

# Kapitel 10

## Verweise und Referenzen



Fachgebiet Eingebettete Systeme und ihre Anwendungen  
Prof. Dr. Andreas Koch



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



# Inhalt von Kapitel 10

## Verweise und Referenzen

### 10.1 Das Schlüsselwort `this`

- Definition
- Verwendungszweck
- Klassenmethoden

### 10.2 Objektreferenzen

- Initialisieren von Referenzen
- Zuweisung von Referenzen
- Referenzen übergeben
- Referenzen zurückgeben





## 10.1 Einleitung

Der Konstruktor der Klasse `LetterCounter` initialisiert einen String `text`. Vorsicht mit dem Scope der Variablen.

- Zugriff auf die Objektvariable mittels des Schlüsselwortes `this`.
- Das Schlüsselwort `this` ist ein Verweis auf ein Datenfeld.

```
1 public class LetterCounter {
2     // Objektvariable text
3     private String text;
4     // Konstruktor initialisiert String text
5     public LetterCounter(String text) {
6         this.text = text; // this.text ist die Objektvariable
7     }
8 }
```

Hinweis Zeile 6: `(Verweis auf Objektvariable).text = (Verweis auf Parameter).text;`





# 10.1 Definition des Schlüsselworts `this`

## Schlüsselwort `this`

Das Schlüsselwort `this` ist innerhalb einer Objektmethode ein Verweis auf das Objekt, mit dem diese Methode aufgerufen wurde.

Insbesondere erfolgt so eine Zugriff auf die

- Objekt- und Klassenvariablen
  - `this.Datenfeld`
- und Objektmethoden.
  - `this.Objektmethode`

Da diese Zugriffe sehr oft vorkommen, darf man `this` auch weglassen

→ d.h. `this.n==n`

▪ Achtung:

- Der Name einer Klassen- oder Objektvariable bzw. -konstante kann innerhalb der Methode für ein anderes Objekt noch einmal vergeben werden.
- Die Deklaration eines solchen Objekts "überdeckt" die Klassen bzw. Objektvariable/ -konstante.

→ Letztere kann von da an nur noch mit Hilfe von `this` angesprochen werden.





# Beispiel 1

## Quellcode

```
public class MeineKlasse2 {  
    int n = 1;  
    public void nocheineMethode( int n ) {  
        n = 27;  
        System.out.print( n );  
        System.out.print(" ");  
        System.out.print( this.n );  
    }  
}
```

## Testprogramm

```
MeineKlasse2 meinObjekt2 = new MeineKlasse2();  
meinObjekt2.nocheineMethode( 12 );
```

Ausgabe ?





# Beispiel 1

## Quellcode

```
public class MeineKlasse2 {  
    → int n = 1;  
    public void nocheineMethode( int n ) {  
        n = 27; ←  
        System.out.print( n ); ←  
        System.out.print(" ");  
        System.out.print( this.n ); ←  
    }  
}
```

n bezieht sich auf die Variable (den Parameter) im momentanen Scope

this.n bezieht sich auf das Datenfeld

## Testprogramm

```
MeineKlasse2 meinObjekt2 = new MeineKlasse2();  
meinObjekt2.nocheineMethode( 12 );
```

Ausgabe ? „27 1“





# Beispiel 2

## Quellcode

```
public class MeineKlasse {
    public int n = 2;
    public void meineMethode1 () {
        int n = 7;
        System.out.print ( n );
        System.out.print ( " " );
        System.out.println ( this.n );
    }
    public void meineMethode2 () {
        (this.n)++;
        System.out.print ( n );
        System.out.print ( " " );
        System.out.println ( this.n );
    }
}
```

## Testprogramm

```
MeineKlasse meinObjekt = new MeineKlasse();
meinObjekt.meineMethode1();
meinObjekt.meineMethode2();
meinObjekt.meineMethode2();
meinObjekt.meineMethode1();
```

## Ausgabe

```
7 2
3 3
4 4
7 4
```





# Verwendungszweck

Warum this? Man könnte den Variablen doch auch verschiedene Namen geben?

