- Die Schnittstelle für Threads ist eine Abstraktion des Betriebssystems (oder einer virtuellen Maschine), die es ermöglicht, mehrere Ausführungsfäden, jeweils mit eigenem Stack und PC ausgestattet, in einem gemeinsamen Adressraum arbeiten zu lassen.
- Der Einsatz lohnt sich insbesondere auf Mehrprozessormaschinen (bzw. Prozessoren mit mehreren Kernen) mit gemeinsamen Speicher.
- Vielfach wird die Fehleranfälligkeit kritisiert wie etwa von C. A. R. Hoare in *Communicating Sequential Processes*: „In its full generality, multithreading is an incredibly complex and error-prone technique, not to be recommended in any but the smallest programs.“

- Wie die *comp.os.research* FAQ belegt, gab es Threads bereits lange vor der Einführung von Mehrprozessormaschinen:
  „The notion of a thread, as a sequential flow of control, dates back to 1965, at least, with the Berkeley Timesharing System. Only they weren't called threads at that time, but processes [Dijkstra, 1965]. Processes interacted through shared variables, semaphores, and similar means. Max Smith did a prototype threads implementation on Multics around 1970; it used multiple stacks in a single heavyweight process to support background compilations."
  `http://www.serpentine.com/blog/threads-faq/the-history-of-threads/`
- UNIX selbst kannte zunächst nur Prozesse, d.h. jeder Thread hatte seinen eigenen Adressraum.

- Zu den ersten UNIX-Implementierungen, die Threads unterstützten, gehörten der Mach-Microkernel (entwickelt at der Carnegie Mellon University, eingebettet in NeXT, später Mac OS X) und Solaris (zur Unterstützung der von Sun ab 1992 hergestellten Multiprozessormaschinen). Heute unterstützen alle UNIX-Varianten einschließlich Linux Threads.
- 1995 wurde durch die *The Open Group* (einer Standardisierungsgesellschaft für UNIX) mit POSIX 1003.1c-1995 eine standardisierte Threads-Bibliotheksschnittstelle publiziert, die 1996 von der IEEE, dem ANSI und der ISO übernommen wurde.
- Diverse andere Bibliotheken für Threads existierten und existieren (u.a. von Microsoft und von Sun), sind aber nicht portabel und daher inzwischen von geringerer Bedeutung.

- Spezifikation der *Open Group*:
  `https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`
- Unterstützt
  - das Erzeugen von Threads und das Warten auf ihr Ende,
  - den gegenseitigen Ausschluss (notwendig, um auf gemeinsame Datenstrukturen zuzugreifen),
  - Bedingungsvariablen (*condition variables*), die einem Prozess signalisieren können, dass sich eine Bedingung erfüllt hat, auf die gewartet wurde,
  - Lese- und Schreibsperren, um parallele Lese- und Schreibzugriffe auf gemeinsame Datenstrukturen zu synchronisieren.
- Freie Implementierungen der Schnittstelle für C:
  - GNU Portable Threads:
    `http://www.gnu.org/software/pth/`
  - Native POSIX Thread Library:
    `http://people.redhat.com/drepper/nptl-design.pdf`

- Seit dem C++-Standard ISO 14882-2012 (C++11) werden POSIX-Threads direkt unterstützt, wobei die POSIX-Schnittstelle in eine für C++ geeignete Weise verpackt ist.
- Diese Schnittstelle basiert auf der zuvor entwickelten Boost-Schnittstelle für Threads.
- Die folgende Einführung bezieht sich auf C++11 bzw. auf die späteren Standards C++14 und C++17.

- Der von einem Thread auszuführende Programmtext wird in C++ immer durch ein sogenanntes Funktionsobjekt repräsentiert.
- In C++ sind alle Objekte Funktionsobjekte, die den parameterlosen Funktionsoperator unterstützen.
- Das könnte im einfachsten Falle eine ganz normale parameterlose Funktion sein:

```
void f() {
   // do something
}
```

- Das ist jedoch nicht sehr hilfreich, da wegen der fehlenden Parametrisierung unklar ist, welche Teilaufgabe die Funktion für einen konkreten Thread erfüllen soll.

```
class Task {
   public:
      Task( /* parameters */ );
      void operator()() {
         // do something in dependence of the parameters
      }
   private:
      // parameters of this task
};
```

