

4.1 Objekte und Objektklassen in Java

4.2 Methoden in Java

4.3 Sichtbarkeit in Java

4.4 Modularität

4.5 Klassenhierarchien

4.6 Polymorphie und Typsicherheit

4.7 Abstrakte und Generische Klassen

4.8 Zeichenfolgen und Reihungen

4.9 Auszug aus der Syntax

4.10 Beispiel



Innensicht

Imperatives Programmieren

(Konstrukte aus Java)

- Praxis: Basistypen, Anweisungen, Verbunde, Felder, Schleifen, Rekursion
- Theorie: Syntaxbeschreibung, Induktion, Schleifeninvarianten
- Ausnahmebehandlung

Objektorientiertes Programmieren

(Java)

- OO-Klassen
- Objekt und Zustände
- Methoden und Vererbung in Java
- Java: Einführung aller restlicher OO-Sprachkonstrukte

Außensicht

Dienste

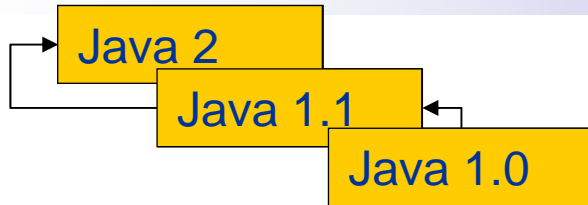
- Funktionalität/Schnittstellen
- Qualitätsparameter: Aufwand, Zuverlässigkeit, Skalierbarkeit, Persistenz
- Algorithmenwahl

Objektorientierung

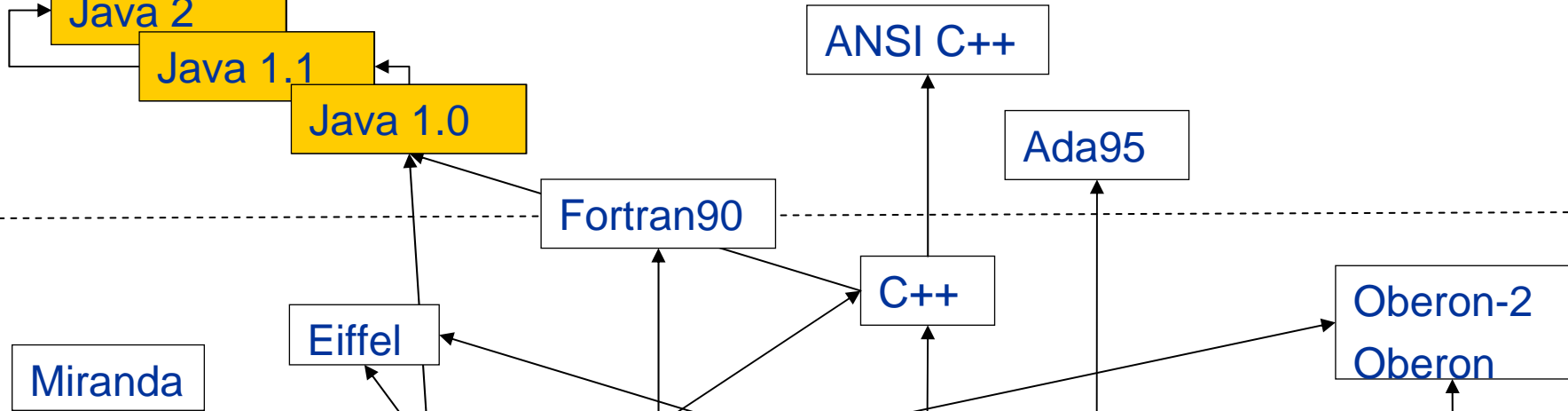
- Objekte, Zustände, Methoden
- Entwurf mittels UML
- Vererbung



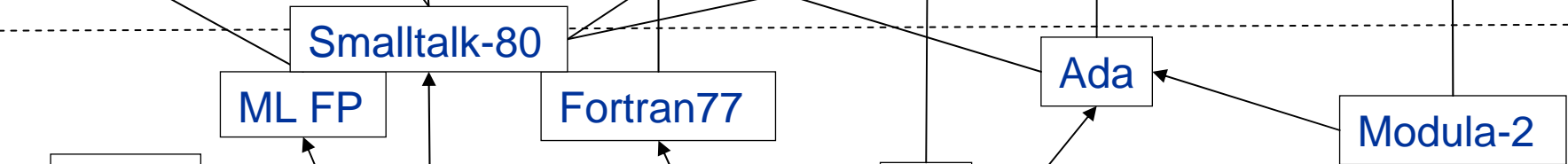
1991-2000



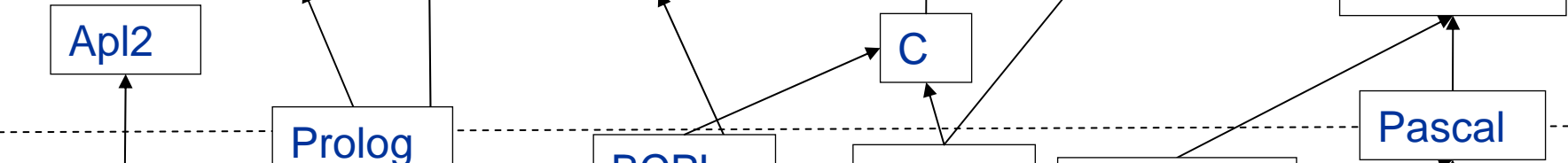
1981-1990



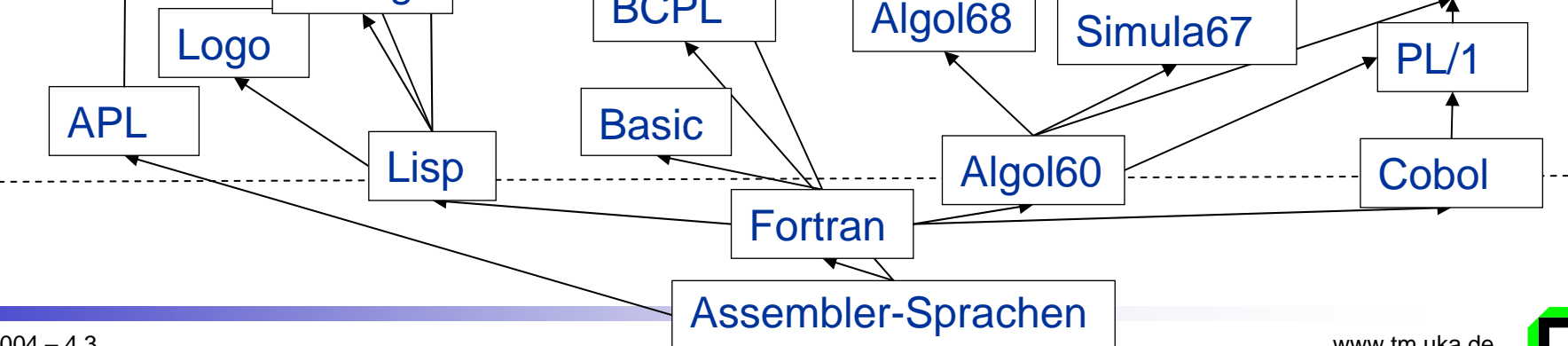
1971-1980



1961-1970



1950-1960



- Eine der vier Hauptinseln der Republik Indonesien im Indischen Ozean
- Eine aromatische Kaffee-Sorte, speziell zur Herstellung von Espresso verwendet (von den Sun Entwicklern bevorzugt)
- Ein Modetanz der 1920er
- Ein grobes, locker eingestelltes Grundgewebe aus Leinen oder Baumwolle für Stickereiarbeiten
- Eine von der Firma Sun Microsystems entwickelte, objektorientierte, plattformunabhängige Programmiersprache

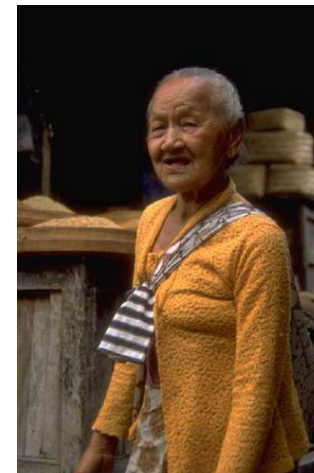
Wir beschäftigen uns natürlich mit der Programmiersprache Java ;-)



Reisfelder auf Java



Teepflücker auf Java



Javanerin im trad. Tarong



<http://www.wikipedia.de>



<http://www.wissen.de>

1990 wollte eine Gruppe von Programmierern bei Sun um **James Gosling** und **Bill Joy** eine an C++ angelehnte, jedoch bei weitem einfachere und für die Programmierung kleiner spezieller Anwendungen geeignete Programmiersprache entwickeln und im „Green Project“ von Sun anwenden.

- Speziell für Consumer Electronics, Interactive TV dort allerdings kein Erfolg
- 1993 Eignung für das Web erkannt
- Erfolg von Java rettete Sun letztendlich vor dem Bankrott.