- Es erleichtert die Programmierarbeit aber erheblich und verbessert die Lesbarkeit eines Programms deutlich, wenn man den eindeutigen Namen verwendet.
- *Konkret.* Manchmal ist es einfach unnatürlich, verschiedenen Variablen, die eigentlich das gleiche beschreiben, unterschiedliche Namen zu geben.
  - *Beispiel:*
    - Eine Klasse `Kreis`, in der ein Kreis durch Mittelpunkt und Radius definiert wird.
    - Die entsprechenden Datenkomponenten heißen sinnvollerweise `x`, `y` und `radius`.
    - Im Konstruktor `Kreis` können die Parameter aber ebenfalls `x`, `y` und `radius` heißen.





# Beispiel - Kreis

## Quellcode

```
public class Kreis {  
    private double x;  
    private double y;  
    private double radius;  
  
    public Kreis ( double x, double y, double radius ) {  
        this.x = x;  
        this.y = y;  
        this.radius = radius;  
    }  
  
    public Kreis ( Kreis k ) {  
        x = k.x;  
        y = k.y;  
        radius = k.radius;  
    }  
}
```

Normaler Konstruktor mit Parametern

Überlagerter Konstruktor mit einem anderen Kreis-objekt zur Initialisierung (Copy-Konstruktor)





# Klassenmethoden und `this`

- Erinnerung: Zum Aufruf einer Klassenmethode bedarf es keines Objekts der Klasse.
- Objektvariable und -konstante gehören aber per Definition zu konkreten Objekten.
- Daher würde es keinen Sinn ergeben, wenn eine Klassenmethode
  - mit `this` auf dieses nicht unbedingt existierende Objekt selbst oder
  - mit oder ohne `this` auf die Objektvariablen und –konstanten dieses Objekts zugreifen oder
  - eine andere Objektmethode mit diesem Objekt aufrufen dürfte.





# Fehlerhaftes Beispiel

## Quellcode

```
public class MeineFehlerhafteKlasse {  
  
    public int n1;  
    public static int n2;  
  
    public static void meineFehlerhafteKlassenMethode1() {  
        System.out.println ( n1 );  
    }  
  
    public static void meineFehlerhafteKlassenMethode2() {  
        System.out.println ( this.n1 );  
    }  
}
```

- Beide Aufrufe sind **falsch**, da eine Klassenmethode nicht auf einem Objekt arbeitet und daher die Komponente **n1** (= eine Objektvariable) keinen Wert hat!





# Fortsetzung Fehlerhaftes Beispiel

## Quellcode

```
public class MeineFehlerhafteKlasse {  
  
    public int n1;  
    public static int n2;  
  
    public static void meineFehlerhafteKlassenMethode1() {  
        System.out.println ( n2 );  
    }  
  
    public static void meineFehlerhafteKlassenMethode2() {  
        System.out.println ( this.n2 );  
    }  
}
```

- Der erste Aufruf ist **korrekt**, da **n2** eine Klassenvariable ist!
  - Klassenvariable sind identisch für alle Objekte einer Klasse
- Der zweite Aufruf ist **falsch**, da für eine Klassenmethode kein Objekt definiert wird und daher **this** keinen Wert haben kann!





# Klassenmethoden und Objekte

## Zusammengefasst:

- Klassenmethoden können nicht direkt oder mittels `this` auf ein Objekt der eigenen Klasse zugreifen

## Aber:

- Es spricht aber nichts dagegen (und ist auch absolut korrekt), wenn eine Klassenmethode auf die Objektvariablen, -konstanten und -methoden eines **benannten Objekts** derselben Klasse zugreift.
  - Siehe Beispiel nächste Folie!





