



# Einführung in die Programmierung

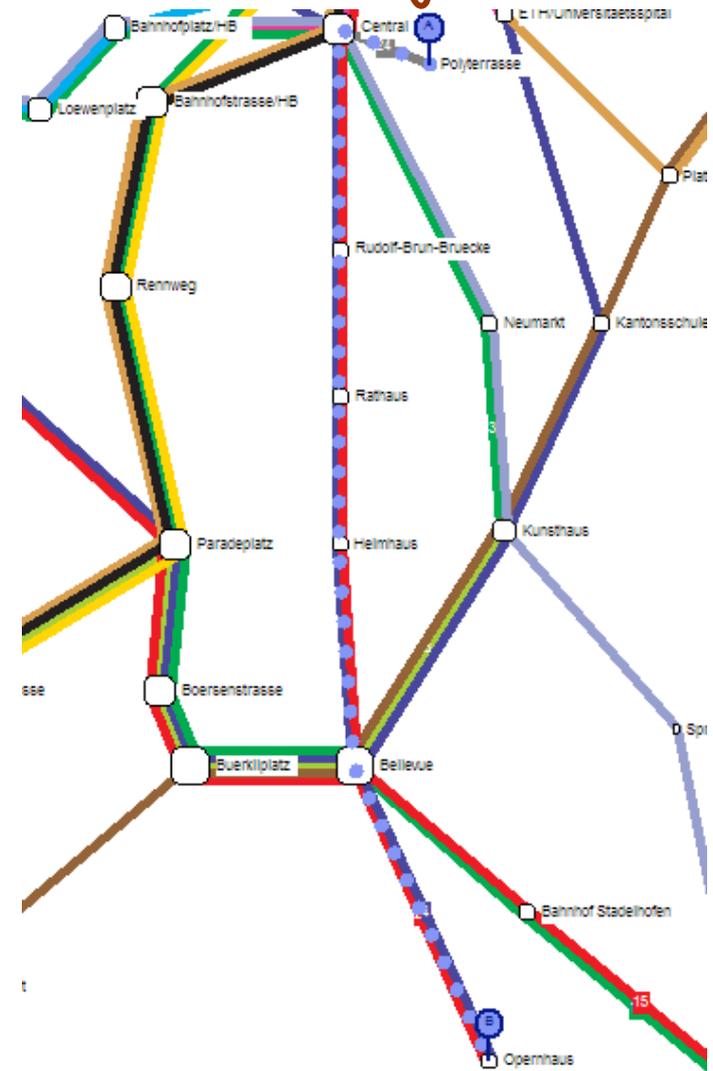
Prof. Dr. Bertrand Meyer

Lektion 6: Objekterzeugung

# Objekte erzeugen

In früheren Beispielen haben *Zurich, Polybahn* etc. jeweils vordefinierte Objekte bezeichnet. *Wir werden jetzt unsere eigenen Objekte erzeugen*

Unser Ziel: Eine Route von Polyterrasse nach Opernhaus erzeugen



# Beispiel: *ROUTE\_BUILDING*



```
class ROUTE_BUILDING inherit  
  ZURICH_OBJECTS
```

```
feature
```

```
  build_route
```

```
    -- Eine Route bauen und damit arbeiten.
```

```
  do
```

```
    -- "Opera_route erzeugen und Teilstrecken hinzufügen"
```

```
    Zurich.add_route (Opera_route)
```

```
    Opera_route.reverse
```

```
    -- "Weiter mit Opera_route arbeiten"
```

```
  end
```

```
    Opera_route: ROUTE
```

```
    -- Eine Route von Polybahn nach Opernhaus.
```

```
end
```

Pseudocode

Pseudocode

Bezeichnet eine Instanz  
der Klasse *ROUTE*



«Pseudocode» bezeichnet Sätze in natürlicher Sprache, die noch nicht geschriebenen Programmcode darstellen

Beispiel:

-- "*Opera\_route* erzeugen und Teilstrecken hinzufügen"

Wir schreiben Pseudocode als Kommentar und in Anführungszeichen

*Stil*: wenn der wirkliche Code geschrieben wird, ist es eine gute Idee, den Pseudocode als normalen Kommentar in dem Programm zu behalten



Ein **Bezeichner** ist ein vom Programmierer gewählter Name, um ein gewisses Programmelement zu repräsentieren. Beispiele sind:

Identifizier

- Eine Klasse, z.B. *ROUTE*
- Ein Feature, z.B. *i\_th*
- Ein Laufzeitwert, wie etwa ein Objekt oder eine Objektreferenz, z.B. *Opera\_route*

Ein Bezeichner, der einen Laufzeitwert bezeichnet, wird **Entität** genannt

Entity

Eine Entität, dessen Wert sich ändern kann, ist eine **Variable**

Variable

Während der Ausführung können Entitäten an Objekte **gebunden** sein

Attached

# An ein Objekt gebundene Entitäten

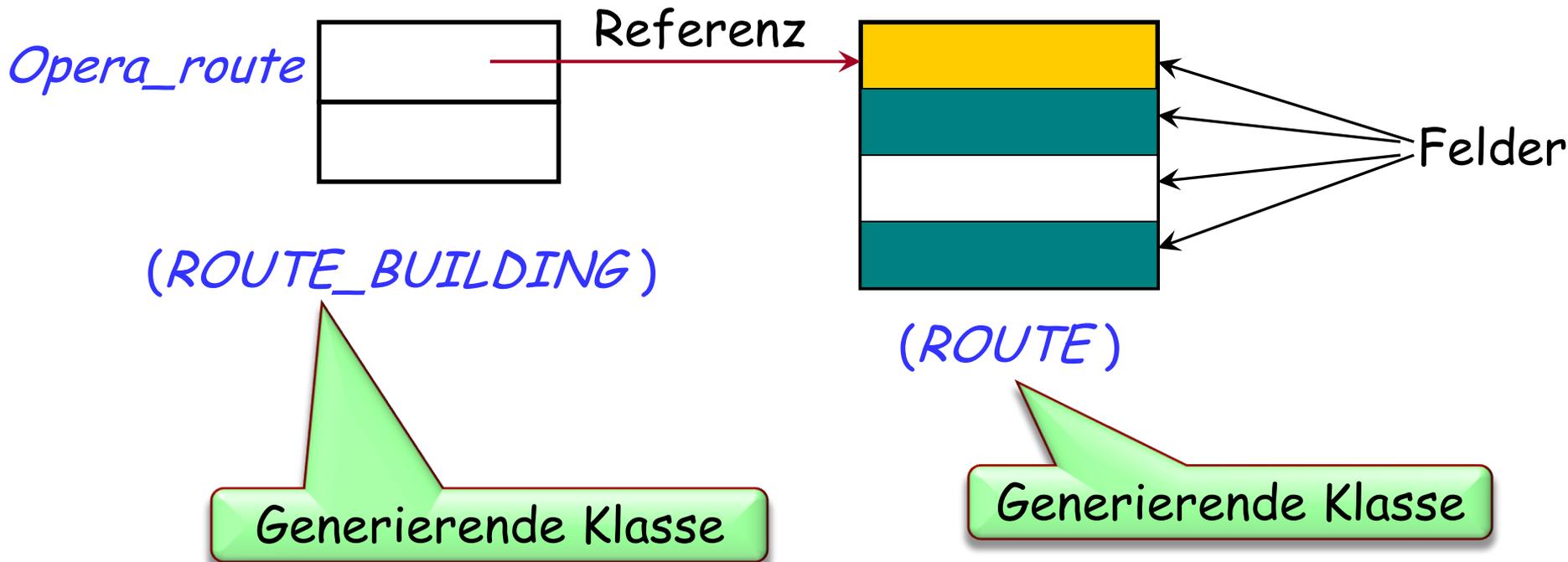


Im Programm: eine Entität, wie z.B. *Opera\_route*

Im Speicher, während der Ausführung: ein Objekt

OBJEKT

OBJEKT



# ROUTE\_BUILDING

---



```
class ROUTE_BUILDING inherit
  ZURICH_OBJECTS
```

```
feature
```

```
  build_route
```

```
    -- Eine Route bauen und damit arbeiten.
```

```
  do
```

```
    -- "Opera_route erzeugen und Teilstrecken hinzufügen"
```

```
    Zurich.add_route (Opera_route)
```

```
    Opera_route.reverse
```

```
    -- "Weiter mit Opera_route arbeiten"
```

```
  end
```

```
Opera_route: ROUTE
```

