Vorlesung „Objektorientierte Softwareentwicklung“

Sommersemester 2010

Kapitel 3. Implementierung objektorientierter Sprachen

Motivation

Speicherbereiche

Umsetzung von Klassendeklarationen

Objekterzeugung

Nachrichten und dynamisches Binden

Prozedurausführung

universitätbonn

Motivation

- Basiskonzepte besser verstehen!
  - Dadurch besser verstehen, warum bestimmte Programmiertechniken wichtig sind!
- Themen
  - Was „statisch“ und „dynamisch“ praktisch bedeutet
  - „Dynamisches Binden“ ist keine Magie und auch nicht ineffizient!
  - Warum man Objekte erzeugen, aber nicht „zerstören“ kann
  - Warum man Variablen, die auf nicht mehr benötigte Objekte zeigen `null` zuweisen sollte!
  - Warum Objekterzeugung teuer ist!

Speicherbereiche

- Statischer Bereich für Klassen und Interfaces
  - Speicher für Klassenvariablen – daher `static`
  - Code von Klassenmethoden
  - Code von Instanzmethoden
  - Sprungtabelle zu Instanzmethoden (für dyn. binding)
- Dynamischer Bereich für Objeke (Halde / heap)
  - Heap-Verwaltung incl. “Garbage Collection“
  - Dynamisches Binden von Nachrichten
- Dynamischer Bereich für Aufrufe (Stapel / stack)
  - Stackframes für Aufrufparameter
  - Stackverwaltung

universitätbonn

Objekterzeugung und Speicherverwaltung

Verwaltung von Objekten im Speicher

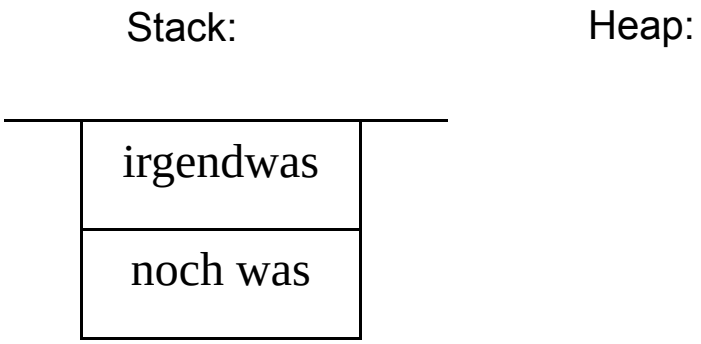
- Der **Stapelspeicher** wird gemäß der **Blockstruktur** des Programms automatisch verwaltet
  - Jeder Aufruf hat ein „Stackframe“
  - Stackframe = Speicher für alle Aufrufparameter
- In Java liegen Objekte grundsätzlich auf dem **Haldenspeicher** (**Heap**)
  - Sie sind über Referenzvariablen zugänglich, die auf dem Stapel liegen und deren Referenzen auf die Halde zeigen
  - Das **Java-Laufzeitsystem** organisiert die Halde mittels einer automatischen Speicherverwaltung

Verwaltung von Objekten im Speicher

- Der Ausdruck `new K();` reserviert den Speicherplatz für ein Objekt der Klasse `K`
  - Außerdem wird das Objekt initialisiert
- Der Ausdruck `new K()` liefert eine Referenz auf das neue geschaffene Objekt

Verwaltung von Objekten im Speicher: Beispiel

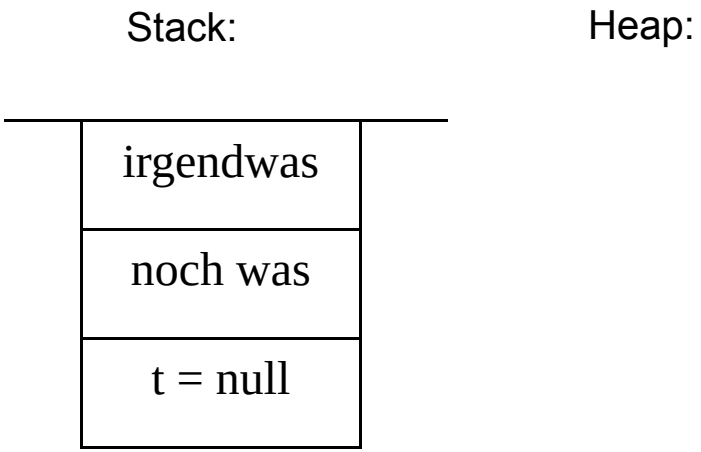
- **Beispiel:** Wir betrachten die Speicherbilder beim Ausführen folgender Codesequenz:




```
{
    Time t;
    t = new Time();
}
```

Verwaltung von Objekten im Speicher: Beispiel

- **Beispiel:** Wir betrachten die Speicherbilder beim Ausführen folgender Codesequenz:



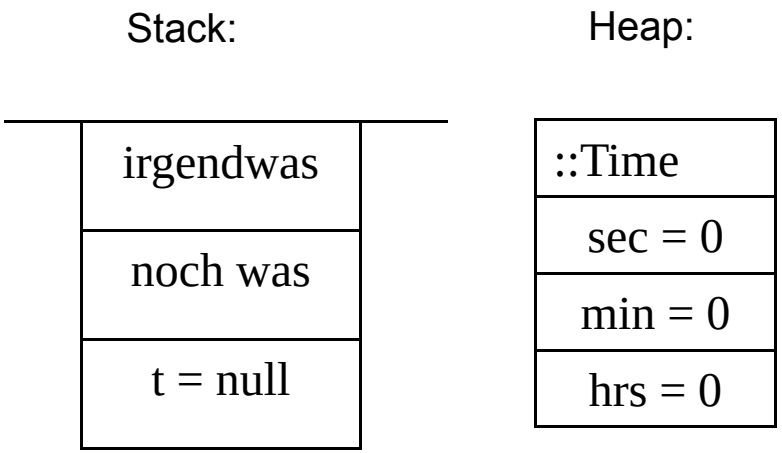

```
{
    Time t;
    t = new Time();
}
```

Bei Eintritt in den Block wird zunächst Platz für die lokalen Variablen (hier: **`Time t`**) auf dem Stack geschaffen und mit Null vorinitialisiert

Verwaltung von Objekten im Speicher: Beispiel

- **Beispiel:** Wir betrachten die Speicherbilder beim Ausführen folgender Codesequenz:

```
{
    Time t;
    t = new Time();
}
```

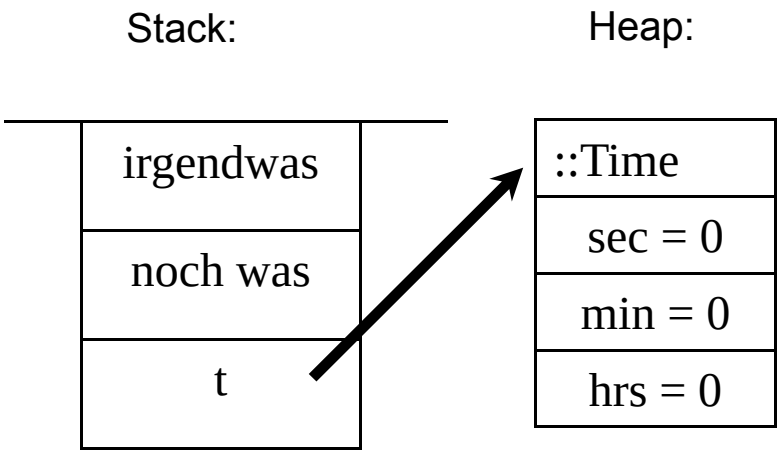


Danach legt **new** ein neues Objekt vom Typ Time mit auf default-Werte vorinitialisierten Membervariablen auf dem Heap an

Verwaltung von Objekten im Speicher: Beispiel

- **Beispiel:** Wir betrachten die Speicherbilder beim Ausführen folgender Codesequenz:

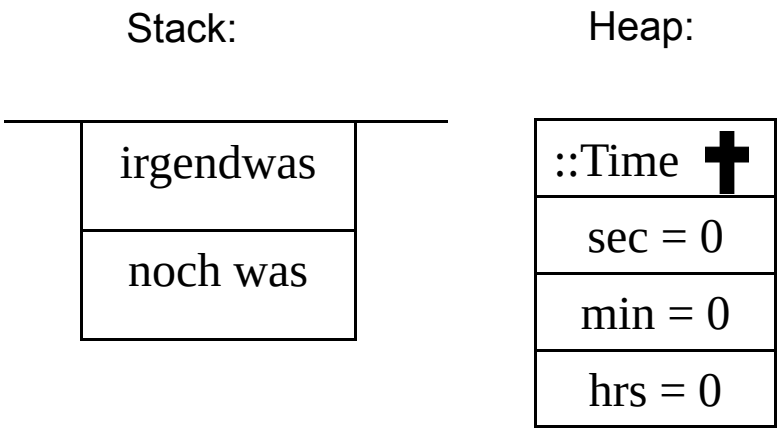
```
{
    Time t;
    t = new Time();
}
```



Als nächstes bekommt die Variable `t` eine Referenz auf das neu erzeugte Objekt zugewiesen

Verwaltung von Objekten im Speicher: Beispiel

- **Beispiel:** Wir betrachten die Speicherbilder beim Ausführen folgender Codesequenz:

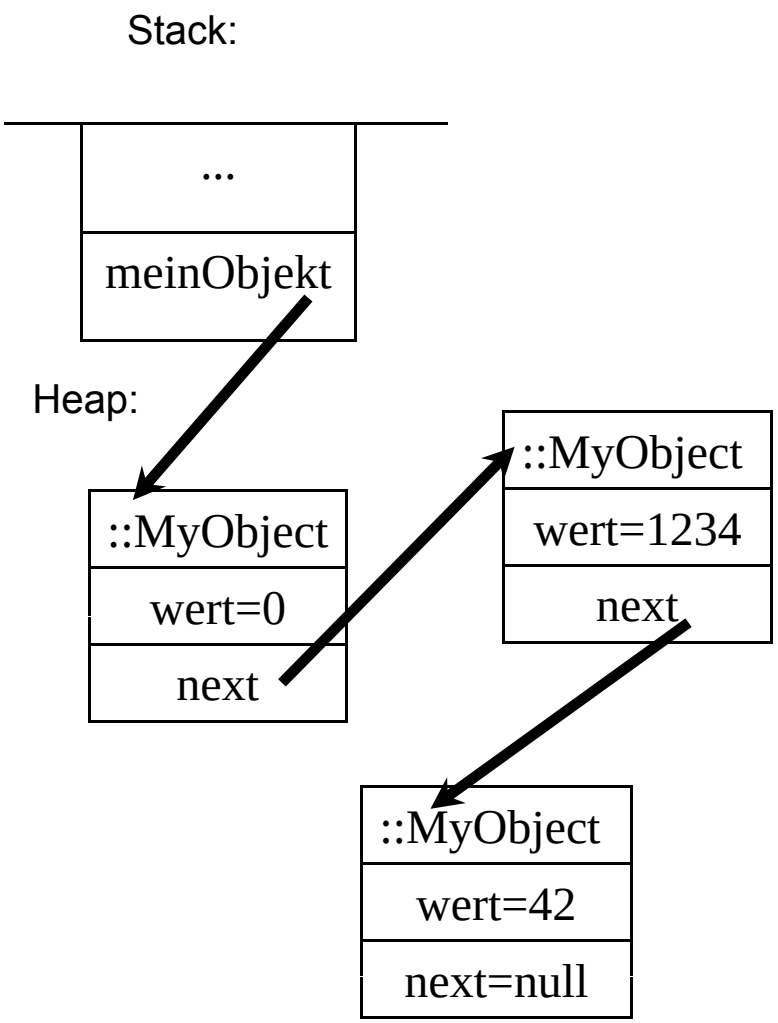



```
{
    Time t;
    t = new Time();
}
```

Zum Abschluss wird der Block verlassen, wodurch die Referenz auf das Objekt verloren geht.
Das Objekt „stirbt“

Verwaltung von Objekten im Speicher: Erreichbarkeit

- Objekte auf der Halde leben (unabhängig von der Blockstruktur des Programms) so lange, wie sie über Referenzvariable **erreichbar** (reachable) sind
  - Ein Objekt ist indirekt erreichbar, wenn es über eine Referenzvariable erreichbar ist, die als Feld in einem anderen erreichbaren Objekt vorkommt
    - Das Objekt, das den Wert 0 enthält ist direkt erreichbar, die weiteren Objekte sind indirekt erreichbar



Verwaltung von Objekten im Speicher

- Tote Objekte auf der Halde bezeichnet man als **Abfall** (garbage)
- Um den Haldenspeicher wieder verwenden zu können, kennt das Java-Laufzeitsystem eine **automatische Speicherbereinigung** oder „Abfallsammlung“ (**garbage collection**)
  - Die es z.B. anstößt, wenn der Haldenspeicher zur Neige geht

C++: Explizite Verwaltung von Objekten im Speicher

- In C++ hingegen muss (bzw. darf) die Speicherrückgabe explizit programmiert werden
- Der Programmierer muss für jede von ihm entworfene Klasse eine spezielle Funktion schreiben, einen sogenannten **Destruktor** (destructor), in der die Rückgabe des Speichers von einem nicht mehr benötigten Objekt beschrieben wird
  - Dieser ruft i.A. einen Operator **`delete`** auf
- Dies ist in vielen Fällen effizienter als eine automatische Speicherbereinigung
  - Ist aber für den Programmierer sehr viel umständlicher und kann leicht zu Programmierfehlern führen

C++: Explizite Verwaltung von Objekten im Speicher

- Insbesondere können dabei in C++ (**nicht in Java**) folgende Programmierfehler vorkommen:
  - Speicher für ein Objekt wird irrtümlicherweise **nicht zurückgegeben**, obwohl es tot ist
  - Speicher für ein Objekt wird fälschlicherweise zurückgegeben wird, obwohl es noch **nicht tot** ist

Immer mehr Speicher wird verbraucht „Speicherleck“ (memory leak)

Solche Fehler sind extrem boshaft, da sie erst dann bemerkt werden, wenn der Speicherplatz erneut zugeteilt und überschrieben wurde

Verwaltung von Objekten im Speicher: Garbage Collection

- Ein einfaches Prinzip der automatischen Speicherbereinigung besteht aus zwei Phasen:
  - **markieren** (mark) und
  - (zusammen-)**kehren** (sweep)
- In der Markierungsphase geht man von unten nach oben über den Stack und markiert rekursiv alle von den Referenzvariablen aus erreichbaren Objekte auf dem Heap
- In der Kehrphase kehrt man allen nicht markierten Speicher auf dem Heap in eine **Freispeicherliste** (free list) zusammen und löscht die Markierungen
  - Man kann sogar die markierten Objekte in eine Ecke der Halde kehren (verschieben), denn der Java-Programmierer bekommt die tatsächlichen Werte der Referenzen nie zu Gesicht

Verwaltung von Objekten im Speicher: Garbage Collection

- Es gibt noch andere Prinzipien der Speicherbereinigung
  - Es ist in Java nicht festgeschrieben, welche benutzt werden
- Mit komplizierteren Methoden kann z. B. erreicht werden, dass Objekte, die oftmals in kurzen Abständen hintereinander benutzt werden (z. B. in einer Schleife), nach einer Speicherbereinigung in benachbarte Stellen im Speicher zu liegen kommen
  - Alle modernen Prozessoren sind mit einem **Cache-Speicher** (cache memory) ausgestattet, auf den sehr viel schneller als auf den Hauptspeicher zugegriffen werden kann
    - Dieser hat eine beschränkte Größe, es können aber zusammenhängende Stücke vom Hauptspeicher sehr schnell zu ihm transferiert werden
    - Somit kann evtl. die Ablaufgeschwindigkeit eines Programms erheblich gesteigert werden, wenn bei der Speicherbereinigung verkettete Objekte in benachbarte Bereiche im Heap gelegt werden
- Weiteres mögliches Problem beim Garbage Collection: Echtzeitanwendungen