- Eine Klasse für Funktionsobjekte muss den parameterlosen Funktionsoperator unterstützen, d.h. **void operator**()().
- Im privaten Bereich der *Task*-Klasse können alle Parameter untergebracht werden, die für die Ausführung benötigt werden.
- Der Konstruktor erhält die Parameter einer *Task* und kann diese dann in den privaten Bereich kopieren.
- Dann kann die parameterlose Funktion problemlos auf ihre Parameter zugreifen.

fork-and-join.cpp

```
class Task {
   public:
      Task(int id) : id(id) {};
      void operator()() {
         std::cout << "task " << id << " is being worked on" <<
    std::endl;
      }
   private:
      const int id;
};
```

- In diesem einfachen Beispiel wird nur ein einziger Parameter für eine *Task* verwendet: *id*
- Häufig genügt bereits ein Parameter, der die Identität festlegt.
- Für Demonstrationszwecke gibt der Funktionsoperator nur seine eigene *id* aus.
- So ein Funktionsobjekt kann auch ohne Threads erzeugt und benutzt werden:
  *Task* $t(7)$; $t()$;

fork-and-join.cpp

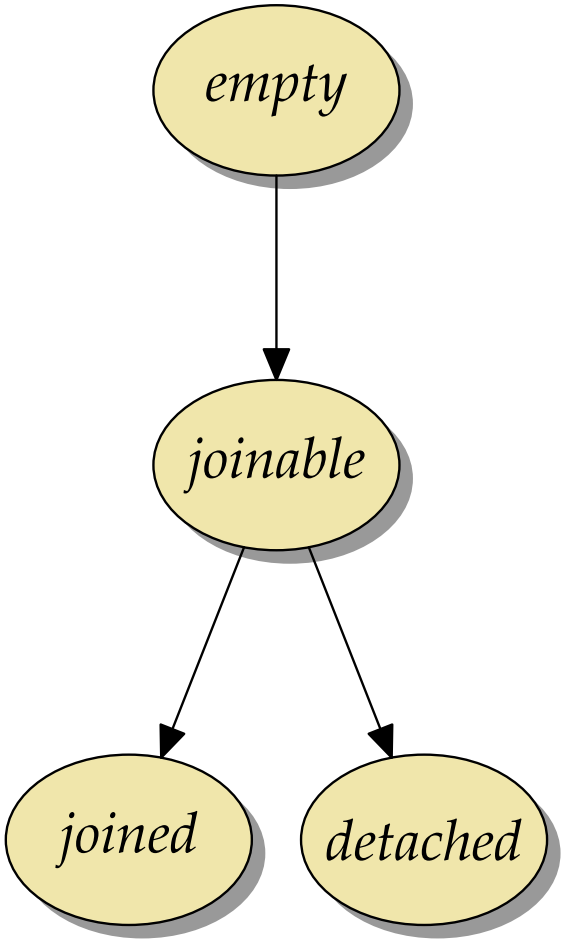
```
#include <iostream>
#include <thread>

// class Task...

int main() {
   // fork off some threads
   std::thread t1(Task(1)); std::thread t2(Task(2));
   std::thread t3(Task(3)); std::thread t4(Task(4));
   // and join them
   std::cout << "Joining..." << std::endl;
   t1.join(); t2.join(); t3.join(); t4.join();
   std::cout << "Done!" << std::endl;
}
```

- Objekte des Typs *std*::*thread* (aus **#include** <thread>) können mit einem Funktionsobjekt initialisiert werden. Die Threads werden sofort aktiv.
- Mit der *join*-Methode wird auf die Beendigung des jeweiligen Threads gewartet.