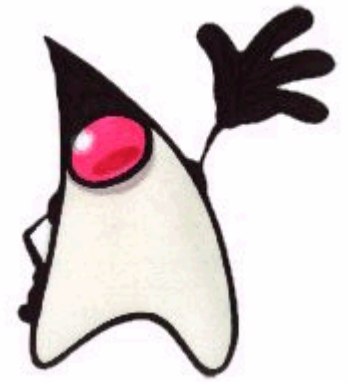
Die erste Java Anwendung *7 („StarSeven“), die sich nicht verkaufte

Java hieß damals noch „Oak“, die interaktive Filmanwendung „MovingWood“

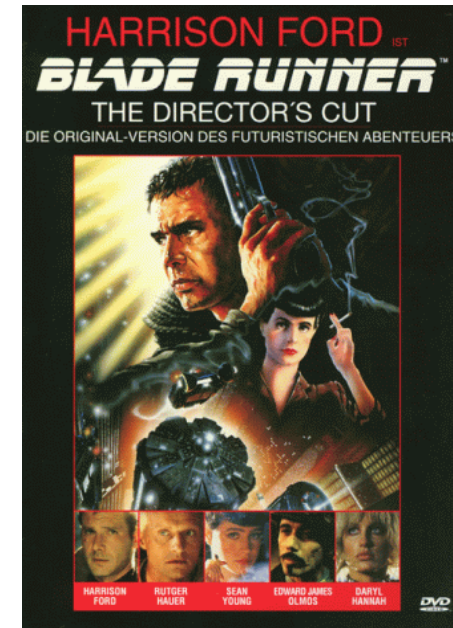
1994 Erste Implementierung von Java in einem Browser: WebRunner (nach „BladeRunner“), später HotJava

1995 Alphaversion (1.0a2) des Java-Quellcodes für die Öffentlichkeit

Durchbruch durch Integration in Netscape Browser (besiegelt durch Handschlag, Ort und Zeit: 4 Uhr morgens in einem Hotelzimmer des Sheraton Palace Hotels in der Nähe des Convention Centers)



Duke, das Java Maskottchen



Blade Runner, der Film



<http://www.wired.com/wired/archive/3.12/java.saga.html>



James Gosling



Gosling über Microsoft:

“It was always like an unspoken dirty secret, [people would say] 'Microsoft, they're our valued partner'. But give an IT manager a few beers and they'll admit: 'Shit, they fucked us so badly!'”

- Vice-President bei Sun
- Bloggt unter <http://today.java.net/jag/>
- Dr. an der Carnegie-Mellon University
- Bachelor in Informatik an der University of Calgary
- Hat Emacs für Unix geschrieben



„I skipped a lot of classes at high school going over to the university. But by and large, my high school instructors were pretty cool about it because most people were skipping class to go off and do drugs, and here I was, writing software for satellite ground stations.”



Objekt

- Besitzt eindeutige Identität
- Autonome, gekapselte Einheit eines bestimmten Typs
- Eigener Zustand, der von außen nicht eingesehen werden kann (lokale Variable)
- Code zur Manipulation des Zustands
- Hat ein Verhalten
- Bietet anderen Objekten Dienstleistungen an

→ Dienstgeber

Zustand (state)

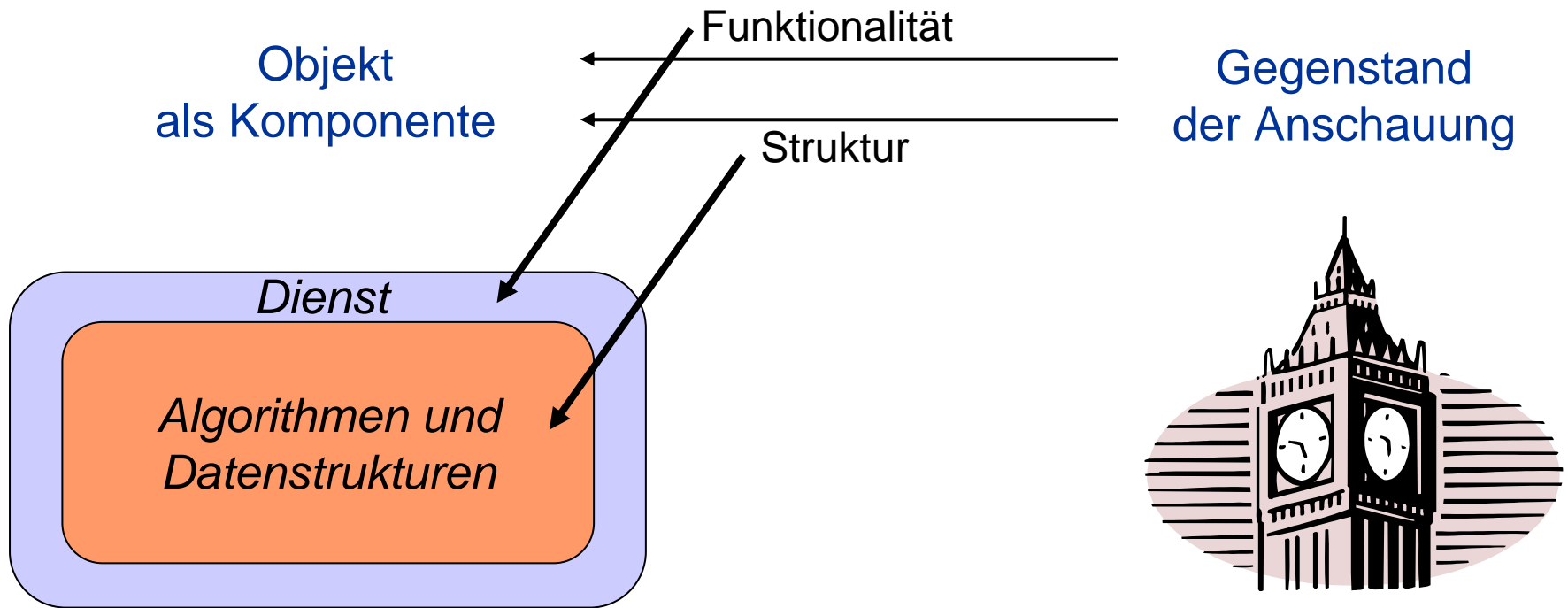
- Beschrieben durch einen Verbund von Attributwerten bzw. Daten
- Kann sich im Laufe der Zeit ändern

Verhalten (behavior)

- Beschrieben durch eine Menge von Handlungsvorschriften zur Erbringung eines Dienstes durch ein Objekt (Methode)
- Die Ausführung eines Dienstes geschieht durch Mitteilung eines Ereignisses samt zugehöriger Daten (Methodenparameter) an das Objekt (Botschaft)



4.1 Objekte und Objektklassen in Java



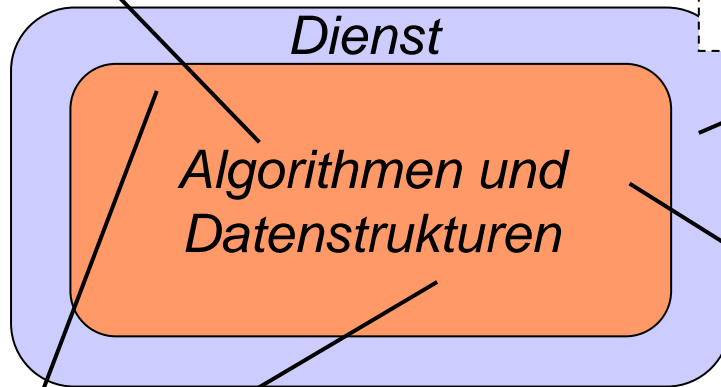
Objekt
als Komponente

Funktionalität

Gegenstand
der Anschauung

Struktur

Schnittstelle (interface)



Operatorensignaturen

Algorithmen und
Datenstrukturen

Klasse (class)

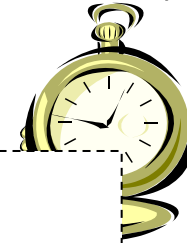
Attribute
Assoziationen
Operatorimplementierungen

Objektklasse
als Komponenten-
Baumuster

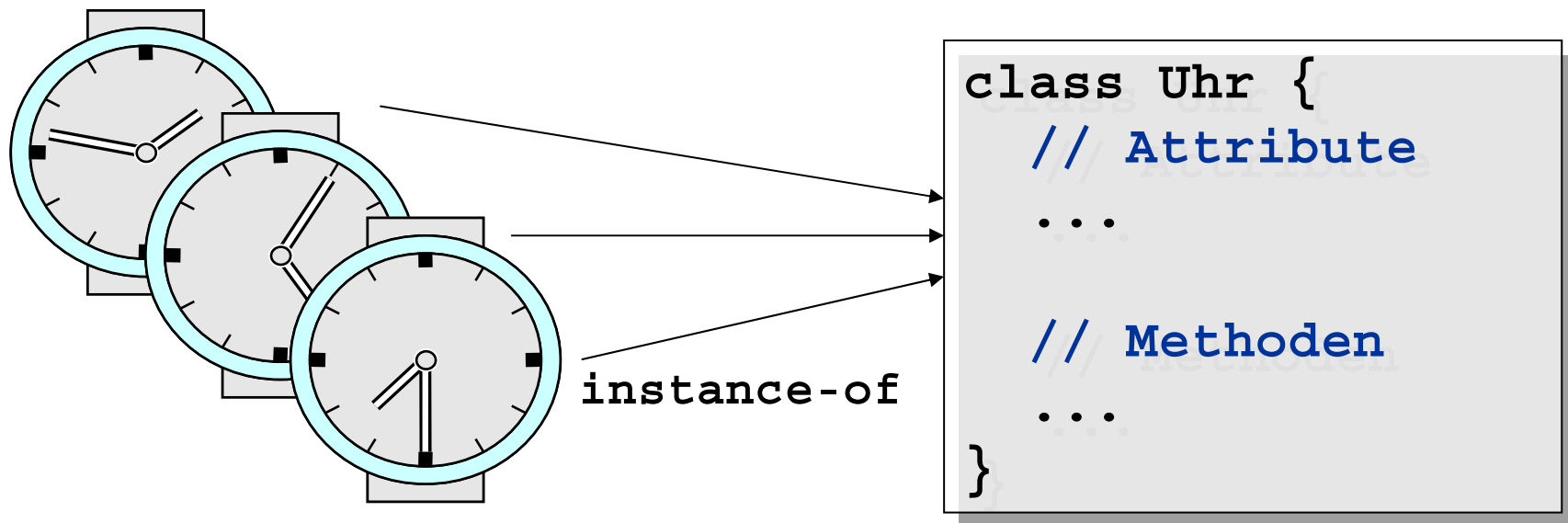
Funktionalität

Menge gleichartiger
Gegenstände der
Anschauung

Struktur



- Struktur- und verhaltensäquivalente Objekte werden zu Objektklassen zusammengefasst.
- Die Objekte bilden dann die Ausprägungen dieser Klasse (instance, class instance).