universitätbonn

Dynamisches Binden bei „überladenen“ und „überschriebenen“ Methoden

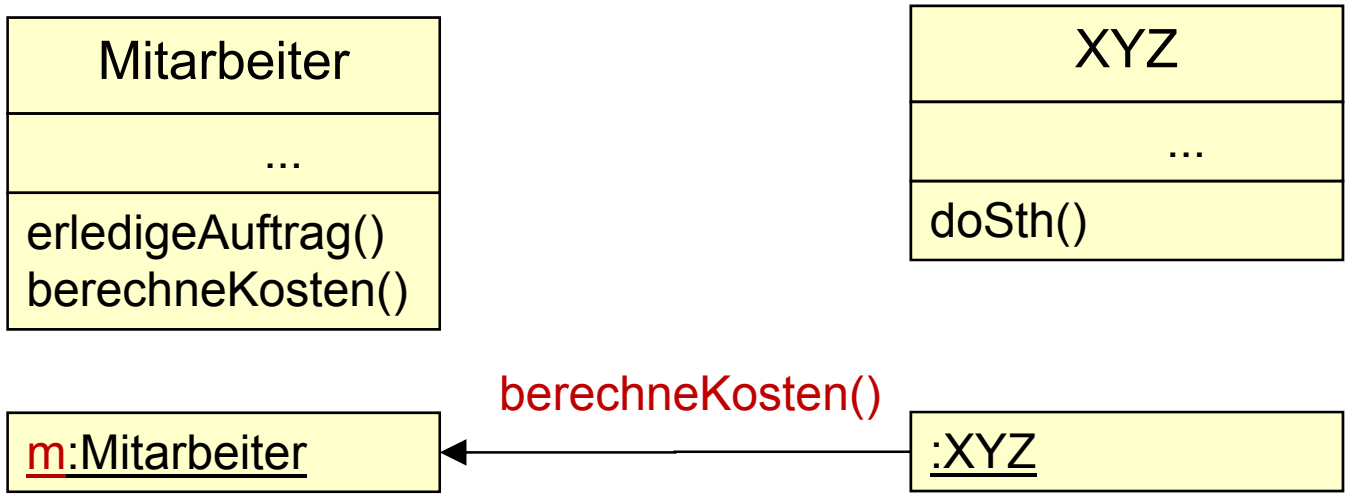
Folgende Beispielszenarien:

1. Gleiche Nachrichtensignatur / Empfängerobjekte verschiedener Klassen
2. Verschiedene Nachrichtensignaturen / Gleiches Empfängerobjekt

Umsetzen von dynamischem Binden

- Folgendes Szenario setzen wir auf den nächsten Folien um. Es geht immer um die Umsetzung des Aufrufs von „berechneKosten()“:

UML



Java

```
class Mitarbeiter {
    erledigeAuftrag() {...}
    berechneKosten(){...}
}
```

```
class XYZ {
    doSth() {
        Mitarbeiter m;
        ...;
        m.berechneKosten();
    }
}
```

Überblick

- Zuerst: Allgemeine Grundlagen
  - Dynamisches Binden in rein objektbasierten (d.h. nicht klassenbasierten) und nicht streng getypten Sprachen
  - Objekte mit Methodentabellenverweisen
- Anschliessend: Java, C++, Eiffel, und Co
  - Anpassung an klassenbasierte, strang typisierte Sprachen
- Abschluss: Anwendung in versch. Szenarien
  - Überladen versus Überschreiben
  - Zusammenspiel mit Vererbung

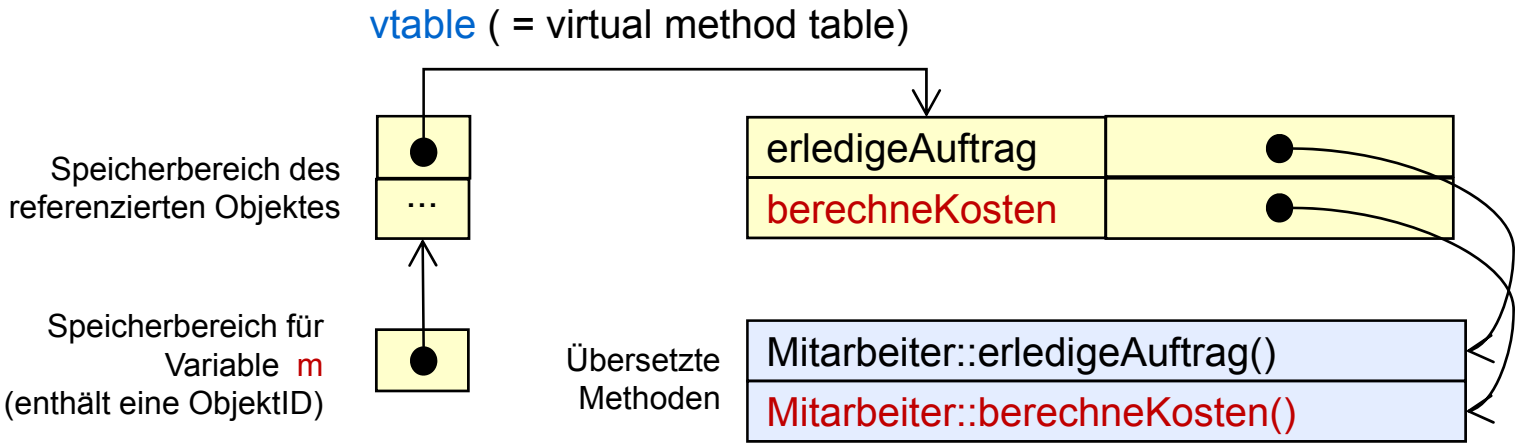
Methodentabellen (rein objektbasiert)

- Szenario (Ursprünglicher Code in fiktiver Objektsprache):

```
object Mitarbeiter = {
    erledigeAuftrag() {...}
    berechneKosten(){...}
}
```

```
object XYZ {
    doSth(m) {
        m.berechneKosten();
    }
}
```

- Umsetzung der Objektdefinitionen
  - Jedes Objekt verweist auf seine Methodentabelle!
  - Jeder Eintrag in einer Methodentabelle verweist auf den übersetzten Code einer Methode!



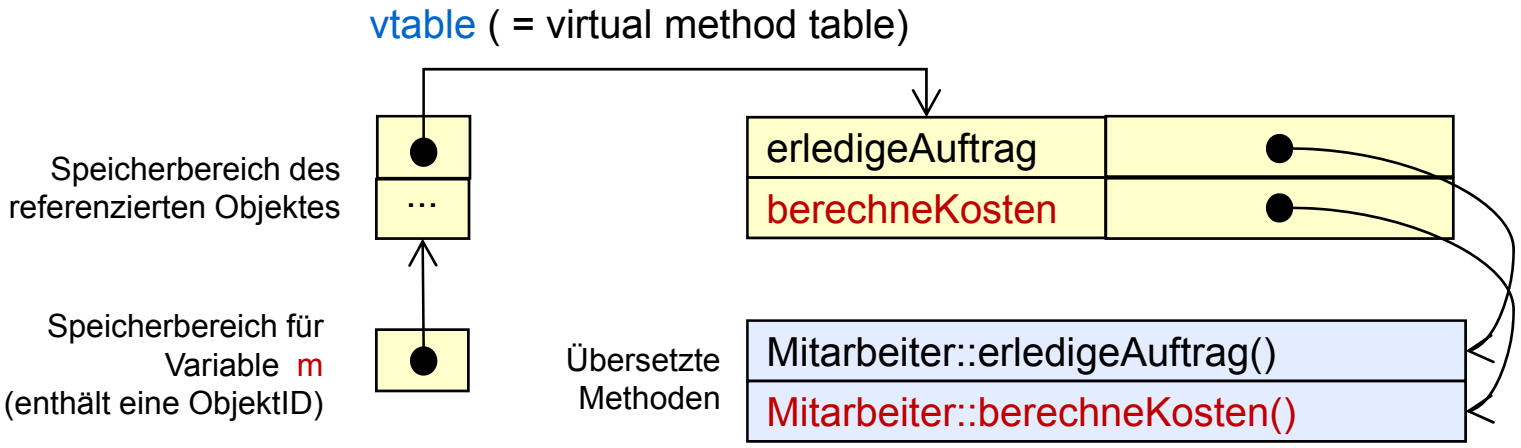
Nachrichten (rein objektbasiert)

