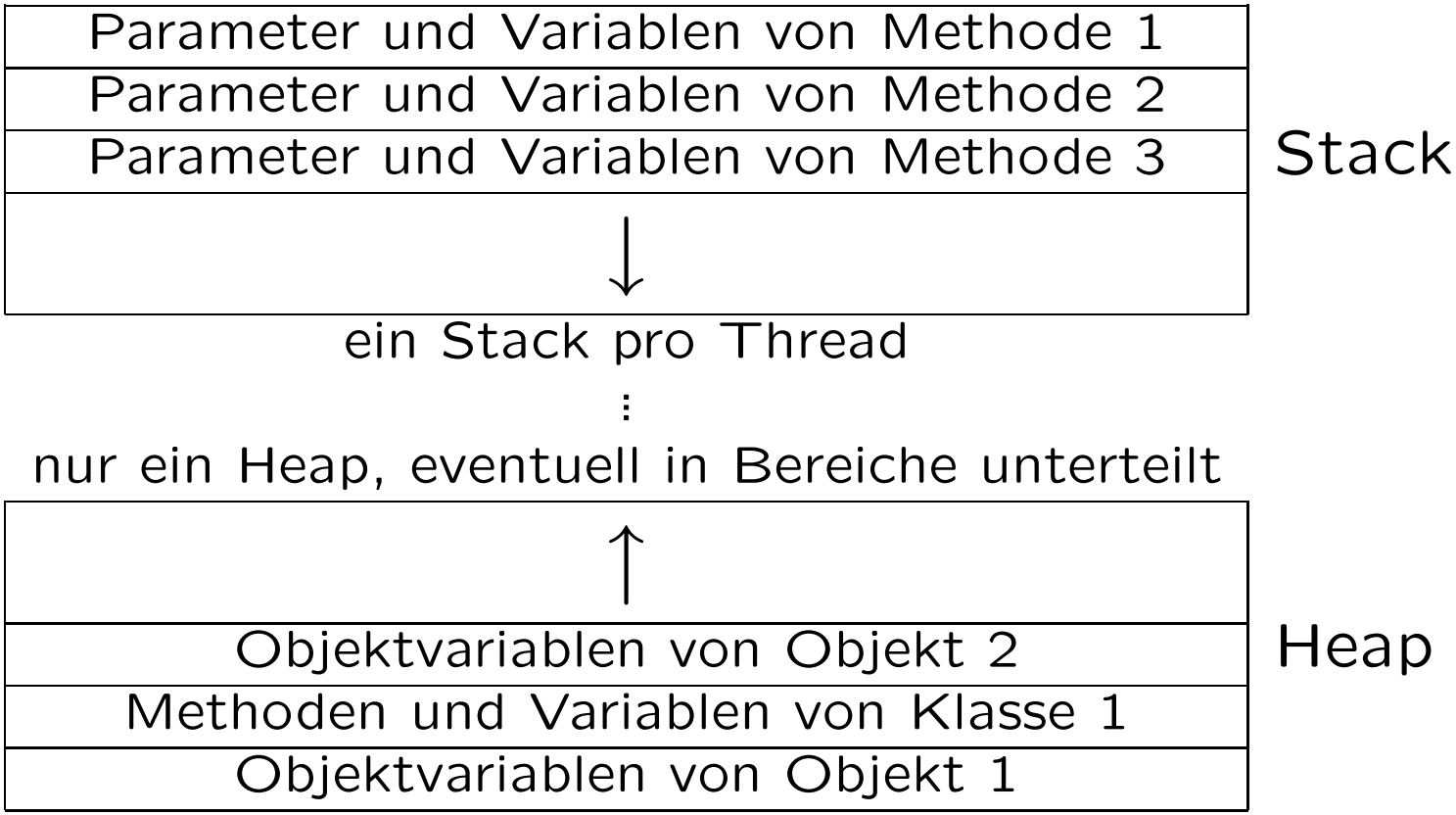
Speicherbereiche (Java & Co)



Verwaltung der Speicherbereiche

Stack:

- Speicherallokation bei Methodenaufruf
- Speicherfreigabe bei Rückkehr aus Methode
- Speicher bleibt durch Stack-Struktur stets kompakt

Heap:

- Speicherallokation bei Objekterzeugung
  (und beim Laden von Klassen)
- Speicherfreigabe durch *Garbage-Collection*
- Schließen freier Löcher bei *Speicherkompaktierung*

Kosten der Objekterzeugung

- Speicherallokation
- Initialisierung
- Garbage-Collection oder explizite Speicherverwaltung
- Finalisierung
- Speicherdeallokation
- Speicherfragmentierung

? wie Objekterzeugung bzw. deren Kosten vermeiden?

Escape-Analyse

- Compiler (auch JIT) analysiert Objektverwendung
- wenn Objekt in lokalem Scope bleibt (Non-Escape)
  - Objektvariablen am Stack bzw. in anderem Objekt
  - nur Initialisierungskosten, weitere Optimierungen
- sonst wenn Objekt nur innerhalb von Thread
  - Objekt am Heap notwendig
  - aber kein Locking nötig (und kein Speicher dafür)
- sonst Global-Escape, volles Objekt nötig
- z.B. von neuerer Java-HotSpot-VM verwendet

Wert- statt Referenztypen

- Struct in C# hat Wert-, nicht Referenzsemantik
- Werte bei Parameterübergabe vollständig kopiert
- ohne `new` am Stack oder als Teil von Objekt erzeugbar
- ähneln Non-Escape-Objekten wenn lokal
- vorteilhaft wenn klein oder nur lokal verwendet
- ebenso: C++-Struct (nicht als Referenz)