Variable von einfachen Typen

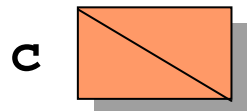
- Mit Variablen einfachen Typs wird bei deren Vereinbarung zugleich ein Behälter für den Wert bereitgestellt
- Mit Angabe des Namens wird unmittelbar der Wert angesprochen (Wertesemantik)



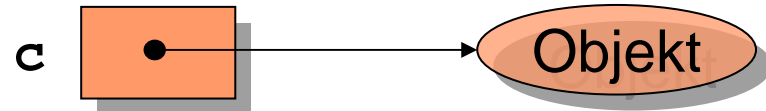
Variable vereinbaren z.B. durch `int a;`
Wert eingebracht z.B. durch `a = 5;`

Referenzvariablen

- Mit Objektvariablen wird nur ein Behälter für einen Zeiger auf ein Objekt des entsprechenden Typs angelegt (Referenzsemantik)
- Der eigentliche Speicherplatz für die Objektinstanz muss dann noch angelegt werden und ist somit nicht fest angebunden



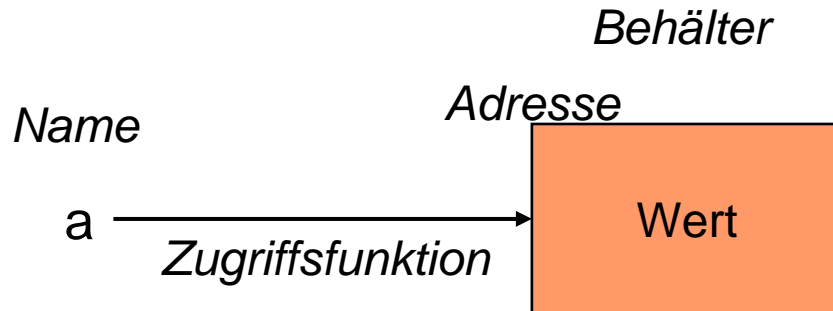
Objekt nicht zugewiesen



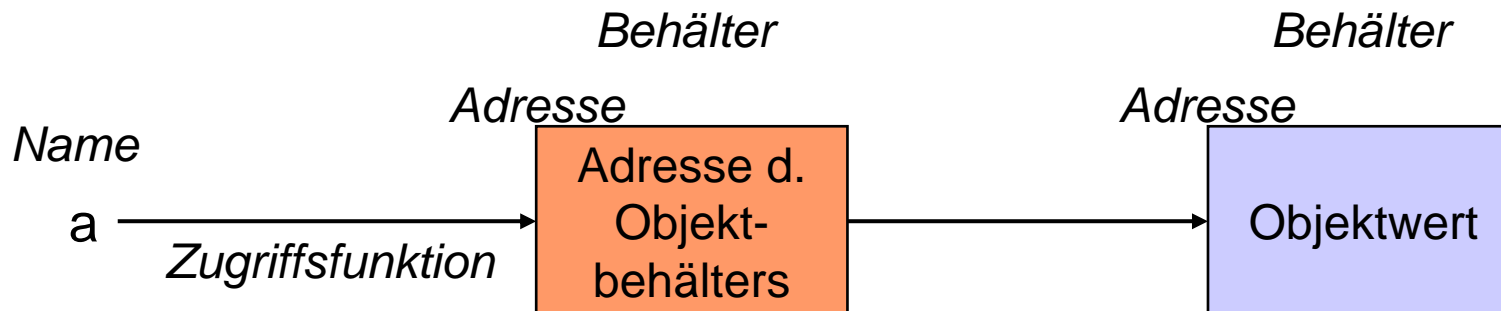
Objekt zugewiesen

Veranschaulichung:

Wertsemantik



Referenzsemantik



- Über die Darstellung von Referenzen ist dem Programm nichts bekannt, mit Referenzen kann man nicht rechnen.

Attribute (attribute, field)

Als Attribute sind Variablen mit Werte- sowie Referenzsemantik erlaubt. Gemeinsam stellen sie den Zustand eines Objektes dar.

```
class Uhr {  
    long minuten;  
    long stunden;  
  
    TimeZone zeitzone;  
}
```

Wertesemantik

Referenzsemantik



- Ein Konstruktor erzeugt ein Objekt einer angegebenen Klasse und belegt den hierfür nötigen Speicherplatz bzw. nimmt weitere Initialisierungen vor.
- Jede Java-Klasse hat den impliziten Konstruktor `<Klassenname>()`.
- Ein Konstruktor wird mit Hilfe des Operators `new` aufgerufen.

```
class Uhr {  
    long minuten;  
    long stunden;  
    TimeZone zeitzone;  
}
```

```
...  
Uhr c = new Uhr();  
...
```

Erzeugt neue Instanz und liefert Referenz auf diese

Behälter für Referenzvariable

Impliziter Konstruktor



Impliziter Konstruktor:
Attribut wird bei
Objekterzeugung mit
dem angegebenen
Wert vorbesetzt

Der Modifikator
final kennzeichnet
Konstanten

Expliziter Konstruktor
mit Initialisierungscode
u. Eingabeparameter

```
class Uhr {  
    long minuten;  
    long stunden;  
    TimeZone zeitzone =  
        new TimeZone ("UTC");  
    final Date herstellungsdatum =  
        new Date () ;  
  
    Uhr (long min) {  
        stunden = min / 60;  
        minuten = min % 60;  
    }  
}
```

Impliziter Konstruktor:
Attribut wird bei
Objekterzeugung mit
dem angegebenen
Wert vorbesetzt

①

```
class Uhr {  
    long minuten;  
    long stunden;  
    TimeZone zeitzone =  
        new TimeZone ( "UTC" );  
    final Date herstellungsdatum =  
        new Date () ;  
  
    Uhr ( long min ) {  
        stunden = min / 60;  
        minuten = min % 60;  
    }  
}
```

Expliziter Konstruktor
mit Initialisierungscode
u. Eingabeparameter

②

```
...  
Uhr c = new Uhr (600);  
...
```



- Bei der Vereinbarung von Objektvariablen wird keine Instanz angelegt.
- Referenzvariablen ohne zugeordnetes Objekt besitzen die leere Referenz `null`, auf die geprüft werden kann.
- Referenzvariablen können bei ihrer Vereinbarung oder später initialisiert werden.

```
Uhr c = null;  
...  
// Falls Objekt noch nicht erzeugt wurde,  
// jetzt Instanz anlegen.  
if ( c == null ) {  
    c = new Uhr ();  
}
```



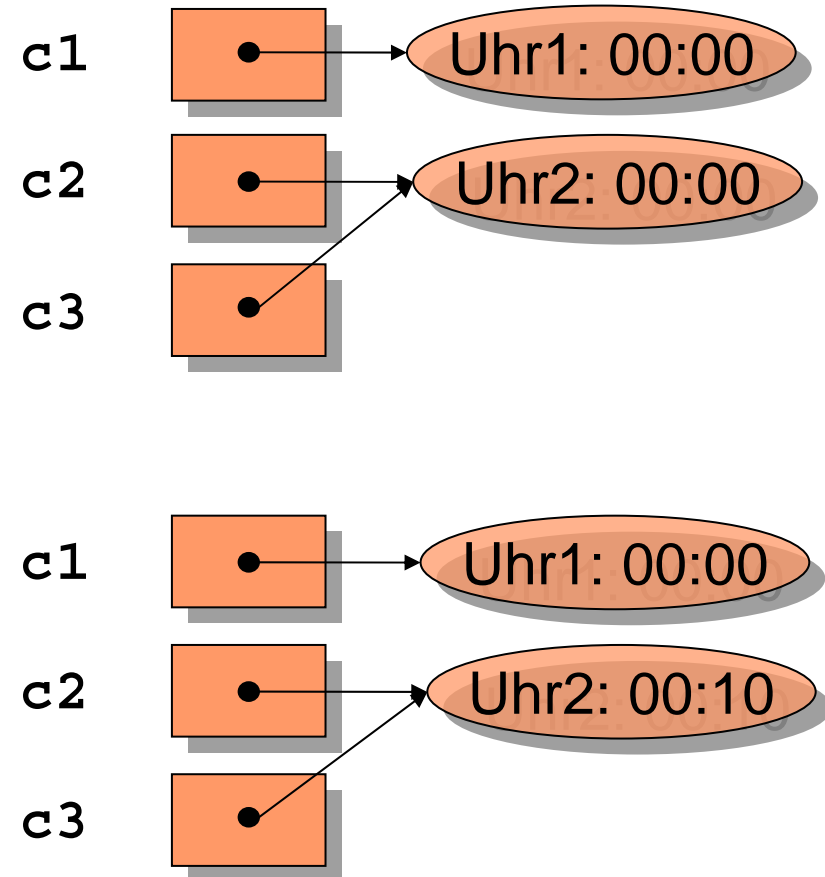
- Ein Vergleich auf Referenzvariablen vergleicht nur die Referenzen selbst und nicht die Objekte, auf die verwiesen wird.

```
Uhr c1 = new Uhr (0);
Uhr c2 = new Uhr (0);
Uhr c3 = c2;

// c1 == c2 liefert false
// c1.minuten == c2.minuten
// liefert true

c3.minuten = c3.minuten + 10;

// c2 == c3 liefert true
// c2.minuten == c3.minuten
// liefert true
// c1.minuten == c2.minuten
// liefert false
```