- Szenario (Ursprünglicher Code in fiktiver Objektsprache):

```
object Mitarbeiter = {
    erledigeAuftrag() {...}
    berechneKosten(){...}
}
```

```
object XYZ {
    doSth(m) {
        m.berechneKosten();
    }
}
```

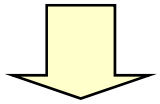
- Umsetzung von Nachrichten
  - Jede Nachricht wird durch ein Stück Code ersetzt, das anhand der Methodentabelle des Empfänger-Objektes den auszuführenden Methodencode bestimmt.
  - m.berechneKosten(..) → m.vtable.getValue(berechneKosten)(..);



Prozeduraufrufe versus Nachrichten

C, Pascal & Co

```
berechneKosten(obj,param1, ..., paramN);;
```



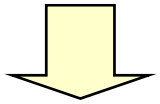
```
... // Parameterübergabe
jump addresseXXXXX;



addresseXXXXX:
... // Code von „berechneKosten“
return;
```

C++, Java & Co

```
object.berechneKosten(...);
```



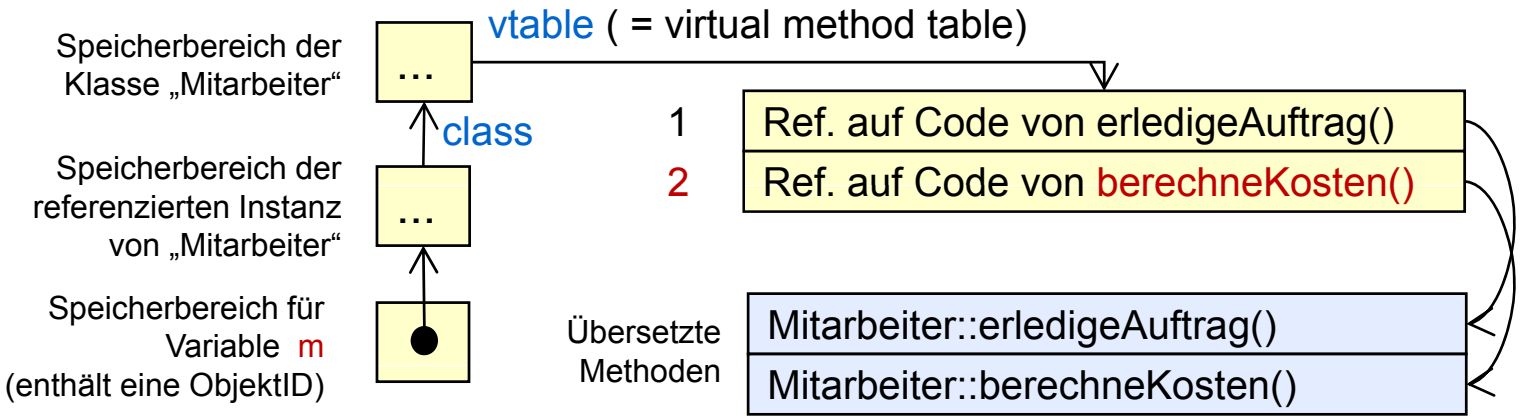
```
... // Parameterübergabe
addresse =
object.vTable.getValue(berechneKosten);
jump addresse;

addresseXXXXX:
... // Code von „berechneKosten“
return;
```

- Nachricht ≠ Prozeduraufruf
  - Die Adresse des auszuführenden Codes ist an der Aufrufstelle nicht bekannt!
  - Auch in verteilten Umgebungen einsetzbar!
  - Eine Nachricht → Verschiedene Effekte, je nach Empfänger!

Methodentabellen (Klassenbasiert, streng typisiert)

- Besonderheiten klassenbasierter Sprachen
  - Klasse definiert gemeinsame Methoden für alle Instanzen
    - Klasse hat Methodentabelle die für alle Instanzen gilt
    - Objekte verweisen auf ihre Klasse und diese verweist auf die Methodentabelle
  - Menge der Methoden und Felder einer Klasse ändern sich nicht zur Laufzeit
    - Methodennamen als explizite Schlüssel für jede Referenz auf Methodencode sind nicht mehr erforderlich
    - Stattdessen kann der Compiler jeder Methode einen festen Index in der Methodentabelle einer bestimmten Klasse vergeben
    - Beispiel: indexOf(Mitarbeiter, berechneKosten) = 2



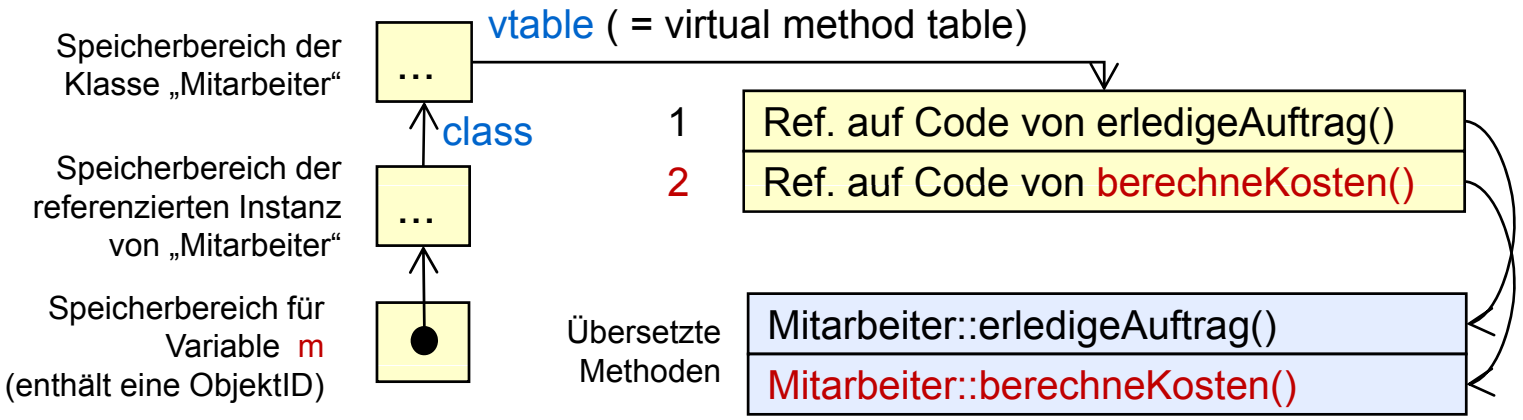
Methodentabellen (Klassenbasiert, streng typisiert)

- Szenario (Ursprünglicher Java-Code):

```
class Mitarbeiter {
    erledigeAuftrag() {...}
    berechneKosten(){...}
}
```

```
class XYZ {
    doSth() {
        Mitarbeiter m;
        ...;
        m.berechneKosten(); }   }
```

- Schema der Umsetzung der Klassendefinitionen
  - Jedes Objekt verweist auf seine Klasse!
  - Jede Klasse verweist auf ihre Methodentabelle!
  - Jeder Eintrag in der Methodentabelle verweist auf den übersetzten Code einer Methode!



