universität innsbruck

Java Virtual Machine

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Übersetzung – Interpretierung (Wiederholung)

- Übersetzung (Kompilierung)
  - Der Quellcode wird in einen Zielcode (z.B. Maschinencode) übersetzt.
  - Laufzeitumgebung führt diesen Code aus.
  - Vorteile
    - Effizienz (sowohl Laufzeit als auch Speicher)
  - Nachteile
    - Nicht portierbar (muss neu kompiliert werden)
- Interpretierung
  - Interpreter liest jede Instruktion.
  - Verifiziert den Code.
  - Gibt den korrekten Code an die Laufzeitumgebung weiter.
  - Vorteile
    - Portierbarkeit
  - Nachteile
    - Effizienz (langsam)

Programmierumgebung

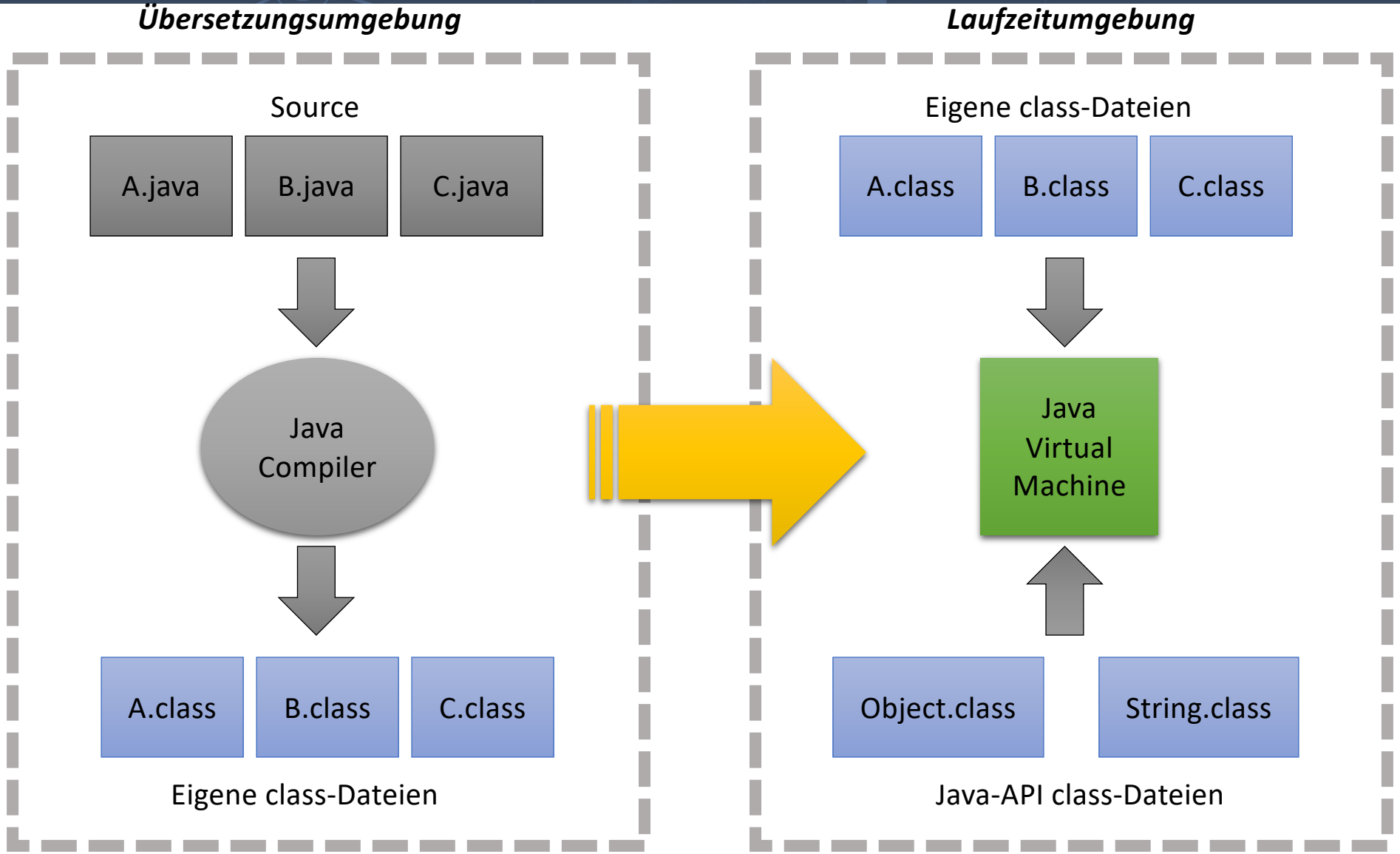


Abbildung entnommen aus https://www.artima.com/insidejvm/ed2/introarchP.html

Bytecode

- Zwischencode der von der JVM interpretiert wird.
- Ein beliebiger dem Standard entsprechender Java-Compiler muss gültigen Bytecode erzeugen (`.class`-Dateien).
- Inhalt
  - Informationen über
    - Klassenhierarchien
    - Methodensignaturen
    - Code von Methoden
    - etc.
  - Referenzen auf andere Klassen
    - Referenzen werden beim Ladeprozess aufgelöst.
- `.class`-Datei
  - Binärdatei
  - Immer big-endian (unabhängig von der darunterliegenden Architektur; most significant bit in niedrigster Adresse)
  - Kompakt (z.B. für Netzwerkübertragung)

Struktur einer .class-Datei

Magic Number
Version
Constant Pool
Access Flags
this class
super class
Interfaces
Fields
Methods
Attributes

Virtuelle Maschine

- Die virtuelle Maschine unterstützt
  - Plattformunabhängigkeit
  - Sicherheit
  - Mobilität (im Netzwerk)
- Die JVM ist eine abstrakte Maschine.
  - Spezifikation gibt einige Teile vor (z.B. JVM muss Bytecode interpretieren können).
  - Implementierung wird aber offen gelassen.
- Hauptaufgaben
  - Klassendateien laden
  - Bytecode in diesen Klassendateien ausführen

Programmausführung

- Reines Interpretieren des Bytecodes (in Software)
- Just-in-time Übersetzung (JIT)
  - Bytecode wird in Maschinencode übersetzt (beim ersten Aufruf einer Methode).
  - Erzeugter Code kommt in einen Cache, um später darauf zuzugreifen (benötigt mehr Speicher!).
- Adaptive Optimizer (dynamische Optimierung)
  - Interpretierung
  - Oft benutzte Teile werden mit der Zeit (Monitoring) in Maschinencode übersetzt (Variable → Konstante).
- JVM in Hardware

JVM in Software (1)