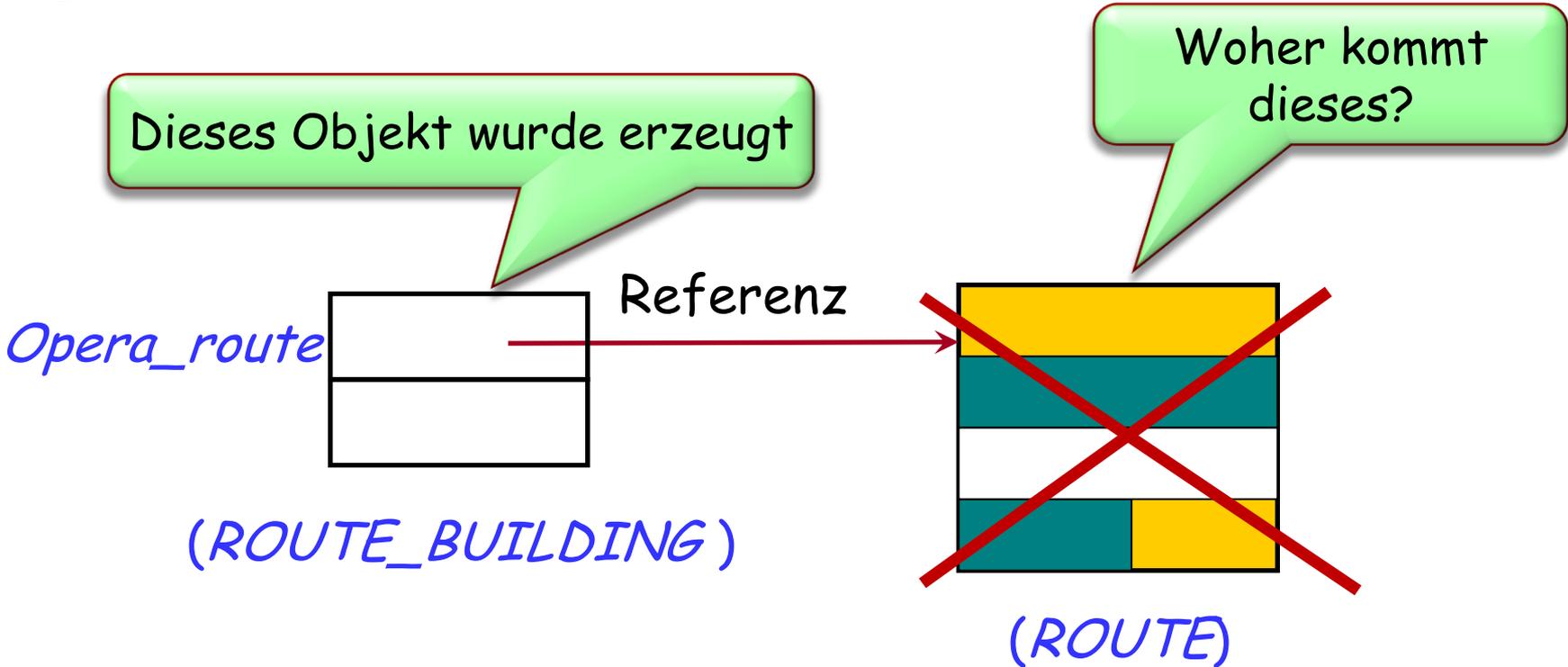
```
    -- Eine Route von Polyterrasse nach Opernhaus.
```

```
end
```

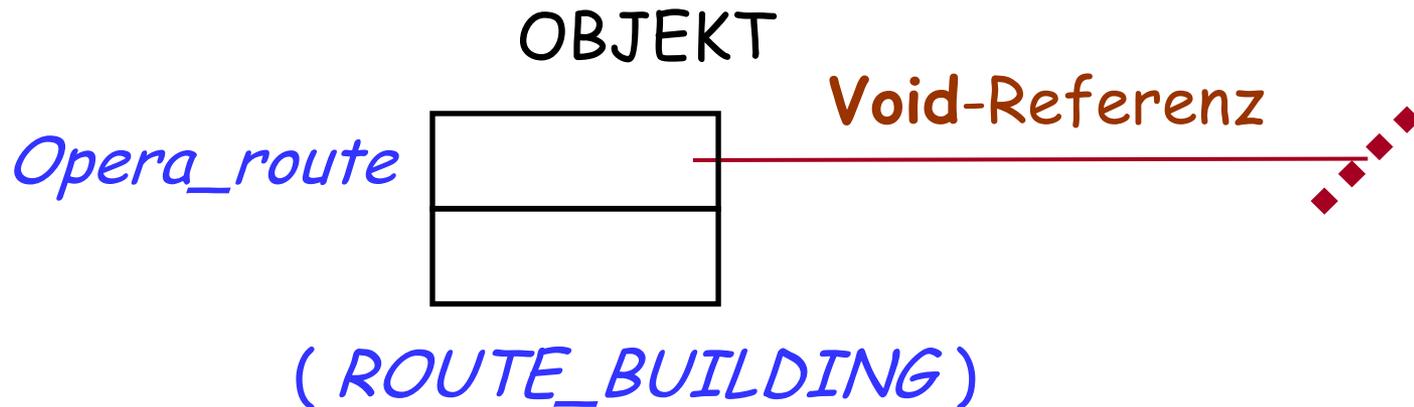
# Der Grundzustand einer Referenz



Können wir in einer Instanz von *ROUTE\_BUILDING* annehmen, dass *Opera\_route* an eine Instanz von *ROUTE* gebunden ist?



Anfangs ist *Opera\_route* nicht an ein Objekt gebunden:  
Es ist eine **Void**-Referenz.



# Warum müssen wir Objekte erzeugen?

---



Können wir nicht annehmen, dass eine Deklaration der Form

*Opera\_route: ROUTE*

eine Instanz von *ROUTE* erzeugt und sie an *Opera\_route* bindet?

(Die Antwort darauf folgt bald...)



Zur Laufzeit ist eine Referenz

- Entweder an ein gewisses Objekt **gebunden**
- Oder **void**

- Um eine Void-Referenz zu bezeichnen: Benutzen sie das reservierte Wort **Void**
- Um herauszufinden, ob **x** void ist, können sie folgende Abfrage benutzen:

**x = Void**

- Die inverse Abfrage (ist **x** an ein Objekt gebunden?):

**x != Void**

# Das Problem mit Void-Referenzen



Der Grundmechanismus von (O-O) Programmen ist der **Featureaufruf**

Feature  $f$  anwenden

$x.f(a, \dots)$

Evtl. mit Argumenten

Auf das an  $x$  gebundene Objekt

Da Referenzen void sein können, kann  $x$  möglicherweise an kein Objekt gebunden sein

**In solchen Fällen ist der Aufruf fehlerhaft!**

# Beispiel: Aufruf auf ein Ziel, das void ist



```
class ROUTE_BUILDING inherit  
  ZURICH_OBJECTS
```

```
feature
```

```
  build_route
```

```
    -- Eine Route bauen und damit arbeiten.
```

```
  do
```

```
    -- "Opera_route erzeugen und Teilstrecken hinzufügen"
```

```
    Zurich.add_route (Opera_route)
```

```
    Opera_route.reverse
```

```
  end
```

Void-Referenz

```
Opera_route: ROUTE
```

```
  -- Eine Route von Polybahn nach Opernhaus.
```

```
end
```

Beispiele von abnormale Ereignisse (*failures*) während der Ausführung:

- "Void call": *Opera\_route.reverse*, wobei *Opera\_route* void ist
- Der Versuch,  $a / b$  auszurechnen, wobei  $b$  den Wert 0 hat

Die Konsequenz ist ein Abbruch; es sei denn, das Programm hat Code, um sich von der Ausnahme zu „erholen“ ("**rescue**" Klausel in Eiffel, "**catch**" in Java)

Jede Ausnahme hat einen **Typ**, der in den Laufzeit-Fehlermeldungen von EiffelStudio ersichtlich ist, z.B.

- **Feature call on void target**
- **Arithmetic underflow**



Um eine Ausnahme zu vermeiden:

- Verändern Sie die Prozedur *build\_route*, so dass sie ein Objekt erzeugt und an *Opera\_route* bindet, bevor es *reverse* aufruft

# Void-Aufrufe – Die ganze Geschichte

---



In ISO Eiffel werden Void-Aufrufe dank dem Begriff des „gebundenen Typs“ (attached type) nicht mehr vorkommen

Der Compiler lehnt jeden Aufruf  $x.f$  ab, bei dem  $x$  in einer Ausführung void sein könnte

Dies ist ein grosser Fortschritt, der allerdings auch Kompatibilitätsprobleme für bereits existierenden Code mit sich führt. Deshalb wird dies schrittweise ab EiffelStudio 6.2 eingeführt.

