
}

public void onNavigationToolbarClicked()
{
  …
}

public void onBookmarksToolbarClicked()
{
  …
}

…
```

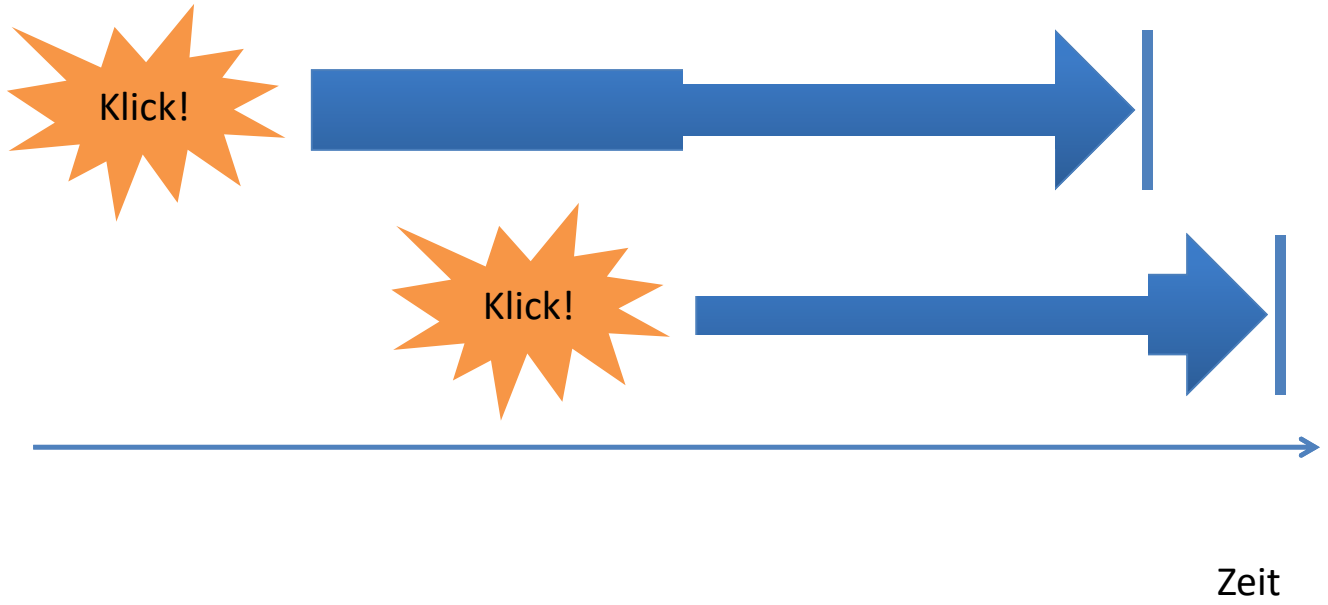
Ereignisgesteuerter Programmablauf

Wird ein Klick nach dem anderen bearbeitet, kann der Eindruck entstehen, dass das Programm „abgestürzt“ ist.



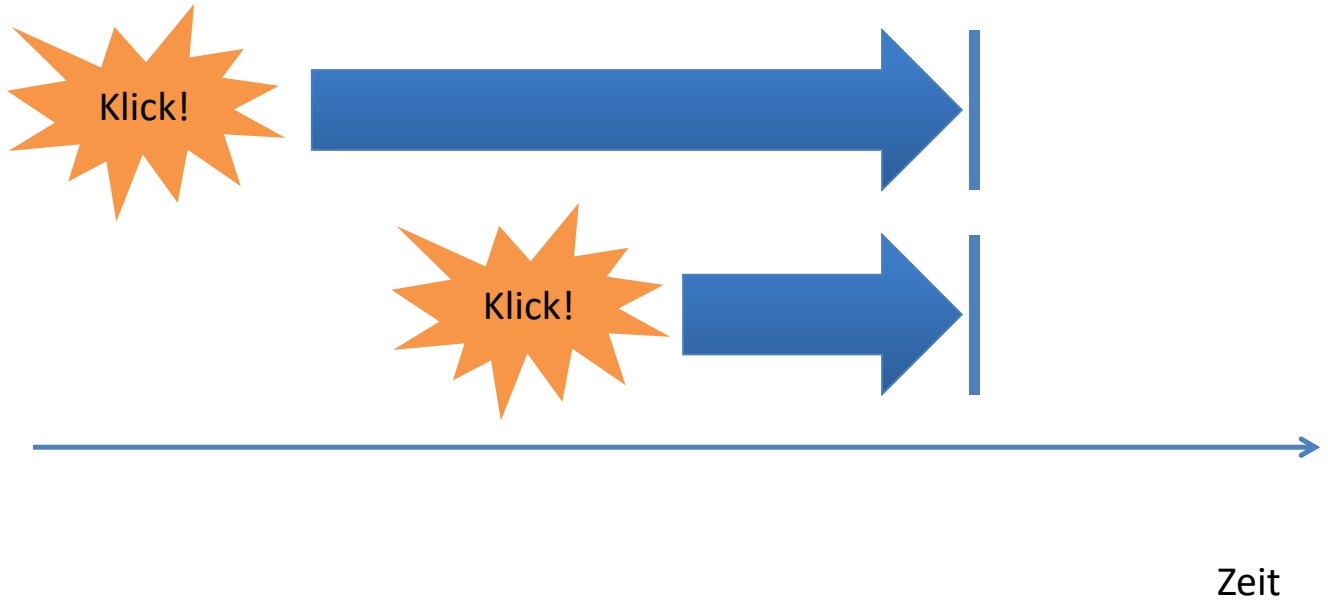
Ereignisgesteuerter Programmablauf

Besser: Quasi-parallele Abarbeitung der Klicks (die Rechenleistung wird geteilt), so dass zumindest ein geringer Fortschritt festzustellen ist.



Ereignisgesteuerter Programmablauf

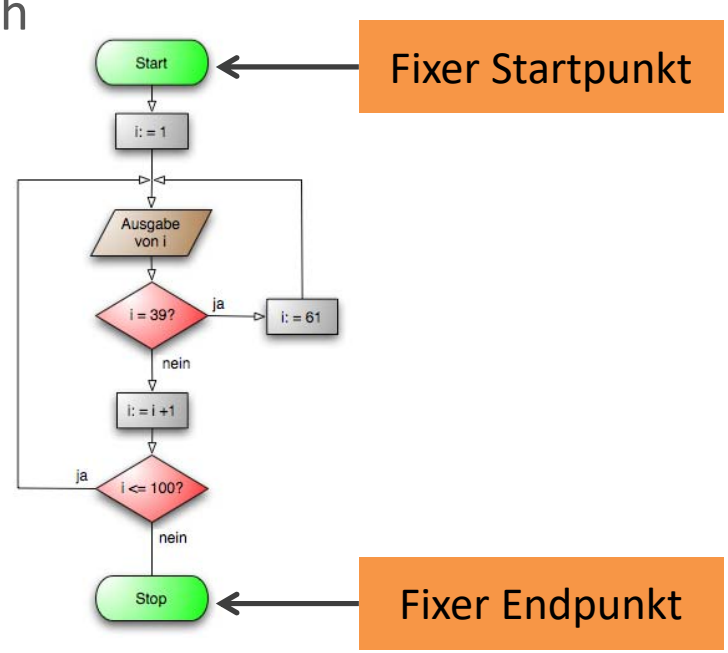
Stehen mehrere Prozessoren bzw. Prozessorkerne zur Verfügung,
