

# Kapitel 6

---

## Kontrollstrukturen und Methoden im Kontext von Klassendeklarationen

## Ziele

- Anweisungen auf den Kontext von Klassendeklarationen erweitern
- Statische Attribute und Methoden verstehen
- Die Klasse `String` kennenlernen

## Überblick Kapitel 3 - 6

### *Kapitel 3*

Grunddatentypen

erweitert um

Werte

erweitert um

Operationen

erweitert um

Ausdrücke

erweitert um

Typisierung

Auswertung bzgl.

Zustand (Stack)

erweitert um

### *Kapitel 4*

Kontrollstrukturen

**erweitert um**

### *Kapitel 5*

Klassendeklarationen

Objekte und Objekthalde (Heap)

Klassentypen

Referenzen und `null`

`==`, `!=` für Referenzen und `null`

Attributzugriff,

Methodenaufruf mit Ergebnis,

Objekterzeugungsausdruck

Objekthalde (Heap)

### *Kapitel 6*

**Return-Anweisung,  
Methodenaufruf, Objekterzeugung**

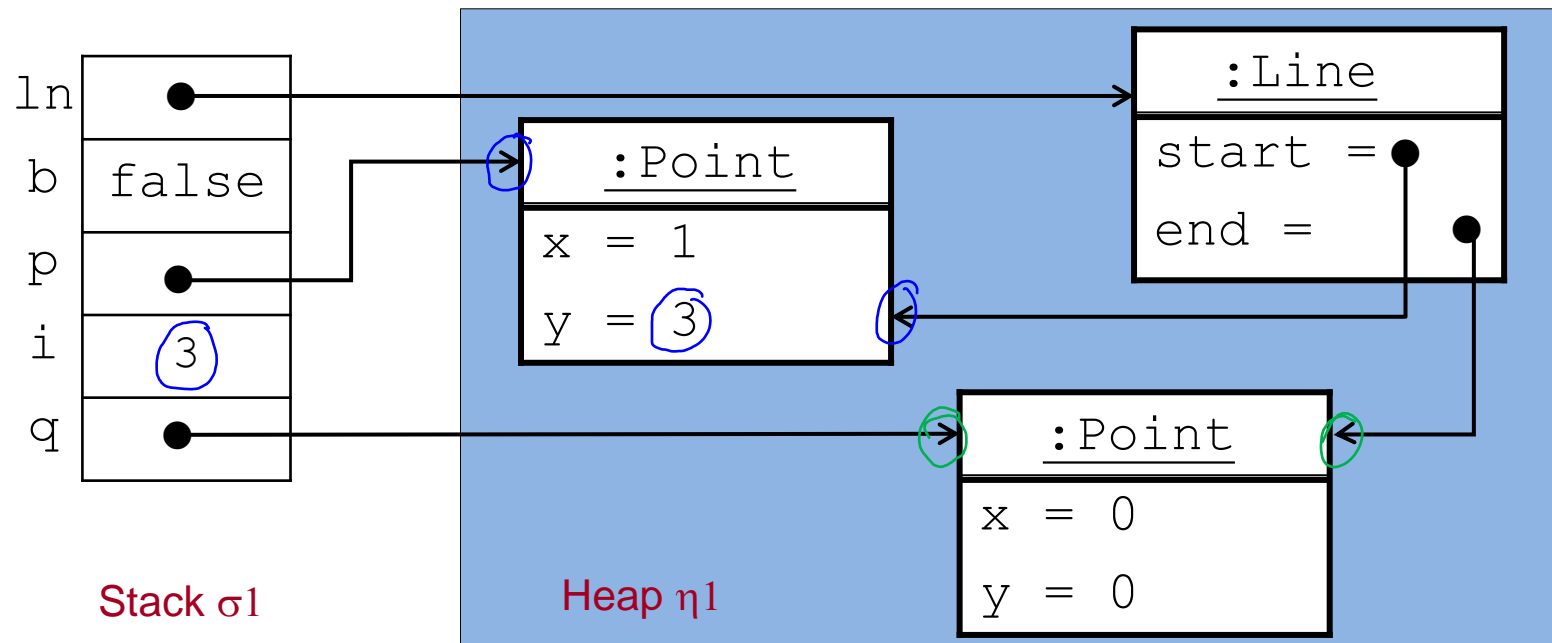
## Erweiterte Grammatik für Anweisungen im Kontext von Klassendeklarationen