- Bei Zuweisung an Referenzvariablen wird der Verweis kopiert und nicht das Objekt, auf das verwiesen wird.

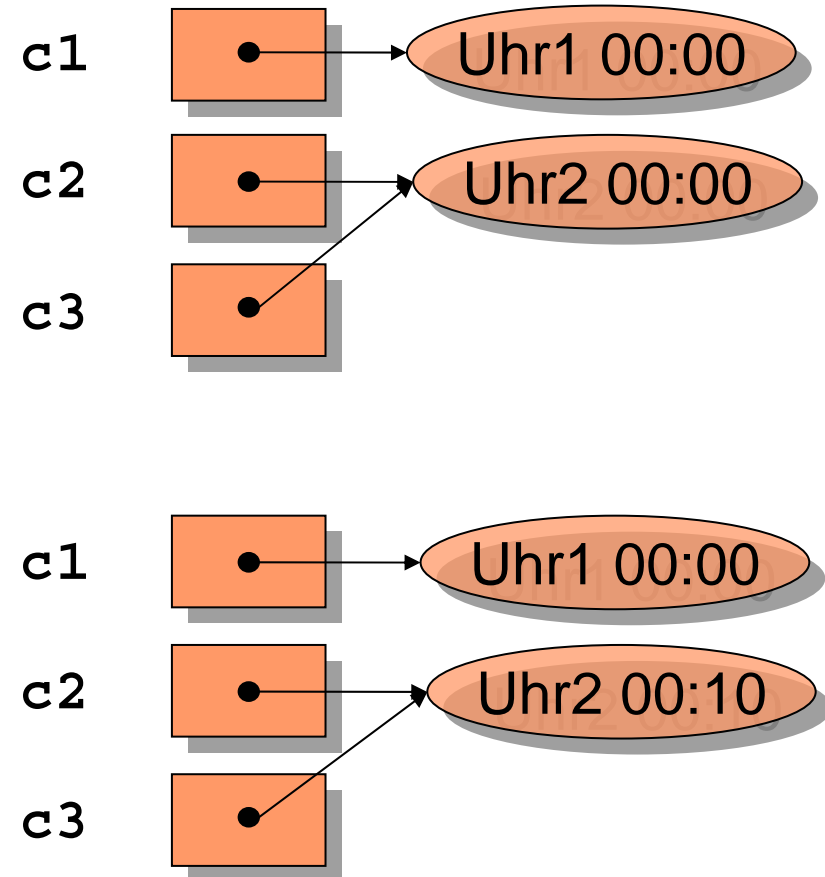
```

Uhr c1 = new Uhr (0);
Uhr c2 = new Uhr (0);
Uhr c3 = c2;

// c1 == c2 liefert false
// c1.minuten == c2.minuten
// liefert true

c3.minuten = c3.minuten + 10;

// c2 == c3 liefert true
// c2.minuten == c3.minuten
// liefert true
// c1.minuten == c2.minuten
// liefert false
    
```



- Referenzkonstanten fixieren den Verweis; das Objekt, auf das verwiesen wird, kann verändert werden.

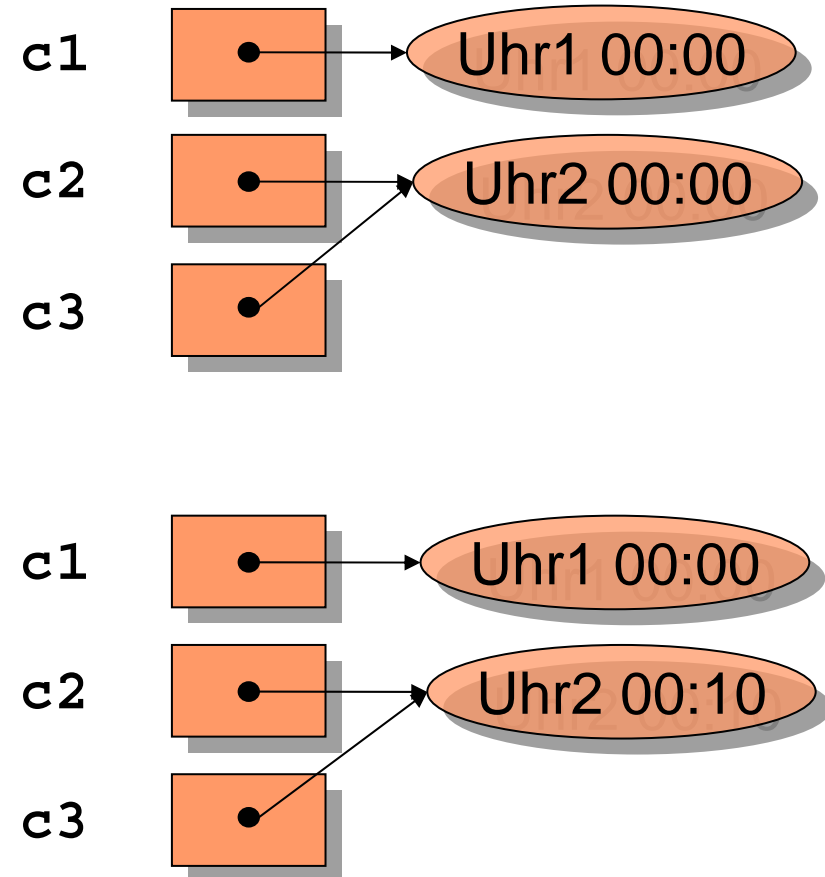
```

Uhr c1 = new Uhr (0);
Uhr c2 = new Uhr (0);
final Uhr c3 = c2;

// c1 == c2 liefert false
// c1.minuten == c2.minuten
// liefert true

c3.minuten = c3.minuten + 10;

// c2 == c3 liefert true
// c2.minuten == c3.minuten
// liefert true
// c1.minuten == c2.minuten
// liefert false
    
```



Spezielle Methode, die ein Objekt vernichtet, d.h. den belegten Speicherplatz freigibt

Fehleranfällig, wenn dieser von Hand im Programmcode aufgerufen werden muss

Daher in Java durch Speicherbereinigung (Garbage Collection) automatisiert:

- Die Speicherbereinigung ist ein spezielles Objekt der Laufzeitumgebung (virtuellen Maschine), das die Objektgraphen im Speicher durchsucht.
- Wird ein Objekt gefunden, das nicht mehr referenziert wird, wird dieses gelöscht und der von diesem belegte Speicher freigegeben.



- Attribute, die der Klasse als Ganzes zugeordnet sind. In Java werden diese mit dem Modifikator `static` eingeleitet.

```
class Uhr {  
    static long zaehler = 0;  
    long        seriennummer;  
  
    Uhr () {  
        seriennummer = zaehler;  
        zaehler++;  
    }  
}
```

```
Uhr c1 = new Uhr ();  
Uhr c2 = new Uhr ();  
// c1.seriennummer ist 0  
// c2.seriennummer ist 1
```



Methode (method, operation):

- In der Objektorientierung senden Objekte Nachrichten, die zur Aktivierung von Methoden führen (siehe UML-Sequenzdiagramm).
- In den meisten Programmiersprachen ist aber das Senden und Empfangen der Botschaften zum Aufruf von Methoden degeneriert.

Nach außen wird eine Methode durch ihre Signatur bestehend aus Ergebnistyp, Name sowie Anzahl und Typen der Parameter beschrieben:

- `<Ergebnistyp> <Name> (<Parameterliste>)`
- Die Deklaration von zwei Methoden mit derselben Signatur in einer Klasse ist nicht erlaubt.
- In Java gehört der Ergebnistyp nicht zur Signatur.

Aufrufe einer Methode:

- `<Objektvariable>.<Name> (<Parameterliste>)`



Übergabe von Eingabeparametern

- Die Spezifikation eines Parameters wird als lokale Vereinbarung einer Variablen im Methodenrumpf aufgefasst.
- Beim Aufruf wird diese Variable mit dem Wert des Arguments vorbesetzt. Parameterausdrücke werden daher zuvor berechnet.

