Kapitel 6: Kontrollstrukturen im Kontext von Klassen und Objekten

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 21. November 2018

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Überblick und Ziele

- Anweisungen im Kontext von Klassen und Objekten
- Verstehen der Methodenaufrufanweisung
- Semantik der Anweisungen
- Statische Attribute und Methoden
- Die Klasse `String`

Überblick Kapitel 3 bis Kapitel 6

Kapitel 3		**Kapitel 5**
		Klassendeklarationen
		Objekte und Objekthalde (Heap)
Grunddatentypen	erweitert um	Klassentypen
Werte	erweitert um	Referenzen und `null`
Operationen	erweitert um	`==`, `!=` für Referenzen und `null`
Ausdrücke (einschließlich Typisierung und Auswertung)	erweitert um	Attributzugriff, Methodenaufruf mit Ergebnis, Objekterzeugungsanweisung
Auswertung bezüglich Zustand (Stack)	erweitert um	Objekthalde (Heap)
Kapitel 4		**Kapitel 6**
Kontrollstrukturen	**erweitert um**	**Return-Anweisung, Methodenaufruf, Objekterzeugung**

Erweiterung der Syntax für Anweisungen

Syntax:

Statement	=	VariableDeclaration	(Deklarationsanweisung)
	|	Assignment	(Zuweisung)
	|	Block	(Block)
	|	Conditional	(Fallunterscheidung)
	|	Iteration	(Wiederholungsanweisung)
	|	ReturnStatement	(Return-Anweisung)
	|	MethodInvocation ";"	(Methodenaufrufanweisung)
	|	ClassInstanceCreation ";"	(Objekterzeugungsanweisung)

Deklarationsanweisungen und Zustandsänderungen

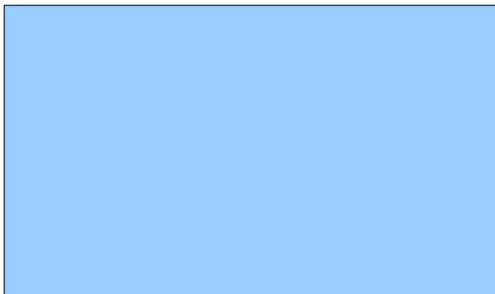
Die Deklarationsanweisungen

Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
```

führen zu folgendem Zustand



Stack

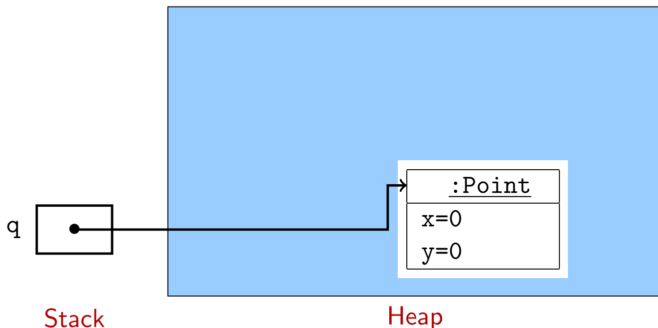
Heap

Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
```

führen zu folgendem Zustand

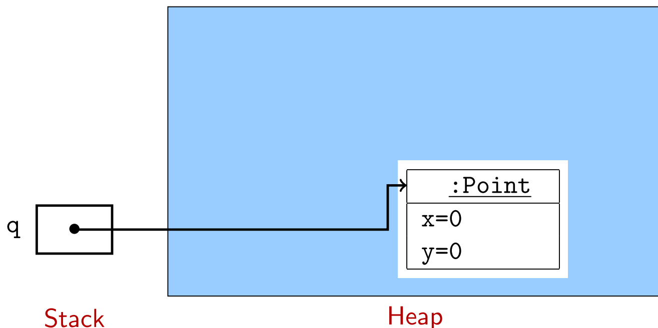


Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
int i = 3;
```

führen zu folgendem Zustand

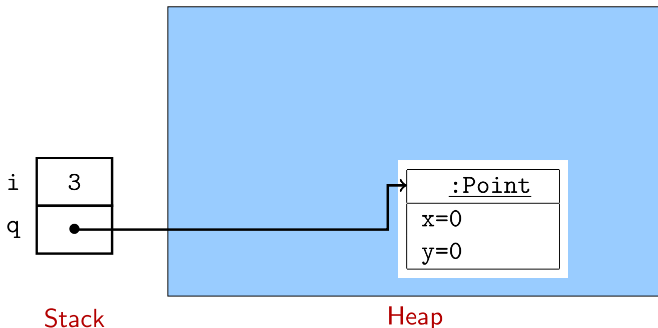


Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
int i = 3;
```

führen zu folgendem Zustand

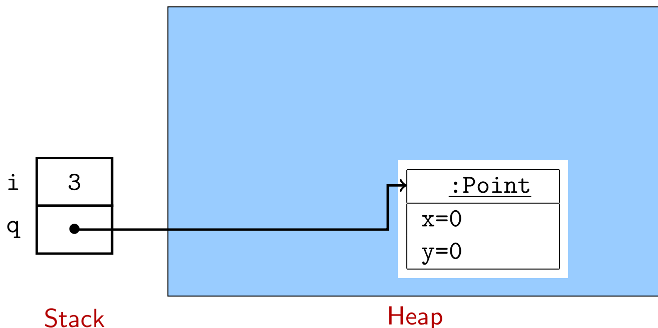


Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
int i = 3;
Point p = new Point(1,i);
```

führen zu folgendem Zustand

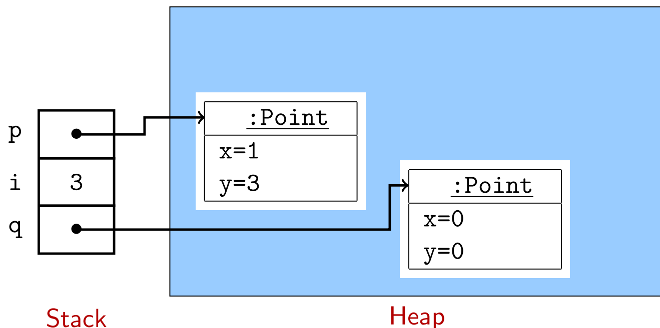


Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
int i = 3;
Point p = new Point(1,i);
```

führen zu folgendem Zustand

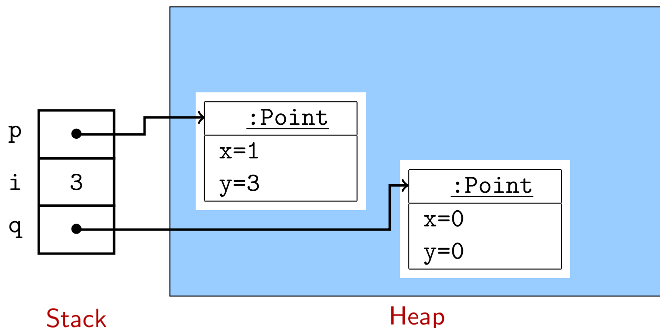


Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
int i = 3;
Point p = new Point(1,i);
boolean b = false;
```

führen zu folgendem Zustand

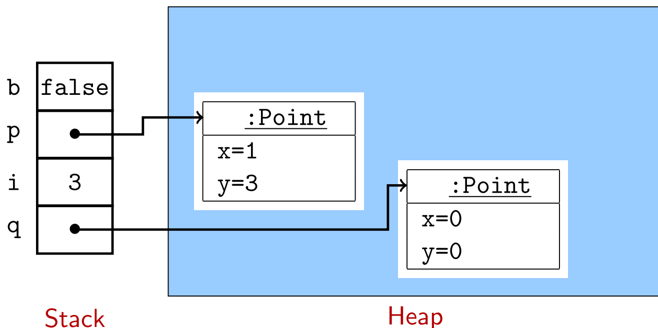


Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
int i = 3;
Point p = new Point(1,i);
boolean b = false;
```