# Korrektes Beispiel

## Quellcode

```
public class MeineKorrekteKlasse {  
    public int n1;  
    public static int n2;  
  
    public void meineObjektMethode () {  
        System.out.println ( n1 );  
        System.out.println ( n2 );  
    }  
  
    public static void meineKorrekteKlassenMethode  
        (MeineKorrekteKlasse weiteresObjekt ) {  
        System.out.println ( n2 );  
        System.out.println ( weiteresObjekt.n1 );  
        System.out.println ( weiteresObjekt.n2 );  
        weiteresObjekt.meineObjektMethode();  
    }  
}
```

Korrekt, da Objektmethode

Erlaubt, da Klassen-  
methoden auf Klassen-  
variablen zugreifen dürfen  
( Zugriff auf n1 ist ungültig)

Alle Ausdruck auch in einer  
Klassenmethode erlaubt,  
da weiteresObjekt ein  
Zeiger auf ein bereits  
definiertes Objekt ist.





## 10.2 Objekte und Referenzen

### Einleitung

- Mit dem Namen eines Objekts von einem eingebauten Typ spricht man das Objekt unmittelbar an
  - d.h. die Variable speichert direkt die zugrundeliegende Repräsentation des Objekts
- Bei zusammengesetzten Objekten ist der Name des Objekts nur als eine Referenz (d.h. Verweis) auf ein anonymes Objekt vom Bausteintyp zu verstehen.
  - d.h. die Variable speichert eine Adresse, die angibt, wo die komplexe Datenstruktur zu finden ist.
- Gründe:
  - Objekte haben keine fixe Größe (können sehr groß sein), dagegen haben Zeiger die Größe einer Speicher-Adresse.
  - bei Übergabe eines Objekts an eine Methode ist es daher viel effizienter, nur die Adresse zu übergeben, anstatt den gesamten Inhalt zu kopieren!

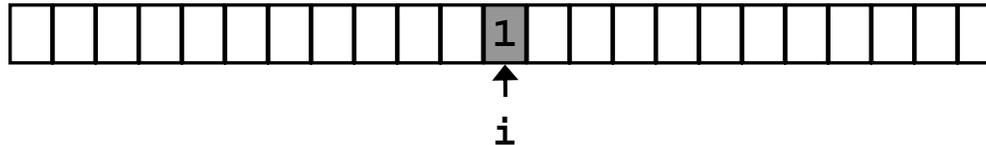




# Veranschaulichung

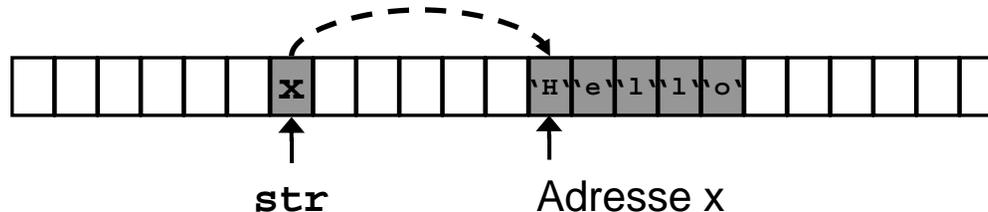
```
int i = 1;
```

- Den Bezeichner `i` kann man sich als einen symbolischen Namen für die Speicheradresse vorstellen, in der der Wert der Variable abgespeichert ist.



```
String str = new String ( "Hello" );
```

- Der Bezeichner `str` ist eher ein symbolischer Name für eine Speicheradresse, deren Inhalt die *Anfangsadresse* `x` der gespeicherten Zeichenkette ist.





# Initialisierung

- Als Konsequenz wird bei der Deklaration eines Objektes eines Bausteintyps nur der Speicherplatz für die Referenz reserviert.
  - Das Objekt selbst wird durch new generiert
  - und erst dann der zugehörige Speicherplatz reserviert.
  
- Bei eingebauten Typen wird das Objekt auf jeden Fall initialisiert!
  - Wenn nicht anders angegeben, dann mit dem Standardwert!