Andere Sprachen kennen dies nicht, aber Spec#, eine auf C# basierende Sprache aus der Forschungsabteilung von Microsoft, spielte eine Vorreiterrolle mit ihren „non-null types“

In diesem Kurs benutzen wir immer noch die alten Regeln



B. Meyer, E. Stapf, A. Kogtenkov: *Avoid a Void: The Eradication of Null Dereferencing*, in *Reflections on the Work of C.A.R. Hoare*, 2010, auf

[docs.eiffel.com/book/papers/void-safety-how-eiffel-removes-null-pointer-dereferencing](https://docs.eiffel.com/book/papers/void-safety-how-eiffel-removes-null-pointer-dereferencing)

oder

<http://bit.ly/8wxXLO>

verfügbar

# Warum müssen wir Objekte erzeugen?

---



Können wir nicht annehmen, dass eine Deklaration der Form

*Opera\_route: ROUTE*

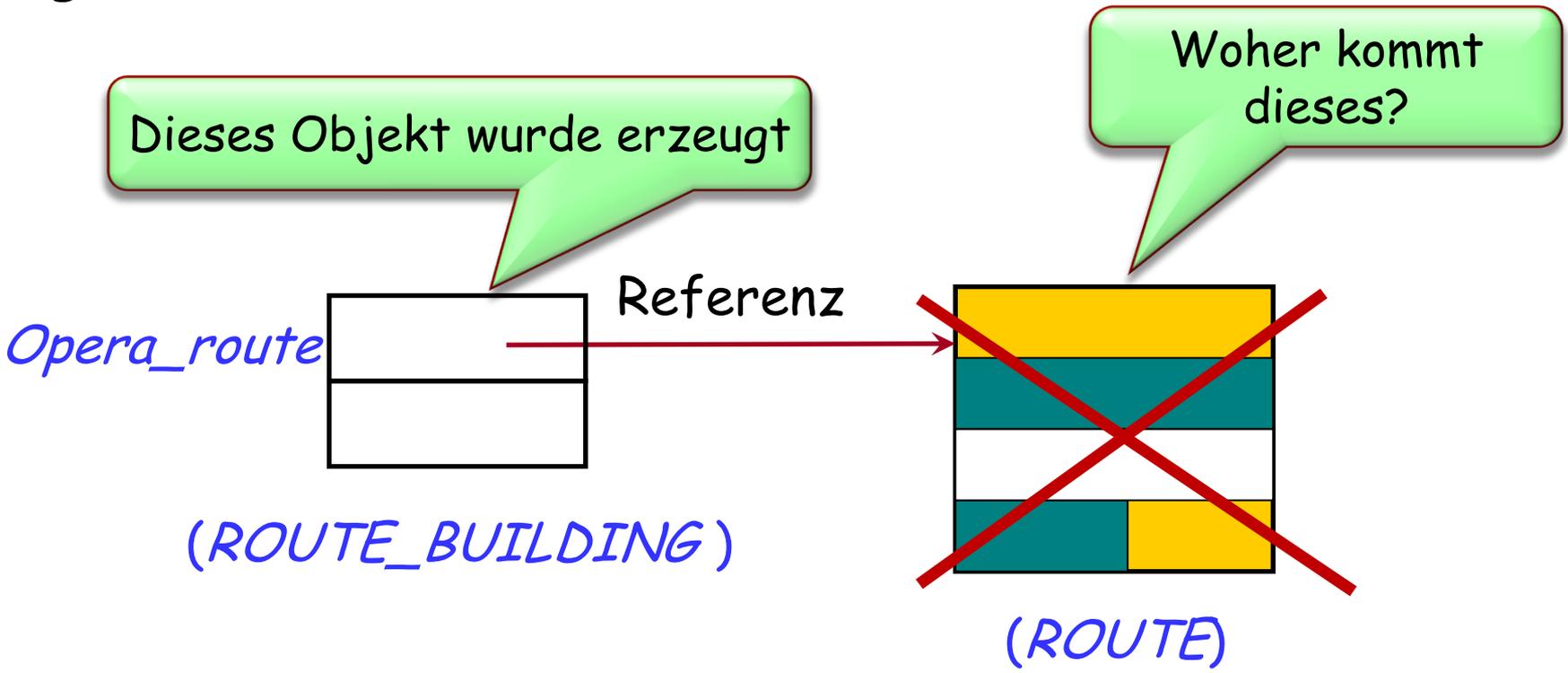
eine Instanz von *ROUTE* erzeugt und sie an *Opera\_route* bindet?

(Die Antwort darauf folgt bald...)

# Der Grundzustand einer Referenz



Können wir in einer Instanz von *ROUTE\_BUILDING* annehmen, dass *Opera\_route* an eine Instanz von *ROUTE* gebunden ist?



# ROUTE\_BUILDING

---



```
class ROUTE_BUILDING inherit
  ZURICH_OBJECTS
```

```
feature
```

```
  build_route
```

```
    -- Eine Route bauen und damit arbeiten.
```

```
  do
```

```
    -- "Opera_route erzeugen und Teilstrecken hinzufügen"
```

```
    Zurich.add_route (Opera_route)
```

```
    Opera_route.reverse
```

```
  end
```

```
Opera_route : ROUTE
```

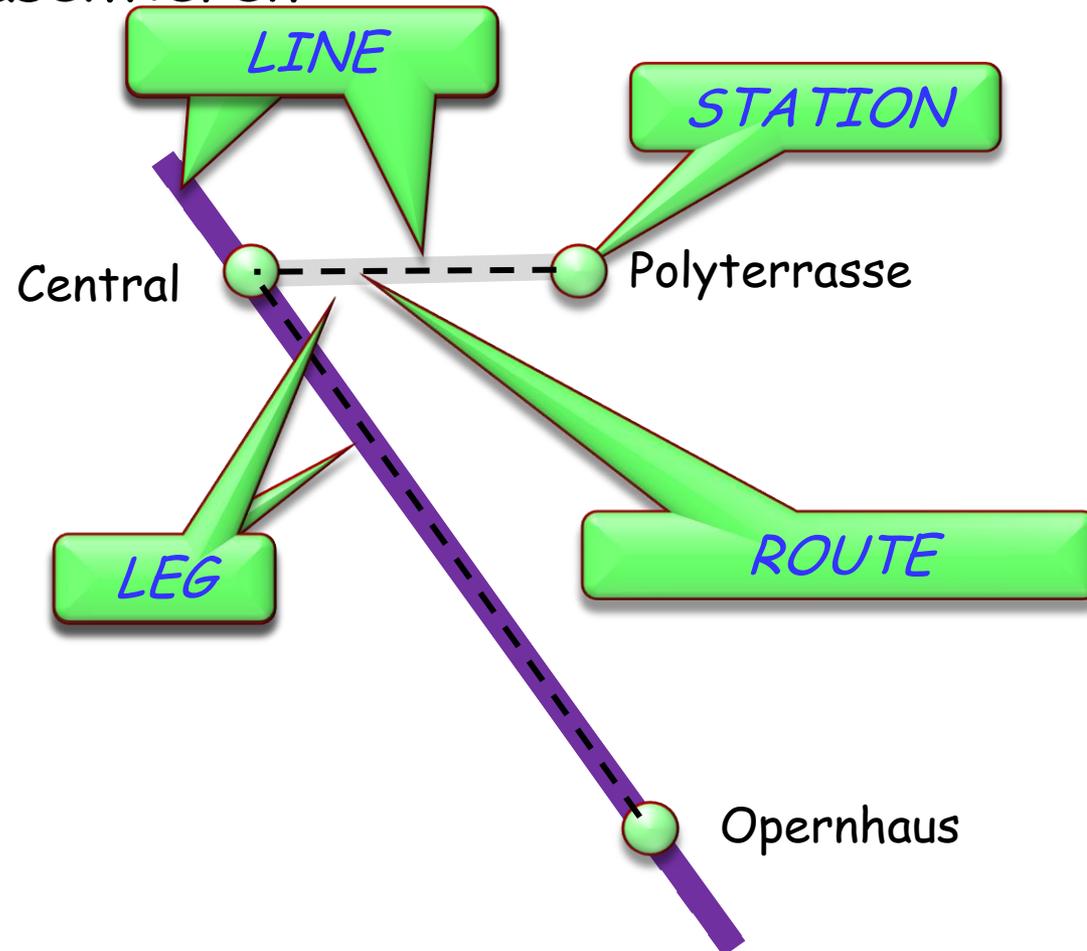
```
  -- Eine Route von Polybahn nach Opernhaus
```

```
end
```