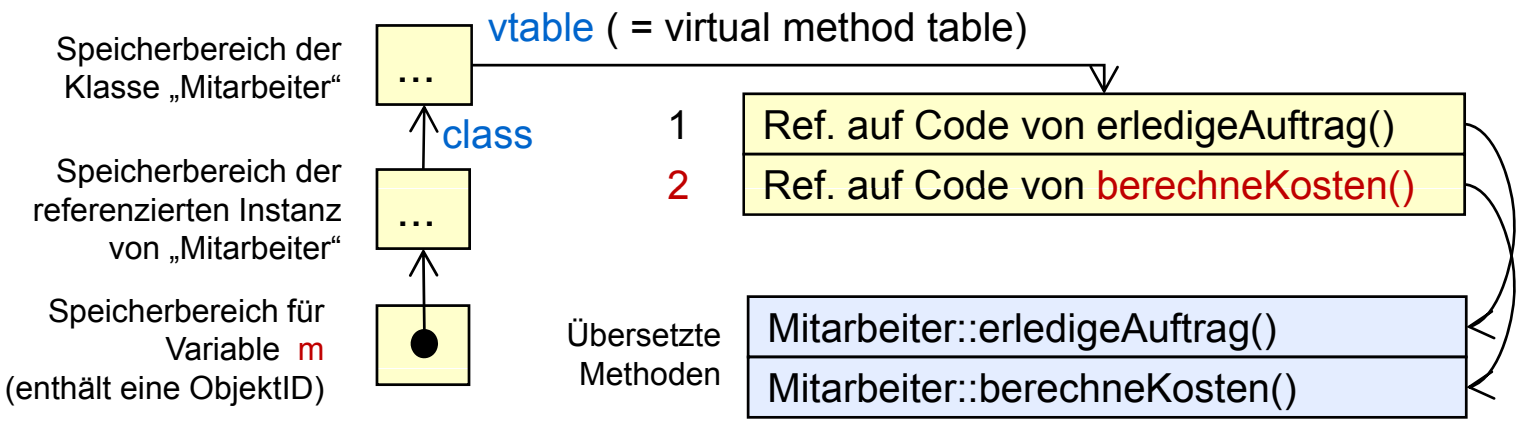
Nachrichten (Klassenbasiert, streng typisiert)

- Szenario (Ursprünglicher Java-Code):

```
class Mitarbeiter {
    erledigeAuftrag() {...}
    berechneKosten(){...}
}
```

```
class XYZ {
    doSth() {
        Mitarbeiter m;
        ...;
        m.berechneKosten(); }   }
```

- Umsetzung von Nachrichten: Schema
  - obj.msg(..) → obj.class.vtable[indexOf(obj.class,msg)] (..);
  - Dank statischer Indexermittlung zu jeder Methode kann für indexOf(....) schon beim Compilieren eine feste Konstante eingesetzt werden
- Umsetzung von Nachrichten: Beispiel
  - m.berechneKosten(..) → m.class.vtable[2](..);



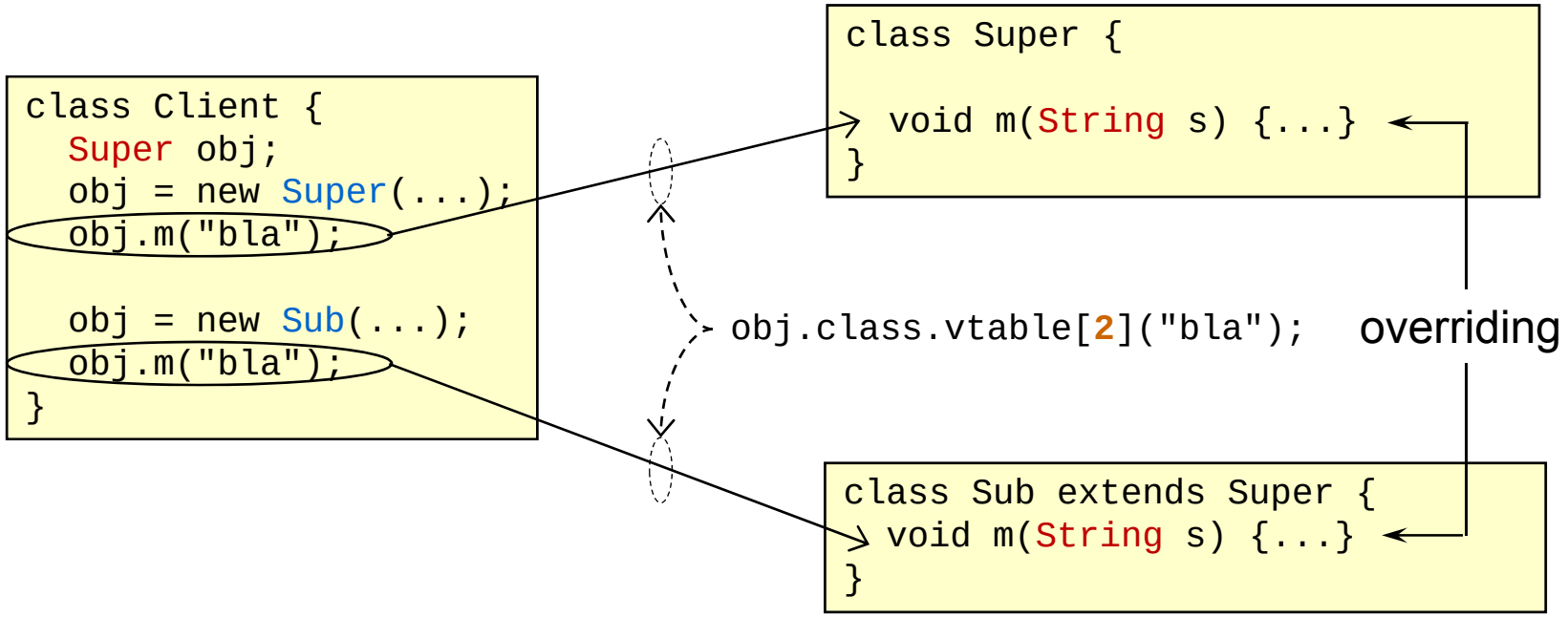
Dynamisches Binden: Effizienz

- Der Zugriff obj.class.vtable[index](..) reduziert die Kosten von dynamischem Binden auf
  - 2 lesende Speicherzugriffe (obj.class.vtable[index]) und ...
  - 1 Sprung an die in der Methodentabelle gefundene Adresse
  - Davon ist der Sprung an eine nicht vorhersehbare Adresse am teuersten!
    - Er bedeutet, dass die in der Anweisungspipeline des Prozessors schon im voraus geladenen Instruktionen nicht wirklich die sind, die man als nächstes braucht. Statt sofort weiterarbeiten zu können muss der Prozessor erst mal warten, bis die von der Sprungadresse geholten Instruktionen bei ihm eintreffen.
- Auch dafür gibt es weitere Optimierungen die aber den Rahmen der Vorlesung sprengen
  - Inlining
  - Polymorphic inline caches (Urs Hölze)
  - „Just-in-time“ Compilierung (Urs Hölzle)
  - Dynamische „HotSpot“ Recompilierung (Urs Hölzle)
  - ➔ http://www.cs.ucsb.edu/~urs/oocsb/papers.shtml

Von der ETH an die Google-Spitze

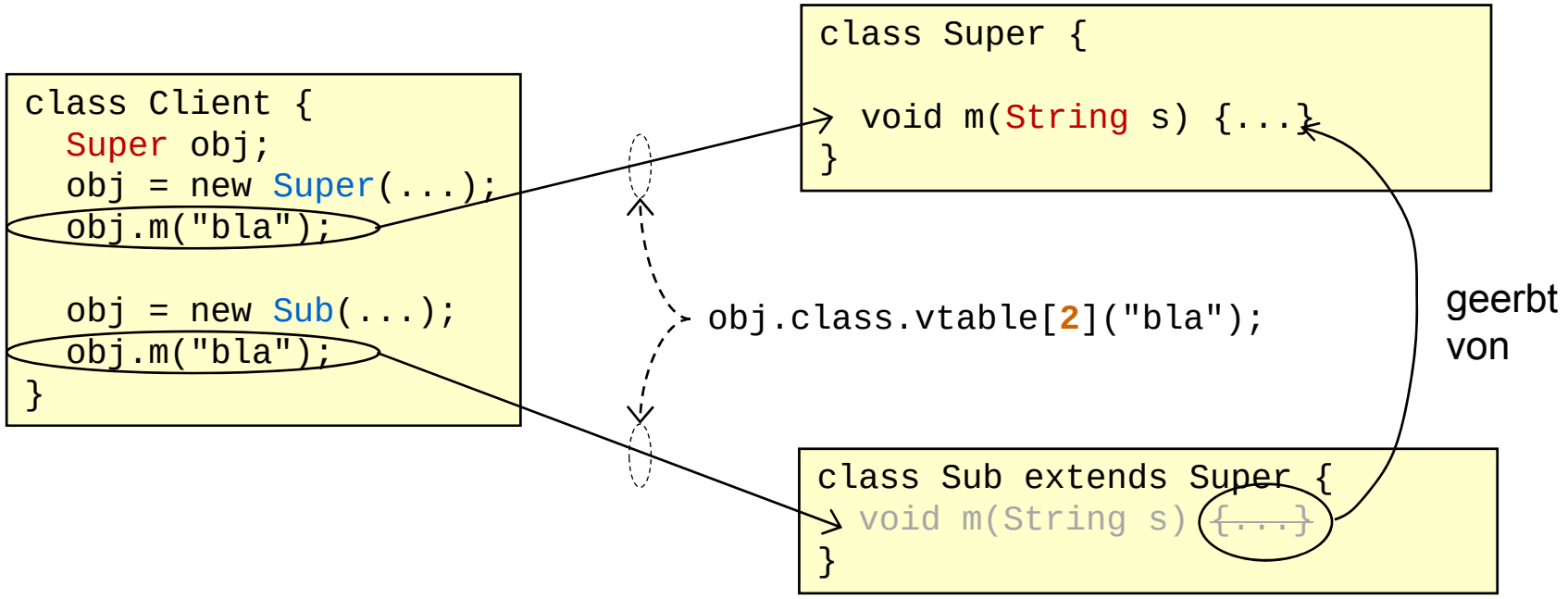
Urs Hölzle studierte von 1984 bis 1988 Informatik an der ETH Zürich. Mit einem Stipendium ging er nach Kalifornien, wo er an der Stanford University promovierte. Bevor er zu Google stiess, war Hölzle Assistenzprofessor an der University of California in Santa Barbara. Er gilt als einer der Pioniere der sogenannten "just-in-time compilation". Bei Google wirkte Hölzle zwei Jahre als "Vice President of Engineering". Jetzt beschäftigt er sich als Google Fellow primär mit neuen Forschungsfragen und der technologischen Weiterentwicklung des Unternehmens.