Wertaufruf (*call-by-value*)

- Wert des Parameters wird in die lokale Variable kopiert (Wertsemantik)

Referenzaufruf (*call-by-reference*)

- Parameter enthält Referenz auf ein Objekt (Referenzsemantik)

Ergebnisrückgabe

- Mit der Anweisung `return <ausdruck>` innerhalb eines Methodenrumpfes wird die Ausführung der Methode beendet und ein Ergebnis zurückgegeben.
- Der Ergebnistyp `void` zeigt an, dass kein Ergebnis zurückgeliefert wird. Die Methode kann (bei Bedarf) mit `return` beendet werden.



Wertaufruf

```
class Uhr {  
    ...  
    void setzeZeit (long std, long min) {  
        stunden = std; minuten = min;  
    }  
    long leseZeitInSekunden () {  
        return (stunden * 3600) + (minuten * 60);  
    }  
    ...  
}
```

```
Uhr c = new Uhr ();  
c.setzeZeit ( 11, 47 );    // setzt die Uhrzeit auf 11:47 Uhr  
long l = c.leseZeitInSekunden (); // l ist 42420
```



Referenzaufruf

```
class Uhr {  
    ...  
    void setzeZeitZone (TimeZone tz) {  
        if ( tz == null ) { return; }    // Test, ob Referenz  
        zeitzone = tz;                  // gültig  
    }  
    TimeZone leseZeitZone () {  
        return zeitzone;  
    }  
    ...  
}
```

```
Uhr c      = new Uhr ();  
TimeZone tz = new TimeZone ("MEZ");  
c.setzeZeitZone (tz);  
tz.setzeIdentifikator ("ET");  
// c.timezone ist jetzt ebenfalls auf "ET"
```



```
class rational {  
    int zähler, nenner;  
  
    rational(int z, int n) {  
        zähler = z; nenner = n;  
    }  
  
    rational rationaladd(rational r) {  
        return new  
            rational(zähler * r.nenner + nenner * r.zähler,  
                    nenner * r.nenner).kürzen();  
    }  
    ...  
}
```

- Alle Methoden enthalten einen ersten, impliziten Parameter **this** mit der vereinbarten Klasse als Typ.
- Beim Versenden einer Botschaft an ein Objekt dieser Klasse wird **this** dann durch dieses Empfängerobjekt aktualisiert.
- Folge: Mit der Botschaft begibt man sich in den Zustandsraum des Empfängerobjektes, um die zugehörige Methode auszuführen.



```
class rational {
    int zähler, nenner;

    rational(int z, int n) {
        zähler = z; nenner = n;
    }

    rational rationaladd(rational r) {
        return new
            rational(zähler * r.nenner + nenner * r.zähler,
                    nenner * r.nenner).kürzen();
    }
    ...
}
```

- Zu interpretieren als `this.zähler`, `this.nenner`.
- Die Attribute des Empfängerobjektes befinden sich in dessen lokalem Gültigkeitsbereich, auf sie kann daher unmittelbar zugegriffen werden.
- `this` kann man explizit verwenden, wenn das Objekt seine eigene Referenz benötigt.

- Botschaft an das durch `r` referenzierte Objekt, um dessen Werte `nenner` und `zähler` zu beschaffen.
- Dazu wird dessen Zustandsraum betreten.



- Statische Methoden sind einer Klasse und nicht einem Objekt zugeordnet.
- Sie können nicht auf den Zustandvariablen, sondern nur auf Klassenvariablen operieren.
- Sie werden mit `static` gekennzeichnet.
- Die statische Methode `main` kennzeichnet den Programmstart.

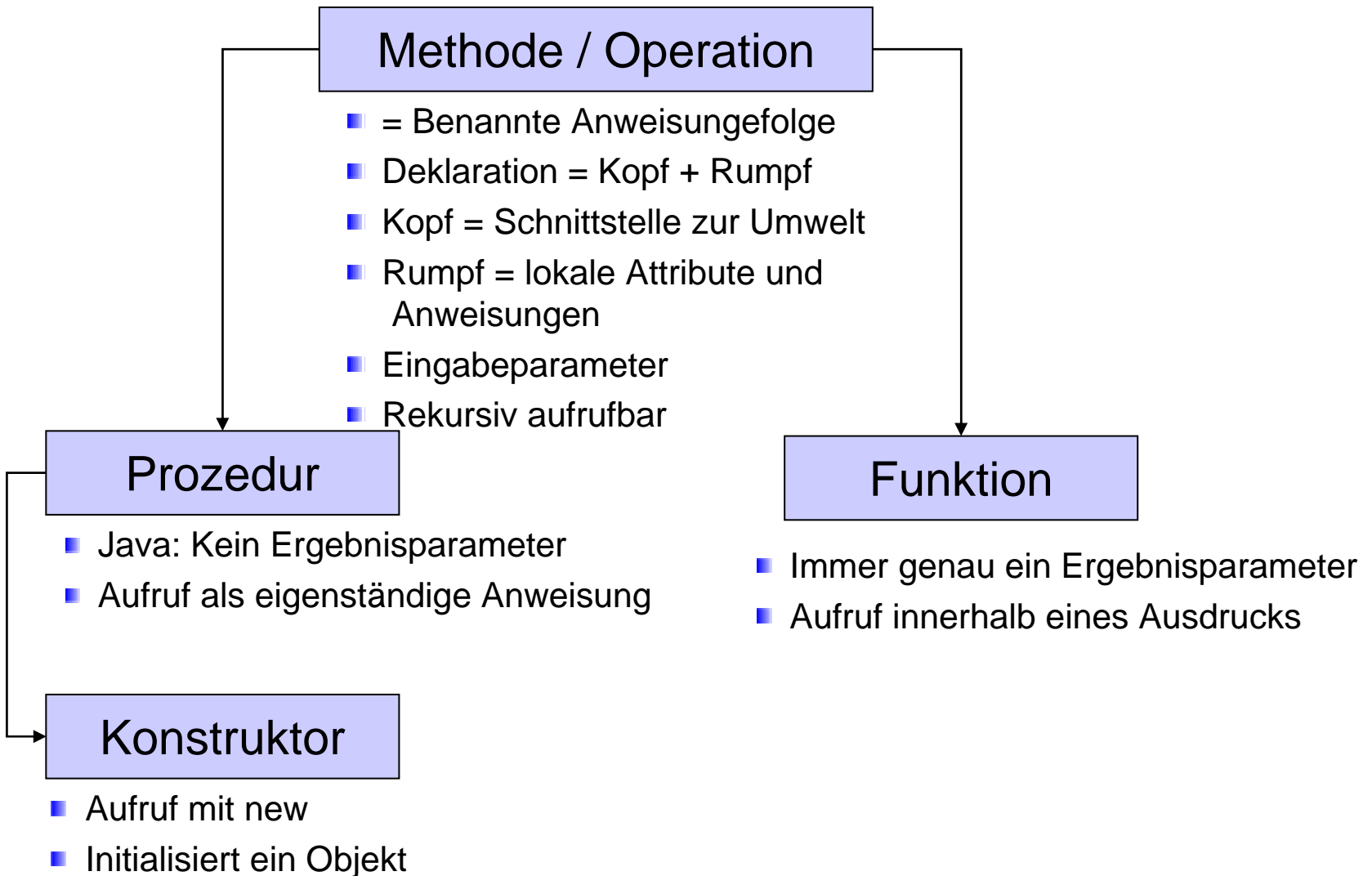
```
class Uhr {  
    ...  
    public static void main (String[] args) { // Programmstart  
        Uhr c = new Uhr ();  
        c.setzeZeit ( 10, 00 );  
    }  
} // Programmende
```



- Eine Methode kann innerhalb einer Klasse mit einer Methode gleichen Namens, aber unterschiedlicher Anzahl von Parametern oder anderen Parametertypen überladen werden.
- Beide besitzen somit unterschiedliche Signaturen.
- Ein Konstruktor kann ebenfalls überladen werden.

```
class Uhr {  
    ...  
    void setzeZeit (long std, long min) {  
        stunden = std; minuten = min;  
    }  
    void setzeZeit (long std) {  
        setzeZeit (std, 0);           // Aufruf der Methode mit  
    }                                 // zwei Parametern  
    ...  
}
```





```
class Uhr {  
    long minuten;  
    long stunden;  
  
    void setzeZeit (long stunden,  
                   long minuten) {  
        this.minuten = minuten;  
        this.stunden = stunden;  
    }  
    Uhr kopiere () {  
        Uhr c = new Uhr ();  
        c.stunden = stunden;  
        c.minuten = minuten;  
        c.zeitzone = zeitzone;  
        return c;  
    }  
    ...  
}
```

minuten, stunden sichtbar
außerhalb von Uhr

minuten, stunden sichtbar
innerhalb von Uhr

Parameter sichtbar in `setzeZeit`

Attribute der Klasse `Uhr` überdeckt
in `setzeZeit`, sichtbar über
Referenz `this`