# Einfache Objekte erzeugen

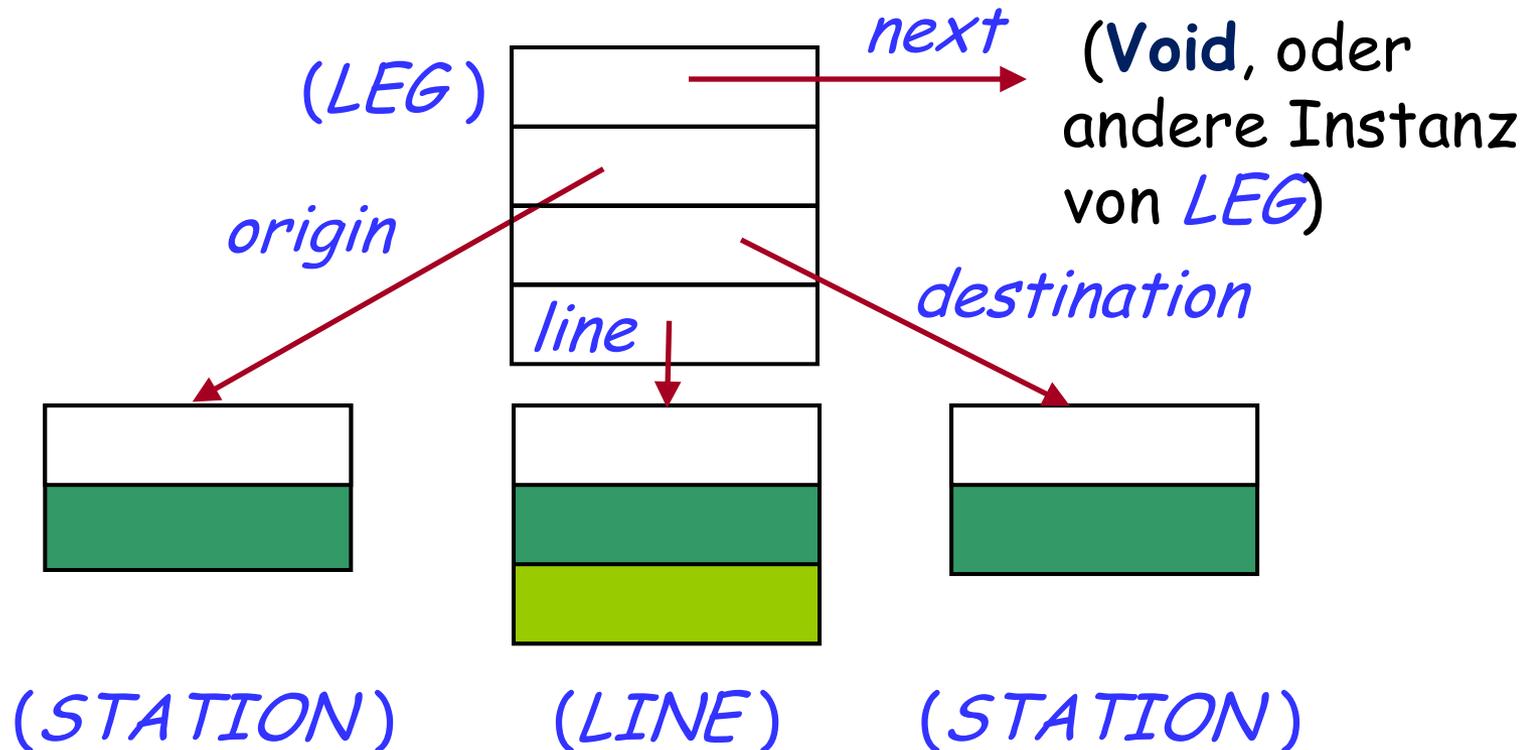


Um *Opera\_route* zu erzeugen, müssen wir zuerst Objekte erzeugen, welche die Teilstrecken (*legs*), sowie Linien und Stationen repräsentieren



Eine Instanz der Klasse *LEG* hat:

- Referenzen zu einer Anfangstation, Endstation und einer Linie; dürfen nicht **Void** sein
- Eine Referenz zur nächsten Teilstrecke; **Void** am Schluss



# Die Schnittstelle der Klasse *LEG* (erste Version)



```
class LEG feature
  origin, destination: STATION
    -- Anfang und Ende der Teilstrecke.

  line: LINE
    -- Die Linie, der die Strecke angehört.

  next: LEG
    -- Nächste Teilstrecke in der Route.

  make(o, d: STATION; l: LINE)
    -- Anfang, Ende und Linie setzen.
    require
      linie_existiert: /= Void
      auf_der_linie: l.has_station(o) and l.has_station(d)
    ensure
      stationen_gesetzt: origin = o and destination = d
      linie_gesetzt: line = l

  link(other: LEG)
    -- other zur nächsten Teilstrecke der Route machen.
    ensure
      naechste_gesetzt: next = other

end
```

# ROUTE\_BUILDING



```
class ROUTE_BUILDING inherit  
  ZURICH_OBJECTS
```

```
feature
```

```
  leg1: LEG
```

```
    -- Erste Teilstrecke der Opera_route.
```

```
  build_route
```

```
    -- Eine Route bauen und damit arbeiten.
```

```
  do
```

```
    Zurich.add_route (Opera_route)  
    Opera_route.reverse
```

```
  end
```

```
  Opera_route: ROUTE
```

```
    -- Eine Route von Polybahn nach Opernhaus.
```

```
end
```

# Eine Instanz von *LEG* erzeugen



```
class ROUTE_BUILDING inherit  
  ZURICH_OBJECTS
```

```
feature
```

```
  leg1: LEG
```

```
    -- Erste Teilstrecke der Opera_route.
```

```
  build_route
```

```
    -- Eine Route bauen und damit arbeiten.
```

```
  do
```

Erzeugungsinstruktion

```
    -- Erzeuge Opera_route und fülle sie mit
```

```
    create leg1
```

```
    -- "Mehr Teilstrecken erzeugen und Opera_route fertig bauen"
```

```
    Zurich.add_route(Opera_route)
```

```
    Opera_route.reverse
```

```
  end
```

```
  Opera_route: ROUTE
```

```
    -- Eine Route von Polybahn nach Opernhaus.
```

```
end
```

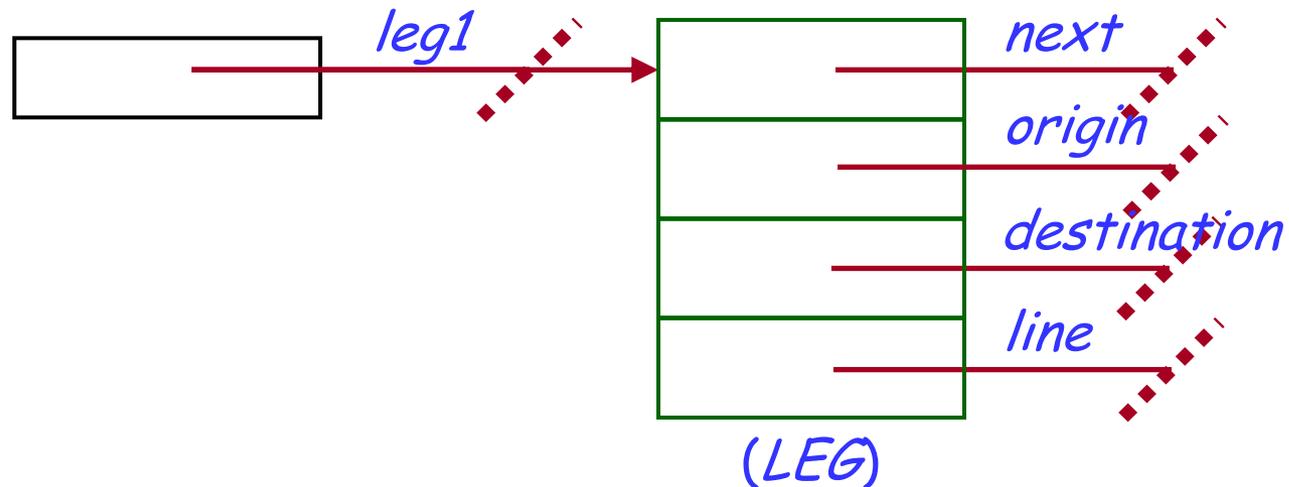
Jetzt ein  
gewöhnlicher  
Kommentar

Neuer Pseudocode

Grundoperation, um Objekte während der Laufzeit zu erzeugen:

- Erzeugt ein neues Objekt im Speicher
- Bindet eine Entität daran

*create leg1*





Jede Entität ist mit einem gewissen Typ **deklariert**:

*leg1: LEG*

Eine Erzeugungsinstruktion

*create leg1*

produziert während der Laufzeit ein Objekt dieses Typs

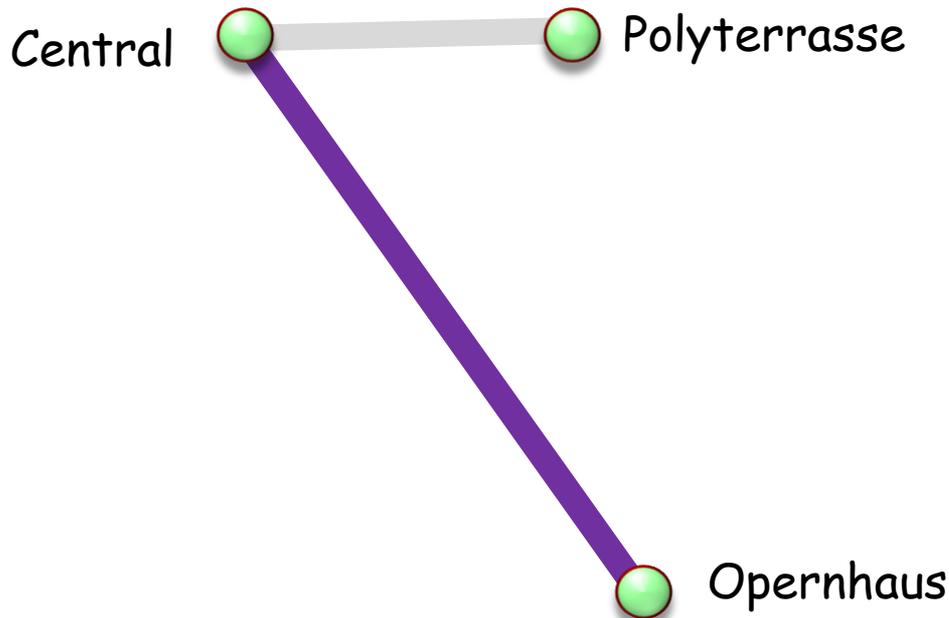
# Eine Route mit zwei Teilstrecken



Wir möchten jetzt, dass *leg1* von Polyterrasse nach Central geht, und eine weitere Teilstrecke nach Opernhaus fortfährt

Als Erstes deklarieren wir die entsprechenden Attribute:

*leg1, leg2: LEG*



# build\_route



*build\_route*

-- Eine Route bauen und damit arbeiten.

**do**

-- Erzeuge die Teilstrecken und setze die Stationen und die Linien.

**create** *leg1*

*leg1* • **make** (*Polyterrasse, Central, Polybahn*)

Vordefiniert in  
*ZURICH\_OBJECTS*

**create** *leg2*

*leg2* • **make** (*Central, Opernhaus, Line4*)

-- Verbinde die erste Teilstrecke mit der zweiten.

*leg1* • **link** (*leg2*)

-- "*Opera\_route* erzeugen und ihr die eben erstellten Teilstrecken zuweisen"

*Zurich* • **add\_route** (*Opera\_route*)

*Opera\_route* • **reverse**

**end**

Immer noch Pseudocode!

# Nochmals: warum müssen wir Objekte erzeugen? ☹

Können wir nicht annehmen, dass eine Deklaration der Form

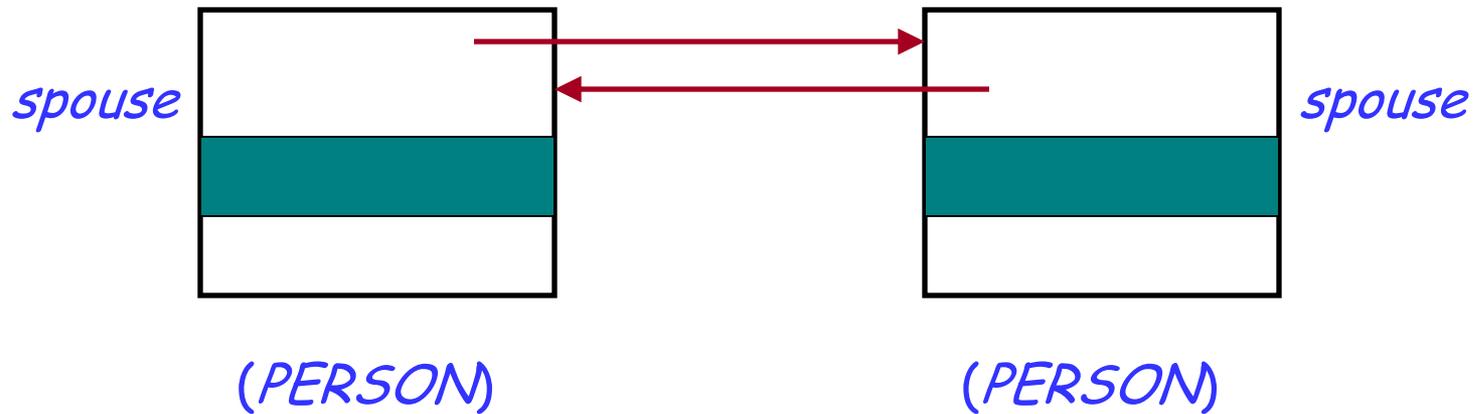
*Opera\_route: ROUTE*

eine Instanz von *ROUTE* erzeugt und sie an *Opera\_route* bindet?

# Void-Referenzen sind nützlich!



Verheiratete Personen:

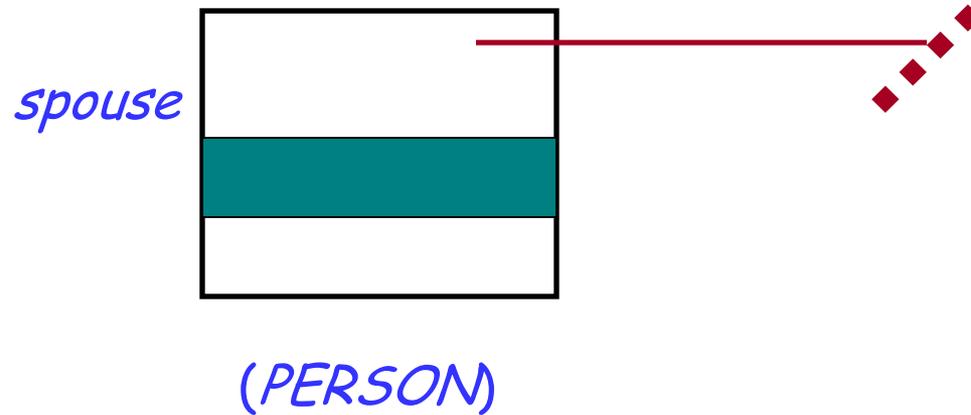


# Void-Referenzen sind nützlich!

---



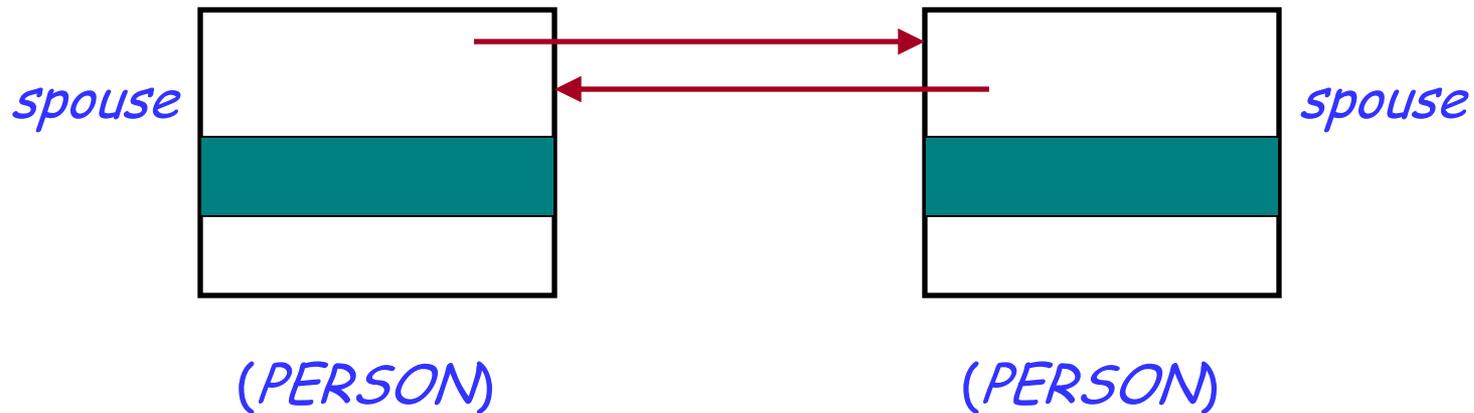
Ledige Personen:



# Void-Referenzen sind nützlich!

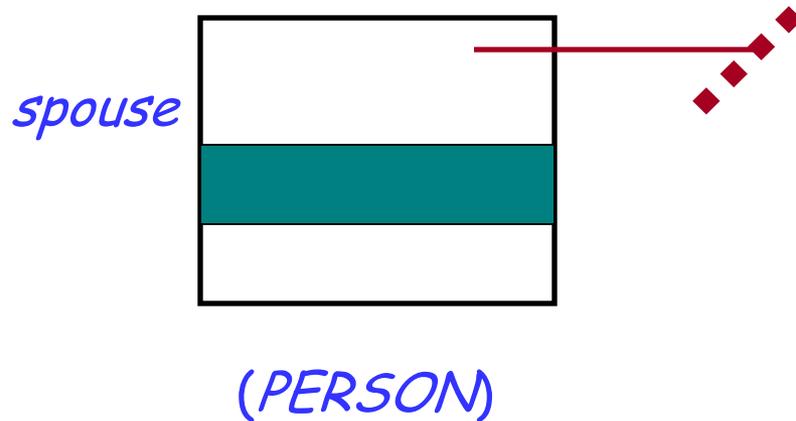


Auch bei verheirateten Personen...