# Initialisierung

```
int i;
```

Objekt `i` ist schon fertig konstruiert und mit dem Standard-Wert 0 initialisiert,

```
int i = 1;
```

Objekt `i` ist schon fertig konstruiert und mit Wert 1 initialisiert.

```
String str;
```

Nur die *Referenz* `str` auf eine Zeichenkette ist eingerichtet worden, aber damit ist noch keine Zeichenkette eingerichtet worden.

```
String str = new String ( "Hello" );
```

Korrekt: Referenz `str` ist eingerichtet und mit einem String-Literal des Inhalts "Hello" initialisiert

→ Alle Objekte müssen mit `new` angelegt werden!





# Standard-Wert von Referenzen

```
int i1 = 1;
int i2;

String str1 = new String ("Hello");
String str2;
```

- Es ist klar, dass das Objekt `i1` den Wert „1“ enthält und `str1` auf ein Objekt mit Inhalt „Hello“ verweist.
- Wir wissen auch schon, dass `i2` den Standard-Wert „0“ hat.
- Das Objekt `str2` wird mit einem symbolischen Referenzwert mit Namen `null` initialisiert
  - keine wirkliche Referenz auf irgendein Objekt,
  - sondern ein "unmöglicher" Wert,
  - der nur anzeigt, dass die Variable momentan auf kein Objekt verweist.





# Nicht initialisierte Referenzen

Es gibt grundsätzlich kein uninitialisiertes Objekt in Java.

- Eine wichtige Quelle für undefiniertes Programmverhalten ist eliminiert.
- Im Gegensatz zu anderen Programmiersprachen wie C und C++.
  - *Allerdings:*
  - Wenn eine Klassenvariable den Wert `null` hat und man trotzdem auf das dahinterstehende Objekt, bzw. auf das eben **nicht** dahinterstehende Objekt zugreift, stürzt das Programm ab:

```
StringBuffer str; // == null  
str.append („test“); // Absturz!
```





# Bug oder Feature

- Dieser Absturz ist kein undefiniertes Programmverhalten sondern es ist "garantiert", dass das Programm sofort abstürzt.
  - Damit ist immerhin garantiert, dass das Programm keinen weiteren Schaden anrichtet.

## Bemerkungen:

- Die Initialisierung von Objekten einer selbstkonstruierten Klasse kann man auch selbst programmieren (bereits bekannt!).
  - Stichwort *Konstruktoren*
- Durch geeignete Java–Konstrukte kann man einen solchen Programmabsturz auch abfangen und behandeln.
  - Stichwort *Exceptions*





# Zuweisung von Referenzen

**Beachte:** Zuweisung bei Klassentypen bedeutet, dass nun zwei Referenzen auf dasselbe anonyme Objekt verweisen (im Gegensatz zu den primitiven Typen)!

- Eine Methode einer Klasse wird zwar mit dem Namen einer Variablen aufgerufen,
- aber sie macht eigentlich gar nichts mit dieser Variable.
- Statt dessen macht sie etwas mit dem *Objekt*, auf das diese Variable verweist.
- Wenn zwei Variable einer Klasse auf dasselbe Objekt verweisen, ist es also logisch, dass der Effekt eines Methodenaufrufs (z.B. `append`) mit einer Variablen (`str2`) zugleich über die andere Variable (`str1`) sichtbar wird.



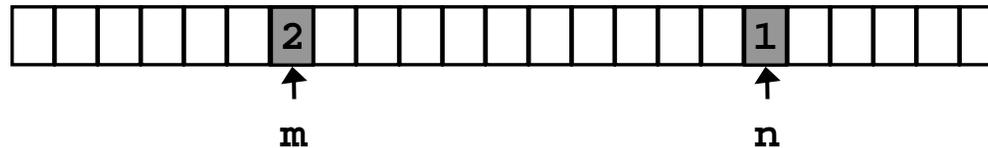


# Veranschaulichung – Eingebaute und Klassentypen

```
int n = 1;
int m = n;
m++;
```

```
System.out.print (m);
System.out.print (n);
```