Freispeicherliste statt Objekterzeugung

- eigene Liste nicht benötigter Objekte pro Klasse/Typ
- Speicherallokation oft aufwendiger (Liste oder Heap)
- Initialisierung trotzdem nötig
- höhere Kosten für Speicherverwaltung
- Deallokation = Einhängen in Freispeicherliste
- Vorteile bezüglich Speicherfragmentierung

$\rightarrow$ bei expliziter Speicherverwaltung manchmal vorteilhaft

$\rightarrow$ bei Garbage-Collection praktisch nie vorteilhaft

Garbage-Collection (GC)

- GC gibt Speicher für nicht mehr zugreifbare Objekte frei, sorgt oft auch für Speicherkompaktierung
- Freigabe verzögert (je nach Speicherbedarf)
- keine Freigabe wenn Objekt noch zugreifbar
  (auch wenn kein Zugriff mehr erwartet wird)
- nicht mehr benötigte Variablen auf `null` setzen: `x = null;`
- Auf-`null`-setzen sinnvoll wenn
  - Variable möglicherweise noch länger gültig ist
  - und darauf sicher kein Zugriff mehr erfolgen wird

Warum Garbage-Collection?

- Effizient und vereinfacht Programmierung wesentlich
- Gute Denkansätze: Objekte leben konzeptuell ewig
  unnötige Referenzen beseitigen
  Ressourcen von Speicher entkoppeln
- Ansätze für Speicherverwaltung funktionieren bei GC kaum (Kontrolle über Speicher, Destruktoren räumen auf)
- Keine Garantie für genug Speicher, daher manchmal trotz Echtzeitfähigkeit wegen dyn. Speicherverwalt. ungeeignet
- Programme sollen auch bei ausgeschalteter GC laufen

Probleme bei Garbage-Collection

- gefühlt immer zum falschen Zeitpunkt
  (effizient, aber manchmal merkbar längere Antwortzeiten)
- Speicherverwaltung machtlos gegen schlechte Algorithmen
- automatische Speicherverwaltung nicht überall sinnvoll
  (z.B. sicherheitskritische Systeme)
- Auf-`null`-setzen kann aufwendig sein

Garbage-Collection-Algorithmen

- *Mark-and-Sweep:* zugreifbare Objekte markiert, danach Rest freigegeben $\rightarrow$ Fragmentierung
- *Mark-and-Compact:* markierte Objekte in sicheren Bereich kopiert, großer freier Bereich bleibt übrig
- *Generational:* mehrere Techniken je nach Bereich (= Alter der Objekte), junge Objekte öfter behandelt als alte
- in C++ *konservativ:* jedes Bitmuster, das Referenz sein könnte, wird als Referenz interpretiert $\rightarrow$ unzuverlässig, wenig benutzt

Eingriffe in Speicherverwaltung

- `StackOverflowError` → `java -Xssn1m Prog` → meist erfolglos
- Heapgröße ändern: `java -Xmsn6m -Xmxn66m Prog` (sinnvoll?)
- Garbage-Collection explizit aufrufen:

```
Runtime r = Runtime.getRuntime();
r.gc();
```

- Parameter der Gargabe-Collection einstellen (kompliziert)
- vor Speicherfreigabe wird `finalize()` ausgeführt
  (verzögert Speicherfreigabe, kaum verwendet)

Sichtbarkeit und Sicherheit

- Sichtbarkeit auf Klassenebene (Java, C#, C++, . . .)
  $\rightarrow$ Konzentration auf Verantwortlichen für Klasse
- Sichtbarkeit auf Objektebene (Variablen in Eiffel)
  $\rightarrow$ Konzentration auf Zusicherungen und Ersetzbarkeit
- Sichtbarkeit als Sicherheitskonzept (theoretisch)
  $\rightarrow$ Objekte nur in einem bestimmten Bereich sichtbar
  - Uniqueness (nur eine Referenz auf Objekt)
  - Ownership (nur von einem Objekt referenziert)
  - Confinement (nur innerhalb einer Zone referenziert)

C++ Templates

- auch Konstanten als Typparameter:

```
template<typename T, int n> class buffer { ... };
```

- partielle/volle Spezialisierung mit „Pattern-Matching“

```
template<int n> class buffer<bool, n> { ... };

template<> class buffer<bool, 0> { ... };
```

- implizite Instanzierung wenn keine passende Spezialisierung
- Methoden nur erzeugt wenn im Code verwendet

Ausdrucksstärke von Templates

- statische Auswertung von Ausdrücken verwendbar:

```
template<int x, int n> struct power {
  static const int r = x * power<x, n-1>::r;
};
template<int x> struct power<x, 0> {
  static const int r = 1;
};
```

- auch algebraische Datentypen ausdrückbar:

```
template<class Head, class Tail> struct Cons { };
struct Nil { };
typedef Cons<int, Cons<float, Nil> > list_of_types;
```

Policy-Based-Programming

```
#include <iostream>
template<typename lang> class HelloWorld : public lang {
   public: void Run() { cout << Message() << endl; }
};
#include <string>
class HelloWorld_Msg_German {
   protected: string Message() { return "Hallo Welt!"; }
};
typedef HelloWorld<HelloWorld_Msg_German> MyHelloWorld;
MyHelloWorld hello;
hello.Run();
```

Beispiel für Strategy (Design-Pattern)

```
interface IMessage { String message(); }
class DMessage implements IMessage {
    public String message() {return "Hallo Welt!";}
}
class HelloWorld {
    private IMessage msg;
    public HelloWorld(IMessage msg) {this.msg = msg;}
    public void run() {System.out.println(msg.message());}
}
class Main {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld(new DMessage());
        hello.run();
    }
}
```