führen zu folgendem Zustand

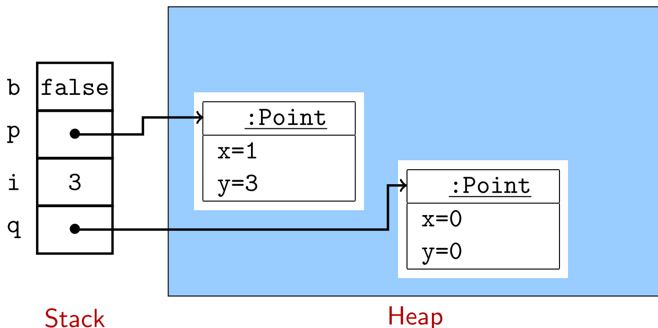


Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
int i = 3;
Point p = new Point(1,i);
boolean b = false;
Line ln = new Line(p,q);
```

führen zu folgendem Zustand

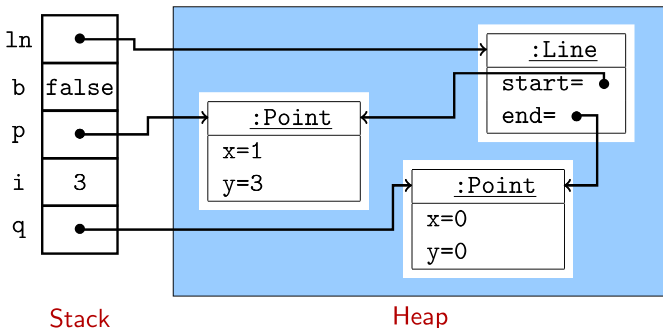


Deklarationsanweisungen und Zustandsänderungen

Die Deklarationsanweisungen

```
Point q = new Point(0,0);
int i = 3;
Point p = new Point(1,i);
boolean b = false;
Line ln = new Line(p,q);
```

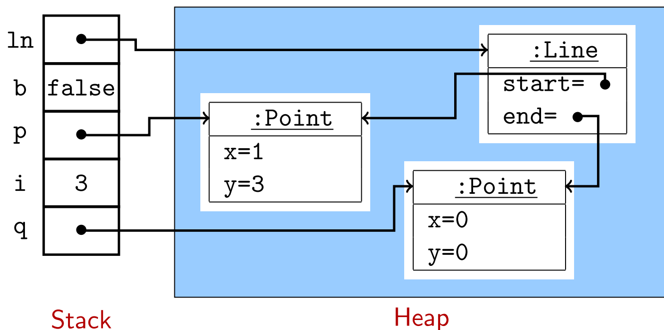
führen zu folgendem Zustand



Zuweisung und Zustandsänderungen

Die folgenden Zuweisungen auf dem Zustand der vorherigen Folie

führen zu folgendem Zustand

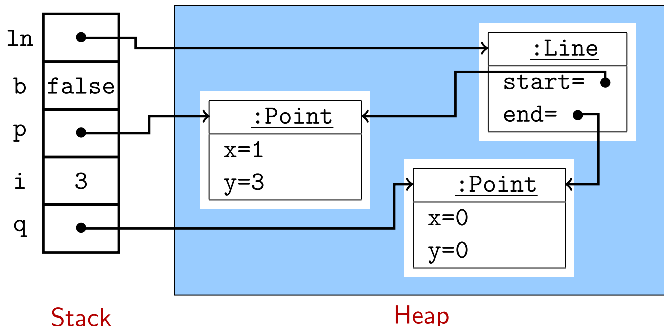


Zuweisung und Zustandsänderungen

Die folgenden Zuweisungen auf dem Zustand der vorherigen Folie

```
q = p; // Aliasing!
```

führen zu folgendem Zustand

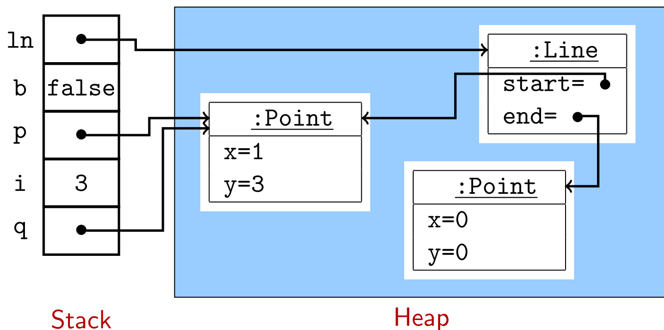


Zuweisung und Zustandsänderungen

Die folgenden Zuweisungen auf dem Zustand der vorherigen Folie

```
q = p; // Aliasing!
```

führen zu folgendem Zustand

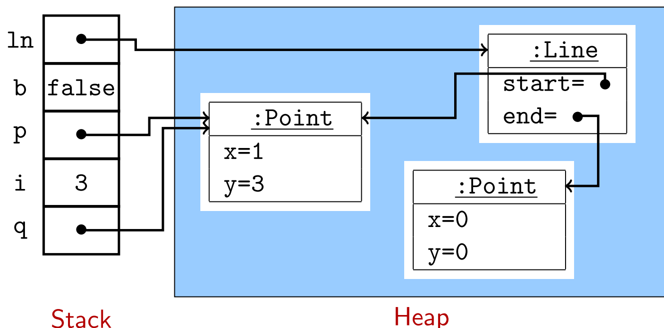


Zuweisung und Zustandsänderungen

Die folgenden Zuweisungen auf dem Zustand der vorherigen Folie

```
q = p; // Aliasing!
p.x = p.x + 1;
```

führen zu folgendem Zustand

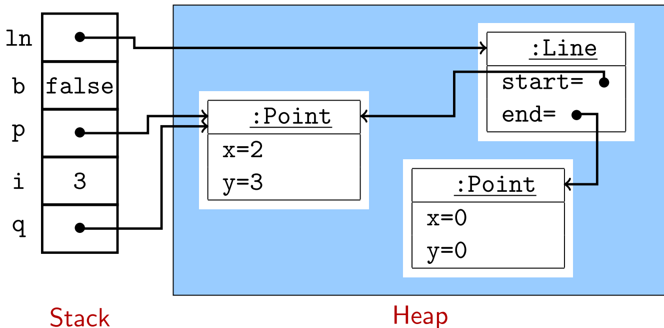


Zuweisung und Zustandsänderungen

Die folgenden Zuweisungen auf dem Zustand der vorherigen Folie

```
q = p; // Aliasing!
p.x = p.x + 1;
```

führen zu folgendem Zustand

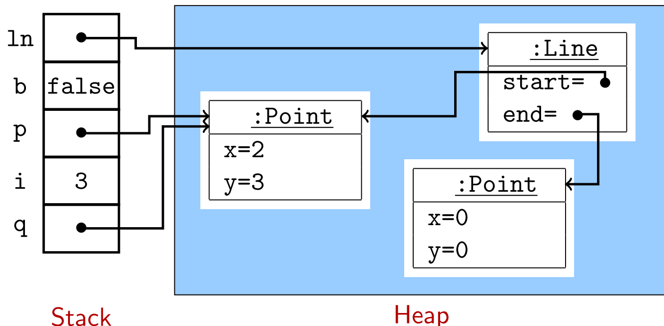


Zuweisung und Zustandsänderungen

Die folgenden Zuweisungen auf dem Zustand der vorherigen Folie