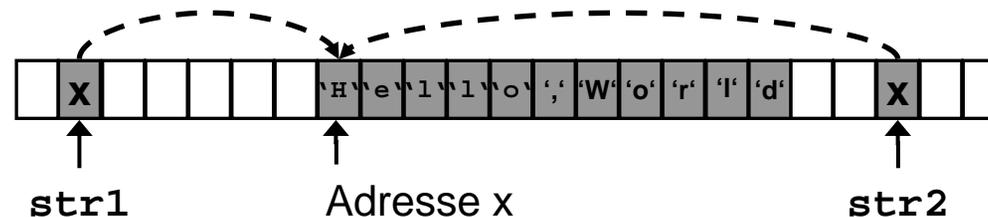
```
// Ausgabe: „2“
// Ausgabe: „1“
```



```
StringBuffer str1 = new StringBuffer ( "Hello" );
StringBuffer str2 = str1;
str2.append ( ",World" );
```

```
System.out.print ( str2 );
System.out.print ( str1 );
```

```
// Ausgabe: "Hello,World"
// Ausgabe: "Hello,World"
```





# Gleichheit von Referenzen

Test mit `==` auf Gleichheit bedeutet

- Bei den eingebauten Typen
  - Test auf **Wertgleichheit**
- im Unterschied zu den Klassentypen (zusammengesetzte Objekte)
  - Test auf **Objektidentität!**
- Jede vorgefertigte Klasse in der Java–Standardbibliothek besitzt für den Test auf Wertgleichheit von Klassentypen eine Methode namens `equals` mit folgender Signatur:
  - ein Rückgabotyp `boolean` (`true/false`) und
  - einem Parameter, dem zu vergleichenden Objekt.





# Beispiel

```
String str1 = new String ("Hello");
String str2 = str1;
String str3 = new String ("Hello");

if ( str1 == str2 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?

if ( str1.equals(str2) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?
```

```
if ( str1 == str3 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?

if ( str1.equals(str3) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?
```





# Beispiel

```
String str1 = new String ("Hello");
String str2 = str1;
String str3 = new String ("Hello");

if ( str1 == str2 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: "Ja"

if ( str1.equals(str2) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?
```

```
if ( str1 == str3 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?

if ( str1.equals(str3) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?
```





# Beispiel

```
String str1 = new String ("Hello");
String str2 = str1;
String str3 = new String ("Hello");

if ( str1 == str2 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: "Ja"

if ( str1.equals(str2) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: "Ja"
```

```
if ( str1 == str3 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?

if ( str1.equals(str3) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?
```





# Beispiel

```
String str1 = new String ("Hello");
String str2 = str1;
String str3 = new String ("Hello");

if ( str1 == str2 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: "Ja"

if ( str1.equals(str2) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: "Ja"
```

```
if ( str1 == str3 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: „Nein“

if ( str1.equals(str3) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: ?
```





# Beispiel

```
String str1 = new String ("Hello");
String str2 = str1;
String str3 = new String ("Hello");

if ( str1 == str2 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: "Ja"

if ( str1.equals(str2) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: "Ja"
```

```
if ( str1 == str3 )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: „Nein“

if ( str1.equals(str3) )
    System.out.println ( "Ja" );
else
    System.out.println ( "Nein" );
    // Ausgabe: "Ja"
```





# Übergabe von Referenzen an Methoden

Argumente von Methoden haben unterschiedliche Bedeutung für eingebaute Typen und Klassentypen.

- Einfaches Beispiel:

```
void f ( int m, StringBuffer str2 ){  
    m++;  
    str2.append ( ",World" );  
}  
  
...  
  
int n = 1;  
StringBuffer str1 = new StringBuffer ( "Hello" );  
f ( n, str1 );  
System.out.println ( n);  
System.out.println (str1);
```

// Ausgabe: ?

// Ausgabe: ?





# Übergabe von Referenzen an Methoden

Argumente von Methoden haben unterschiedliche Bedeutung für eingebaute Typen und Klassentypen.