dann kann die Bearbeitung tatsächlich parallel erfolgen.



Wie kann dem Programmcode mitgeteilt werden,
dass er parallel zum „normalen“ Programm
ablaufen soll?

Was ist ein Thread?

- Ein **Thread** ist ein **sequentieller Ausführungsfaden** innerhalb eines Prozesses und ist gekennzeichnet durch einen
  - Anfangspunkt
  - sequentiellen Ablauf
  - Endpunkt

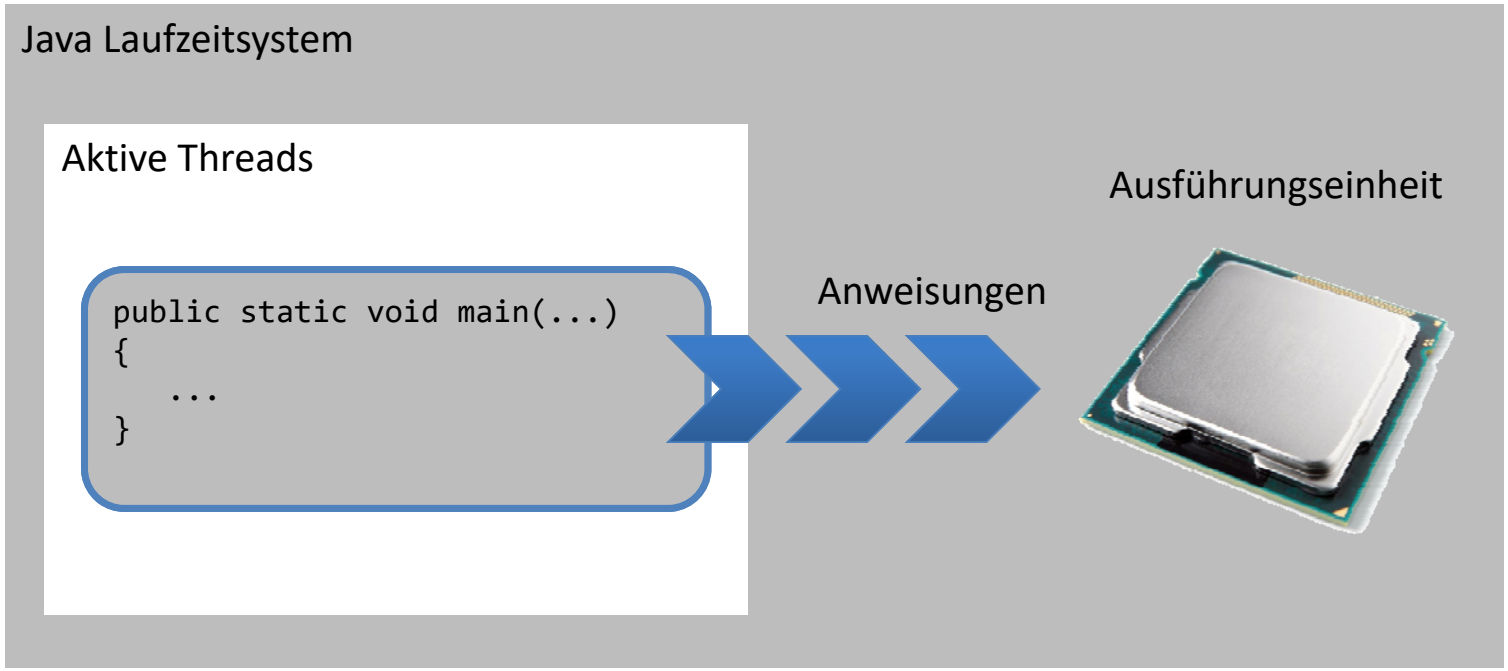


- Jeder Prozess erhält bei seiner Erzeugung i.d.R. einen Steuerungs-Thread, der weitere Threads erzeugen kann.

Was passiert beim Programmstart in Java?

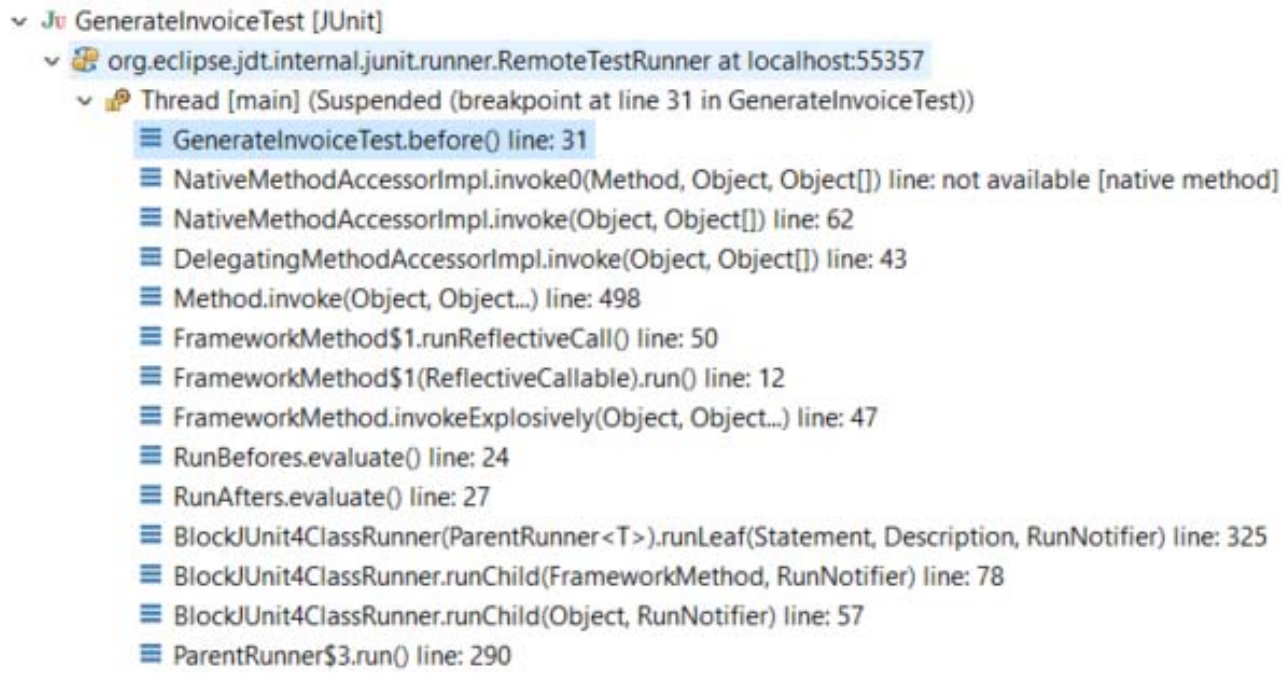
Unmittelbar mit dem Programmstart wird von der Java-Laufzeitumgebung ein sog. Thread angelegt und gestartet.