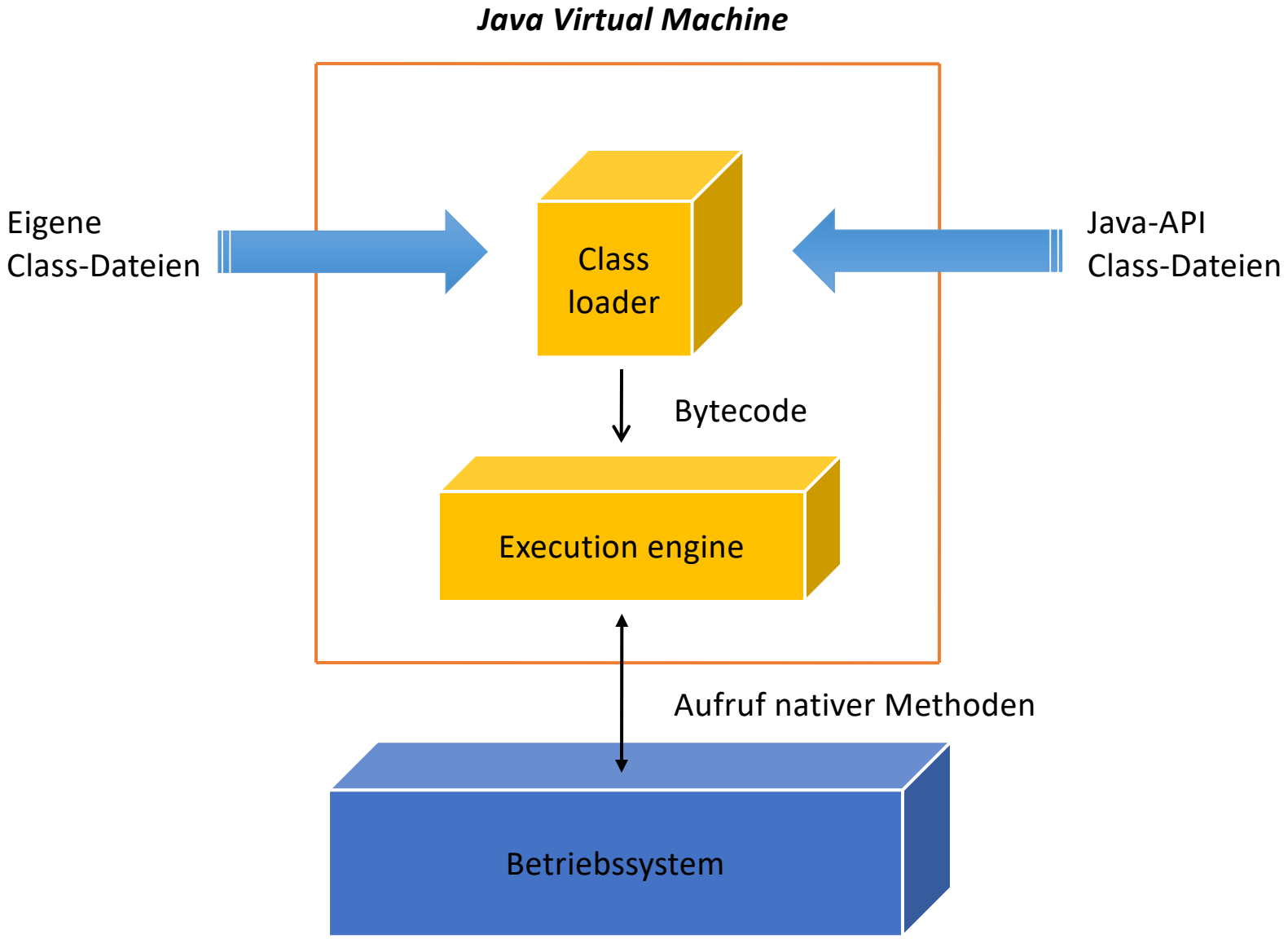


Abbildung entnommen aus https://www.artima.com/insidejvm/ed2/introarchP.html

JVM in Software (2)

- Java-Programm interagiert mit dem Betriebssystem über native Methoden.
- Java hat zwei Arten von Methoden
  - Java-Methoden (z.B. im eigenen Java-Programm implementiert)
  - Native Methoden (in C, C++, Assembler und entsprechend übersetzt)
- Java API stellt unter anderem Methoden bereit, die auf bestimmte Ressourcen des darunterliegenden Systems zugreifen.
  - Diese Methoden rufen ihrerseits native Methoden auf.
  - Von der Java API verwendete native Methoden werden für jede Plattform bereitgestellt.
- Eigene native Methoden
  - JNI (Java Native Interface)
  - Damit ist das entsprechende Programm nicht mehr plattformunabhängig!

JVM – Struktur

Virtuelle Maschine

- Spezifikation beschreibt das Verhalten anhand von
  - Subsystemen
  - Speicherbereichen
  - Datentypen
  - Instruktionen
- Abstrakte Beschreibung
  - Beschreibt das Verhalten der JVM.
  - Beschreibt nicht die genaue innere Architektur!
  - Implementierung hängt vom Design der JVM ab.

Virtuelle Maschine (Architektur)