Dynamic Binding und Overriding



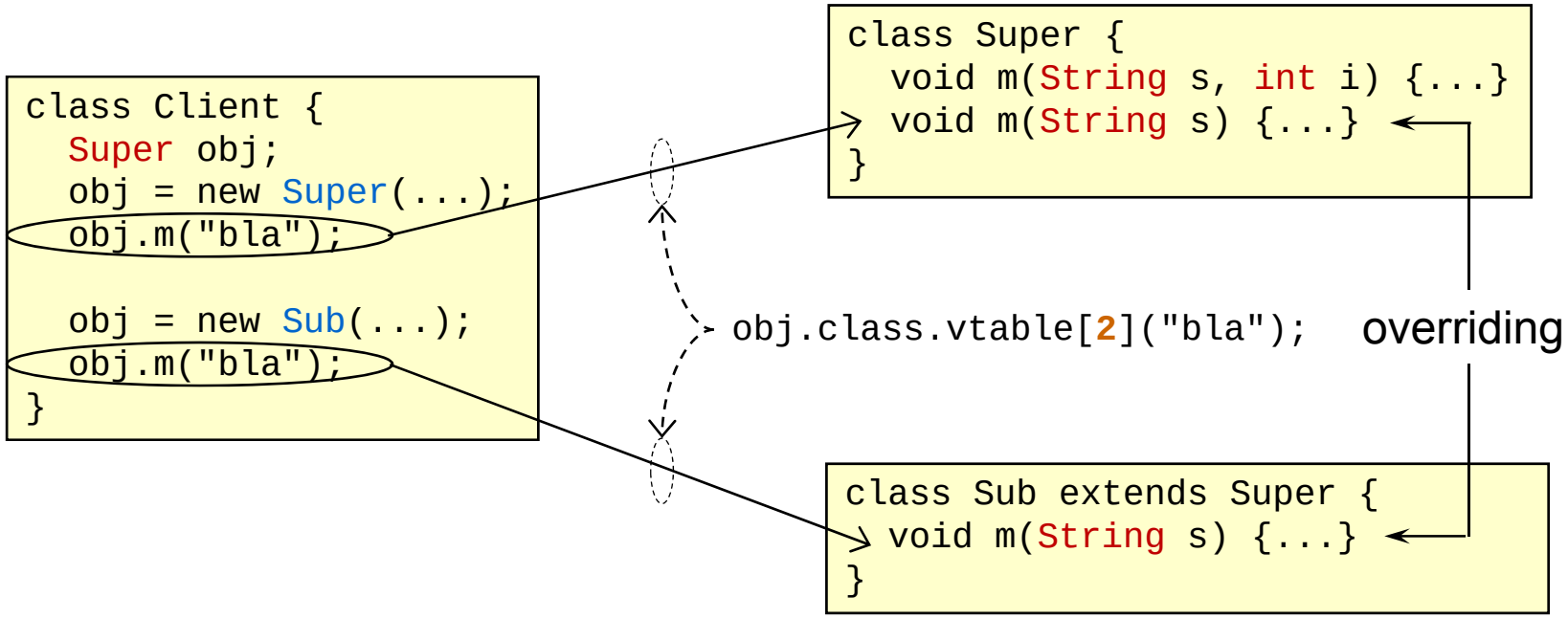
- Gleiche Signatur $\Rightarrow$ gleicher Methodentabellen-Index
- Methodentabelle der Unterklasse verweist auf der gleichen Indexposition auf die eigene Methode

Dynamic Binding und Vererbung



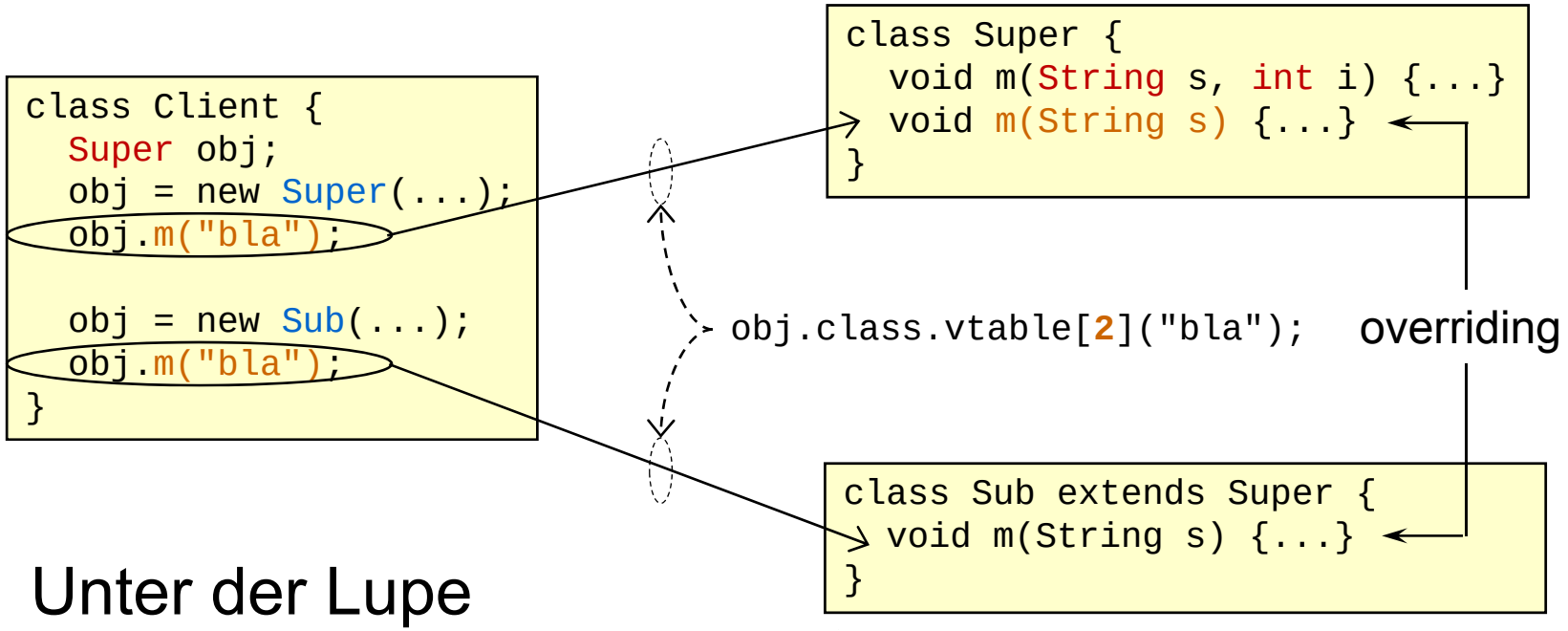
- Gleiche Signatur $\Rightarrow$ gleicher Methodentabellen-Index
- Methodentabelle der Unterklasse verweist auf der gleichen Indexposition auf die geerbte Methode