```
q = p; // Aliasing!
p.x = p.x + 1;
b = (q.getX() == 2);
```

führen zu folgendem Zustand

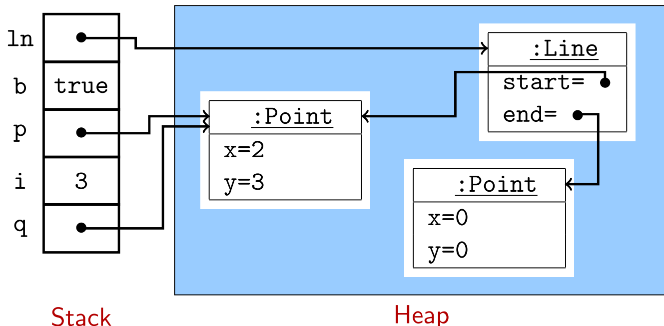


Zuweisung und Zustandsänderungen

Die folgenden Zuweisungen auf dem Zustand der vorherigen Folie

```
q = p; // Aliasing!
p.x = p.x + 1;
b = (q.getX() == 2);
```

führen zu folgendem Zustand



Aliasing (1)

Aliasing = Vorhandensein mehrerer Variablen,
die auf den gleichen Speicherort verweisen.

Im Beispiel

```
q = p;
```

sind `p` und `q` Referenzen auf dasselbe Objekt.

Problematik:

- Fehlerquelle, da Änderungen über eine Variable z.B.

  ```
  p.x = p.x+1;
  ```

  auch zu Änderungen für die andere Variable führen (auch `q.x` wurde um 1 inkrementiert), ohne dass man es richtig sieht.
- Programme, die Aliasing verwenden sind oft schlecht nachvollziehbar.

Aliasing (2)

Beachte, dass bei Grunddatentypen in Java **kein Aliasing** auftritt:

```
public class TestAliasingGrundtyp {
  public static void main(String[] args) {
          int i = 3;
          int j = i;
          i++;
          System.out.println("i = " + i);
          System.out.println("j = " + j);
  }
}
```

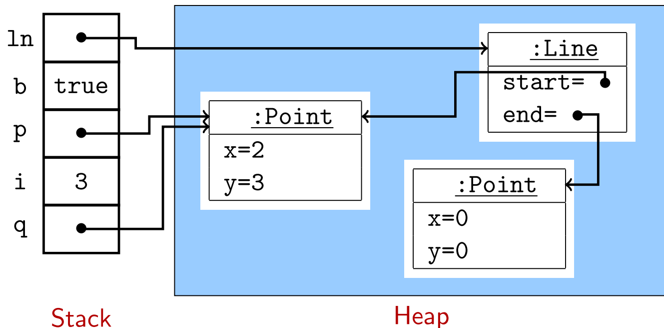
(Da bei Grunddatentypen die Werte direkt im Stack gespeichert, und nicht als Referenz auf den Heap)

Datenmüll

Die folgende Zuweisung auf dem Zustand von vorher

```
ln.end = p;
```

führt zu folgendem Zustand

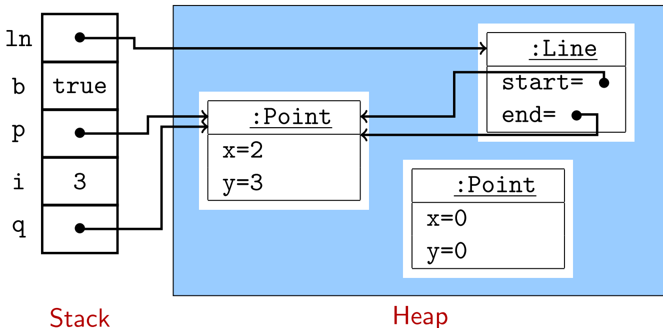


Datenmüll

Die folgende Zuweisung auf dem Zustand von vorher

```
ln.end = p;
```

führt zu folgendem Zustand

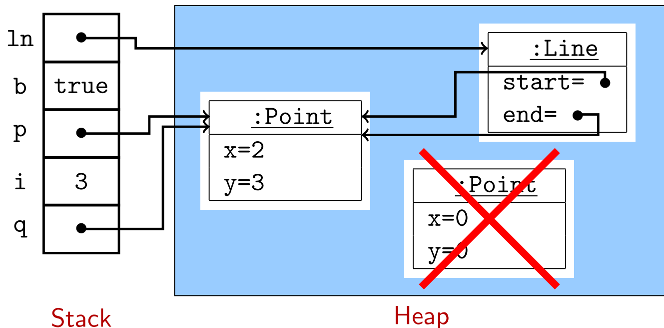


Datenmüll

Die folgende Zuweisung auf dem Zustand von vorher