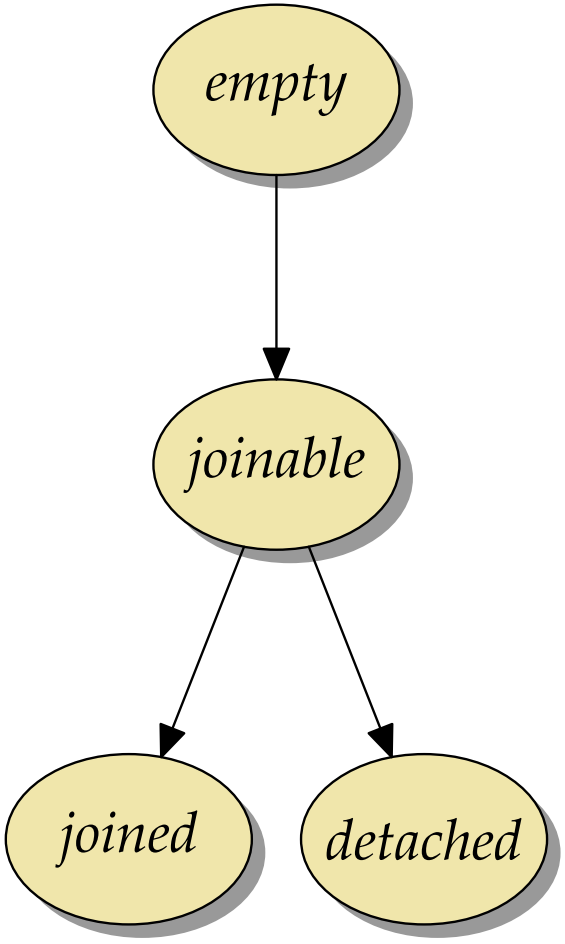
$$P_i = (fork \rightarrow join \rightarrow SKIP)$$

- Beim Fork-And-Join-Pattern werden beliebig viele einzelne Threads erzeugt, die dann unabhängig voneinander arbeiten.
- Entsprechend bestehen die Alphabete nur aus *fork* und *join*.
- Das Pattern eignet sich für Aufgaben, die sich leicht in unabhängig voneinander zu lösende Teilaufgaben zerlegen lassen.
- Die Umsetzung in C++ sieht etwas anders aus mit $\alpha P_i = \{fork_i, join_i\}$ und $\alpha M = \alpha P = \bigcup_{i=1}^{n} \alpha P_i$:

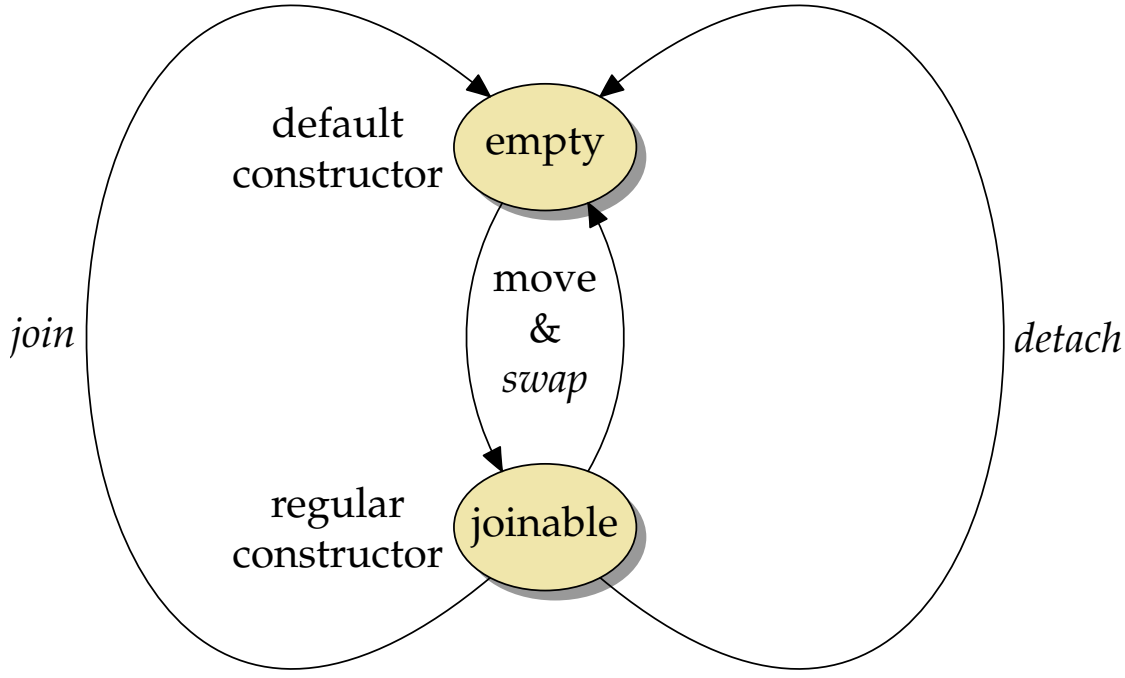
$$P = M \parallel P_1 \parallel \ldots \parallel P_n$$
$$M = (fork_1 \rightarrow \cdots \rightarrow fork_n \rightarrow join_1 \rightarrow \cdots \rightarrow join_n \rightarrow SKIP)$$
$$P_i = (fork_i \rightarrow join_i \rightarrow SKIP)$$