*Statement* =  
    *VariableDeclaration*  
| *Assignment*  
| *Block*  
| *Conditional*  
| *Iteration*  
| ***ReturnStatement*** (← neu)  
| ***MethodInvocation*** ";" (← neu)  
| ***ClassInstanceCreation*** ";" (← neu)

## Deklarationsanweisungen und Zustandsänderung

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

führt zu folgendem Zustand:

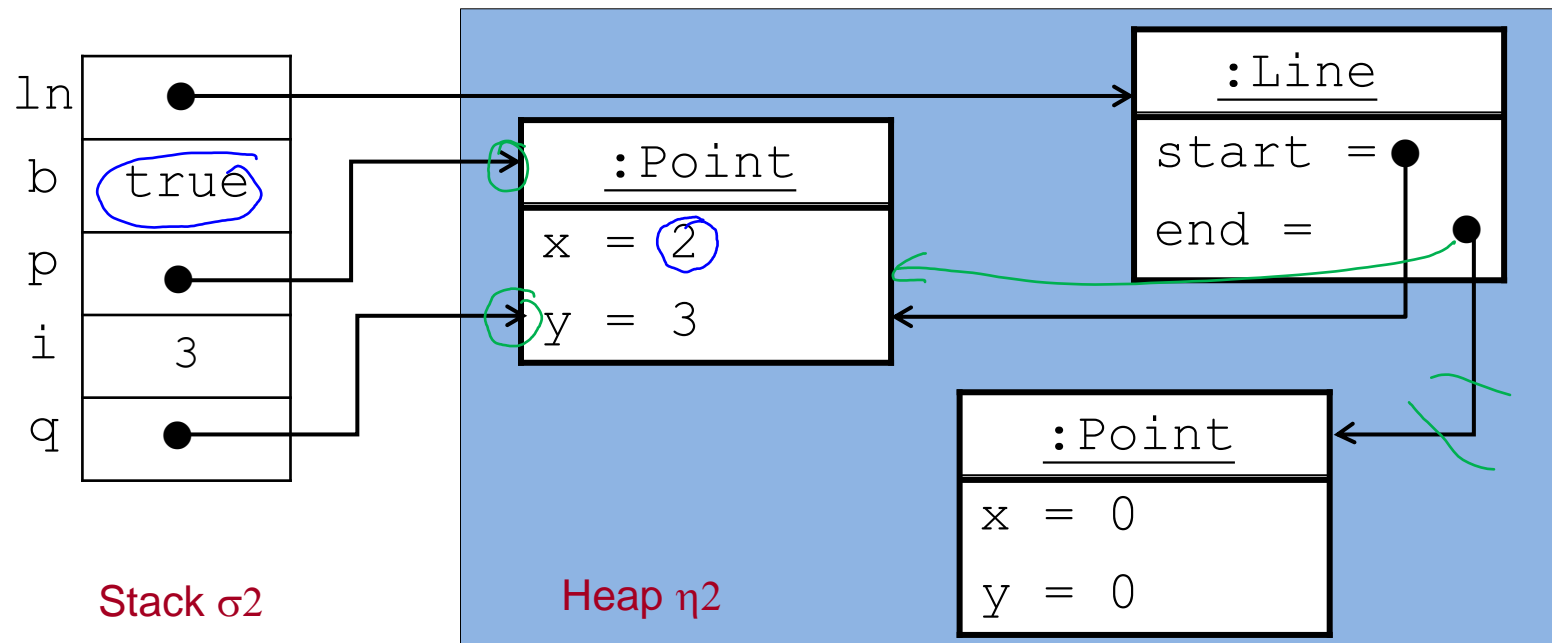


## Zuweisungen und Zustandsänderung

Im Zustand  $(\sigma_1, \eta_1)$  der letzten Folie werden folgende Zuweisungen durchgeführt:

```
q = p; // Aliasing! q und p zeigen auf dasselbe Objekt!  
p.x = p.x + 1;  
b = (q.getX() == 2);
```

Dies führt zu folgendem Zustand:

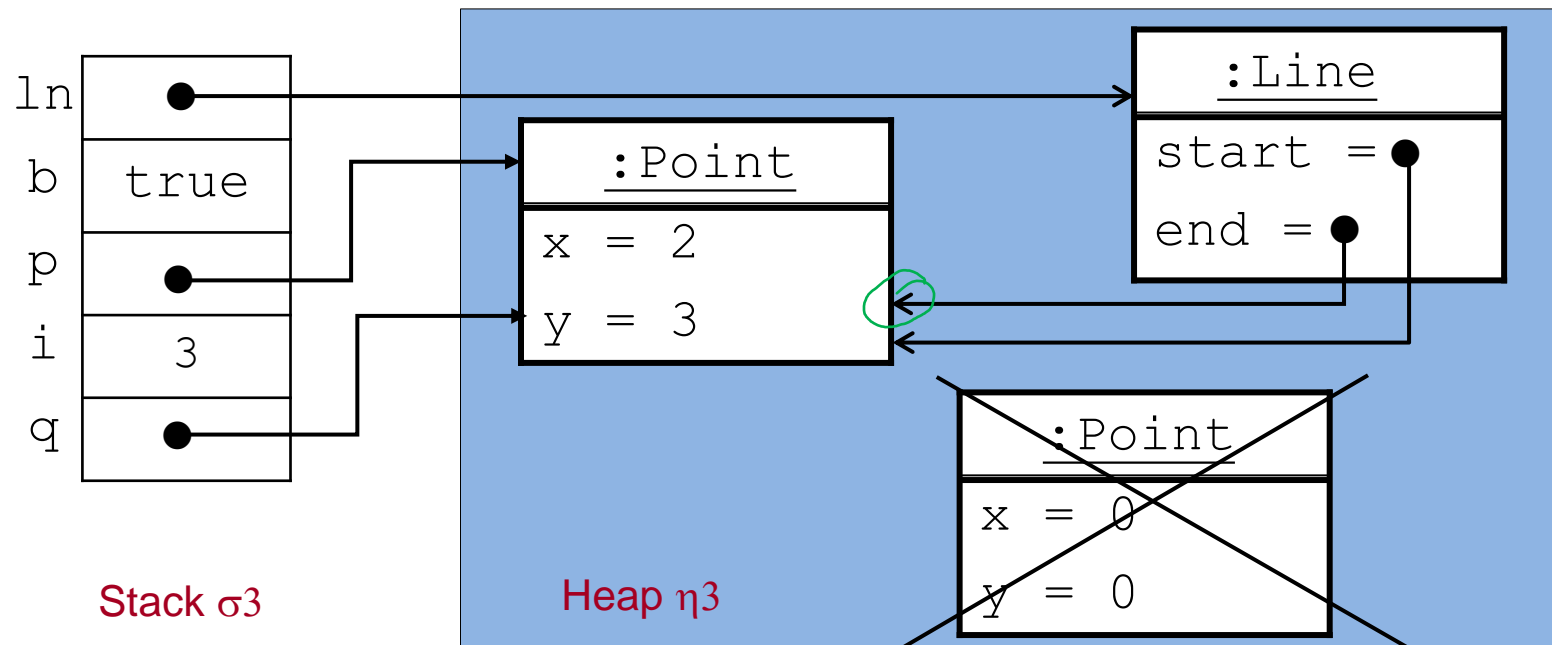


## Datenmüll

Im Zustand  $(\sigma_2, \eta_2)$  der letzten Folie führen wir durch: `l.end = p;`

Im Zustand  $(\sigma_3, \eta_3)$  nach der Zuweisung ist ein Objekt unerreichbar geworden.

- Keine Referenz zeigt mehr darauf.
- Es ist Müll (engl.: Garbage) und wird automatisch vom Speicherbereinigungsalgorithmus („Garbage Collector“) gelöscht.



## Return-Anweisung

Syntax: *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.



## Methodenaufruf-Anweisung

Syntax: *MethodInvocation* ";"

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

Eine Methodenaufruf-Anweisung hat also die Form

$e.m(a_1, \dots, a_n);$

- Der Ausdruck  $e$  muss einen Klassentyp haben und der Identifier  $m$  muss der Name einer Methode der Klasse (oder einer Oberklasse, vgl. später) sein:

`void m (T1 x1, ..., Tn xn) {body}    oder`

`Type m (T1 x1, ..., Tn xn) {body}`

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

Beispiel: Sei  $e$  ein Ausdruck vom Typ `Point`.