`c` sichtbar in `kopiere`

Attribute von `c` sichtbar in `kopiere`

`this` nicht nötig, da Attribute der
Klasse `Uhr` nicht überdeckt

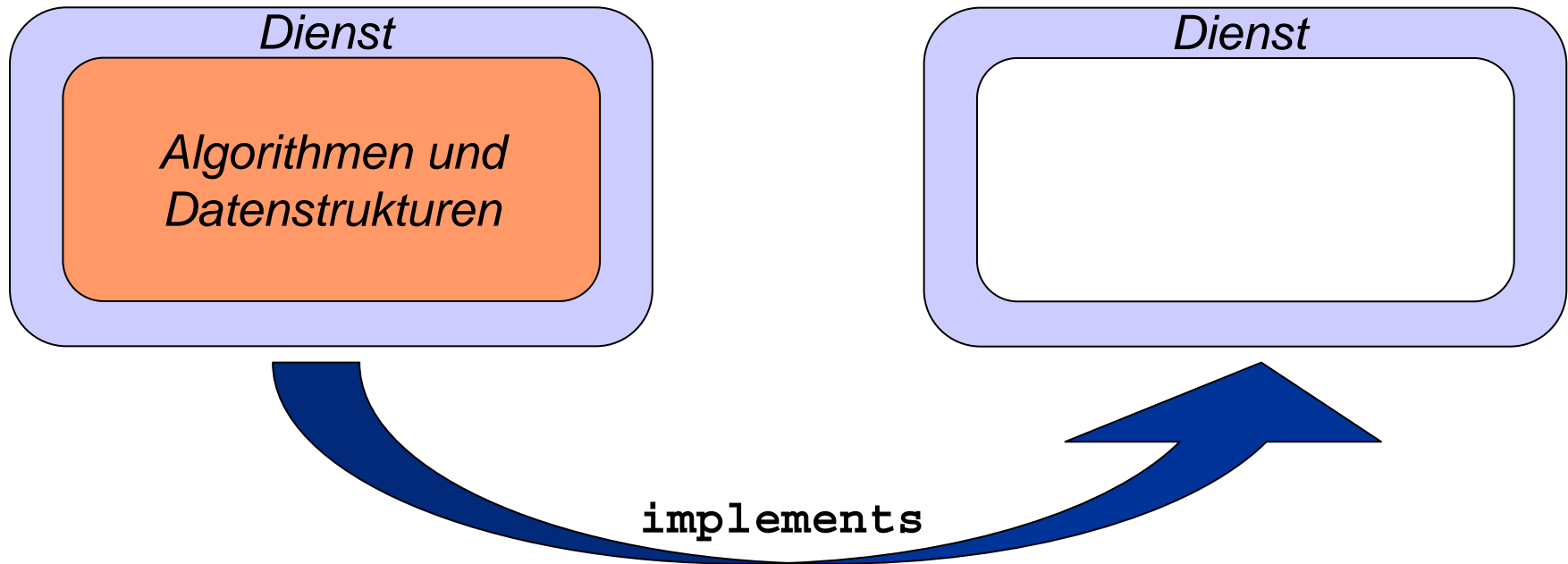
Dem Benutzer einer Klasse sollten die Interna der Klasse nicht zugänglich sein:

- Geheimnisprinzip:
 - Kenntnisse über Attribute und Methodenimplementierungen gelangen nicht nach außen.
- Unabhängigkeit:
 - Programmierung möglich ohne Rücksicht auf andere; Änderung einer Implementierung hat keine Effekte auf andere.
- Für die Benutzer dürfen nur Schnittstellen nach außen sichtbar sein; das sind folglich die Methodensignaturen (Benutzersicht).
- Aber: Die Klassenvereinbarungen belassen volle Einsicht in die Implementierung. Diese dürfen daher nur einem Ersteller der Klasse bekannt sein (Anbietersicht).



Klasse (`class`)

Schnittstelle (`interface`)



Eine Schnittstelle ist an keine Methoden-Implementierungen gebunden; sie beschreibt nur deren Signaturen.

Eine Implementierung der angegebenen Methoden muss also in einer Klasse erfolgen.

Eine Klasse kann ein oder mehrere Schnittstellen implementieren.

Bei Objektvariablen kann als Typ statt einer Klasse eine Schnittstelle angegeben werden.

- Die Referenz kann aber nur auf ein Objekt verweisen
- Wird ihr ein Objekt zugewiesen, so muss dessen Klasse die Schnittstelle implementieren



- Schnittstellen werden in Java durch das Schlüsselwort `interface` eingeleitet.
- Durch `implements` wird angegeben, dass eine Klasse mindestens die in der Schnittstelle angegebenen Methoden implementiert.

```
interface Taktgeber {  
    long leseZeitInSekunden ();  
}
```

```
class Uhr implements Taktgeber {  
    ...  
    long leseZeitInSekunden () {  
        return (stunden*3600)+(minuten*60);  
    }  
    ...  
}
```

```
Taktgeber t = new Uhr ();  
long l = t.leseZeitInSekunden ();
```



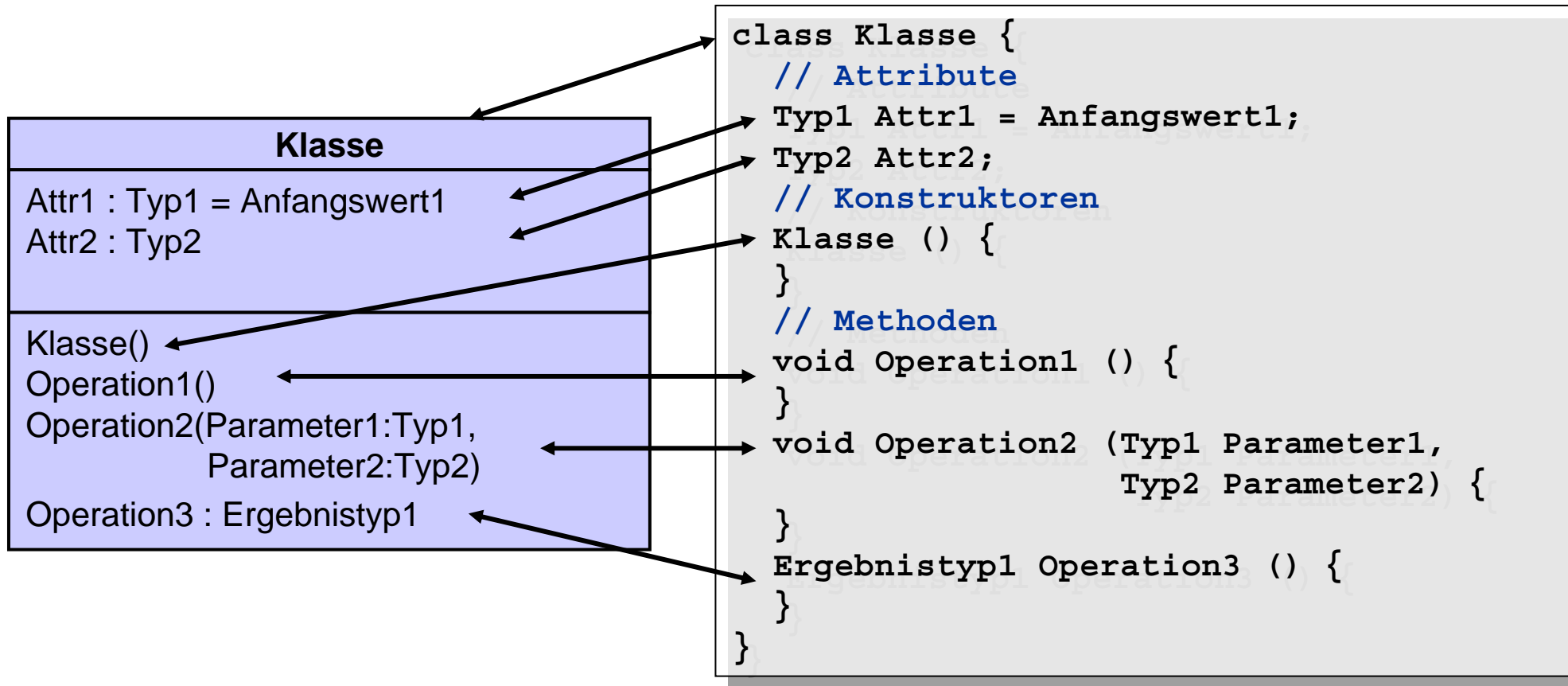
Abschwächung: Der Zugriff auf Klassen, Attribute und Methoden von außen kann durch den Klassenimplementierer mittels Modifikatoren (modifier) eingeschränkt werden.

- `private` gibt an, dass nur von innerhalb der Klasse selbst auf das Attribut bzw. die Methode zugegriffen werden kann.
- `public` gibt an, dass ein Zugriff von überall auf das Attribut bzw. die Methode möglich ist.

... und ohne Modifikator?

```
public class Uhr {  
    // Zugriff auf Attribute von außen nicht möglich  
    private long minuten;  
    private long stunden;  
    // Zugriff von außen nur über Methoden möglich  
    public Uhr () { ... }  
    public void setzeZeit (long std, long min) { ... }  
    ...  
}
```





Modul:

- Ein Modul ist eine Komponente mit einer gewissen Abgeschlossenheit: Es erbringt Dienste, nimmt aber kaum welche in Anspruch.
- Ein Modul ist somit ein Teilsystem mit einer klaren und schmalen Grenze zu seiner Umgebung.

Modularisierung:

- Eine Modularisierung eines Systems erfolgt zum Zwecke der Übersichtlichkeit, Arbeitsteilung, Wiederverwendung, ...