In diesem Thread beginnt die Ausführung der main-Methode.



Was passiert beim Programmstart in Java?

Innerhalb des Threads wird mit einem Befehlszähler festgehalten, welche Anweisung gerade ausgeführt wird.

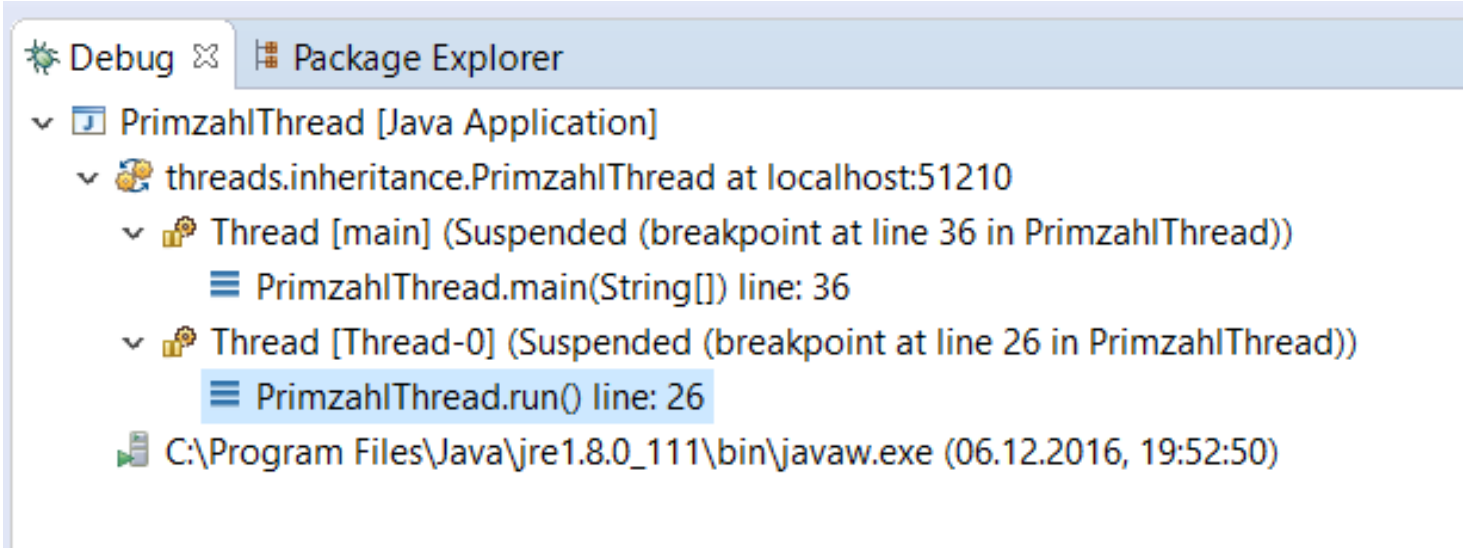


Nach der Ausführung der Anweisung wird der Befehlszähler weitergeschaltet bzw. bei Methodenaufrufen oder Schleifen auf die nächste Anweisung gesetzt.

Der Thread beendet sich („terminiert“), wenn der Befehlszähler auf keine weitere Anweisung mehr verweist, d.h. main abgearbeitet ist.

Während des Programmablaufs können weitere Threads erzeugt werden.