Methodenaufruf-Anweisung: `e.move(10, 15);`

## Methodenaufruf-Anweisung: Wirkung

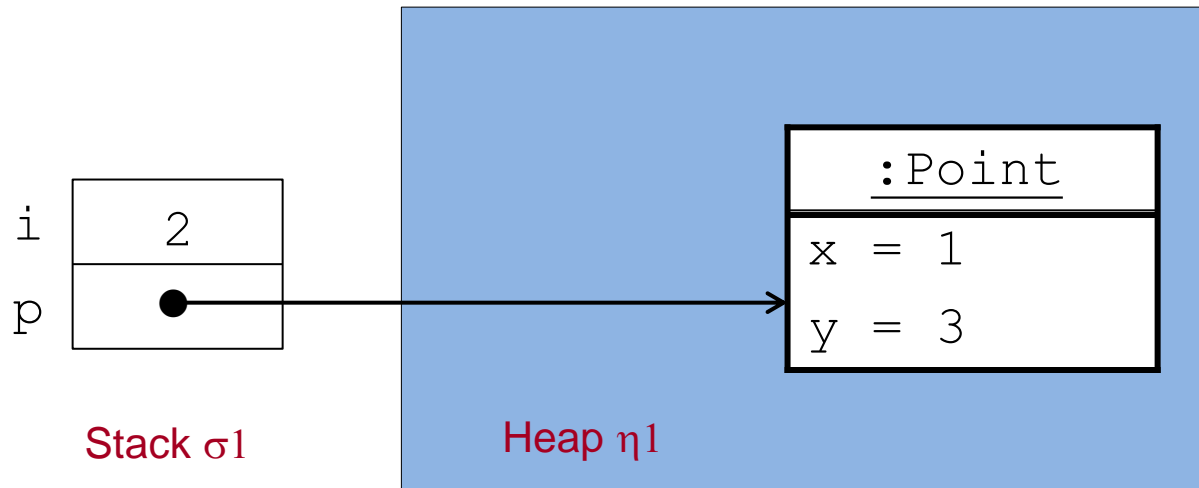
$e.m(a_1, \dots, a_n)$ ; hat folgende Wirkung:

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

1. Der Ausdruck  $e$  wird im aktuellen Zustand ausgewertet.  
Falls der Wert `null` ist, erfolgt ein Laufzeitfehler (`NullPointerException`), andernfalls wird eine lokale Variable `this` vom Typ  $C$  angelegt und mit der erhaltenen Objektreferenz initialisiert.
2. Analog werden die Werte aller aktuellen Parameter  $a_1, \dots, a_n$  berechnet, lokale Variable für die formalen Parameter der Methode angelegt und mit den erhaltenen Werten der aktuellen Parameter initialisiert („**Call by Value**“).
3. Der Rumpf der Methode wird (als Block) ausgeführt.
4. Die lokalen Variablen `this`,  $x_1, \dots, x_n$  werden vom Stack genommen.