Overriding versus Overloading: Szenario 1



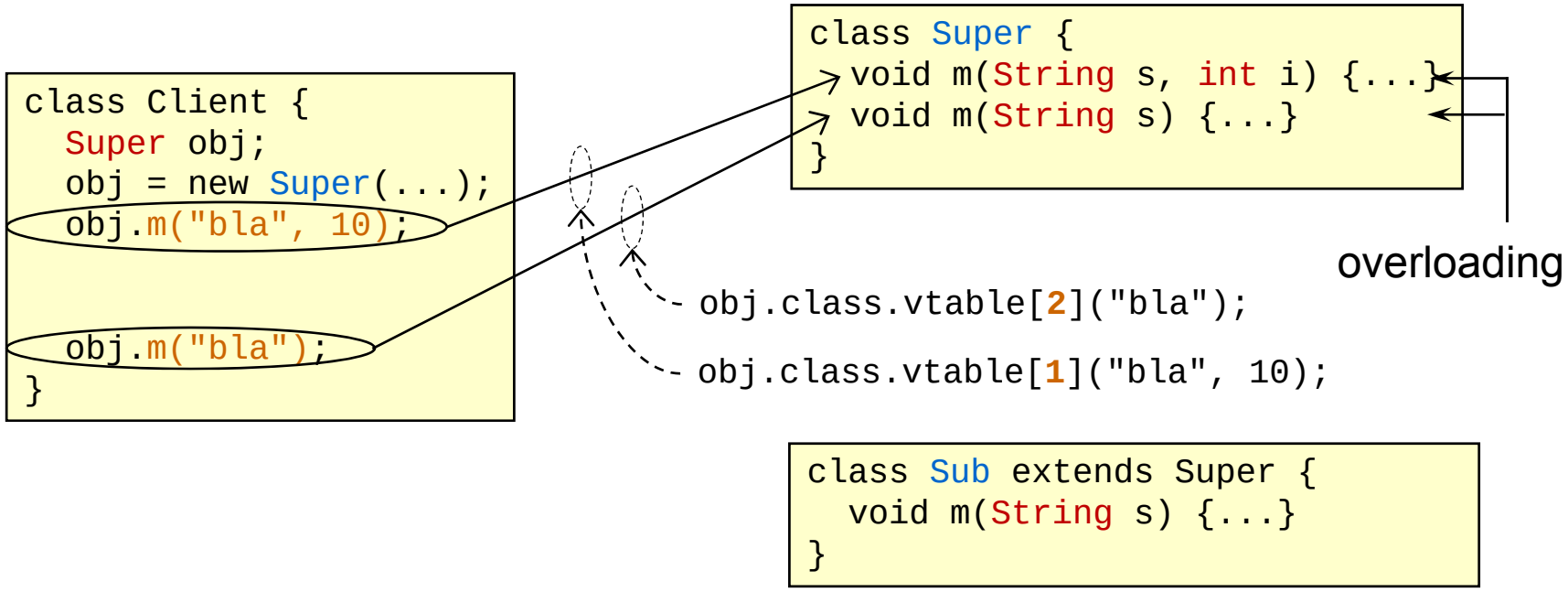
- Overriding
  - Beziehung zwischen Methoden in Oberklasse und Unterklasse
  - Gleiche Signatur ⇒ gleiche Operation
    - bekommen gleichen Methodentabellen-Index
  - ➔ Die spezifischste Methode (= Implementierung der Operation) gilt!
    - ➔ Methodentabelle der Unterklasse verweist auf Indexposition 2 auf die eigene Methode statt die Überschriebene

Overriding versus Overloading: Szenario 1



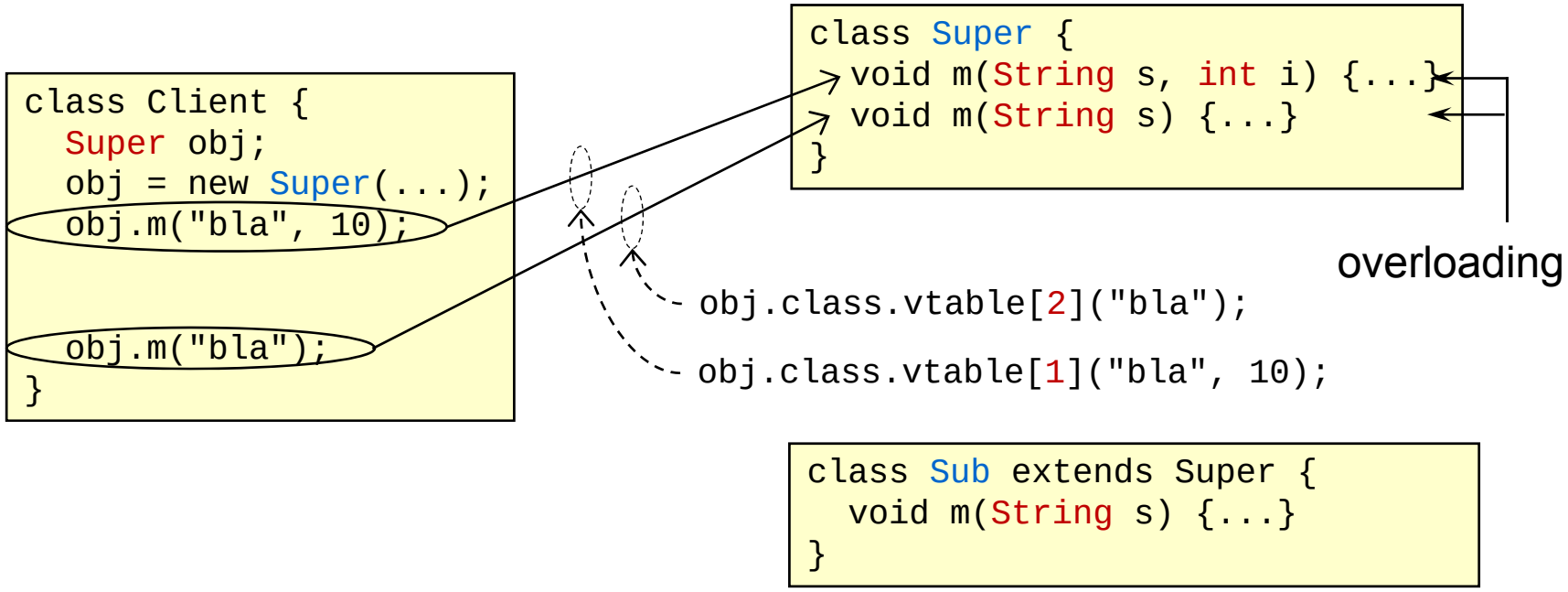
- Unter der Lupe
  - Der Empfänger „obj“ hat in beiden Nachrichten den statischen Typ „Super“
  - ... m(„bla“) hat die Aufrufsignatur m(String)
  - ... die spezifischte Obersignatur von m(String) in Super ist m(String)
  - ... der Index von m(String) ist 2
  - Also wird beide male der gleiche Code generiert:
    `obj.class.vtable[2]("bla");`
  - Der dynamische Typ des Empfängers ist in der ersten Nachricht „Super“ und in der zweiten Nachricht „Sub“.