```
ln.end = p;
```

führt zu folgendem Zustand



- Das Objekt wird nicht mehr referenziert und ist daher Müll (Garbage).
- Es wird von der **automatischen Speicherverwaltung** (Garbage Collector) gelöscht.

Die Return-Anweisung

Syntax:

Statement = ...
  | ReturnStatement (Return-Anweisung)

...

ReturnStatement = `"return"` [Expression] `";"`

- Eine Return-Anweisung mit einem Ergebnisausdruck muss in jedem Ausführungspfad einer Methode mit Ergebnis vorhanden sein.
- Der Typ von Expression muss zum Ergebnistyp der Methode passen.

Die Return-Anweisung

Syntax:

Statement = ...
    | ReturnStatement (Return-Anweisung)

...

ReturnStatement = `"return"` [Expression] `";"`

- Eine Return-Anweisung mit einem Ergebnisausdruck muss in jedem Ausführungspfad einer Methode mit Ergebnis vorhanden sein.
- Der Typ von Expression muss zum Ergebnistyp der Methode passen.

Wirkung:

- Die Ausführung des Methodenrumpfs wird beendet.
- Bei Methoden mit Ergebnistyp wird der Ausdruck Expression im zuletzt erreichten Zustand ausgewertet und dessen Wert als Ergebnis bereit gestellt.

Die Methodenaufrufanweisung

Syntax:

Statement = ...
  | MethodInvocation ";"

MethodInvocation = Expression "." Identifier "(" [ActualParameters] ")"

- Eine Methodenaufrufanweisung hat also die Form `e.m(`a_1`,...,`a_n`);`

Die Methodenaufrufanweisung

Syntax:

Statement = ...
| MethodInvocation ";"

MethodInvocation = Expression "." Identifier "(" [ActualParameters] ")"

- Eine Methodenaufrufanweisung hat also die Form `e.m(`a_1`,...,`a_n`);`
- Der Ausdruck e muss einen Klassentyp haben und der Identifier `m` muss ein Name einer Methode der Klasse (oder einer Oberklasse, vgl. später) sein:

  `void m (`T_1 x_1`,...,`T_n x_n`) {body}` oder
  `Type m (`T_1 x_1`,...,`T_n x_n`) {body}`

- Die aktuellen Parameter a_1`,...,`a_n sind Ausdrücke, die in Anzahl und Typ zu den formalen Parametern der Methodendeklaration passen müssen.

Beispiel: Sei e ein Ausdruck vom Typ `Point`.
Methodenaufrufanweisung: `e.move(10,15);`

Die Wirkung der Methodenaufrufanweisung

`e.m(a`$_1$`,...,a`$_n$`);` hat folgende Wirkung:

Sei e ein Ausdruck mit Klassentyp `C`.

1. e wird im aktuellen Zustand ausgewertet. Falls der Wert `null` ist, erfolgt ein Laufzeitfehler (`NullPointerException`), anderenfalls sei *ref* die erhaltene Objektreferenz.
2. Berechne die Werte von a_1 bis a_n nacheinander. Seien $w_1,\ldots,w_n$ die erhaltene Werte.
3. Lege lokale Variable `this` an, die mit *ref* initialisiert wird, und lege lokale Variablen $x_1,\ldots,x_n$ für die formalen Parameter an, die mit $w_1,\ldots,w_n$ initialisiert werden („**Call by Value**“).
4. Der Rumpf der Methode wird als Block ausgeführt.
5. Die lokalen Variablen `this`, $x_1,\ldots,x_n$ werden vom Stack genommen.

Die Wirkung der Methodenaufrufanweisung

`e.m(a`$_1$`,...,a`$_n$`);` hat folgende Wirkung:

Sei `e` ein Ausdruck mit Klassentyp `C`.

1. `e` wird im aktuellen Zustand ausgewertet. Falls der Wert `null` ist, erfolgt ein Laufzeitfehler (`NullPointerException`), anderenfalls sei *ref* die erhaltene Objektreferenz.
2. Berechne die Werte von `a`$_1$ bis `a`$_n$ nacheinander.
   Seien $w_1,\ldots,w_n$ die erhaltene Werte.
3. Lege lokale Variable `this` an, die mit *ref* initialisiert wird, und lege lokale Variablen `x`$_1$`,...,x`$_n$ für die formalen Parameter an, die mit $w_1,\ldots,w_n$ initialisiert werden („**Call by Value**").
4. Der Rumpf der Methode wird als Block ausgeführt.
5. Die lokalen Variablen `this`, `x`$_1$`,...,x`$_n$ werden vom Stack genommen.

Die Wirkung der Methodenaufrufanweisung

`e.m(a`$_1$`,...,a`$_n$`);` hat folgende Wirkung:

Sei `e` ein Ausdruck mit Klassentyp `C`.

1. `e` wird im aktuellen Zustand ausgewertet. Falls der Wert `null` ist, erfolgt ein Laufzeitfehler (`NullPointerException`), anderenfalls sei *ref* die erhaltene Objektreferenz.
2. Berechne die Werte von `a`$_1$ bis `a`$_n$ nacheinander.
   Seien $w_1,\ldots,w_n$ die erhaltene Werte.
3. Lege lokale Variable `this` an, die mit *ref* initialisiert wird, und lege lokale Variablen `x`$_1$`,...,x`$_n$ für die formalen Parameter an, die mit $w_1,\ldots,w_n$ initialisiert werden („**Call by Value**“).
4. Der Rumpf der Methode wird als Block ausgeführt.
5. Die lokalen Variablen `this`, `x`$_1$`,...,x`$_n$ werden vom Stack genommen.

Die Wirkung der Methodenaufrufanweisung

`e.m(a_1,...,a_n);` hat folgende Wirkung:

Sei e ein Ausdruck mit Klassentyp `C`.

1. e wird im aktuellen Zustand ausgewertet. Falls der Wert `null` ist, erfolgt ein Laufzeitfehler (`NullPointerException`), anderenfalls sei *ref* die erhaltene Objektreferenz.
2. Berechne die Werte von a_1 bis a_n nacheinander. Seien $w_1,\ldots,w_n$ die erhaltene Werte.
3. Lege lokale Variable `this` an, die mit *ref* initialisiert wird, und lege lokale Variablen $x_1,\ldots,x_n$ für die formalen Parameter an, die mit $w_1,\ldots,w_n$ initialisiert werden („**Call by Value**“).
4. Der Rumpf der Methode wird als Block ausgeführt.
5. Die lokalen Variablen `this`, $x_1,\ldots,x_n$ werden vom Stack genommen.

Die Wirkung der Methodenaufrufanweisung

`e.m(a`$_1$`,...,a`$_n$`);` hat folgende Wirkung:

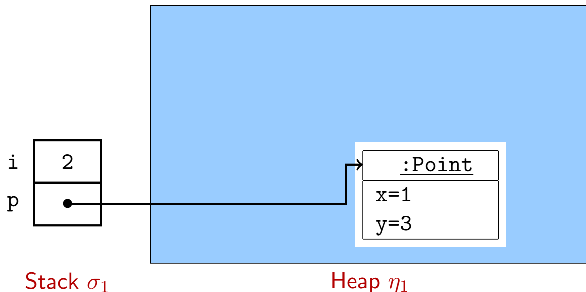
Sei `e` ein Ausdruck mit Klassentyp `C`.

1. `e` wird im aktuellen Zustand ausgewertet. Falls der Wert `null` ist, erfolgt ein Laufzeitfehler (`NullPointerException`), anderenfalls sei *ref* die erhaltene Objektreferenz.
2. Berechne die Werte von `a`$_1$ bis `a`$_n$ nacheinander. Seien $w_1,\ldots,w_n$ die erhaltene Werte.
3. Lege lokale Variable `this` an, die mit *ref* initialisiert wird, und lege lokale Variablen `x`$_1$`,...,x`$_n$ für die formalen Parameter an, die mit $w_1,\ldots,w_n$ initialisiert werden („**Call by Value**“).
4. Der Rumpf der Methode wird als Block ausgeführt.
5. Die lokalen Variablen `this`, `x`$_1$`,...,x`$_n$ werden vom Stack genommen.

Beachte: Von einer anderen Klasse aus, sind Methodenaufrufe nur gemäß den spezifizierten Sichtbarkeiten zulässig.

Call-by-Value Parameterübergabe: Beispiel (1)

Zustand (σ_1, η_1):



Im Zustand (σ_1, η_1) werde

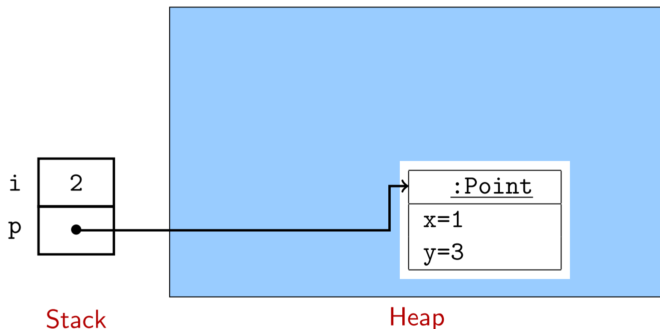
```
p.move(i,2+2);
```

aufgerufen.

Zur Erinnerung:

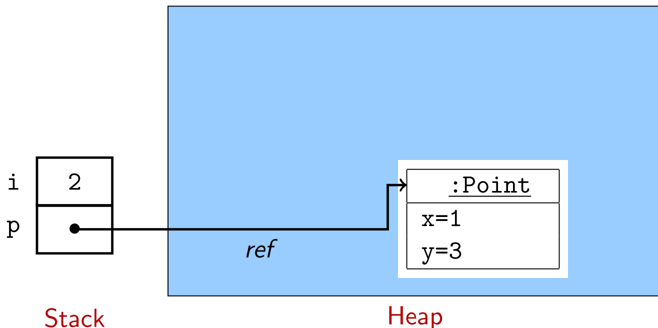
```
public class Point {
 private int x,y;
 ...
 public void move(int dx, int dy){
  this.x += dx;
  this.y += dy;
 }
```

Call-by-Value Parameterübergabe: Beispiel (2)



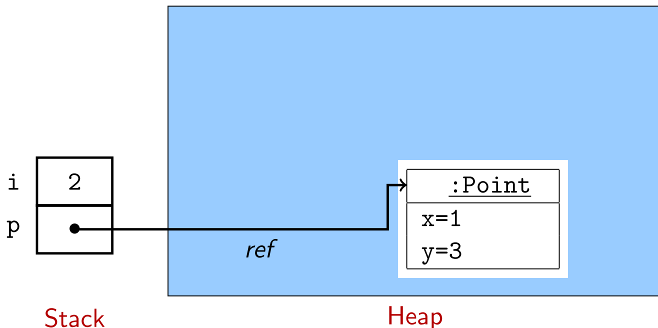
Call-by-Value Parameterübergabe: Beispiel (2)

- Auswertung von `p`: ist bereits eine Objektreferenz (das *ref* ist der entsprechende Zeiger)



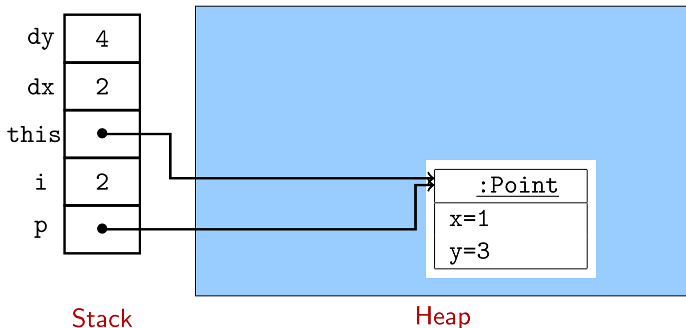
Call-by-Value Parameterübergabe: Beispiel (2)

- Auswertung von `p`: ist bereits eine Objektreferenz (das *ref* ist der entsprechende Zeiger)
- Auswertung von `i` ergibt 2 und Auswertung von 2+2 ergibt 4

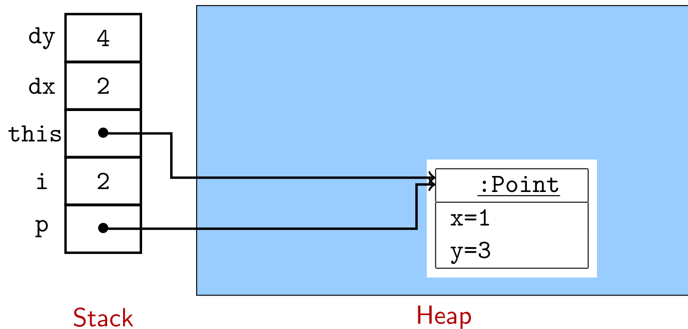


Call-by-Value Parameterübergabe: Beispiel (2)

- Auswertung von `p`: ist bereits eine Objektreferenz (das *ref* ist der entsprechende Zeiger)
- Auswertung von `i` ergibt 2 und Auswertung von 2+2 ergibt 4
- Nach Erzeugung der lokalen Variablen `this`, `dx` `dy` ergibt sich:



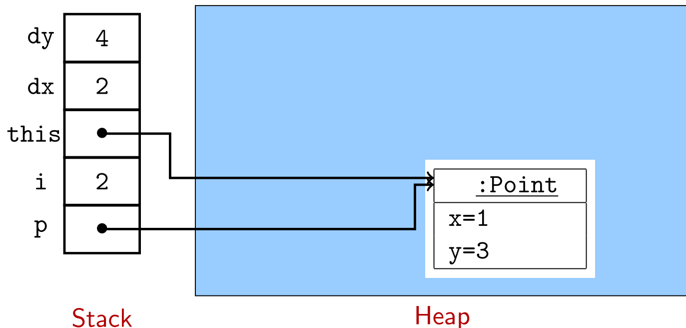
Call-by-Value Parameterübergabe: Beispiel (3)



Call-by-Value Parameterübergabe: Beispiel (3)

Ausführung des Methodenrumpfs als Block

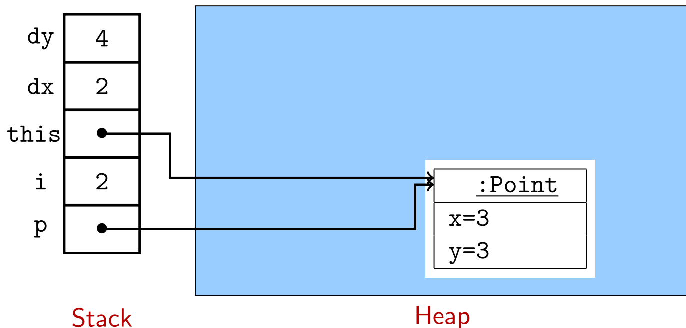
```
this.x += dx;
this.y += dy;
```



Call-by-Value Parameterübergabe: Beispiel (3)

Ausführung des Methodenrumpfs als Block

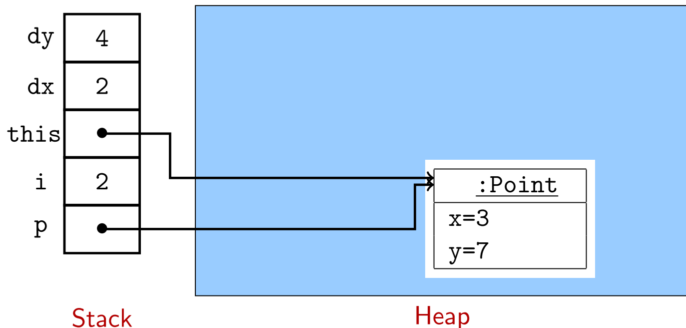
```
this.x += dx;
this.y += dy;
```



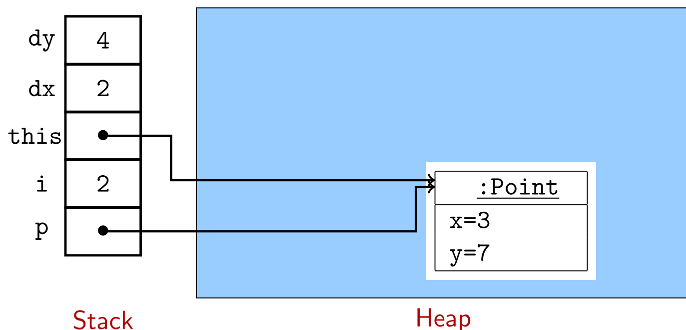
Call-by-Value Parameterübergabe: Beispiel (3)

Ausführung des Methodenrumpfs als Block