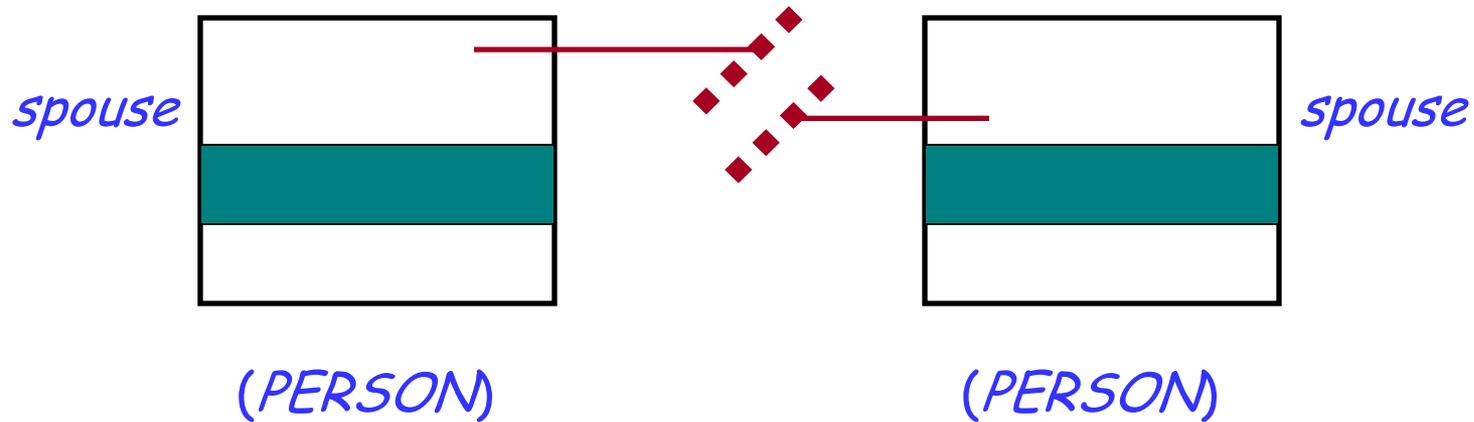


... sollten wir nicht jedes Mal, wenn wir eine Instanz von `PERSON` erzeugen, auch ein Objekt für `spouse` erzeugen (**Warum?**)

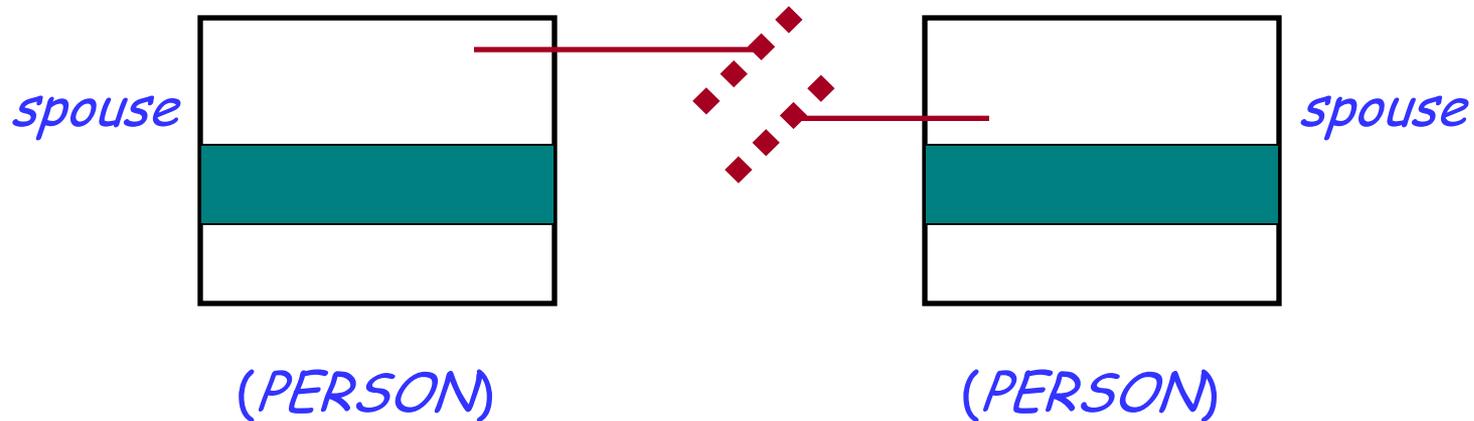
Jedes *PERSON*-Objekt wird mit einer Void-Referenz *spouse* erzeugt



Jedes *PERSON*-Objekt wird mit einer Void-Referenz *spouse* erzeugt

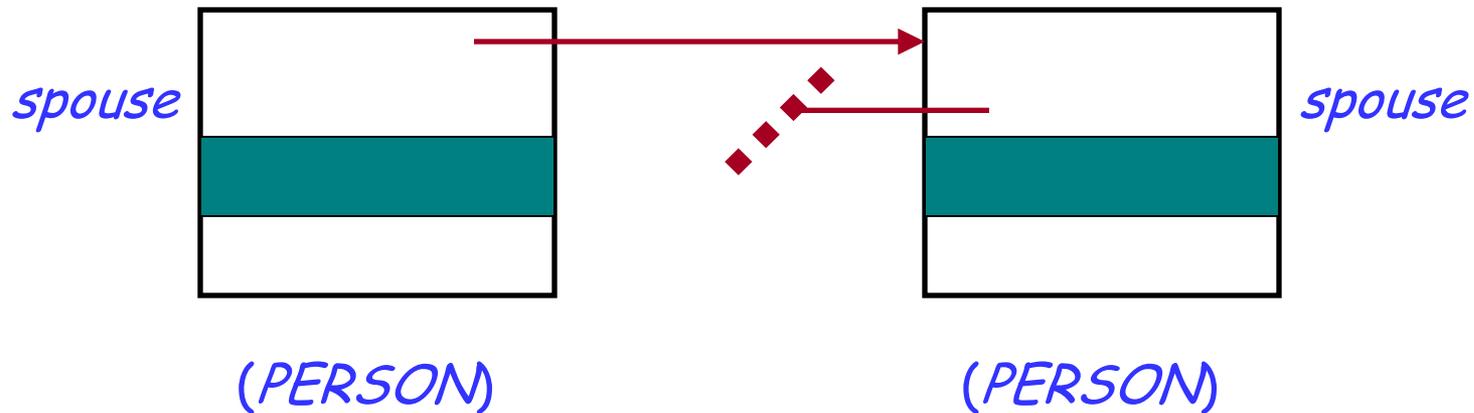


Jedes *PERSON*-Objekt wird mit einer Void-Referenz *spouse* erzeugt...



... und danach werden die *spouse* - Referenzen durch entsprechende Instruktionen gebunden

Jedes *PERSON*-Objekt wird mit einer Void-Referenz *spouse* erzeugt...