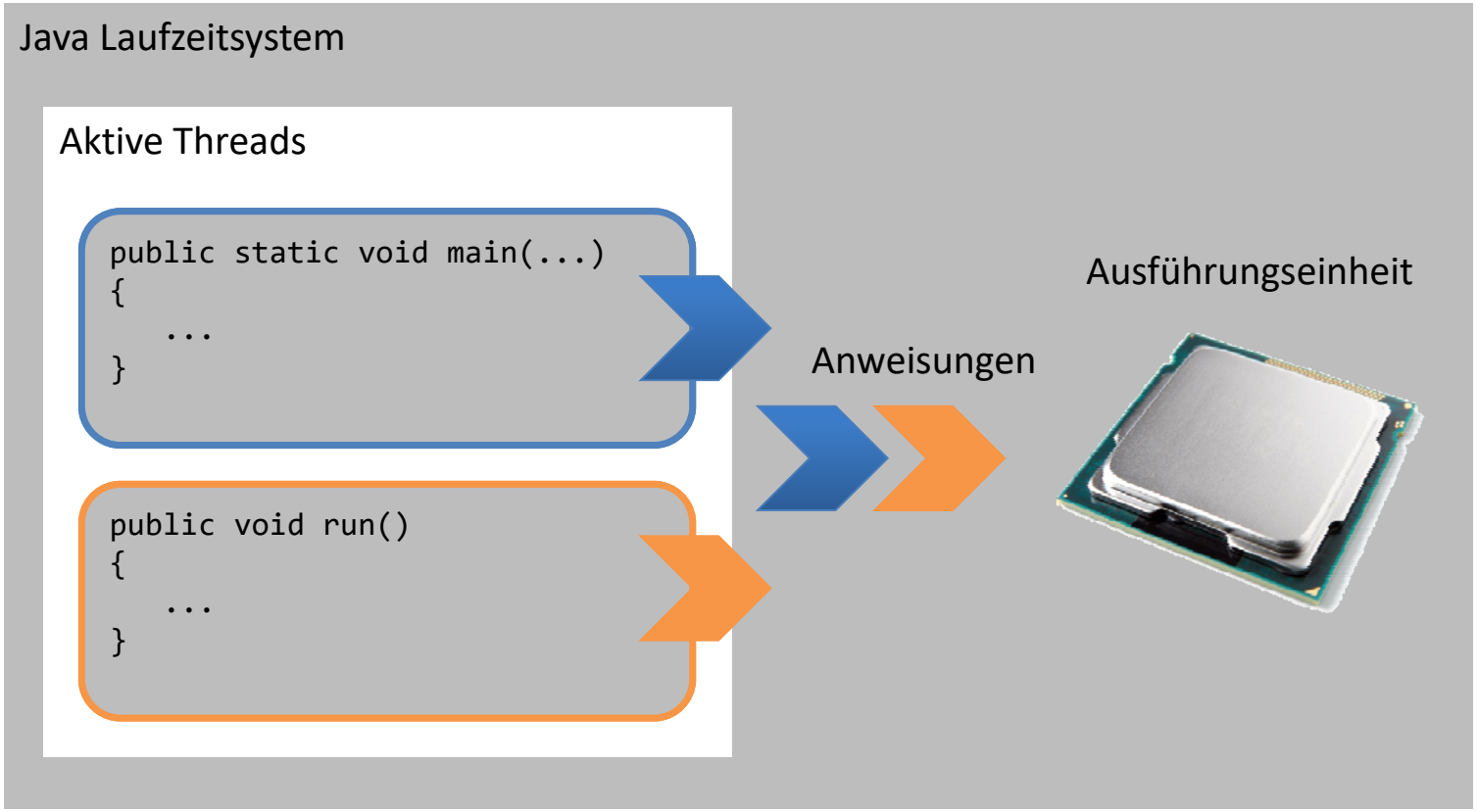
Diese besitzen einen eigenen Befehlszähler
und einen eigenen Stack.



Im Debugger kann man die parallel ausgeführten Threads
sehen, inkl. Zeile, in der sie gerade stehen.

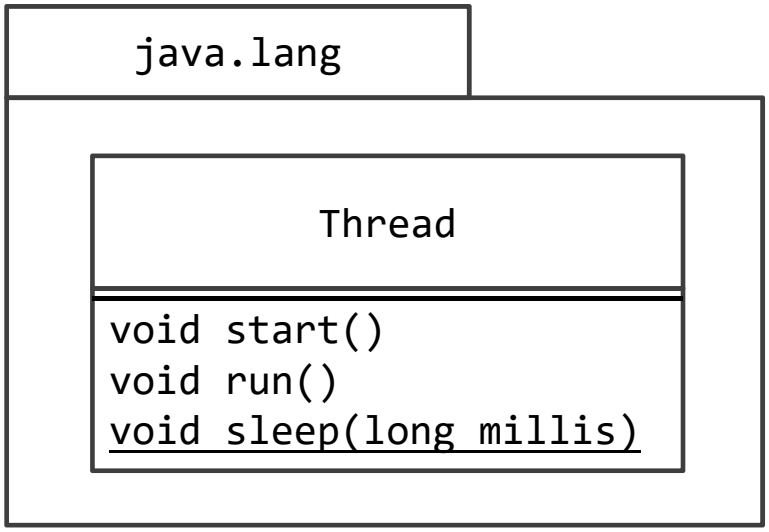
Der von einem neu erzeugten Thread auszuführende Code kann festgelegt werden, d. h. der Thread startet nicht die **main**-Methode...

...sondern Code in einer vorgegebenen Methode, der run-Methode()



Die JVM sorgt über einen sog. Dispatcher dafür, dass alle aktiven Threads (abwechselnd) zum Zuge kommen und Anweisungen an die Ausführungseinheit weiterreichen können.