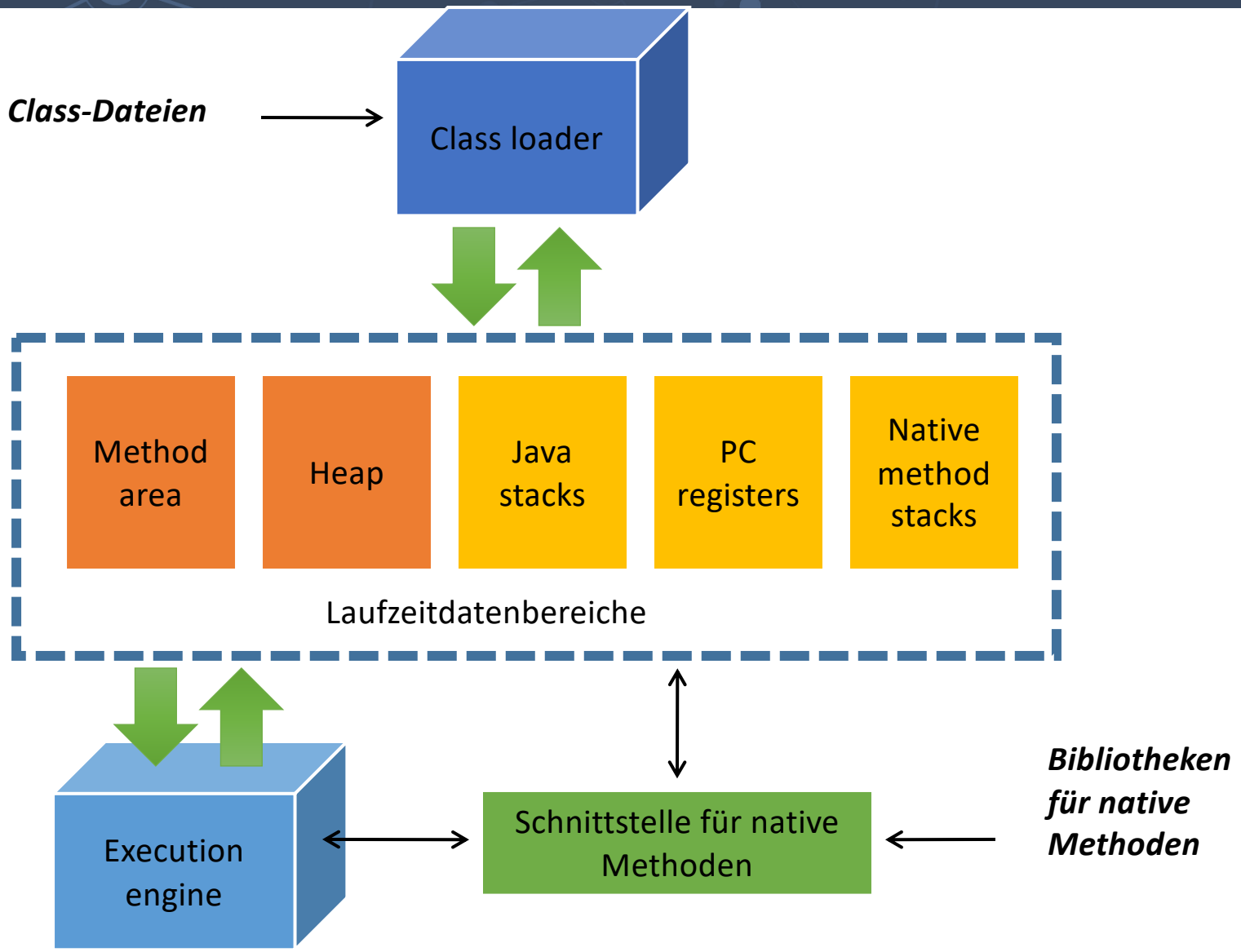


Abbildung entnommen aus https://www.artima.com/insidejvm/ed2/jvmP.html

Laden von Klassen (Class loading) in der JVM

Loading

- Lokalisierung der binären Repräsentation eines „Typs“ (Klasse, Interface)
- Laden der Daten

Linking

- Aufnahme in die aktuelle Verwaltung der JVM
  - Verifikation des Codes
  - Vorbereitung
  - Auflösung (von symbolischen Referenzen)

Initialization

- Aufruf des Initialisierers

Wann wird eine Klasse geladen?

- Nicht festgelegt
  - Flexible JVM-Spezifikation
    - Loading vor Linking
    - Linking vor Initialization
    - Initialization vor erster Verwendung („active use“)
  - „Active use“
    - `new`-Anweisung, Aufruf einer statischen Methode, Initialisierung einer Subklasse, Aufruf der `main`-Methode usw.
- Normalerweise wird das Laden möglichst lange hinausgezögert!
  - Ressourcen werden erst dann belegt, wenn sie wirklich benötigt werden.
  - Ist aber keine Anforderung, d.h. Laden kann auch zu einem früheren Zeitpunkt erfolgen!

Klassenlader

- Damit der Bytecode von der Ausführungseinheit (execution engine) verarbeitet werden kann, muss ein Klassenlader (class loader) die benötigte Klasse laden.
- Alle Referenztypen (außer Arrays) werden entweder
  - durch den **Bootstrap-Klassenlader** oder
  - durch einen **benutzerdefinierten Klassenlader** geladen.
- Bootstrap-Klassenlader ist Teil der JVM.
- Benutzerdefinierte Lader („normale“ Java-Klassen)
  - Werden von `java.lang.ClassLoader` abgeleitet.
  - Können bestimmte Zusatzaufgaben übernehmen.

Method Area

- Die JVM extrahiert Informationen und speichert diese Informationen über den Typ in der sogenannten **Method Area**.
- Implementierung
  - Hängt vom JVM Design ab (Datenstruktur, Größe, Verwaltung …).
- Für jeden Typ
  - Basisinformationen wie Name (voll qualifiziert), Name der Superklasse, Unterscheidung ob Klasse oder Interface, Zugriffsmodifikatoren, Liste von Interfaces (voll qualifiziert) …
  - Zusätzlich: Constant Pool, Variablen- und Methodeninformationen, alle statische Variablen (nicht Konstanten), Referenz auf den Klassenlader, Referenz auf Klasse `Class`
  - Methodentabelle (Dispatch-Tabelle)

Heap

- Innerhalb einer JVM gibt es einen Heap.
  - Alle Threads eines Programms teilen sich diesen.
  - Jedes Programm läuft aber in einer eigenen JVM-Instanz!
- Bei der Objekterzeugung (auch Array) wird Speicher vom Heap angefordert
  - Die JVM hat eine Instruktion um Speicher anzufordern.
  - Es existiert aber keine Instruktion für die Freigabe.
  - Speicherfreigabe erfolgt durch den sogenannten Garbage-Collector.

Objektrepräsentation (1)

- Objektrepräsentation ist nicht festgelegt.
- Beispiel: Monolithische Repräsentation