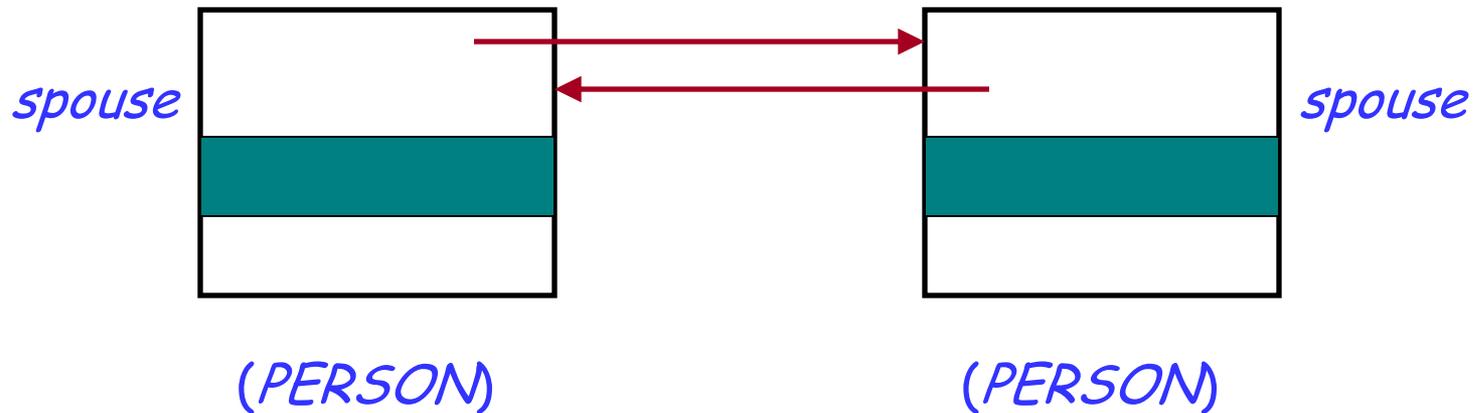


... und danach werden die *spouse* - Referenzen durch entsprechende Instruktionen gebunden

# Der Gebrauch von Void-Referenzen



Jedes *PERSON*-Objekt wird mit einer Void-Referenz *spouse* erzeugt...



... und danach werden die *spouse* - Referenzen durch entsprechende Instruktionen gebunden

# (Nochmals) Die Schnittstelle der Klasse *LEG*



**class** *LEG* **feature**

*origin, destination: STATION*

-- Anfang und Ende der Teilstrecke.

*line: LINE*

-- Die Linie, der die Strecke angehört.

*next: LEG*

-- Nächste Teilstrecke in der Route.

*make(o, d: STATION; l: LINE)*

-- Anfang, Ende und Linie setzen.

**require**

*linie\_existiert: /= Void*

*auf\_der\_linie: l.has\_station(o) and l.has\_station(d)*

**ensure**

*stationen\_gesetzt: origin = o and destination = d*

*linie\_gesetzt: line = l*

*link(other: LEG)*

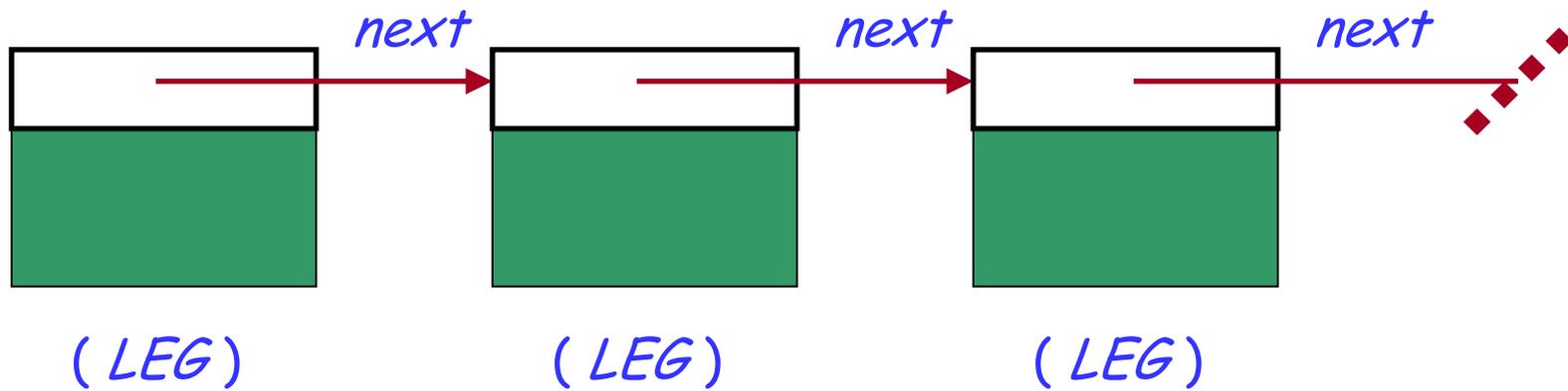
-- *other* zur nächsten Teilstrecke der Route machen.

**ensure**

*naechste\_gesetzt: next = other*

**end**

# Referenzen in verketteten Strukturen



Die Liste wird durch eine *next*-Referenz, die void ist, beendet

# build\_route



## build\_route

-- Eine Route bauen und sie auf der Karte hervorheben.

do

-- Erzeuge die Teilstrecken und setze die Stationen und die Linien:

create leg1

leg1.make (Polyterrasse, Central, Polybahn)

create leg2

leg2.make (Central, Opernhaus, Line4)

-- Verbinde die erste Teilstrecke mit der zweiten.

leg1.link(leg2)

-- "Opera\_route erzeugen und ihr die eben erstellten Teilstrecken zuweisen"

Zurich.add\_route (Opera\_route)

Opera\_route.reverse

end

Vordefiniert in  
*ZURICH\_OBJECTS*

Immer noch Pseudocode!

# Die Schnittstelle der Klasse *LEG* (erste Version)



```
class LEG feature
  origin, destination: STATION
    -- Anfang und Ende der Teilstrecke.

  line: LINE
    -- Die Linie, der die Strecke angehört.

  next: LEG
    -- Nächste Teilstrecke in der Route.

  make(o, d: STATION; l: LINE)
    -- Anfang, Ende und Linie setzen.
    require
      linie_existiert: /= Void
      auf_der_linie: l.has_station(o) and l.has_station(d)
    ensure
      stationen_gesetzt: origin = o and destination = d
      linie_gesetzt: line = l

  link(other: LEG)
    -- other zur nächsten Teilstrecke der Route machen.
    ensure
      naechste_gesetzt: next = other

end
```

# Der Invariant der Klasse *LEG*

---



linie\_existiert: *line* != **Void**

anfang\_auf\_der\_linie: *line.has\_station(origin)*

ende\_auf\_der\_linie: *line.has\_station(destination)*

Erzeugung und Initialisierung eines *LEG* Objektes:

```
create leg  
leg.make(station1, station2, line)
```

Nach der Erzeugung:  
Invariante nicht erfüllt!

Die Invariante der Klasse:

**invariant**

linie\_existiert: *line*  $\neq$  **Void**

anfang\_auf\_der\_linie: *line.has\_station(origin)*

ende\_auf\_der\_linie: *ine.has\_station(destination)*



Eine bessere Lösung:

- Deklarieren sie *make* als **Erzeugungsprozedur**, um Initialisierung mit Erzeugung zu verbinden:

```
create leg1.make (Polyterrasse, Central, Polybahn)
```

-- Gleicher Effekt wie die zwei vorherigen Instruktionen

- **Einfachheit**: Initialisierung bei Erzeugung
- **Korrektheit**: Die Invariante wird von Anfang an erfüllt.

Erzeugungsprozeduren heissen auch **Konstruktoren**  
(z.B. in Java oder C#)

# Die Schnittstelle der Klasse *LEG*



```
class LEG create  
  make  
feature
```

```
  origin, destination: STATION  
    -- Anfang und Ende der Teilstrecke.  
  line: LINE  
    -- Linie, der die Strecke angehört.  
  next: LEG  
    -- Nächste Teilstrecke der Route.
```

Auflistung der Erzeugungsprozeduren

Jetzt auch als Erzeugungsprozedur verfügbar

```
  make (o, d: STATION; l: LINE)  
    -- Anfang, Ende und Linie setzen.  
    require  
      linie_existiert: /= Void  
      auf_der_linie: l.has_station(o) and l.has_station(d)  
    ensure  
      stationen_gesetzt: origin = o and destination = d  
      linie_gesetzt: line = l
```

```
  link (other: LEG)  
    -- other zur nächsten Teilstrecke der Route machen.  
    ensure  
      naechste_gesetzt: next = n
```

```
  invariant  
    linie_existiert: line /= Void  
    auf_der_linie: line.has_station(origin) and line.has_station(destination)  
end
```



Falls eine Klasse eine nicht-triviale Invariante hat, muss sie eine oder mehrere Erzeugungsprozeduren definieren, die sicherstellen, dass jede Instanz nach der Ausführung einer Erzeugungsinstruktion die Invariante erfüllt

Dies erlaubt dem Autor der Klasse, zu erzwingen, dass alle von Klienten erzeugten Instanzen korrekt initialisiert werden

# Erzeugungsprozeduren



Auch wenn keine starke Invariante vorhanden ist, sind Erzeugungsprozeduren nützlich, um Initialisierung und Erzeugung zu kombinieren

```
class POINT create
```

```
    default_create, make_cartesian, make_polar
```

```
feature
```

```
    ...
```

```
end
```



Vererbt an alle Klassen, macht standardmässig nichts

Gültige Erzeugungsinstruktionen:

```
create your_point.default_create
```

```
create your_point
```

```
create your_point.make_cartesian(x, y)
```

```
create your_point.make_polar(r, t)
```

Um ein Objekt zu erzeugen:

- Falls die Klasse keine **create**-Klausel hat, benutzen sie die Grundform: **create x**
- Falls die Klasse eine **create**-Klausel hat, die eine oder mehrere Prozeduren auflistet, benutzen sie

**create x.make (...)**

wobei **make** eine der Erzeugungsprozeduren ist und "**(...)**" für allfällige Argumente steht



Um mit dem Prinzip „Design by Contract“ („Entwurf gemäss Vertrag“) übereinzustimmen, müssen wir von jeder Instruktion genau wissen:

- Wie man die Instruktion richtig benutzt: die **Vorbedingung**
- Was wir dafür garantiert bekommen: die **Nachbedingung**

Zusammen definieren diese beiden Eigenschaften (zusammen mit der Invarianten) die **Korrektheit** eines Sprachmechanismus

Wie lautet die Korrektheitsregel für eine Erzeugungsinstruktion?