In Java sind Threads in die Sprache integriert und werden durch die Klasse Thread repräsentiert.



`start()`

- übernimmt Parallelisierung
- startet neuen Thread
- neuer Thread ruft `run()`-Methode auf

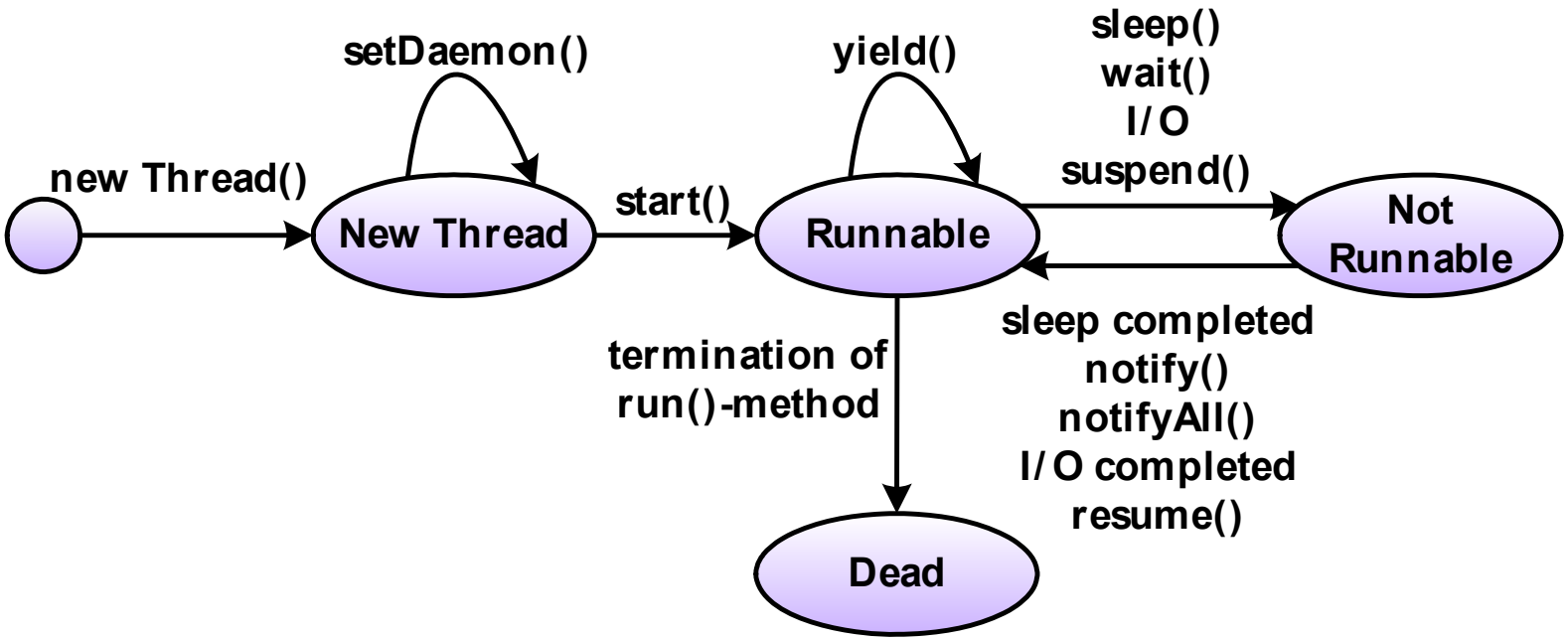
`run()`

- beinhaltet Anweisungen, welche parallel ausgeführt werden sollen

`sleep()`

- Der aktuelle Thread wird für *millis* Millisekunden „schlafen“ gelegt.

Threadzustände in Java



Threads in Java programmieren

Wie kann Programmcode mitgeteilt werden, dass er parallel zum „normalen“ Programm ablaufen soll?

Wir untersuchen das, indem wir einen Thread schreiben, der überprüft, ob eine Zahl eine Primzahl ist.

Zuerst entwickeln wir die Primzahlberechnung:

```
public class PrimzahlThread {
  int zahlZumTesten;


  public PrimzahlThread(int zahlZumTesten) {
    this.zahlZumTesten = zahlZumTesten;
  }

  public boolean testeObPrimzahl() {
    if (zahlZumTesten < 2) return false;
    for (int divisor = 2; divisor < zahlZumTesten; divisor++)
      if (zahlZumTesten % divisor == 0) return false;
    return true;
  }
}
```

Als nächstes müssen wir unseren Code parallel ausführen.

Es gibt mehrere Möglichkeiten, Threads zu verwenden:

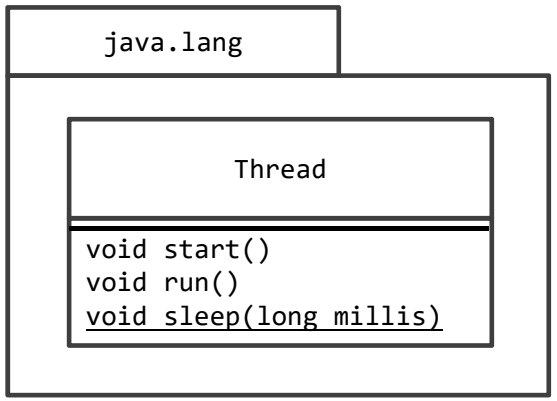
- **durch Vererbung**
- durch Implementierung des Interface-Runnable
- durch eine anonyme Klasse
- durch einen Lambda-Ausdruck

Threads schreiben durch Vererbung

```
public class PrimzahlThread extends Thread {
  int zahlZumTesten;


  public PrimzahlThread(int zahlZumTesten) {
    this.zahlZumTesten = zahlZumTesten;
  }

  public boolean testeObPrimzahl() {
    …
  }

  @Override
  public void run() {
    boolean istPrimzahl = testeObPrimzahl();
    System.out.println(zahlZumTesten + " ist "
                       + (istPrimzahl ? "eine " : "keine ")
                       + "Primzahl.");
  }
}
```

run-Methode von Thread geerbt

Wie starten wir den Thread?




```
public static void main(String[] args)
{
    PrimzahlThread pt1 = new PrimzahlThread(341);
    pt1.start();

    PrimzahlThread pt2 = new PrimzahlThread(633910099);
    pt2.start();
}
```

Was ist, wenn wir das Ergebnis wieder einsammeln wollen?

Was ist, wenn wir das Ergebnis wieder einsammeln wollen?

```
public class PrimzahlThread extends Thread {
  int zahlZumTesten;
  String ergebnis;

  public PrimzahlThread(int zahlZumTesten) {
    this.zahlZumTesten = zahlZumTesten;
  }

  public boolean testeObPrimzahl() {
    …
  }

  @Override
  public void run() {
    boolean istPrimzahl = testeObPrimzahl();
    ergebnis = zahlZumTesten + " ist "
                        + (istPrimzahl ? "eine " : "keine ")
                        + "Primzahl.");
  }
}
```

Wir können auf das Ergebnis nur zugreifen, wenn wir sicher sind, dass die Berechnung fertig ist.

```
public static void main(String[] args)
{
    PrimzahlThread pt1 = new PrimzahlThread(341);
    pt1.start();
    PrimzahlThread pt2 = new PrimzahlThread(633910099);
    pt2.start();
    System.out.println(pt1.ergebnis);
    System.out.println(pt2.ergebnis);
}
```

Wir können auf das Ergebnis nur zugreifen, wenn wir sicher sind, dass die Berechnung fertig ist.

```
public static void main(String[] args)
{
    PrimzahlThread pt1 = new PrimzahlThread(341);
    pt1.start();
    PrimzahlThread pt2 = new PrimzahlThread(633910099);
    pt2.start();
    try {
      pt1.join();
      pt2.join();


    }
    catch (InterruptedException e) {
      e.printStackTrace();
    }
}
```

Mit join wartet der aktuelle Thread (hier: der, der die main-Methode ausführt) auf das Ende eines Threads (pt1 bzw. pt2)

```
public static void main(String[] args)
{
    PrimzahlThread pt1 = new PrimzahlThread(341);
    pt1.start();
    PrimzahlThread pt2 = new PrimzahlThread(633910099);
    pt2.start();
    try {
      pt1.join();
      pt2.join();