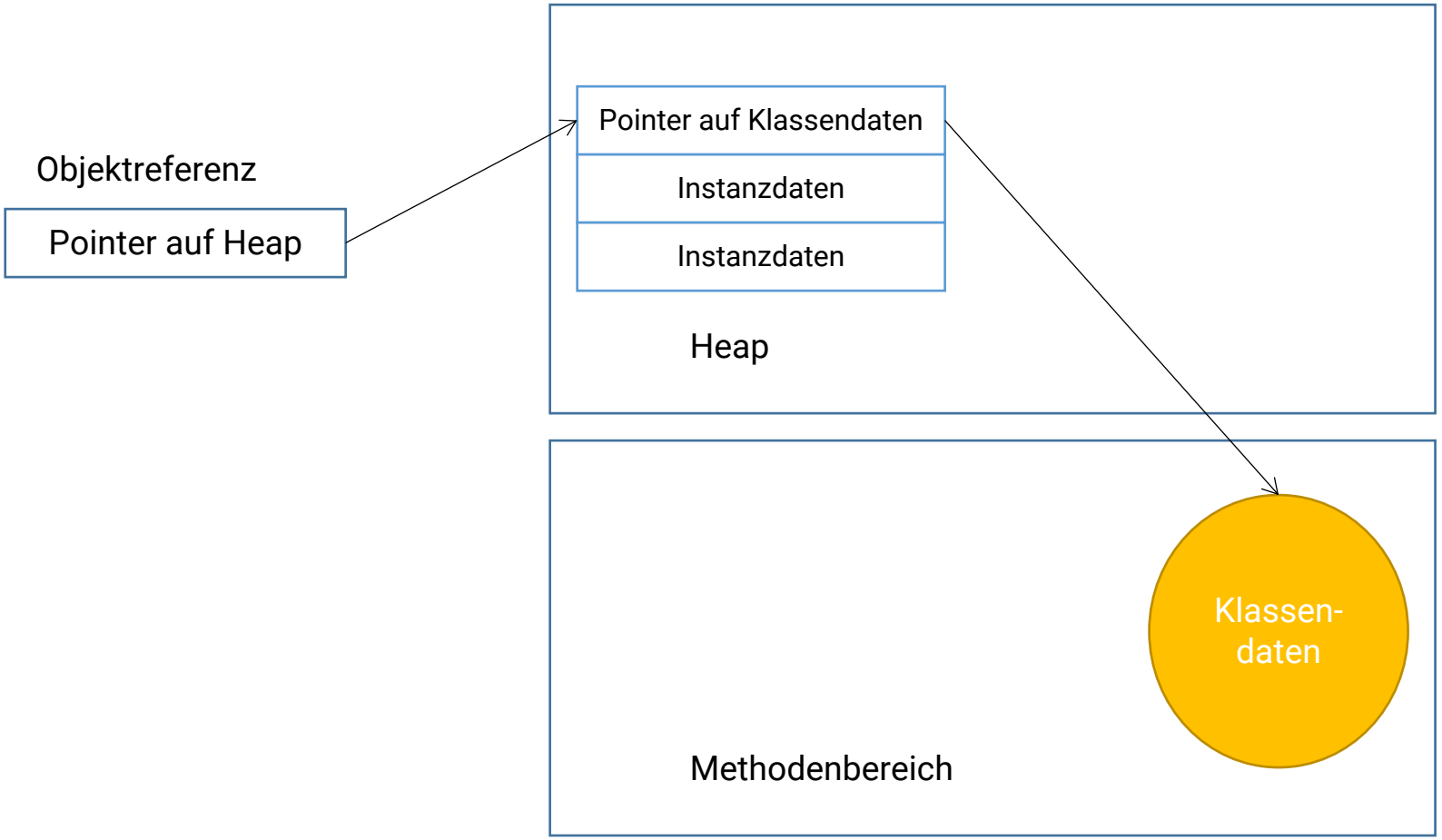


Abbildung entnommen aus https://www.artima.com/insidejvm/ed2/jvmP.html

Objektrepräsentation (2)

- Beispiel: Geteilte Repräsentation

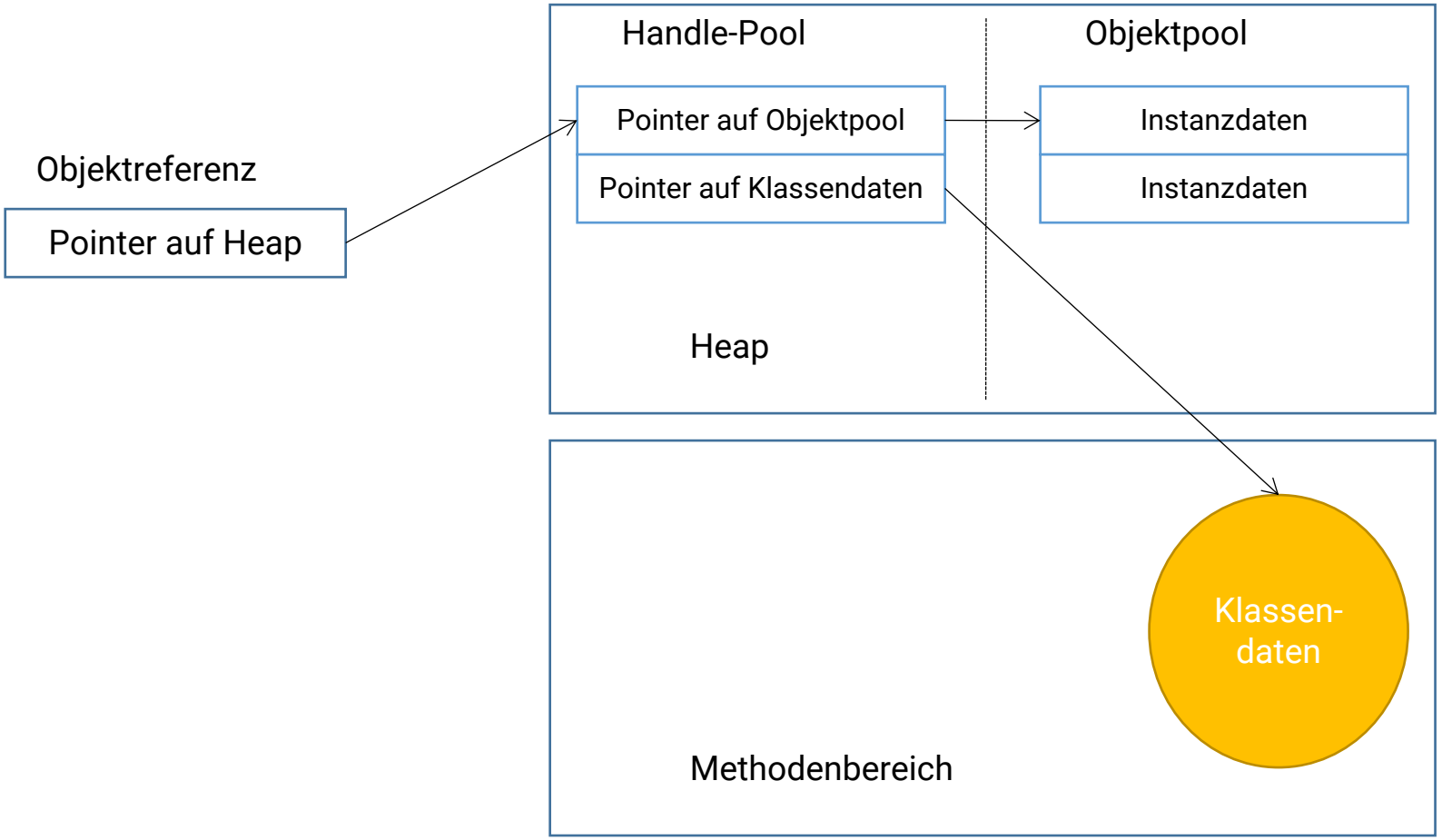


Abbildung entnommen aus https://www.artima.com/insidejvm/ed2/jvmP.html

Vererbung

- Code für Methoden wird bei der Vererbung nicht dupliziert!
  - Beispiel: Klasse C1 und C2 teilen sich den Code von `m1()`

```
class C1 {
    protected int x;
    public void m1() {…}
}
```

```
class C2 extends C1 {
    protected int y;
    public void m2() {…}
}
```

```
C1 o1 = new C1();
C2 o2 = new C2();
o2.m1();

o2.x = 10;
o2.y = 10;
```