```
this.x += dx;
this.y += dy;
```

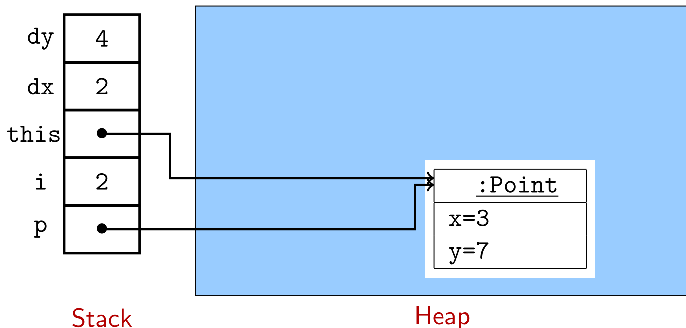


Call-by-Value Parameterübergabe: Beispiel (4)



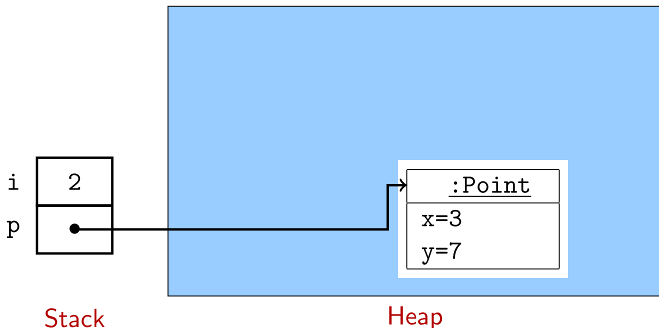
Call-by-Value Parameterübergabe: Beispiel (4)

Beenden des Methodenaufrufs: Entfernen der lokalen Variablen vom Stack



Call-by-Value Parameterübergabe: Beispiel (4)

Beenden des Methodenaufrufs: Entfernen der lokalen Variablen vom Stack



Objekterzeugungsanweisung

Syntax:

Statement = ...
| ClassInstanceCreation ";"

ClassInstanceCreation = "new" ClassType "(" [ActualParameters] ")"

Eine Objekterzeugungsanweisung hat also die Form