## Korrektheitsregel für Erzeugungsinstruktionen

*Vor* der Erzeugungsinstruktion:

1. Die Vorbedingung der Erzeugungsprozedur (falls vorhanden) muss erfüllt sein

*Nach* der Erzeugungsinstruktion mit dem Ziel  $x$  vom Typ  $C$ :

2.  $x \neq \text{Void}$  gilt
3. Die Nachbedingung der Erzeugungsprozedur ist erfüllt
4. Das an  $x$  gebundene Objekt erfüllt die Invariante von  $C$

# Aufeinanderfolgende Erzeugungsinstruktionen



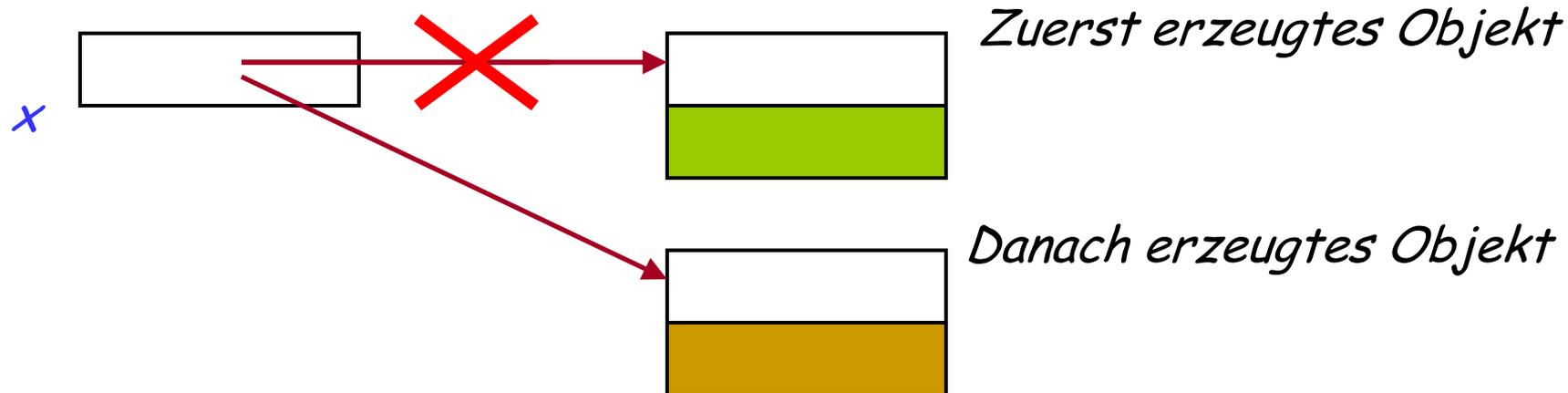
Die Korrektheitsregel erfordert nicht, dass  $x$  void ist:

create  $x$

-- Hier ist  $x$  nicht void.

...

create  $x$





- $x$  ist nach der Erzeugungsinstruktion nicht void (egal ob es vorher void war oder nicht)
- $x$  ist an dem eben erzeugten Objekt Objekt gebunden
- Falls es eine Erzeugungsprozedur gibt, ist ihre Nachbedingung für das neue Objekt erfüllt
- Das Objekt erfüllt die Klasseninvariante



Die Ausführung eines Systems beginnt mit der Erzeugung eines **Wurzelobjektes**, das eine Instanz einer vom System vorgesehenen Klasse (die **Wurzelklasse**) ist, mittels einer definierten Erzeugungsprozedur dieser Klasse (die **Wurzelerzeugungprozedur** oder einfach **Wurzelprozedur**)

Eine Wurzelprozedur kann:

- Neue Objekte erzeugen
- Auf diesen Features aufrufen, die wiederum neue Objekte erzeugen können
- Etc.

# Ausführung eines Systems



*Wurzelobjekt*



— *Wurzelprozedur*



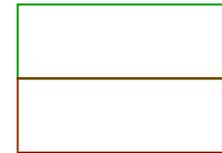
*create obj1.r1* *obj1*



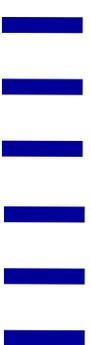
*r1*



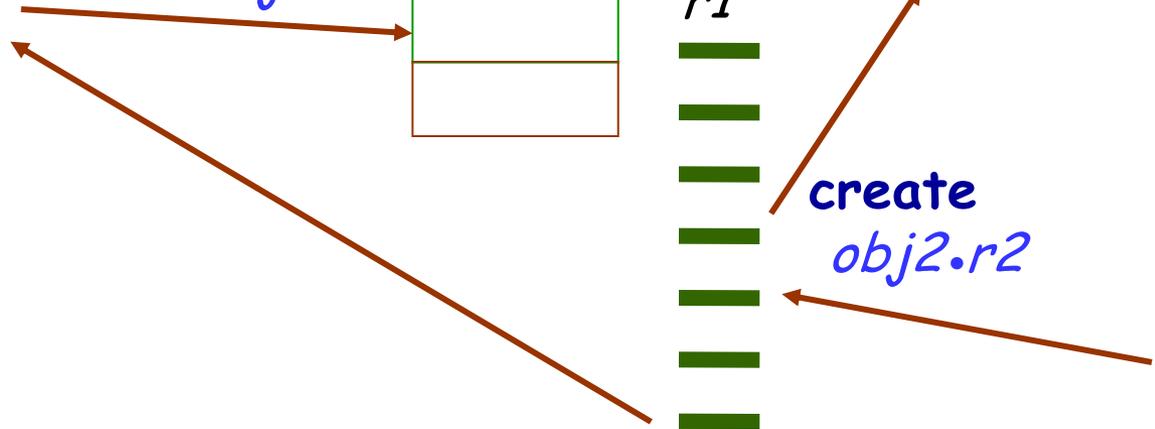
*obj2*



*r2*



*create obj2.r2*



# Das aktuelle Objekt (current object)

---



Zu jeder Zeit während einer Ausführung gibt es ein **aktuelles Objekt**, auf welchem das aktuelle Feature aufgerufen wird

Zu Beginn ist dies das Wurzelobjekt

Während eines „qualifizierten“ Aufrufs  $x.f(a)$ , ist das neue aktuelle Objekt dasjenige, das an  $x$  gebunden ist

Nach einem solchen Aufruf übernimmt das vorherige aktuelle Objekt wieder diese Rolle



Ein System ist eine bestimmte Gruppe von gewissen Klassen, wobei eine Klasse als Wurzelklasse dient

Die Klassen können auch ausserhalb des Systems wertvoll sein: sie sind dann **wiederverwendbar**



- **Erweiterbarkeit:** Die Einfachheit, mit welcher es möglich ist, ein System den Wünschen eines Benutzers entsprechend anzupassen
- **Wiederverwendbarkeit:** Die Einfachheit, bestehende Software für neue Applikationen wiederzuverwenden

Ältere Software-Engineering-Ansätze, die auf einem Hauptprogramm und Unterprogrammen beruhen, beachten diese Bedürfnisse weniger



Wie spezifiziert man die **Wurzelklasse** und die **Wurzel-Erzeugungsprozedur** eines Systems?

Benutzen sie EiffelStudio



- Klasseninvarianten
- Das Konzept "Design by Contract"
- Den Begriff einer Ausnahme
- Objekterzeugung
- Erzeugungsprozeduren
- Die Beziehung zwischen Erzeugungsprozeduren und Invarianten
- Der Vorgang einer Systemausführung