Code von C1

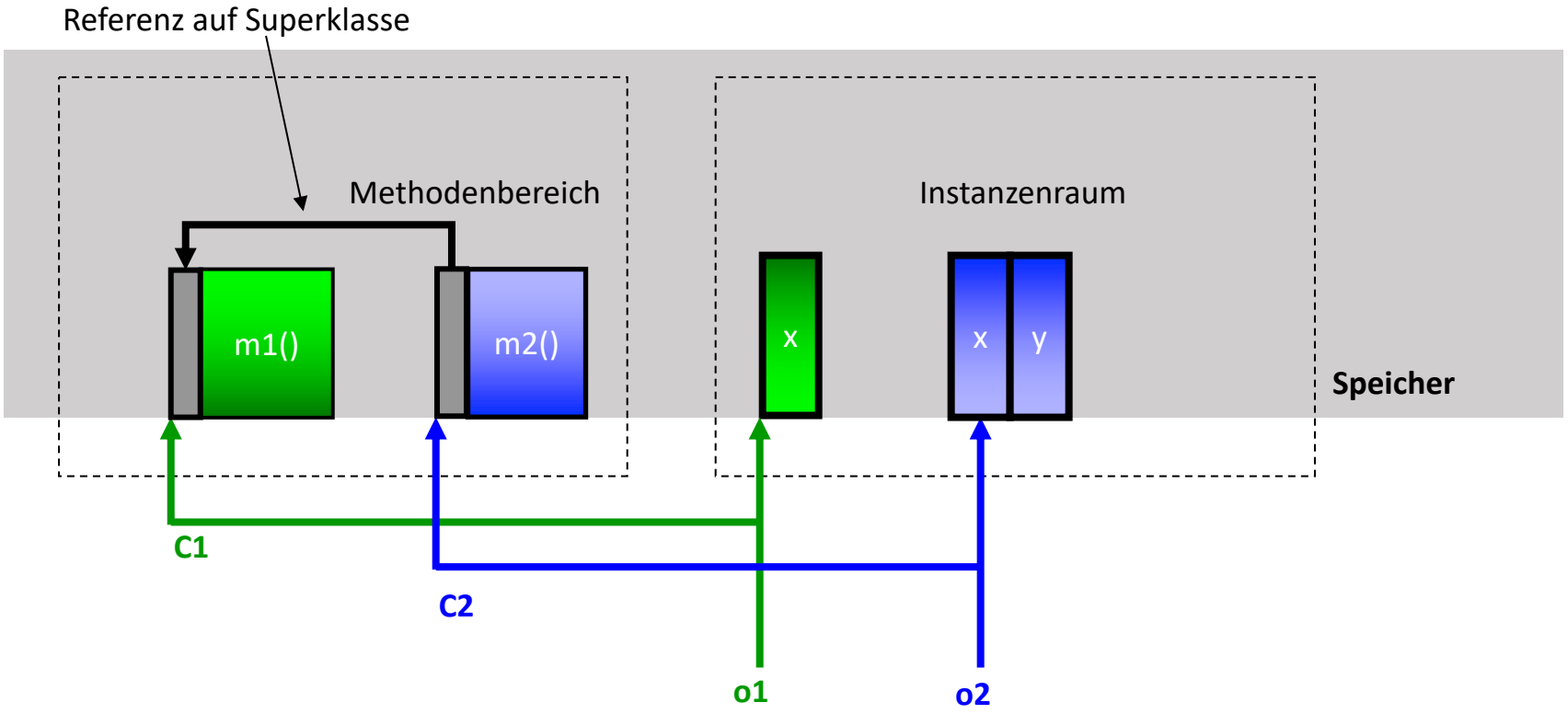
C2 hat ein eigenes lokales x!

Vererbung (Implementierung)

```
class C1 {
    protected int x;
    public void m1() {…}
}
```

```
class C2 extends C1 {
    protected int y;
    public void m2() {…}
}
```

```
C1 o1 = new C1();
C2 o2 = new C2();
```



Dynamisches Binden (1)

```
class C1 {
    protected int x;
    public void m1() {…}
    public boolean isNull() {…}
}
```

```
class C2 extends C1 {
    protected int y;
    public void m2() {…}
    public boolean isNull() {…}
}
```

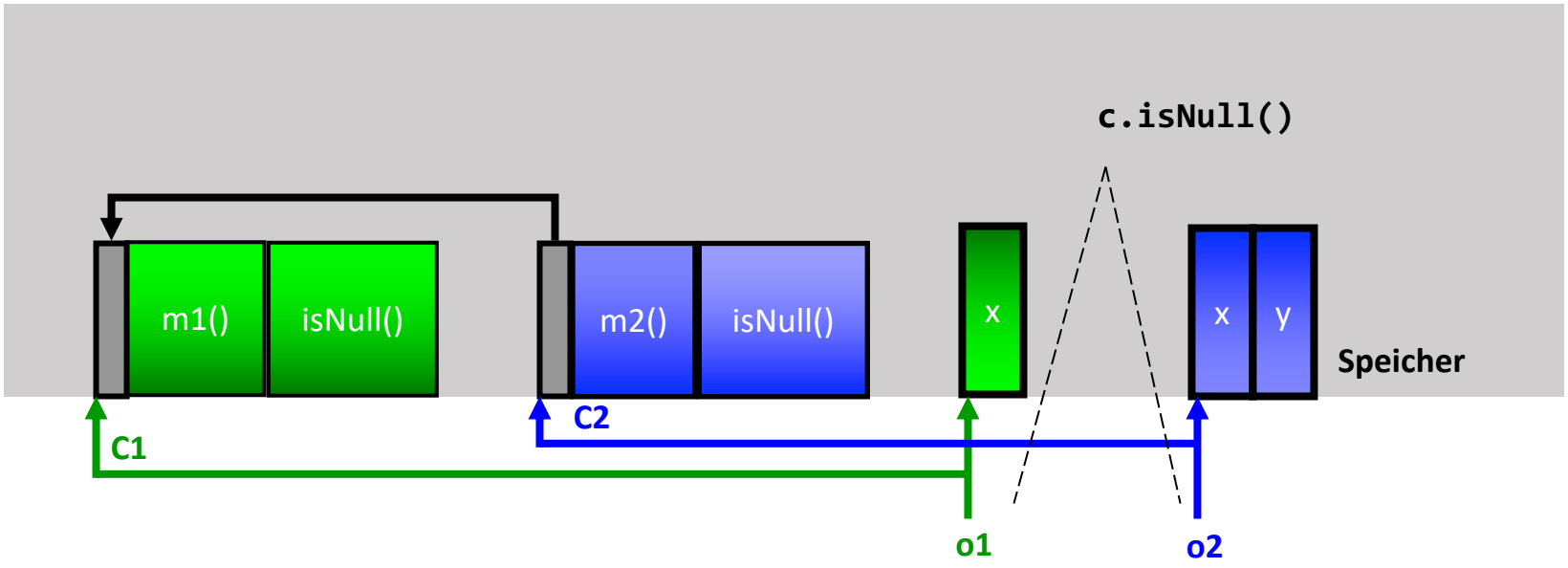
```
C1 o1 = new C1();
C2 o2 = new C2();
…
C1 c = (…) ? o1 : o2;
boolean b = c.isNull();
```