      System.out.println(pt1.ergebnis);
      System.out.println(pt2.ergebnis);
    }
    catch (InterruptedException e) {
      e.printStackTrace();
    }
}
```


Ab dieser Stelle sind beide Threads durchgelaufen und deren Ergebnis kann ausgegeben werden.

Durch das Erben von java.lang.Thread war es unserer Klasse möglich als Thread ausgeführt zu werden.

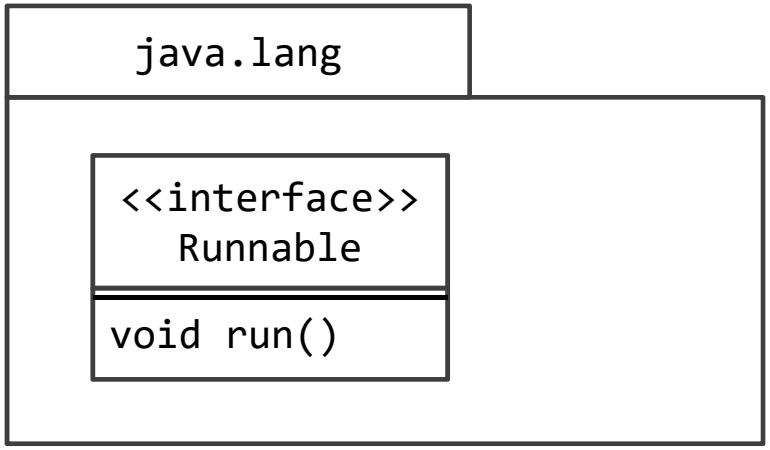
Was, wenn unsere Klasse bereits von einer anderen Klasse erbt?

Es gibt mehrere Möglichkeiten, Threads zu verwenden:

- durch Vererbung
- **durch Implementierung des Interface-Runnable**
- durch eine anonyme Klasse
- durch einen Lambda-Ausdruck

Ausschnitt aus der Klasse `java.lang.Thread`

Der Klasse Thread kann ein Objekt im Konstruktor übergeben werden, das das Runnable-Interface implementiert.



Beim Aufruf der run()-Methode (die beim Starten des Threads automatisch aufgerufen wird) wird die run()-Methode des Runnable-Objekts ausgeführt, wenn eines vorhanden ist.

```
public class Thread implements Runnable
{
  …
  /* What will be run. */
  private Runnable target;

  public Thread(Runnable target)
  {
    …
    this.target = target;
    …
  }

  public void run()
  {
    if (target != null) target.run();
  }
  …
}
```

Threads schreiben durch Implementieren des Interface Runnable

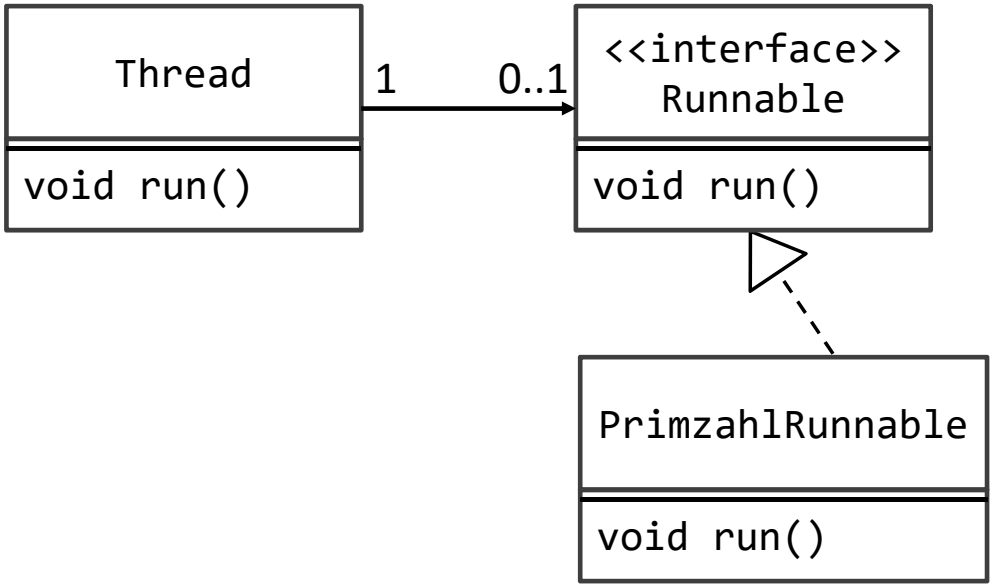
```
public class PrimzahlRunnable implements Runnable {
  int zahlZumTesten;
  String ergebnis;

  public PrimzahlRunnable(int zahlZumTesten) {
    this.zahlZumTesten = zahlZumTesten;
  }

  public boolean testeObPrimzahl() {
    …
  }

  @Override
  public void run() {
    boolean istPrimzahl = testeObPrimzahl();
    ergebnis = zahlZumTesten + " ist "
                        + (istPrimzahl ? "eine " : "keine ")
                        + "Primzahl.");
  }
}
```

```
public static void main(String[] args)
{
  PrimzahlRunnable pr1 = new PrimzahlRunnable(341);
  PrimzahlRunnable pr2 = new PrimzahlRunnable(633910099);
  Thread t1 = new Thread(pr1);
  Thread t2 = new Thread(pr2);
  t1.start();
  t2.start();

  try
  {
    t1.join();
    t2.join();