- Zusammenstellung der Vereinbarungen von als zusammengehörig betrachteten Java-Klassen und Java-Interfaces
- Ein Paket besitzt einen Namen, der (wie von Dateiverzeichnissen her bekannt) hierarchisch aufgebaut sein kann:
 - `uhren`
 - `com.apple.quicktime.v2`
 - `edu.cmu.cs.bovik.cheese`
- Der Paketname definiert einen Namensraum für die im Paket vorkommenden Klassendeklarationen.



- Bei der Klassendeklaration kann das Paket, in das die Klasse kommen soll, mit `package` angegeben werden
- Der Zugriff auf eine Klasse in einem Paket erfolgt über den Paketnamen + Klassennamen, oder der Paketname kann mittels `import` einem Aufrufer bekannt gemacht werden.

```
package uhren;  
interface Taktgeber {  
    long leseZeitInSekunden ();  
}
```

```
package uhren;  
public class Uhr implements Taktgeber {  
    ...  
}
```

```
uhren.Taktgeber t = new uhren.Uhr ();
```

```
import uhren;  
Taktgeber t = new Uhr ();
```

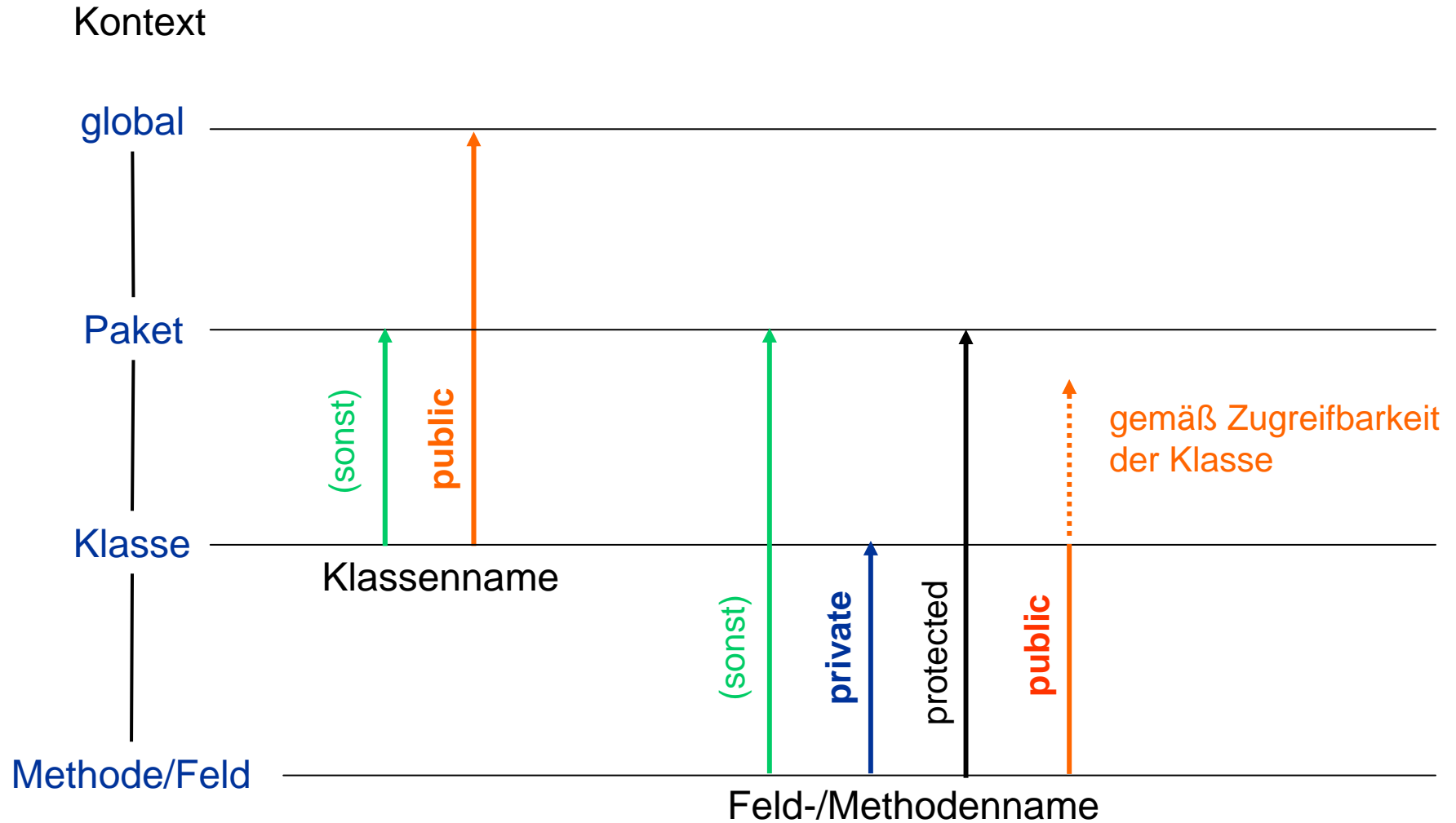


- Sammlung vordefinierter, häufig verwendeter Klassen, auf die bei der Programmierung zugegriffen werden kann.
- In Java ist dies eine Menge von Paketen.

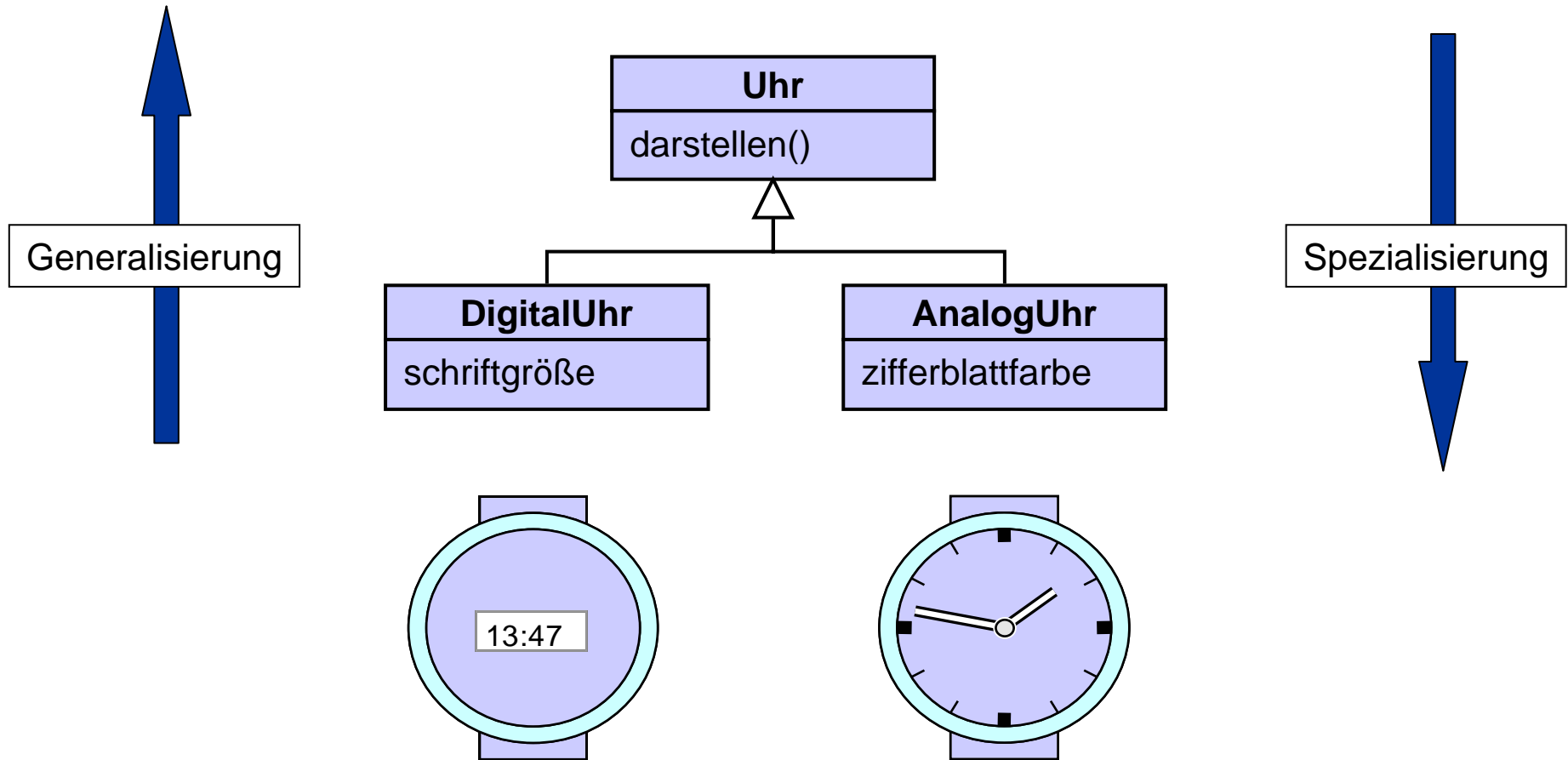
Java selbst liefert zum Beispiel die Pakete:

- `java.awt` Klassen für GUIs (Abstract Window Toolkit)
- `java.io` Ein-/Ausgabe in Dateien o.ä.
- `java.lang` Zentrale Klasse von Java (z.B. Object)
- `java.net` Verwaltung von Netzwerkverbindungen
- `java.sql` Datenbankansbindung
- `java.util` Nützliche Klassen (z.B. Kalender, Listen)
- `javax.swing` Graphische Oberfläche