Dynamisches Binden!

Dynamisches Binden (2)

- Ziel: Aufruf mit Methode verknüpfen.
- Wird die Methode nicht in der Methodentabelle der eigenen Klasse gefunden wird beim super-Typ weiter gesucht.

```
C1 c = (…) ? o1 : o2;
boolean b = c.isNull();
```



Garbage-Collection

Speicherplatzfreigabe

- In C
  - `free()` gibt belegten Speicherplatz frei.
  - Fehleranfällig!
- In C++
  - Konstruktor/Destruktor (Erzeugung, Freigabe)
  - Auch fehleranfällig!
- In einem objektorientierten System sollten Objekte verfügbar sein, solange sie benutzt werden.
- Die Laufzeitumgebung sollte nicht mehr benutzte Objekte entfernen (und damit den Speicherplatz freigeben).

Garbage

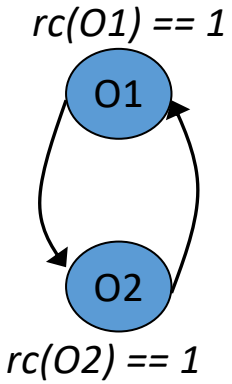
- Garbage-Collector
  - Ist ein spezieller Prozess.
  - Sammelt periodisch nicht benutzte Objekte ein und gibt den Speicher frei.
  - Erweiterungen werden noch besprochen.
- JVM
  - Art der Garbage-Collection ist nicht explizit vorgegeben.
  - Es muss nicht einmal ein Garbage-Collector vorhanden sein.

Garbage-Collection allgemein

- Zwei Gründe für Garbage-Collection
  - Speicher freigeben
    - JVM kann diese Aufgabe übernehmen (automatisieren).
  - Fragmentierung des Heaps unterbinden bzw. verringern.
    - Für eine neue Allokation wird zusammenhängender Speicher benötigt.
    - Freigabe mehrerer Objekte muss nicht zusammenhängenden freien Speicher freigeben.
- Nachteil
  - Zusätzlicher Speicheraufwand
  - CPU-Aufwand wenn der Garbage-Collector anläuft
    - Längere Laufzeit
- Unterschiedliche Algorithmen
  - Basierend auf Referenzzähler
  - Tracing-Algorithmen (basierend auf Erreichbarkeitsgraphen)

Algorithmus 1: Reference Counting

- Jedes Objekt hat einen Zähler rc.
  - Bei der Erzeugung: rc=1
  - Kommt eine neue Referenz hinzu: rc++
  - Wird eine Referenz entfernt: rc --
  - Wenn rc==0 dann kann der Garbage-Collector das Objekt einsammeln.
- Vorteile
  - Verteilung der Last (aber viel Last!)
  - Gute Lokalität, schnelle Reaktion
- Nachteile
  - Zusätzlicher Speicherbedarf
  - Zyklen werden nicht erkannt!
    - O1 und O2 werden nicht mehr von P (im Programm) referenziert.
    - Referenzieren sich aber gegenseitig, d.h. rc==1.



Algorithmus 2: Mark-and-Sweep

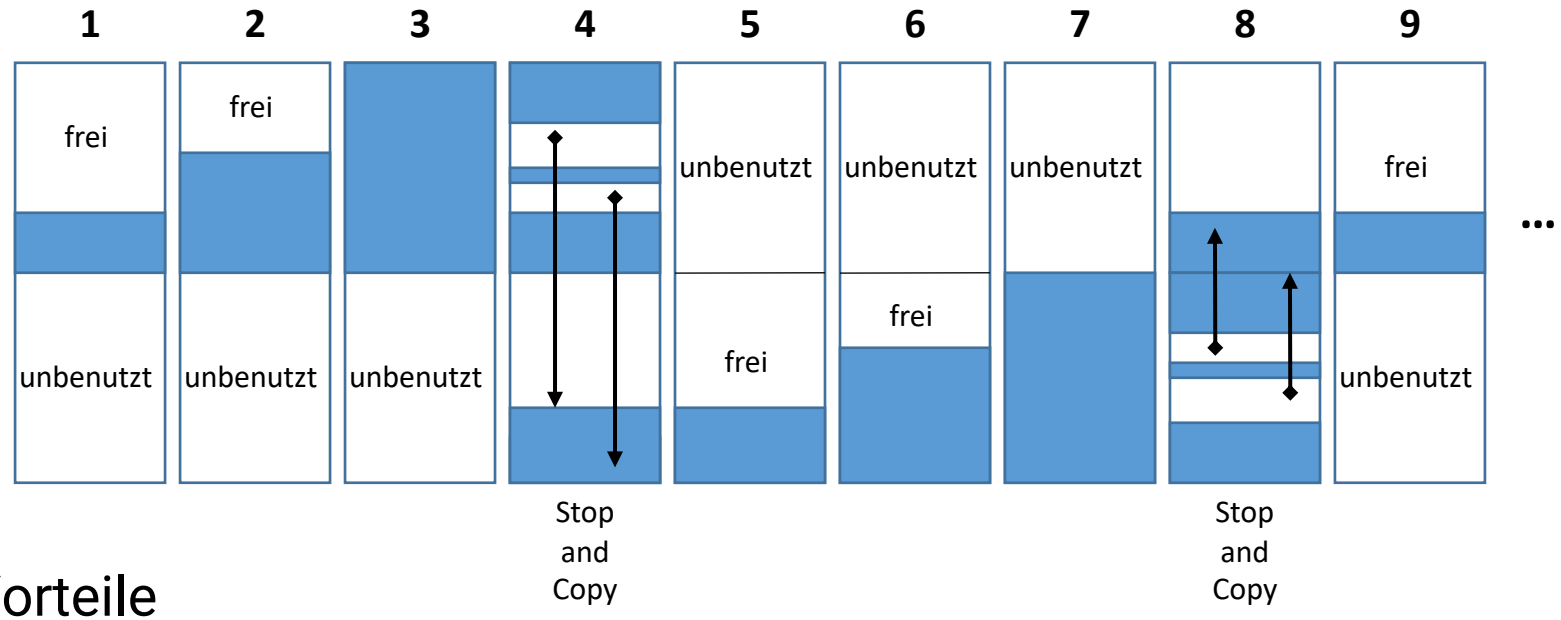
- Definiert Wurzel und überprüft Erreichbarkeit.
  - Wurzel (root set) = Objekte die immer existieren
  - Objekte, die man von der Wurzel aus nicht erreichen kann, werden entfernt.
- Mark and Sweep
  - Graph von der Wurzel aus aufbauen.
  - Alle erreichten Objekte markieren.
  - Entferne nicht markierte Objekte (linear im Speicher).
  - Setze alle markierten Objekte auf nicht-markiert.
- Vorteil
  - Kein Problem mit Zyklen
- Nachteile
  - Fragmentierung
  - Aufwand proportional zur Heap-Größe