```
new C(a1,...,an);
```

wobei `new C(a1,...,an)` ein Objekterzeugungsausdruck ist.

Wirkung der Objekterzeugungsanweisung

`new C(a`$_1$`,...,a`$_n$`);` hat folgende Wirkung

1. Ein Objekt der Klasse `C` wird erzeugt und auf den Heap gelegt.
2. Die Felder des Objekts werden mit den default-Werten intialisiert (`0` bei `int`, `false` bei `boolean`, `null` bei Klassentypen.
3. Die Referenz auf das Objekt wird bereitgestellt.

Wirkung der Objekterzeugungsanweisung

`new C(a1,...,an);` hat folgende Wirkung

1. Ein Objekt der Klasse `C` wird erzeugt und auf den Heap gelegt.
2. Die Felder des Objekts werden mit den default-Werten intialisiert (`0` bei `int`, `false` bei `boolean`, `null` bei Klassentypen.
3. Die Referenz auf das Objekt wird bereitgestellt.

Falls ein benutzerdefinierter Konstruktor aufgerufen wurde,
erfolgt vor Schritt 3:

1. Die Werte der aktuellen Parameter $\texttt{a}_1,\ldots,\texttt{a}_n$ werden berechnet,
2. Eine lokale Variable `this` mit Typ `C` wird angelegt und mit der Referenz auf das neue Objekt initialisiert und lokale Variablen für die formalen Parameter $\texttt{x}_1,\ldots,\texttt{x}_n$ des Konstruktor werden angelegt und mit den erhaltenen Werten für $\texttt{a}_1,\ldots,\texttt{a}_n$ initialisiert.
3. Der Rumpf des Konstruktors wird als Block ausgeführt.
4. Die lokalen Variablen `this`, $\texttt{x}_1,\ldots,\texttt{x}_n$ werden vom Stack genommen.

Benutzung von Klassen und Objekten

Objekte werden (meist) in Methoden von anderen Klassen erzeugt und benutzt. Die Benutzung geschieht (meist) durch Methodenaufruf.

```
public class PointMain {
 public static void main(String[] args) {
  Point p1 = new Point(10, 20);
  Point p2 = new Point(0,0);
  int x1 = p1.getX(), y1 = p1.getY();
  int x2 = p2.getX(), y2 = p2.getY();
  System.out.println("p1=(" + x1 + ", " + y1 + ")");
  System.out.println("p2=(" + x2 + ", " + y2 + ")");
  p1.move(10, 10);
  System.out.println("p1=(" + p1.getX() + ", " + p1.getY()
      + ")");
 }
}
```

Aufruf einer Methode mit Rückgabewert (bei `p1.getY();`)

Methodenaufruf (ohne Rückgabewert) (bei `p1.move(10, 10);`)

Die Klasse Point mit öffentlichen Attributen

```
public class Point {
        public int x,y;
        public Point(int x0, int y0){
                this.x = x0;
                this.y = y0;
        }
        public void move(int dx, int dy){
                this.x += dx;
                this.y += dy;
        }
        public int getX(){ return this.x; }
        public int getY(){ return this.y; }
}
```

- Auf öffentliche Attribute kann von anderen Objekten aus zugegriffen werden!
- Das verletzt die Idee des **Geheimnisprinzips**!
- Geheimnisprinzips: Änderungen an Objektzuständen finden nur durch Methodenaufrufe statt.

Benutzung der Objekte bei öffentlichem Attributzugriff

```
public class PointMain {
  public static void main(String[] args) {
    Point p1 = new Point (10,20);
    Point p2 = new Point (0,0);
    int x1 = p1.x, y1 = p1.y;
    int x2 = p2.x, y2 = p2.y;
    System.out.println("p1=(" + x1 ", " + y1 + ")");
    System.out.println("p2=(" + x2 + ", " + y2 + ")");
    p1.x = p1.x + 10;
    System.out.println("p1=(" + x1 ", " + y1 + ")");
  }
}
```

Zugriff auf das Attribut eines anderen Objekts

Ändern des Attributwerts eines anderen Objekts

Methodenimplementierung: Abkürzung

Innerhalb einer Methodenimplementierung ist der Name von `this` eindeutig und kann weggelassen werden, wenn keine Namenskonflikte auftreten:

```
public void move(int dx, int dy){
       x += dx;
       y += dy;
}
```

Methodenimplementierung: Abkürzung

Innerhalb einer Methodenimplementierung ist der Name von `this` eindeutig und kann weggelassen werden, wenn keine Namenskonflikte auftreten:

```
public void move(int dx, int dy){
        x += dx;
        y += dy;
}
```

Aber: Parameter und lokale Variablen überdecken Attribute gleichen Namens. Die folgende Implementierung von `move` benötigt die explizite Verwendung von `this`.

```
public void move(int x, int y){
        this.x += x;
        this.y += y;
}
```

Call-by-Value und nicht Call-by-Reference

- In Java wird Call-by-Value-Parameterübergabe verwendet, d.h. beim Methodenaufruf $\texttt{m(a}_1\texttt{,...,a}_n\texttt{)}$ werden die aktuellen Parameter $\texttt{a}_i$ erst ausgewertet und dann (Kopien) der Werte übergeben.
- Beachte: Dies gilt auch, wenn $\texttt{a}_i$ einen Klassentyp hat: In diesem Fall wird, als Wert die Referenz auf das Objekt bestimmt, und anschließend eine Kopie der Referenz übergeben.
- Das Erstellen der Kopie geschieht durch Anlegen der lokalen Variablen auf dem Stack!

Beachte: **Andere** Programmiersprachen verwenden teilweise die Call-by-Reference-Übergabe: Dort werden nur Zeiger auf die aktuellen Parameter (ohne Kopie) übergeben. Alle Änderungen an den Objekten wirken sich in diesem Fall auf die ursprünglichen Werte aus.

Beispiel zur Parameterübergabe (1)

```
class Point {
  ... // wie vorher

  public void setX(int newX){
   this.x = newX;
  }
  public void setY(int newY){
   this.y = newY;
  }
  public String toString(){
      return ("(" + this.x + "," + this.y + ")");
 }
  public void transferXYToPoint(Point p) {
     p.setX(this.x);
     p.setY(this.y);
  }
  public void transferPointToPoint(Point p) {
     p = this; // funktioniert nicht, wegen
         call-by-value Parameteruebergabe
  }
}
```

Beispiel zur Parameterübergabe (2)

```
public class PointParamTest {
    public static void main(String args[]) {
        Point p1 = new Point(1,2);
        Point p2 = new Point(3,4);
        Point p3 = new Point(5,6);
        System.out.println("p1 = " + p1.toString());
        System.out.println("p2 = " + p2.toString());
        System.out.println("p3 = " + p3.toString());
        System.out.println("-----------------");
        p1.transferXYToPoint(p2);
        System.out.println("p1 = " + p1.toString());
        System.out.println("p2 = " + p2.toString());
        System.out.println("p3 = " + p3.toString());
        System.out.println("-----------------");
        p1.transferPointToPoint(p3);
        System.out.println("p1 = " + p1.toString());
        System.out.println("p2 = " + p2.toString());
        System.out.println("p3 = " + p3.toString());
        System.out.println("-----------------");

    }
```

Beispiel zur Parameterübergabe (3)