    System.out.println(pr1.ergebnis);
    System.out.println(pr2.ergebnis);
  }
  catch (InterruptedException e)
  {
    e.printStackTrace();
  }
}
```

Der Klasse java.lang.Thread wird das Runnable übergeben.

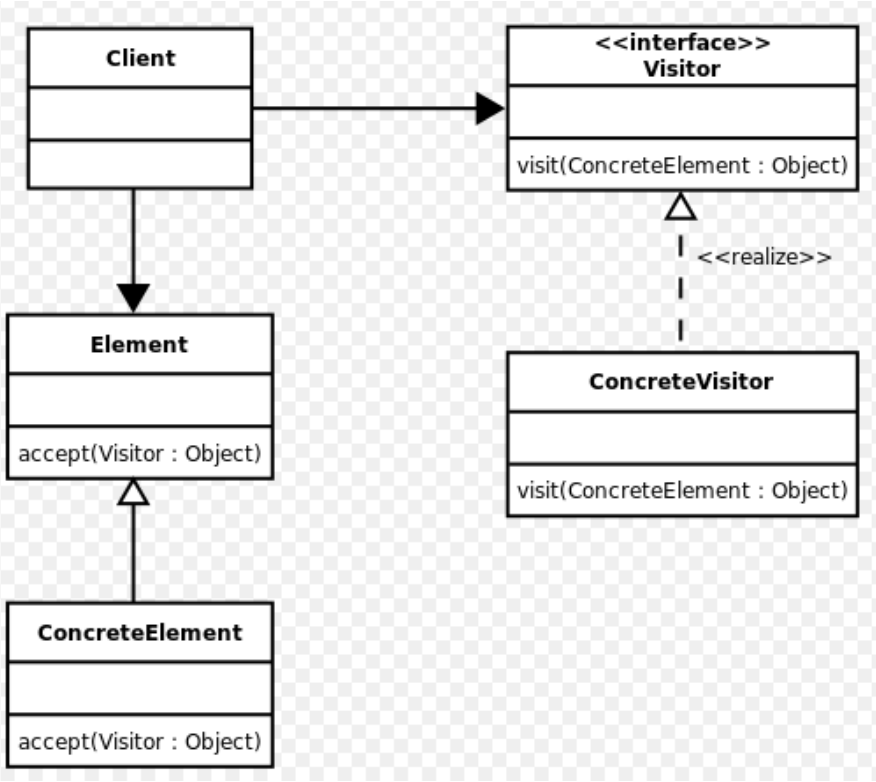
Ähnlicher Aufbau wie das Visitor Pattern

Thread akzeptiert im Konstruktor einen Visitor (Objekt vom Typ Runnable) und führt in der eigenen run-Methode dessen run-Methode aus.



Visitor Pattern

Element akzeptiert einen Visitor in der accept Methode und ruft dort vom Visitor visit() auf.



Wenn wir nur **einen** Thread starten, der eine einfache Aufgabe übernimmt, brauchen wir keine separate Klasse anlegen.

Es gibt mehrere Möglichkeiten, Threads zu verwenden:

- durch Vererbung
- durch Implementierung des Interface-Runnable
- **durch eine anonyme Klasse**
- durch einen Lambda-Ausdruck

```
public class PrimzahlAnonymerThread
{
  public static void main(String[] args)
  {
    int zahlZumTesten = 633910099;
    Thread t1 = new Thread()
    {
      @Override
      public void run()
      {
        boolean istPrimzahl;
        if (zahlZumTesten < 2) istPrimzahl = false;
        else istPrimzahl = true;
        for (int divisor = 2; divisor < zahlZumTesten; divisor++)
          if (zahlZumTesten % divisor == 0) istPrimzahl = false;

        System.out.println(zahlZumTesten + " ist "
                           + (istPrimzahl ? "eine " : "keine ") + "Primzahl.");
      }
    };
    t1.start();
  }
}
```

Eine **anonyme Klasse** ist eine Klassendefinition mit gleichzeitiger Instanziierung.

Bei der Instanziierung wird eine Referenz auf ein Objekt zurückgegeben, das kompatibel zum Basistyp ist (in unserem Bsp. Thread). Der konkrete Typ bleibt anonym.

```
public class PrimzahlAnonymerThread
{
  public static void main(String[] args)
  {
    int zahlZumTesten = 633910099;
    Thread t1 = new Thread()
    {
      @Override
      public void run()
      {
        boolean istPrimzahl;
        if (zahlZumTesten < 2) istPrimzahl = false;
        else istPrimzahl = true;
        for (int divisor = 2; divisor < zahlZumTesten; divisor++)
          if (zahlZumTesten % divisor == 0) istPrimzahl = false;

        System.out.println(zahlZumTesten + " ist "
                           + (istPrimzahl ? "eine " : "keine ") + "Primzahl.");
      }
    };
    t1.start();
  }
}
```

Wir sehen, dass es schwierig ist auf diese Weise ein Ergebnis einzusammeln, da wir nur auf die Methoden von Thread zugreifen können.

Die Definition eigener öffentlicher Methoden innerhalb einer anonymen Klasse ist sinnlos, da nur die Schnittstelle des Basistyps nach außen sichtbar ist.

```
public class PrimzahlLocalClassThread {
  public static void main(String[] args) throws InterruptedException {
    int zahlZumTesten = 633910099;
    class LocalClassThread extends Thread {
      boolean istPrimzahl;

      public boolean testeObPrimzahl() { … }

      @Override
      public void run() {
        boolean istPrimzahl = testeObPrimzahl();
        ergebnis = zahlZumTesten + " ist " + (istPrimzahl ? "eine " :  "keine ") + "Primzahl.");
      }

      public boolean getErgebnis() {
        return istPrimzahl;
      }
    }
    //main ctd.
    LocalClassThread t = new LocalClassThread();
    t.start();
    t.join();
    System.out.println(t.getErgebnis());
  }
}
```

Durch eine lokale Klasse (im Gegesatz zu einer anonymen Klasse) kann die Schnittstelle **erweitert** werden.

Die Verwendung von geschachtelten Klassen sollte wohlüberlegt erfolgen.

Durch die Schachtelung von Klassen werden einzelne Klassendateien länger und damit unübersichtlicher.