- Objekte des Typs *std*::*thread* sind RAII-Objekte (RAII = *resource acquisition is initialization*), die mit einer Ressource (hier der POSIX-Thread) fest verknüpft sind.
- Wenn ein *std*::*thread*-Objekt mit einem Funktionsobjekt erzeugt wird, dann wird bereits beim Konstruieren der POSIX-Thread erzeugt und das Objekt ist im Zustand *joinable*.
- Mit der *join*-Methode erreicht das Objekt den Zustand *joined*, in der der Thread beendet ist.




- Mit der *detach*-Methode kann ein *std*::*thread*-Objekt dauerhaft von dem zugehörigen Thread getrennt werden. Der Thread läuft dann im Hintergrund weiter. Eine Synchronisierung ist dann nicht mehr möglich.
- Die Zustände *joined* und *detached* sind äquivalent zum Zustand *empty*.
- Mit der *joinable*-Methode kann abgefragt werden, ob sich ein *std*::*thread*-Objekt im *joinable*-Zustand befindet.
- Wenn beim Abbau eines *std*::*thread*-Objekts dieses sich im *joinable*-Zustand befindet, wird vom Destruktor *std*::*terminate* aufgerufen.




- Prinzipiell gibt es nur zwei Zustände: *empty* und *joinable*.
- Mit dem Default-Konstruktor wird ein „leeres“ Objekt erzeugt; mit dem Konstruktor, der ein Funktionsobjekt erhält, wird unmittelbar ein Thread erzeugt, der dieses ausführt.
- Der Move-Konstruktor, das Move-Assignment und die *swap*-Operation sind allesamt äquivalent.

destructing-joinable-thread.cpp

```
int main() {
   {
      std::thread t1(Task(1));
      // t1 will now be destructed in the joinable state
   }
}
```

- *std*::*thread*-Objekte dürfen nicht abgebaut werden, wenn sie noch *joinable* sind. Andernfalls wird *std*::*terminate* aufgerufen:

```
clonmel$ destructing-joinable-thread
terminate called without an active exception
Abort (core dumped)
clonmel$
```

assigning-joinable-thread.cpp

```
int main() {
   {
      std::thread t1(Task(1)); // joinable state
      std::thread t2; // empty state
      t2 = std::thread(Task(2)); // move assignment
      std::thread t3; // empty state
      t3 = std::move(t2); // t3 now joinable, t2 empty
      /* reached */
      t1 = std::move(t3); // not permitted as t1 is joinable
      /* not reached */
   }
}
```

- Auf der linken Seite der Zuweisung darf kein *std*::*thread*-Objekt im Zustand *joinable* sein.

detached-thread.cpp

```
#include <chrono>
#include <iostream>
#include <thread>

class Task {
   public:
      Task(int id) : id(id) {};
      void operator()() {
         std::cout << "task " << id << " starts" << std::endl;
         std::this_thread::sleep_for(std::chrono::seconds(2));
         std::cout << "task " << id << " ends" << std::endl;
      }
   private:
      const int id;
};

int main() {
   {
      std::thread t1(Task(1));
      t1.detach();
   }
   std::this_thread::sleep_for(std::chrono::seconds(1));
   std::cout << "main exits" << std::endl;
}
```

- Ein *std*::*thread*-Objekt kann im Zustand *detached* abgebaut werden.
- Der C++11-Standard verrät nichts darüber, was passiert, wenn solche Threads noch weiterlaufen, wenn *main* endet.
- Da auf einem POSIX-System der Thread nicht länger als der Prozess laufen kann, terminiert dieser, wenn es explizit oder implizit zu *exit* kommt.
- Da nach dem Ende von *main* auch noch die statischen Objekte abgebaut werden, kann es zu undefinierten Effekten kommen, wenn der Thread auf diese noch zugreifen sollte.
- Somit ist *detach* normalerweise nicht empfehlenswert.

detached-thread.cpp

```
std::this_thread::sleep_for(std::chrono::seconds(2));
```