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

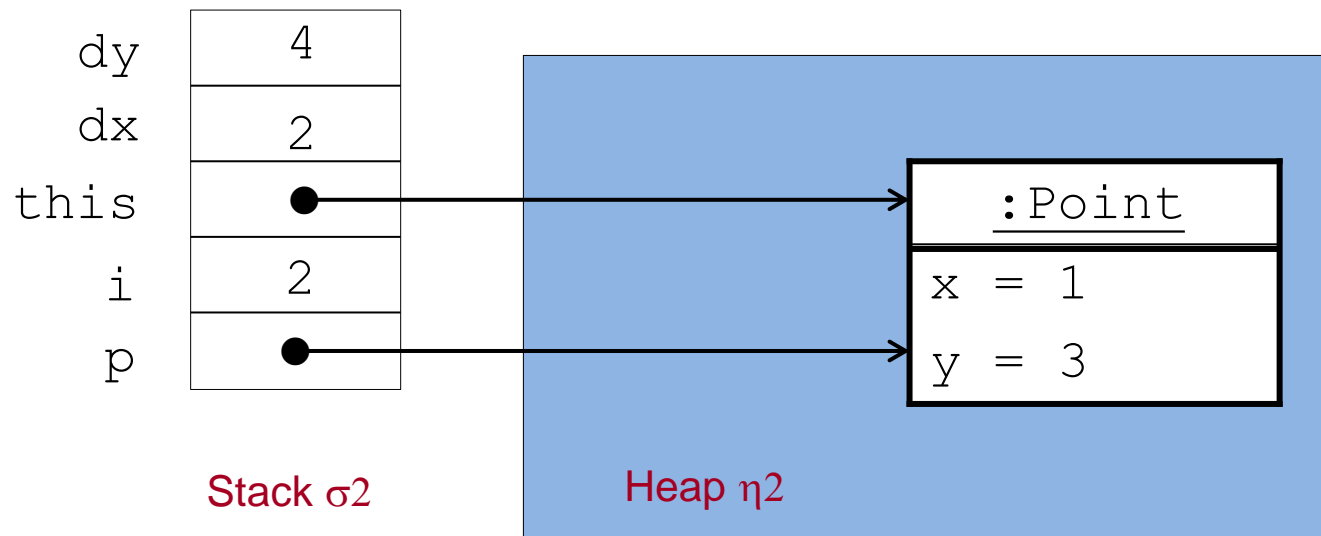
## Call-by-Value Parameterübergabe: Beispiel (1)



Im Zustand  $(\sigma_1, \eta_1)$  werde `p.move(i, 2+2);` aufgerufen.

## Call-by-Value Parameterübergabe: Beispiel (2)

Zustand nach Parameterübergabe:

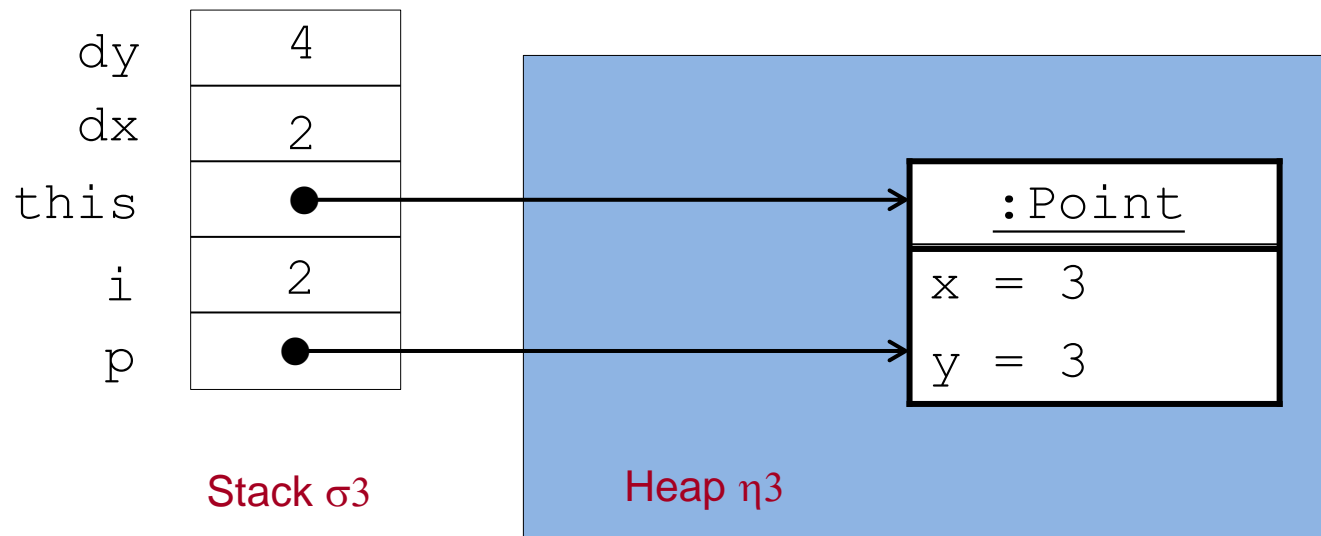


Nun wird der Rumpf der Methode `move` ausgeführt:

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

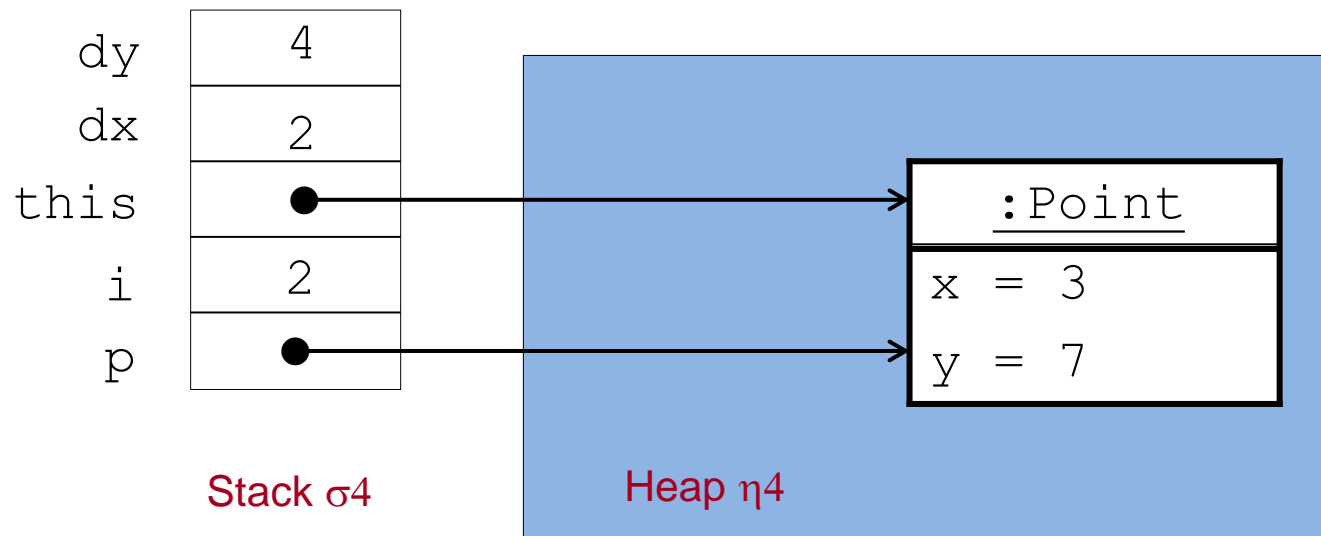
## Call-by-Value Parameterübergabe: Beispiel (3)

Zustand nach Ausführung von `this.x = this.x + dx;` :



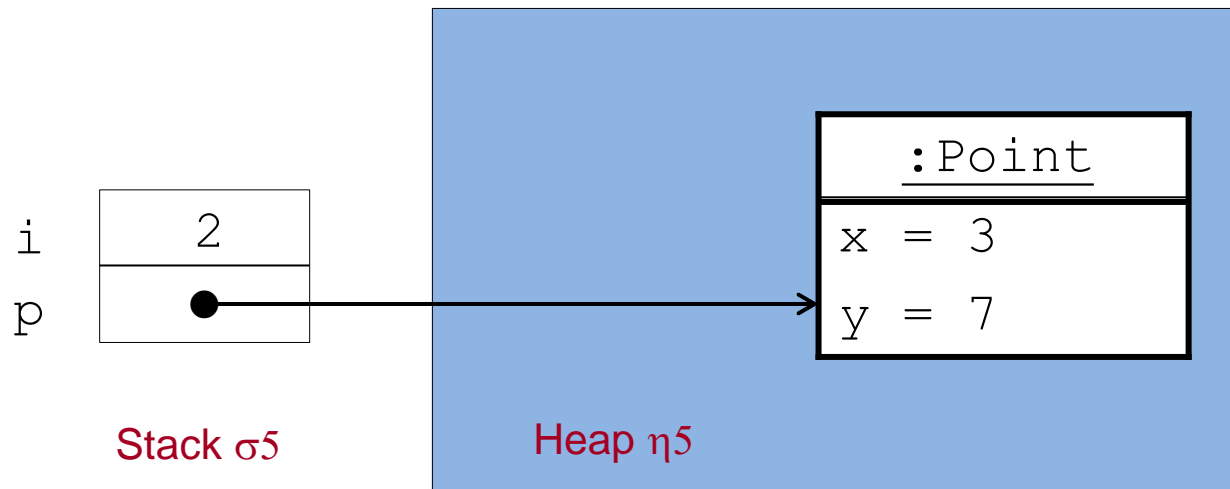
## Call-by-Value Parameterübergabe: Beispiel (4)