Algorithmus 3: Stop-and-Copy (1)

- Heap wird in 2 gleich große Bereiche unterteilt.
  - `fromspace` und `tospace`
- Bei einem Durchlauf werden die Rollen der Bereiche vertauscht.
- Bei einem Durchlauf werden die aktuellen Teile vom alten Bereich (`fromspace`) in den neuen Bereich (`tospace`) kopiert – Programmvariablen benutzen den neuen Bereich.
- Garbage im `fromspace` wird einfach entfernt.
- Im `tospace` wird beim Kopieren automatisch Fragmentierung unterbunden.

Algorithmus 3: Stop-and-Copy (2)

- Beispiel



- Vorteile
  - Allokation schnell und billig
  - Keine Fragmentierung
- Nachteile
  - 50% des Heaps nicht benutzt
  - Langlebige Objekte werden immer wieder kopiert.

Abbildung entnommen aus https://www.artima.com/insidejvm/ed2/gcP.html

Generational Collectors (1)

- Ein Problem (neben der nicht effizienten Auslastung des Speichers) von Stop-and-Copy ist, dass jedes erreichbare Objekt immer wieder kopiert werden muss.
- Dies kann verbessert werden, wenn folgende empirischen Erkenntnisse berücksichtigt werden:
  - Die meisten Objekte haben eine sehr kurze Lebenszeit und werden sehr bald nach der Erzeugung nicht mehr benötigt (z.B. Iteratoren).
  - Einige Objekte haben eine sehr lange Lebenszeit und sind der Hauptgrund für die Ineffizienz von Stop-and-Copy (werden immer wieder kopiert).

Generational Collectors (2)

- Vorgehensweise (allgemein)
  - Heap wird in 2 oder mehrere Sub-Heaps unterteilt.
  - Jeder Sub-Heap verwaltet eine „Generation“ von Objekten.
  - Die jüngste Generation wird am öftesten überprüft.
  - Die meisten Objekte werden sehr schnell freigegeben und können eingesammelt werden.
  - Übersteht ein Objekt eine bestimmte Anzahl von Garbage-Collection-Durchläufen, dann wird es in den nächsten Heap verschoben.
  - Je älter die Generation in einem Heap, desto seltener wird dieser Heap überprüft.
- Kombination mit herkömmlichen Methoden
  - Mark-and-Sweep
  - Stop-and-Copy

Anmerkungen (1)

- Garbage-Collection kann sehr aufwändig sein.
  - Algorithmus benötigt bestimmte Zeit für das Abarbeiten des Heaps (der Heaps).
  - Laufendes Programm wird angehalten („Stop the world“-Prozess).
  - Großes Problem bei interaktiven Programmen.
  - Sehr großes Problem bei Echtzeitsystemen.
- Inkrementelle Anwendung
  - Algorithmus bearbeitet nicht den gesamten Heap (nur Teile davon).
  - Die Zeit für den kleineren Durchlauf kann leichter nach oben beschränkt werden.
- Heapgrößen für unterschiedliche Generationen festlegen.
  - Die Zeit für den Durchlauf eines Heaps kann beschränkt werden.
  - Die „älteste“ Generation darf aber nicht beschränkt sein.
  - Für diese muss man sich einen speziellen Algorithmus überlegen.

Anmerkungen (2)

- Garbage-Collection ist nicht vorhersagbar!
  - In viele Fällen macht sich die Garbage-Collection nicht bemerkbar.
  - Aber schlecht für Echtzeitsysteme!
- Allgemeine Regeln
  - Nur notwendigen Speicher anfordern!
  - Nicht Referenzen horten!
    - z.B. nicht benötigte Einträge in einem Array auf null setzen!
- Aufruf von `System.gc()`
  - Manchmal sinnvoll
  - Was macht der Garbage-Collector an dieser Stelle?
    - *"Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a* ***best effort*** *to reclaim space from all discarded objects."*

Garbage-Collection in der JVM (1)

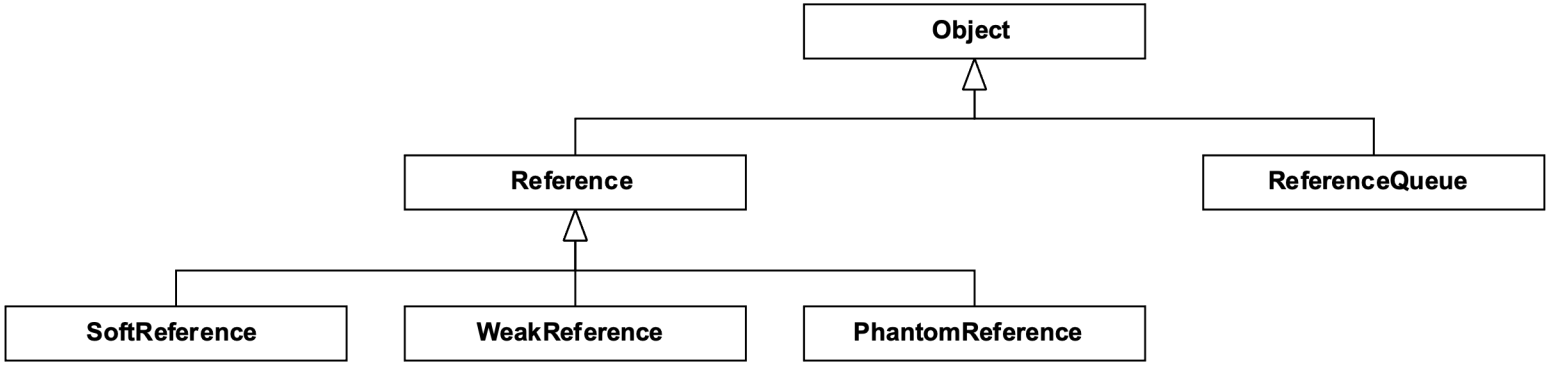
- Java-Implementierung bieten verschiedene Garbage-Collectors an.
- OpenJDK 11– implementierte Algorithmen:
  - Serial GC
  - Parallel GC
  - G1 GC
  - Concurrent Mark and Sweep GC (Deprecated)