- *std::thread::this_thread* ist ein Namensraum mit einigen Funktionen, die sich implizit auf den aufrufenden Thread beziehen.
- Angebotene Funktionen:

| | |
|---|---|
| *thread*::*id get_id*() | liefert die ID des aktuellen Threads (ist implementierungsabhängig) |
| **void** *yield*() | signalisiert, dass andere ausführungsbereite Threads eher ausgeführt werden sollten |
| **void** *sleep_until*(...) | bis zu einem Zeitpunkt suspendieren |
| **void** *sleep_for*(...) | für die genannte Zeitperiode suspendieren |

fork-and-join2.cpp

```
// fork off some threads
std::thread threads[10];
for (int i = 0; i < 10; ++i) {
   threads[i] = std::thread(Task(i));
}
```

- Wenn Threads in Datenstrukturen unterzubringen sind (etwa Arrays oder beliebigen Containern), dann werden sie dort normalerweise im leeren Zustand konstruiert.
- Später kann dann mit Hilfe eines Move-Assignments ein Thread zugewiesen wird.
- Hier ist auf der rechten Seite ein temporäres Objekt mit dem Typ *std*::*thread*&&, das ohne weiteres Zutun zur Verwendung des Move-Assignments führt.
- Im Anschluss an die Zuweisung hat die linke Seite den Verweis auf den Thread, während die rechte Seite dann nur noch eine leere Hülle ist.

fork-and-join2.cpp

```
// and join them
std::cout << "Joining..." << std::endl;
for (int i = 0; i < 10; ++i) {
   threads[i].join();
}
```

- Das vereinfacht dann auch das Zusammenführen all der Threads mit der *join*-Methode.

simpson.cpp

```
double simpson(double (*f)(double), double a, double b,
      std::size_t n) {
   assert(n > 0 && a <= b);
   double value = f(a)/2 + f(b)/2;
   double xleft; double x = a;
   for (std::size_t i = 1; i < n; ++i) {
      xleft = x; x = a + i * (b - a) / n;
      value += f(x) + 2 * f((xleft + x)/2);
   }
   value += 2 * f((x + b)/2); value *= (b - a) / n / 3;
   return value;
}
```

- *simpson* setzt die Simpsonregel für das in $n$ gleichlange Teilintervalle aufgeteilte Intervall $[a, b]$ für die Funktion $f$ um:

$$S(f, a, b, n) = \frac{h}{3}\left(\frac{1}{2}f(x_0) + \sum_{k=1}^{n-1} f(x_k) + 2\sum_{k=1}^{n} f(\frac{x_{k-1} + x_k}{2}) + \frac{1}{2}f(x_n)\right)$$

mit $h = \frac{b-a}{n}$ und $x_k = a + k \cdot h$.

simpson.cpp

```
class SimpsonTask {
   public:
      SimpsonTask(double (*f)(double), double a, double b,
                  std::size_t n, double& rp) :
            f(f), a(a), b(b), n(n), rp(rp) {
      }
      void operator()() {
         rp = simpson(f, a, b, n);
      }
   private:
      double (*f)(double);
      double a, b;
      std::size_t n;
      double& rp;
};
```

- Jedem Objekt werden nicht nur die Parameter der *simpson*-Funktion übergeben, sondern auch noch einen Zeiger auf die Variable, wo das Ergebnis abzuspeichern ist.

simpson.cpp

```
double mt_simpson(double (*f)(double), double a, double b,
      std::size_t n, std::size_t nofthreads) {
   // divide the given interval into nofthreads partitions
   assert(n > 0 && a <= b && nofthreads > 0);
   std::size_t nofintervals = n / nofthreads;
   std::size_t remainder = n % nofthreads;
   std::size_t interval = 0;

   std::vector<std::thread> threads(nofthreads);
   std::vector<double> results(nofthreads);

   // fork & join & collect results ...
}
```

- *mt_simpson* ist wie die Funktion *simpson* aufzurufen – nur ein Parameter *nofthreads* ist hinzugekommen, der die Zahl der zur Berechnung zu verwendenden Threads spezifiziert.
- Dann muss die Gesamtaufgabe entsprechend in Teilaufgaben zerlegt werden.

simpson.cpp

```
std::vector<std::thread> threads(nofthreads);
std::vector<double> results(nofthreads);
```