- Einfaches Beispiel:

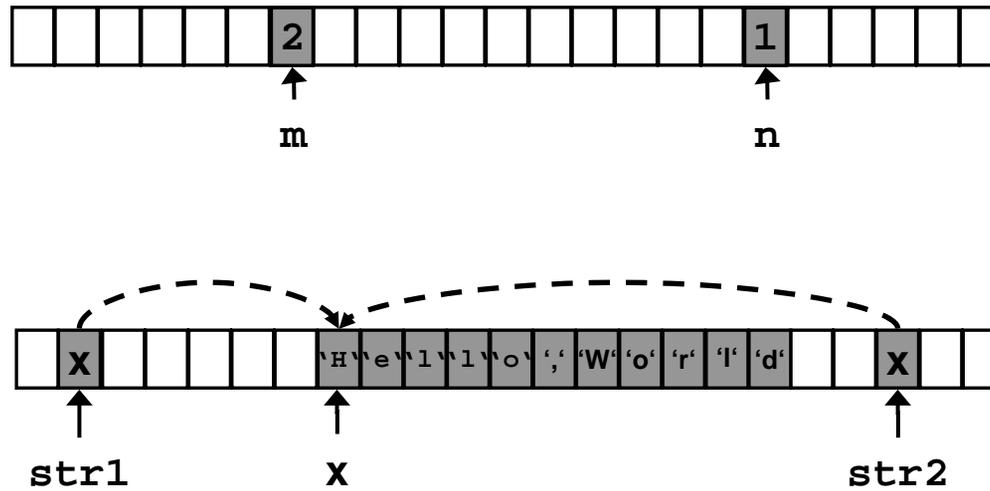
```
void f ( int m, StringBuffer str2 ){  
    m++;  
    str2.append ( ",World" );  
}  
  
...  
  
int n = 1;  
StringBuffer str1 = new StringBuffer ( "Hello" );  
f ( n, str1 );  
System.out.println ( n);  
System.out.println ( str1);
```

// Ausgabe: „1“  
// Ausgabe: „Hello,World“





# Veranschaulichung





# Referenzen als Rückgabewert

Rückgabewerte von Methoden haben unterschiedliche Bedeutung für eingebaute Typen und Klassentypen.

- Betrachten Sie folgendes Beispiel:

```
int f1 ( int n ){
    return n;
}

StringBuffer f2 ( StringBuffer string ){
    return string;
}

int m1 = 1;
StringBuffer str1 = new StringBuffer ( "Hello" );

int m2 = f1 ( m1 );
StringBuffer str2 = f2 ( str1 );

m1++;
str1.append ( ", World" );

System.out.print ( m1 );
System.out.print ( m2 );
System.out.print ( str1 );
System.out.print ( str2 );
```

```
// Ausgabe: ?
// Ausgabe: ?
// Ausgabe: ?
// Ausgabe: ?
```



# Referenzen als Rückgabewert

Rückgabewerte von Methoden haben unterschiedliche Bedeutung für eingebaute Typen und Klassentypen.

- Betrachten Sie folgendes Beispiel:

```
int f1 ( int n ){
    return n;
}

StringBuffer f2 ( StringBuffer string ){
    return string;
}

int m1 = 1;
StringBuffer str1 = new StringBuffer ( "Hello" );

int m2 = f1 ( m1 );
StringBuffer str2 = f2 ( str1 );

m1++;
str1.append ( ", World" );

System.out.print ( m1 );
System.out.print ( m2 );
System.out.print ( str1 );
System.out.print ( str2 );
```

```
// Ausgabe: "2"
// Ausgabe: "1"
// Ausgabe: "Hello, World"
// Ausgabe: "Hello, World"
```



## Kontrollfragen zu diesem Kapitel

1. Warum kann man innerhalb von statischen Methoden auf keine Datenfelder der gleichen Klasse mittels `this` zugreifen?
2. Eine Variable von einem primitiven Typ verweist direkt auf ein Objekt im Speicher. Auf was verweist eine Variable eines komplexen Typs im Speicher?
3. Was teste ich mit `==` wenn ich eingebaute Typen vergleiche? Was teste ich wenn es sich um Klassentypen handelt?