Overriding versus Overloading: Szenario 2



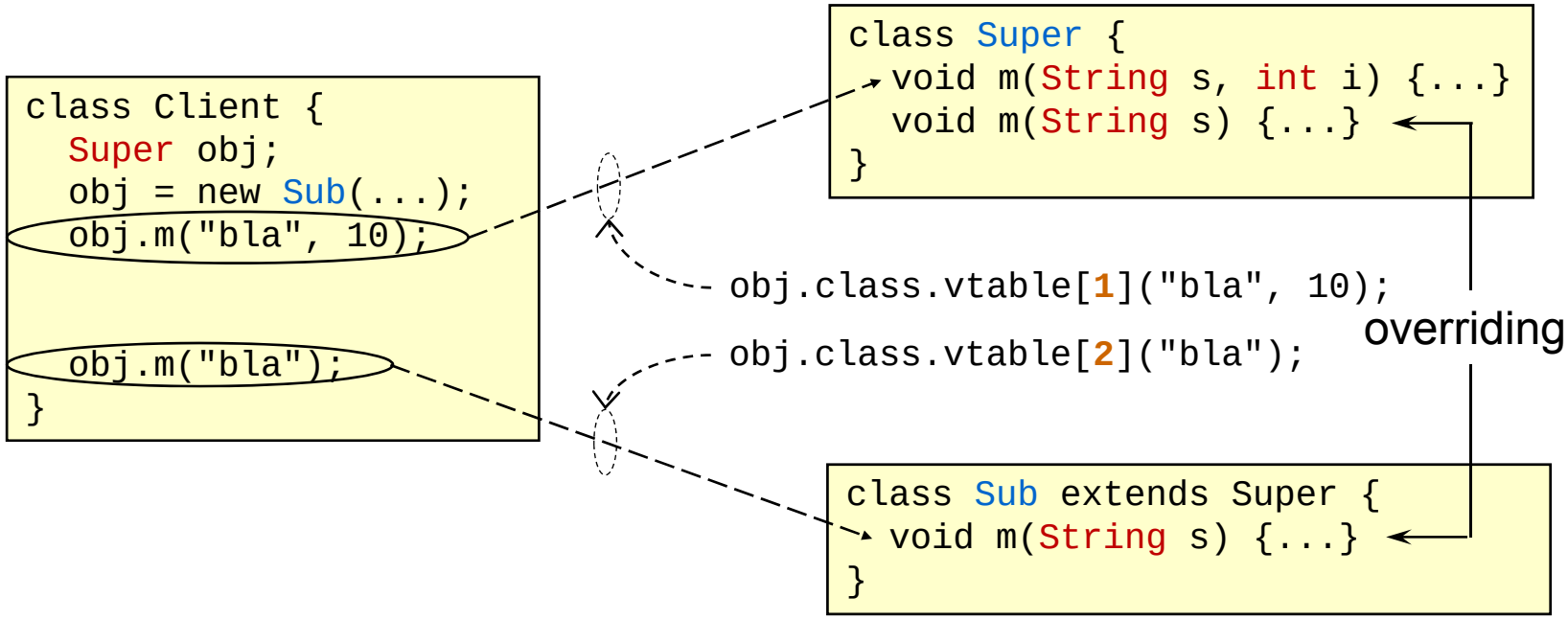
- Overloading
  - Beziehung zwischen Methoden innerhalb einer Klasse
  - Methoden haben gleichen Namen aber verschiedene Signatur ⇒ Sie stellen verschiedene Operationen dar
    - sie bekommen verschiedene Methodentabellen-Indices

OOP: Overriding versus Overloading (Szenario 2)



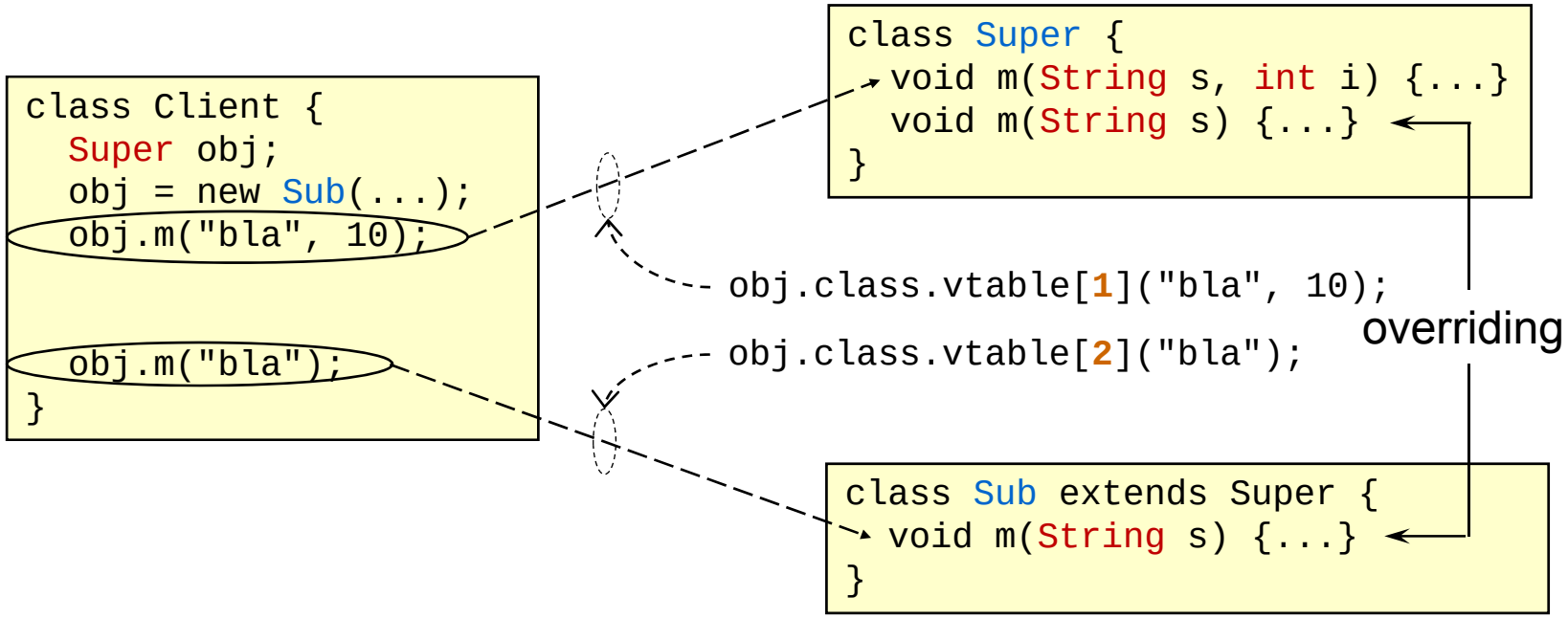
- Unter der Lupe
  - Hausaufgabe (5 Minuten): Lesen Sie sich die vorangegangenen Folien dieses Abschnitts noch einmal aufmerksam durch.
  - Hausaufgabe (2 Minuten): Versuchen Sie nach vorherigem Schema (Seite 3-29) die Erklärung selbst zu rekonstruieren

OOP: Overriding versus Overloading (Szenario 2b)



- Overriding und Overloading zusammen
  - Gleiche Signatur in Oberklasse und Unterklasse
  - ➔ Gleiche Operation → gleicher Index in Methodentabellen
  - ➔ Die spezifischste Methode (= Implementierung der Operation) gilt!
    - ➔ Methodentabelle der Unterklasse verweist auf Indexposition 2 auf die eigene Methode statt die Überschriebene

OOP: Overriding versus Overloading (Szenario 2b)



- Unter der Lupe
  - Hausaufgabe (3 Minuten): Versuchen Sie nach vorherigem Schema die Erklärung selbst zu rekonstruieren

Fazit

- „Dynamisches Binden“ ist nicht teuer
  - Durch die statisch indizierten Methodentabellen sehr effizient!
- Fehlende „Destruktoren“
  - Objekte können nicht explizit „zerstört“ werden um die Konsistenz des Speichers zu gewährleisten
- Erreichbarkeit ist Kriterium für „lebendige“ Objekte
  - Variablen, die auf nicht mehr benötigte Objekte zeigen sollte man `null` zuweisen, um dem „garbage Collector die Erkennung von „Müll“ zu ermöglichen!
- Objekterzeugung ist teuer
  - Aufwand automatischer Speicherverwaltung
  - Programmiertechnik zur Vermeidung überflüssiger Objekterzeugenung
    - Objektpools → Wiederverwendung von Objekten
    - StringBuffer (veränderlich) statt String (unveränderlich, immer wieder neu erzeugt → "a" +"b" erzeugt neuen String "ab" statt die Parameter zu verketten!)