- Variabel lange Arrays auf dem Stack sind in C++ nicht zulässig.
- C++ ist hier restriktiver als C99 bzw. C11 und dies wird sich auch nicht ändern, da dies inzwischen als eine Fehlentwicklung angesehen wird.
- *g++* lässt es zwar zu – dennoch sollte es vermieden werden.
- Stattdessen empfiehlt sich die Verwendung von *std::vector* aus *<vector>*. Beim Konstruktor kann die gewünschte Dimensionierung angegeben werden (in runden Klammern!).
- Die im Vektor enthaltenen Objekte leben dann auf dem Heap.

simpson.cpp

```
double x = a;
for (std::size_t i = 0; i < nofthreads; ++i) {
   std::size_t intervals = nofintervals;
   if (i < remainder) ++intervals;
   interval += intervals;
   double xleft = x; x = a + interval * (b - a) / n;
   threads[i] = std::thread(SimpsonTask(f,
      xleft, x, intervals, results[i]));
}
```

- Für jedes Teilproblem wird ein entsprechendes Funktionsobjekt temporär angelegt, an *std*::*thread* übergeben, womit ein Thread erzeugt wird und schließlich an *threads*[*i*] mit der Verlagerungs-Semantik zugewiesen.
- Hierbei wird auch implizit der Verlagerungs- oder Kopierkonstruktor von *SimpsonThread* verwendet.

simpson.cpp

```
double sum = 0;
for (std::size_t i = 0; i < nofthreads; ++i) {
   threads[i].join();
   sum += results[i];
}
return sum;
```

- Wie gehabt erfolgt die Synchronisierung mit der *join*-Methode.
- Danach kann das entsprechende Ergebnis abgeholt und aggregiert werden.

simpson.cpp

```
cmdname = *argv++; --argc;
if (argc > 0) {
   std::istringstream arg(*argv++); --argc;
   if (!(arg >> N) || N <= 0) usage();
}
if (argc > 0) {
   std::istringstream arg(*argv++); --argc;
   if (!(arg >> nofthreads) || nofthreads <= 0) usage();
}
if (argc > 0) usage();
```

- Es ist sinnvoll, die Zahl der zu startenden Threads als Kommandozeilenargument (oder alternativ über eine Umgebungsvariable) zu übergeben, da dieser Parameter von den gegebenen Rahmenbedingungen abhängt (Wahl der Maschine, zumutbare Belastung).
- Zeichenketten können in C++ wie Dateien ausgelesen werden, wenn ein Objekt des Typs *std*::*istringstream* damit initialisiert wird.
- Das Einlesen erfolgt in C++ mit dem überladenen >>-Operator, der als linken Operanden einen Stream erwartet und als rechten eine Variable.

simpson.cpp

```
double sum = mt_simpson(f, a, b, N, nofthreads);
std::cout << std::setprecision(14) << sum << std::endl;
std::cout << std::setprecision(14) << M_PI << std::endl;
```

- Testen Sie Ihr Programm zuerst immer ohne Threads, indem die Funktion zur Lösung eines Teilproblems verwendet wird, um das Gesamtproblem unparallelisiert zu lösen. (Diese Variante ist hier auskommentiert.)
- *std::cout* ist in C++ die Standardausgabe, die mit dem <<-Operator auszugebende Werte erhält.
- *std*::*setprecision*(14) setzt die Zahl der auszugebenden Stellen auf 14. *endl* repräsentiert einen Zeilentrenner.
- Zum Vergleich wird hier *M_PI* ausgegeben, weil zum Testen $f(x) = \frac{4}{1+x^2}$ verwendet wurde, wofür $\int_0^1 f(x)dx = 4 \cdot \arctan(1) = \pi$ gilt.

simpson2.cpp

```
threads[i] = std::thread([=,&results]() {
   results[i] = simpson(f, xleft, x, intervals);
});
```

- Die Lösung vereinfacht sich, wenn die Klasse *SimpsonTask* durch einen Lambda-Ausdruck ersetzt wird.
- Die *capture*, d.h. der Teil in den eckigen Klammern zu Beginn des Lambda-Ausdrucks ([=,&*results*]) spezifiziert, dass per Voreinstellung alle aus der Umgebung benötigten lokalen Variablen kopiert werden, *results* jedoch als Referenz bezogen wird.
- Lambda-Ausdrücke führen zum Erzeugen einer entsprechenden anonymen Klasse, wobei die *capture* festgelegt wird, was wie in ein zu konstruierendes Objekt übernommen wird.