Zustand nach Ausführung von `this.y = this.y + dy;` :



## Call-by-Value Parameterübergabe: Beispiel (5)

Anschließend werden die lokalen Variablen `this`, `dx`, `dy` vom Stack entfernt:



## Objekterzeugungs-Anweisung

Syntax: *ClassInstanceCreation* ";"

Wdh.: *ClassInstanceCreation* = "new" *ClassType* "(" [*ActualParameters*] ")" "

Eine Objekterzeugungs-Anweisung hat also die Form

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

wobei `new C (a1, ..., an)` ein Objekterzeugungs-Ausdruck ist (vgl. oben).



## Objekterzeugungs-Anweisung: Wirkung

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

1. Ein neues Objekt der Klasse `C` wird erzeugt und auf den Heap gelegt.
2. Die Felder des Objekts werden mit Default-Werten initialisiert.  
(`0` bei `int`, `false` bei `boolean`, `null` bei Klassentypen).
3. Die Referenz auf das neue Objekt wird als Ergebniswert bereit gestellt.

Falls ein benutzerdefinierter Konstruktor aufgerufen wird, erfolgt vor 3.:

- i. Eine lokale Variable `this` mit Typ `C` wird angelegt und mit der Referenz auf das neue Objekt initialisiert.
- ii. Die Werte aller aktuellen Parameter `a1, ..., an` werden berechnet, lokale Variable für die formalen Parameter des Konstruktors werden angelegt und mit den erhaltenen Werten der aktuellen Parameter initialisiert.
- iii. Der Rumpf des Konstruktors wird (als Block) ausgeführt.
- iv. Die lokalen Variablen `this, x1, ..., xn` 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.

### Beispiel "Point":

```
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() + ")");  
        new Point(-6, 13);  
    }  
}
```

*Objekt erzeugungs-  
ausdruck*

*aktuelle Parameter*

*Aufruf einer Methode mit Rückgabewert*

*Methodenaufruf (ohne Rückgabewert)*

*Objekt erzeugungsanweisung*

## 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 = this.x + dx;
        this.y = 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!  
Dies verletzt die Idee des **Geheimnisprinzips**, nach dem Änderungen an Objektzuständen nur unter Kontrolle von Methodenaufrufen geschehen sollen.

# Benutzung von Objekten/Klassen bei öffentlichem Attributzugriff

## Beispiel "Point":

```
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=(" + p1.x + ", " + p1.y + ")");  
    }  
}
```

Zugriff auf das Attribut eines anderen Objekts

Änderung 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 = x + dx;  
    y = 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 = this.x + x;  
    this.y = this.y + y;  
}
```

## 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, die 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

```
class BankKonto {
    private double kontoStand;
    private int kontoNr;
    private static int letzteNr = 0;

    public BankKonto() {
        this.kontoNr = BankKonto.neueNr();
    }
    private static int neueNr() {
        return BankKonto.letzteNr++;
    }
    ...
}
```

## 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;  
}  
}
```

### Benutzung:

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



## Konstanten

- **Konstanten** sind Klassenattribute mit einem festen, unveränderlichen Wert.

- Syntax:

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

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

- Beispiel:

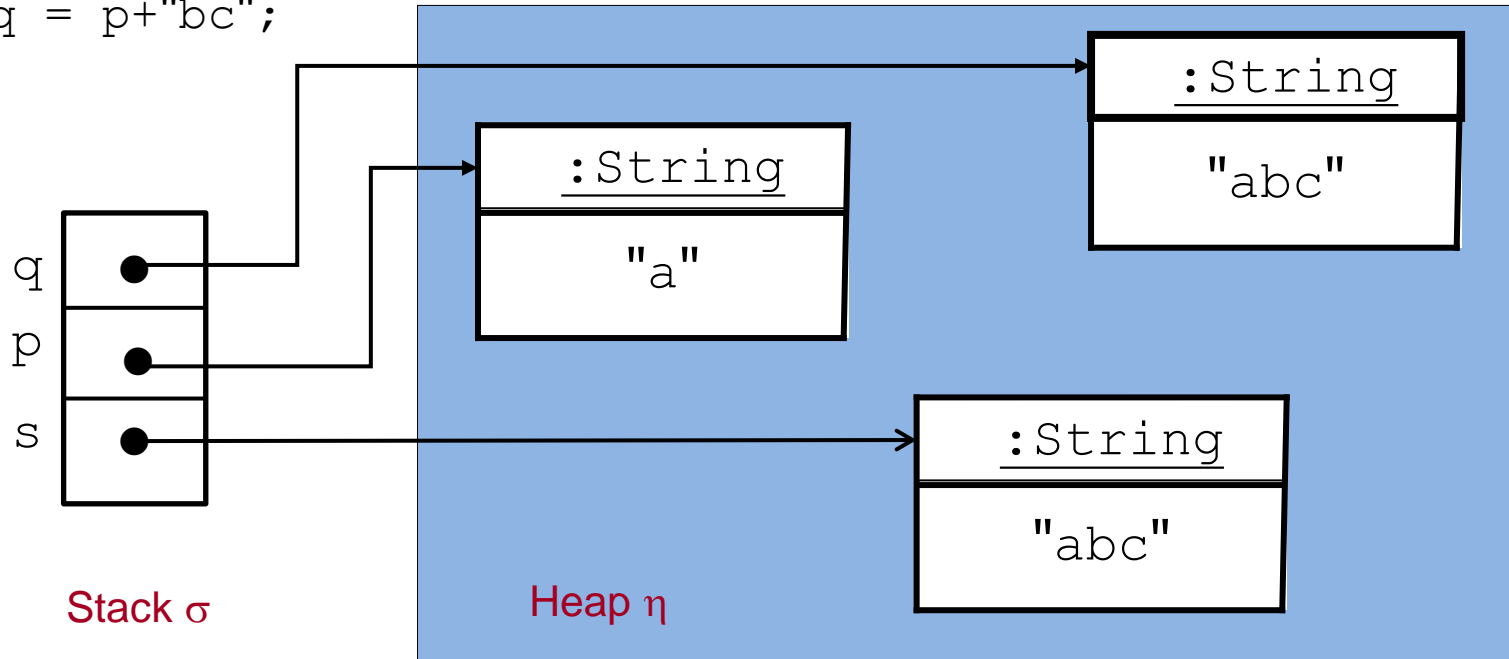
```
class Math {  
    public static final double PI = 3.14159265358979323846;  
    ... }
```

## 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 2011/12"`, `"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";
```



- 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**

# Ausschnitt aus der Java-Dokumentation der Klasse String

Method Summary	
char	<a href="#">charAt(int index)</a> Returns the char value at the specified index.
boolean	<a href="#">isEmpty()</a> Returns true if, and only if, <a href="#">length()</a> is 0.
int	<a href="#">length()</a> Returns the length of this string.
<a href="#">String</a>	<a href="#">replace(char oldChar, char newChar)</a> Returns a new string resulting from replacing all occurrences of <code>oldChar</code> in this string with <code>newChar</code> .
<a href="#">String</a>	<a href="#">substring(int beginIndex, int endIndex)</a> Returns a new string that is a substring of this string.
<a href="#">String</a>	<a href="#">toString()</a> 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`, **etc.**

**z.B.** `String s = ... ; int x = Integer.parseInt(s);`

Der String `s` muss eine ganze Zahl repräsentieren; ansonsten kommt es zu einem Laufzeitfehler (`NumberFormatException`).

Nötig beim Einlesen von numerischen Werten aus Textfeldern.

(z.B. Methoden

`public static String showInputDialog(Object message)` throws ..  
`public String getText()` **der Klassen** `JTextField`, `JTextArea`,

vgl. später).

## Umwandlungen 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`, etc.

**z.B.** `int x = ... ; String s = Integer.toString(x);`

Nötig beim Ausgeben von numerischen Werten in Textfeldern (Methode `public void setText(String t)` der Klassen `JTextField`, `JTextArea`, vgl. später).

Nicht nötig für Ausgaben mit `System.out.println`.

**Methode** `public String toString()`

kann auf Objekte aller Klassen angewendet werden.

**z.B.** `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`