```
> javac PointParamTest.java
> java PointParamTest

p1 = (1,2)
p2 = (3,4)
p3 = (5,6)
-----------------
p1 = (1,2)
p2 = (1,2)
p3 = (5,6)
-----------------
p1 = (1,2)
p2 = (1,2)
p3 = (5,6)
-----------------
```

Wie wurde der Zustand geändert?

Nach Ausführung der Objekterzeugungsanweisungen ist der Zustand (σ_0, η_0) mit

$$\begin{aligned}
\sigma_0 &= \texttt{[(p3,@003),(p2,@002),(p1,@001)]} \\
\eta_0 &= \{\langle\texttt{(@001,Point),[(x,1),(y,2)]}\rangle, \\
&\quad\ \langle\texttt{(@002,Point),[(x,3),(y,4)]}\rangle, \\
&\quad\ \langle\texttt{(@003,Point),[(x,5),(y,6)]}\rangle\}
\end{aligned}$$

Wie wurde der Zustand geändert?

Nach Ausführung der Objekterzeugungsanweisungen ist der Zustand (σ_0, η_0) mit

$$
\begin{aligned}
\sigma_0 &= \texttt{[(p3,@003),(p2,@002),(p1,@001)]}\\
\eta_0 &= \{\langle\texttt{(@001,Point),[(x,1),(y,2)]}\rangle,\\
&\quad\ \langle\texttt{(@002,Point),[(x,3),(y,4)]}\rangle,\\
&\quad\ \langle\texttt{(@003,Point),[(x,5),(y,6)]}\rangle\}
\end{aligned}
$$

Ausführung des Methodenaufrufs `p1.transferXYToPoint(p2)`

Direkt vor Aufruf des Rumpfs: (σ_1, η_1) mit

$\sigma_0 = \texttt{[(p,@002),(this,@001),(p3,@003),(p2,@002),(p1,@001)]}$
$\eta_2 = \eta_0$

Wie wurde der Zustand geändert? (2)

Nach Ausführung des Methodenaufrufs: (σ_2, η_2) mit

$$\begin{aligned}
\sigma_2 &= \texttt{[(p3,@003),(p2,@002),(p1,@001)]} \\
\eta_2 &= \{\langle\texttt{(@001,Point),[(x,1),(y,2)]}\rangle, \\
&\quad\ \langle\texttt{(@002,Point),[(x,1),(y,2)]}\rangle, \\
&\quad\ \langle\texttt{(@003,Point),[(x,5),(y,6)]}\rangle\}
\end{aligned}$$

Wie wurde der Zustand geändert? (2)

Nach Ausführung des Methodenaufrufs: (σ_2, η_2) mit

$$\begin{aligned}\sigma_2 &= \texttt{[(p3,@003),(p2,@002),(p1,@001)]}\\ \eta_2 &= \{\langle\texttt{(@001,Point),[(x,1),(y,2)]}\rangle,\\ &\quad\ \langle\texttt{(@002,Point),[(x,1),(y,2)]}\rangle,\\ &\quad\ \langle\texttt{(@003,Point),[(x,5),(y,6)]}\rangle\}\end{aligned}$$

Ausführung des Methodenaufrufs `p1.transferPointToPoint(p3)`

Direkt vor Aufruf des Rumpfs: (σ_3, η_3) mit $\eta_3 = \eta_2$ und

$$\sigma_3 = \texttt{[(p,@003),(this,@001),(p3,@003),(p2,@002),(p1,@001)]}$$

Wie wurde der Zustand geändert? (2)

Nach Ausführung des Methodenaufrufs: (σ_2, η_2) mit

$$\sigma_2 = \texttt{[(p3,@003),(p2,@002),(p1,@001)]}$$
$$\begin{aligned}\eta_2 = \{&\langle\texttt{(@001,Point),[(x,1),(y,2)]}\rangle,\\ &\langle\texttt{(@002,Point),[(x,1),(y,2)]}\rangle,\\ &\langle\texttt{(@003,Point),[(x,5),(y,6)]}\rangle\}\end{aligned}$$

Ausführung des Methodenaufrufs `p1.transferPointToPoint(p3)`

Direkt vor Aufruf des Rumpfs: (σ_3, η_3) mit $\eta_3 = \eta_2$ und

$$\sigma_3 = \texttt{[(p,@003),(this,@001),(p3,@003),(p2,@002),(p1,@001)]}$$

Nach `p = this;` ist der Zustand (σ_4, η_4) mit $\eta_4 = \eta_2$ und

$$\sigma_4 = \texttt{[(p,@001),(this,@001),(p3,@003),(p2,@002),(p1,@001)]}$$

Wie wurde der Zustand geändert? (2)

Nach Ausführung des Methodenaufrufs: (σ_2, η_2) mit

$$\begin{aligned}\sigma_2 &= \texttt{[(p3,@003),(p2,@002),(p1,@001)]}\\ \eta_2 &= \{\langle\texttt{(@001,Point),[(x,1),(y,2)]}\rangle,\\ &\quad\ \langle\texttt{(@002,Point),[(x,1),(y,2)]}\rangle,\\ &\quad\ \langle\texttt{(@003,Point),[(x,5),(y,6)]}\rangle\}\end{aligned}$$

Ausführung des Methodenaufrufs `p1.transferPointToPoint(p3)`

Direkt vor Aufruf des Rumpfs: (σ_3, η_3) mit $\eta_3 = \eta_2$ und

$$\sigma_3 = \texttt{[(p,@003),(this,@001),(p3,@003),(p2,@002),(p1,@001)]}$$

Nach `p = this;` ist der Zustand (σ_4, η_4) mit $\eta_4 = \eta_2$ und

$$\sigma_4 = \texttt{[(p,@001),(this,@001),(p3,@003),(p2,@002),(p1,@001)]}$$

Nach Beenden des Methodenaufrufs: (σ_5, η_5) mit $\eta_5 = \eta_2$ und

$$\sigma_5 = \texttt{[(p3,@003),(p2,@002),(p1,@001)]}$$

Statische Attribute und statische Methoden

- **Statische Attribute** (**Klassenattribute**) sind (globale) Variablen einer Klasse, die **unabhängig von Objekten** Werte speichern.
- **Statische Methoden** (**Klassenmethoden**) sind Methoden einer Klasse, **unabhängig von Objekten** aufgerufen und ausgeführt werden.

Syntax:

```
class C {
  private static type attribute = ...;
  public static void method (...) { body };
  ...
```

- Im Rumpf einer statischen Methode dürfen keine Instanzvariablen verwendet werden.
- Zugriff auf ein Klassenattribut: `C.attribute` z.B. `System.out`
- Aufruf einer Klassenmethode: `C.method(...)` z.B. `Math.sqrt(7)`

Klassenattribute und -methoden: Beispiel

```
public class BankKonto {
  private double kontoStand;
  private int kontoNr;
  private static int letzteNr = 0;
  private static int neueNr () {
    int number = BankKonto.letzteNr;
    BankKonto.letzteNr++;
    return number;
  }
  public BankKonto(double betrag) {
    this.kontoStand = betrag;
    this.kontoNr = BankKonto.neueNr();
 }
 ...
}
```

Klassenmethoden: Beispiele

```
class NumFunktionen {
 public static int quersumme(int x){
   int qs = 0;
   while (x > 0) {
     qs = qs + x % 10;
     x = x / 10; }
   return qs;                     }
 public static int fakultaet(int n){
   int akk = 1;
   while (n > 1) {
     akk = akk * n;
     n--;        }
   return akk;                    }
}

public class NumAnwendung {
 public static void main(String[] args){
  int x = 352;
  int q = NumFunktionen.quersumme(x);
  System.out.println("Quersumme von " + x + ": " + q);
  int y = 6;
  System.out.println("Fakultaet von " + x + ": " +
      NumFunktionen.fakultaet(y)); }
}
```

Konstanten

Konstanten sind Klassenattribute
mit einem festen, unveränderlichen Wert.

Syntax:

```
class C {
  public static final type attribute = value;
  ...
  }
```

Schlüsselwort final kennzeichnet Konstante

Konstanten werden meist mit Großbuchstaben geschrieben und meist als `public` deklariert.

Beispiel:

```
class Math {
  public static final double PI = 3.14159265358979323846;
  ...
}
```

Konstanten: Beispiel

```
class Roboter {
 public static final char NORD = 'N';
 public static final char SUED = 'S';
 public static final char WEST = 'W';
 public static final char OST  = 'O';
 private int x,y;
 private char richtung;
 public Roboter(int x0, int y0, char richtung0) {
  this.x = x0;
  this.y = y0;
  this.richtung = richtung0;                         }
 public void dreheRechts()                           {
  if (this.richtung == NORD) this.richtung = OST;
  else if (this.richtung == OST) this.richtung = SUED;
        else if (this.richtung == SUED) this.richtung = WEST;
              else this.richtung= NORD;              }
 ...
}
public class RoboterTest {
 public static void main(String[] args) {
  Roboter r = new Roboter(0,0,Roboter.SUED);
  ...
 }
}
```

Final bei Attributen

- Das Schlüsselwort `final` sorgt dafür, dass nur einmal ein Wert zugewiesen kann.
- Auch bei nicht-statischen Attributen kann man `final` verwenden.

Beispiel:

```
class Animal {
 private final Animal mother;  // Mutter kann nur einmal
                               // zugewiesen werden
 private String name;
 public Animal(String name,Animal mother) {
   this.mother=mother;
   this.name=name;                          }
 public Animal (String name) {
   this.mother = null; // unschoen
   this.name  = name;          }
}
public class AnimalTest {
 public static void main (String[] args)             {
  Animal katzeHelga = new Animal("Helga");
  Animal katzeHorst = new Animal("Horst",katzeHelga); }
}
```

Final bei Attributen (2)

- Wenn wir

  `private final Animal mother;`

  durch

  `private static final Animal mother = ...;`

  austauschen, hätten quasi alle die selbe Mutter.

- Hinzufügen der Methode

```
public void setMother(Animal mother) {
    this.mother = mother;
    }
```

  führt zum Fehler:

```
error: cannot assign a value to final variable mother
   this.mother = mother;
       ^
1 error
```

Die Klasse String

- Zeichenketten (Strings) werden in Java durch Objekte der Klasse `String` repräsentiert. Diese Objekte speichern eine (unveränderbare) Folge von Zeichen (Characters).
- Infolgedessen sind die Werte des Klassentyps `String` Referenzen auf String-Objekte.
- Referenzen auf String-Objekte können durch String-Literale angegeben werden:
  z.B. `"WS 2018/19"`, `"M-XY 789"`, `"\""`, `""` (leerer String).

Die Klasse String

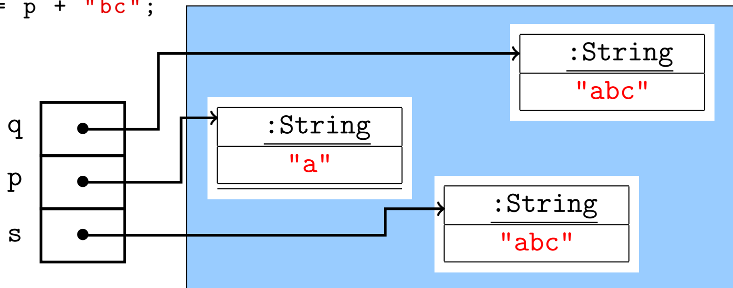
- Zeichenketten (Strings) werden in Java durch Objekte der Klasse `String` repräsentiert. Diese Objekte speichern eine (unveränderbare) Folge von Zeichen (Characters).
- Infolgedessen sind die Werte des Klassentyps `String` Referenzen auf String-Objekte.
- Referenzen auf String-Objekte können durch String-Literale angegeben werden:
  z.B. `"WS 2018/19"`, `"M-XY 789"`, `"\""`, `""` (leerer String).
- **Operationen** auf Strings sind:

`==`, `!=`	Vergleich von Referenzen (nicht empfohlen!)
`+`	Zusammenhängen zweier Strings zu einem neuen String

- Die Klasse String enthält eine Vielzahl von Konstruktoren und Methoden, z.B. `public boolean equals(Object anObject)` für den Vergleich der Zeichenketten („Inhalte“) zweier String-Objekte (empfohlen!).

Gleichheit von Strings

```
String s = "abc";
String p = "a";
String q = p + "bc";
```



Stack σ Heap η

- Gleichheit von String-Referenzen:
  `(s==p)` $=_{(\sigma,\eta)}$ `false`, `(s==q)` $=_{(\sigma,\eta)}$ `false`, (!),
- Gleichheit von String-Inhalten (Zeichenketten):
  `s.equals(p)` $=_{(\sigma,\eta)}$ `false`, `s.equals(q)` $=_{(\sigma,\eta)}$ `true`

Auszug aus der Dokumentation zur Klasse String

Method Summary	
`char`	`charAt(int index)` Returns the char value at the specified index.
`boolean`	`isEmpty()` Returns `true` if, and only if, `length()` is 0.
`int`	`length()` Returns the length of this string.
`String`	`replace(char oldChar, char newChar)` Returns a string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
`String`	`substring(int beginIndex, int endIndex)` Returns a string that is a substring of this string.
`String`	`toString()` This object (which is already a string!) is itself returned.

Umwandlung von Strings in Werte der Grunddatentypen

Statische Methoden

- `public static int parseInt(String s)` der Klasse `Integer`
- `public static double parseDouble(String s)` der Klasse `Double`
- ...

Beispiel:

```
String s = "64";
int x    = Integer.parseInt(s);
```

Wenn der String keine Ganzzahl repräsentiert, kommt es zu einem Laufzeitfehler (`NumberFormatException`)

Beispiel

```
public class ParseBeispiel {
 public static void main(String[] args) {
 String s = "64";
 int x = Integer.parseInt(s);
 System.out.println(x + 10);
 x = Integer.parseInt("das ist keine Zahl");
 }
}
```

Ausführung:

```
> java  ParseBeispiel
74
Exception in thread "main" java.lang.NumberFormatException:
For input string: "das ist keine Zahl" at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:
   at java.lang.Integer.parseInt(Integer.java:580)
   at java.lang.Integer.parseInt(Integer.java:615)
   at ParseBeispiel.main(ParseBeispiel.java:6)
```

Umwandlung in Strings

Statische Methoden (zur Umwandlung von Werten von Grunddatentypen)

- `public static String toString(int i)` der Klasse `Integer`
- `public static String toString(double d)` der Klasse `Double`
- ...

Beispiel

```
int x = 64;
String s = Integer.toString(x);
```

Umwandlung in Strings (2)

Nicht statische Methode

`public String toString()`

kann auf Objekte aller Klassen angewendet werden.

Beispiel

```
BankKonto b = new BankKonto();
String s = b.toString();
```

liefert einen String, bestehend aus dem Namen der Klasse, zu der das Objekt gehört, dem Zeichen `@` sowie einer Hexadezimal-Repräsentation des Objekts, z.B. `BankKonto@a2b7ef43`.

Zusammenfassung

- Anweisungen bei Klassen / Objekten / Methoden: Return-Anweisung, Methodenaufruf, Objekterzeugung
- Syntax und Semantik der Anweisungen
- Call-by-Value Parameterübergabe
- Statische Attribute und Methoden (static)
- Konstanten (final)
- Die Klasse `String`