Garbage-Collection in der JVM (2)

- Parameter
  - Garbage-Collection kann parametrisiert werden (bei Aufruf von java).
  - Beispiele
    - `-Xms`: Anfängliche Heap-Größe
    - `-Xmx`: Maximale Heap-Größe
    - `-XX:MinHeapFreeRatio`: Minimales Verhältnis freier/belegter Speicher
    - `-XX:MaxHeapFreeRatio`: Maximales Verhältnis freier/belegter Speicher
    - `-XX:UseSerialGC`: Serial GC als GC (analog für andere drei GC)
    - Weitere Beispiele: http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html
  - Parameter für Generationen
  - Parameter für unterschiedliche Typen

Objektreferenzen

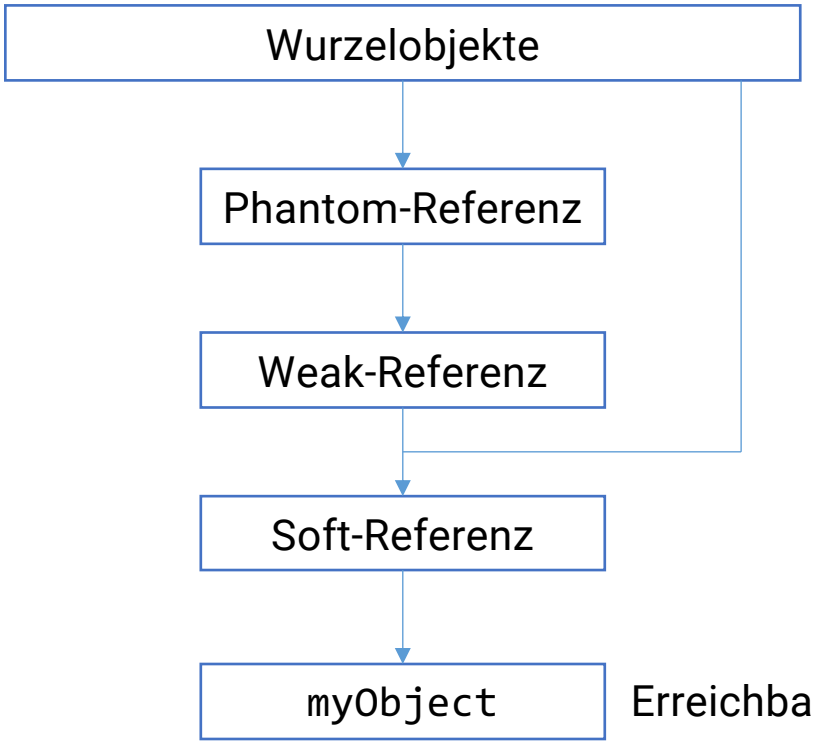
- Unterschiedliche Arten von Referenzen:



- Stufen der Erreichbarkeit
  - Strongly reachable
    - Bisher besprochen
    - Garbage-Collector kann Objekte mit solchen Referenzen nicht einsammeln.
  - Softly reachable
  - Weakly reachable
  - Phantomly reachable
  - Unreachable
    - Bisher besprochen
    - Wenn auf keine andere Art erreichbar

Referenzpfade

- Objekte können von einer Menge an Wurzelobjekten ausgehend über mehrere Pfade erreicht werden.
- Die schwächste Referenz auf dem stärksten Pfad bestimmt die Erreichbarkeit.
- Beispiel:



Erreichbarkeit von `myObject`: *Soft*

Soft-Referenzen

- Objekte die darüber referenziert werden, werden erst dann eingesammelt, wenn der Speicher knapp wird.
- Spezifikation verlangt nur, dass diese Objekte eingesammelt werden, bevor es zu einer `OutOfMemoryError` Exception kommt.
  - Soft-Referenzen werden für Caching verwendet.
- Ältere Objekte werden zuerst eingesammelt.
- Mit der Methode `get()` bekommt man das referenzierte Objekt oder `null`, wenn das Objekt vom Garbage-Collector eingesammelt wurde.
- Beispiel (Ausgeben des Inhalts)

```
SoftReference<Integer> i = new SoftReference<Integer>(1000);
System.out.println(i.get());
```

Weak-Referenzen

- Objekte die darüber referenziert werden, werden eingesammelt, wenn sie nur mehr über Folgen von Referenzen referenziert werden, die alle zumindest eine Weak-Referenz enthalten (keine starke Referenz darauf).
- Sobald der Garbage-Collector anläuft, werden diese Objekte eingesammelt.
- Beispiel

```
WeakReference<Integer> i2 = new WeakReference<Integer>(1200);
System.out.println(i2.get());
```

Phantom-Referenzen

- Auf diese kann man nicht mehr zugreifen.
- Ermöglichen „sauberes“ Einsammeln der Objekte.
- Ähnlich wie `finalize`-Methode, aber flexibler.
- `get()` liefert immer `null`.
- Anwendungsgebiete sind Abschlussaktionen allgemeiner Art, die sich nicht auf ein bestimmtes Exemplar beziehen und bei denen es ausreicht, festzustellen, dass ein Objekt finalisiert wurde, ohne zu wissen, welches konkrete Objekt es war oder zu welcher Klasse es gehörte.

finalize

- Methode `finalize()` kann in jeder Klasse überschrieben werden (von Object geerbt).
- Wird vom Garbage-Collector ausgeführt.
  - Daher kann man nie genau sagen, wann `finalize()` ausgeführt wird.
- Sollte vermieden werden (wenn nicht unbedingt notwendig)!
- Sollte möglichst wenig Zeit und Ressourcen verbrauchen!
- Seit Java 9 als deprecated markiert.

- Allgemeine Form

```
protected void finalize() throws Throwable {
    try {... /*Freigabe etc.*/ }
    finally { super.finalize(); }
}
```