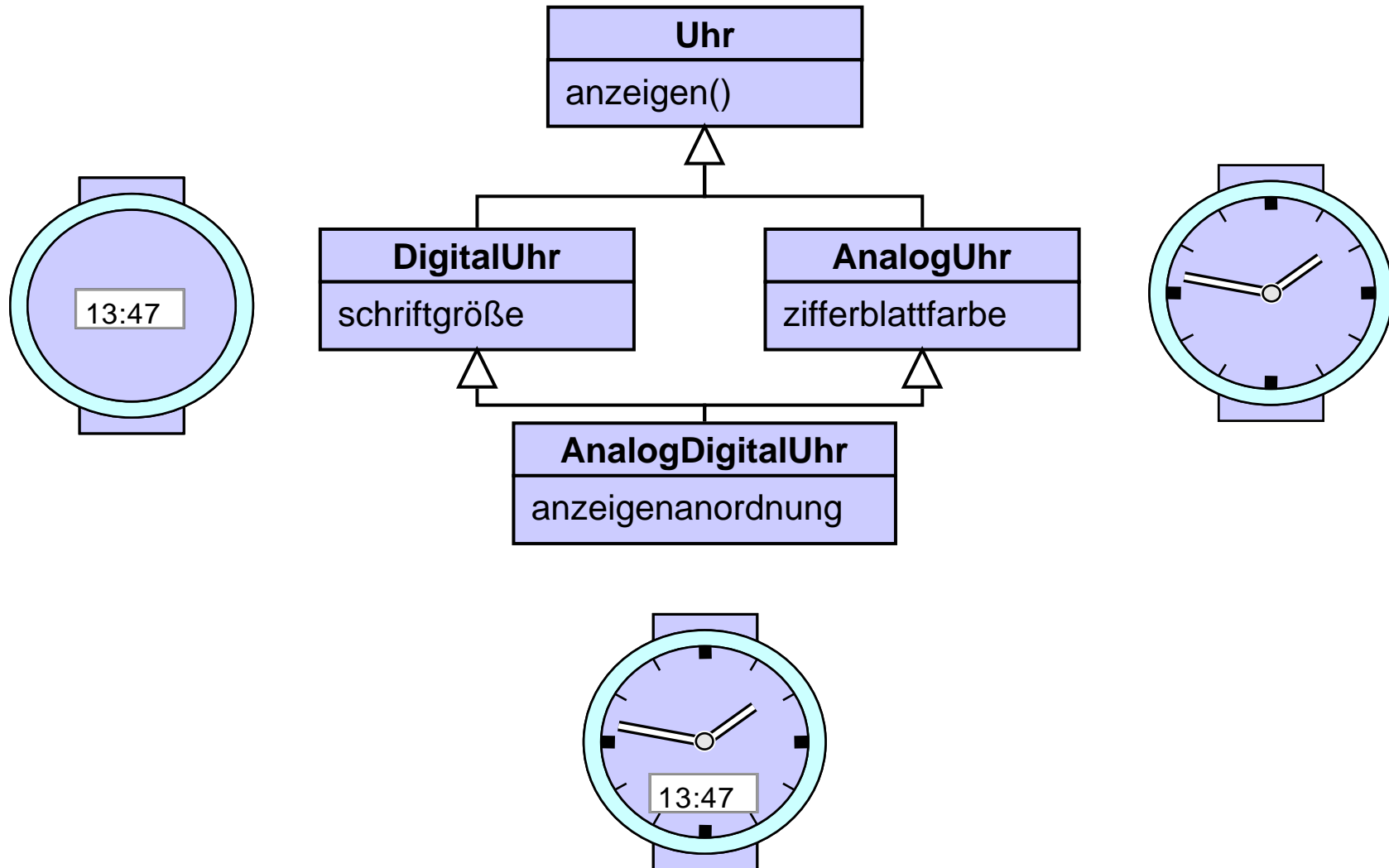




Einfachvererbung



Mehrfachvererbung



Oberklasse (super class)

- Eine Oberklasse einer Objektklasse K ist eine aus K unmittelbar oder mittelbar generalisierte Objektklasse.

Unterklasse (subclass)

- Eine Unterklasse einer Objektklasse K ist ein aus K unmittelbar oder mittelbar spezialisierte Objektklasse.

Vererbung (inheritance)

- Übernahme aller Eigenschaften der Oberklasse A durch die Unterklasse B:
 - Eine Kerneigenschaft von A wird von der Klasse selbst festgelegt.
 - Eine abgeleitete Eigenschaft von A wird von B übernommen (Verhaltensgleichheit).
 - Eine überschriebene Eigenschaft von A übernimmt die Signatur von B, stellt aber eine neue Implementierung bereit (Spezialisierung im engeren Sinn).



Vererbung zwischen Klassen

- Einfachvererbung wird zwischen Klassen unterstützt, wobei **extends** die Oberklasse angibt:

```
class AnalogUhr extends Uhr {  
    ...  
}
```

Die Umsetzung in Java erfolgt durch Ableitung der Eigenschaften:

- Die Unterklasse übernimmt automatisch die Attribute sowie die Methodensignaturen und deren Implementierungen von der Oberklasse.
- Bei Spezialisierung durch Überschreiben einer Eigenschaft wird die Methode in der Unterklasse erneut vereinbart und implementiert.



Vererbung zwischen Schnittstellen

- Zwischen zwei Schnittstellen wird sowohl Einfach- als auch Mehrfachvererbung unterstützt, wobei **extends** die Schnittstelle angibt, deren Signaturen übernommen werden:

```
interface TaktgeberMillisekunden extends Taktgeber {  
    ...  
}
```

Werden zwei gleich benannte Methoden über verschiedene Pfade vererbt, so wird

- bei identischen Signaturen eben diese Signatur übernommen und
- bei unterschiedlichen Signaturen beide übernommen und als Überladen behandelt.



Wurzel der Hierarchie

- Die Oberklasse aller Java-Klassen ist `java.lang.Object`.

Zugriffsbeschränkungen (Ergänzung)

- Will man für eine Klasse keine Unterklassen zulassen, so wird sie mit **final** vereinbart.
- Darf eine Methode in einer Unterklasse nicht überschrieben werden, wird sie mit **final** gekennzeichnet.
- Erweiterung von **protected**: Bedeutet Zugreifbarkeit innerhalb der Klasse und all ihrer Unterklassen.



Ersetzungsprinzip:

- Sei T eine Variable mit Referenzsemantik. Dann dürfen v Verweise auf Objekte der Klasse T oder einer beliebigen Unterklasse zugewiesen werden.
- Die Unterklasse verfügt über alle Eigenschaften der Oberklasse und kann daher deren Platz einnehmen.

```
Uhr c = new Uhr();  
c = new AnalogUhr();  
  
// Alle Methoden von Uhr können auf c aufgerufen werden
```

Das Ersetzungsprinzip garantiert Typsicherheit, d.h. dass eine auf T definierte Methode m auf jedes Objekt angewendet werden kann, auf das v aktuell verweist.



Überschreiben von Methoden (method overriding)

- Eine Unterklasse kann eine neue Implementierung für eine vererbte Methode bereit halten
- Dann kann immer noch auf die Implementierung der direkten Oberklasse über die Referenz `super` zugegriffen werden

```
class Uhr {  
    ...  
    public void anzeigen() {  
    }  
}
```

```
class AnalogUhr extends Uhr {  
    ...  
    // Selbe Signatur wie in Oberklasse  
    public void anzeigen() {  
        super.anzeigen(); // führt Code von Uhr.anzeigen() aus  
        ...  
    }  
}
```



```
Uhr c = null;  
...  
// c wird Objekt der Klasse Uhr oder AnalogUhr zugewiesen  
...  
c.anzeigen ();
```

Folge des Ersetzungsprinzips:

- Wegen der überschriebenen Methode `anzeigen()` ist zur Übersetzungszeit nicht bekannt, ob die Implementierung für `Uhr` oder für `AnalogUhr` zur Ausführung kommt.
- Der Aufrufer gibt nur an, dass er eine Methode aufrufen möchte. Welche Implementierung zum Einsatz kommt, hängt von der Klasse des aktuellen Objekts ab, das der Variablen `c` zugewiesen wurde.

Polymorphie: Beim Niederschreiben wird offen gelassen, welcher aus einer gegebenen Menge von Typen zur Laufzeit angetroffen wird.

- Zur Laufzeit wird also über die richtige Implementierung für einen Methodenaufruf entschieden.



- Die auszuführende Implementierung einer Methode wird erst während der Programmausführung festgelegt.
- Hierzu muss die Klasse des aktuellen Objektes bekannt sein.
- Nach einer Implementierung wird zur Laufzeit zuerst bei dieser Klasse und dann aufsteigend in den Oberklassen gesucht.
- Bei Mehrfachvererbung nicht eindeutig, daher in Java nicht zugelassen.

```
Uhr c = new Uhr ();  
c.setzeZeit (11, 42);    // Implementierung in Uhr  
c.anzeigen ();           // Implementierung in Uhr  
  
c = new AnalogUhr ();  
c.setzeZeit (11, 42);    // Implementierung in Uhr  
c.anzeigen ();           // Implementierung in AnalogUhr
```

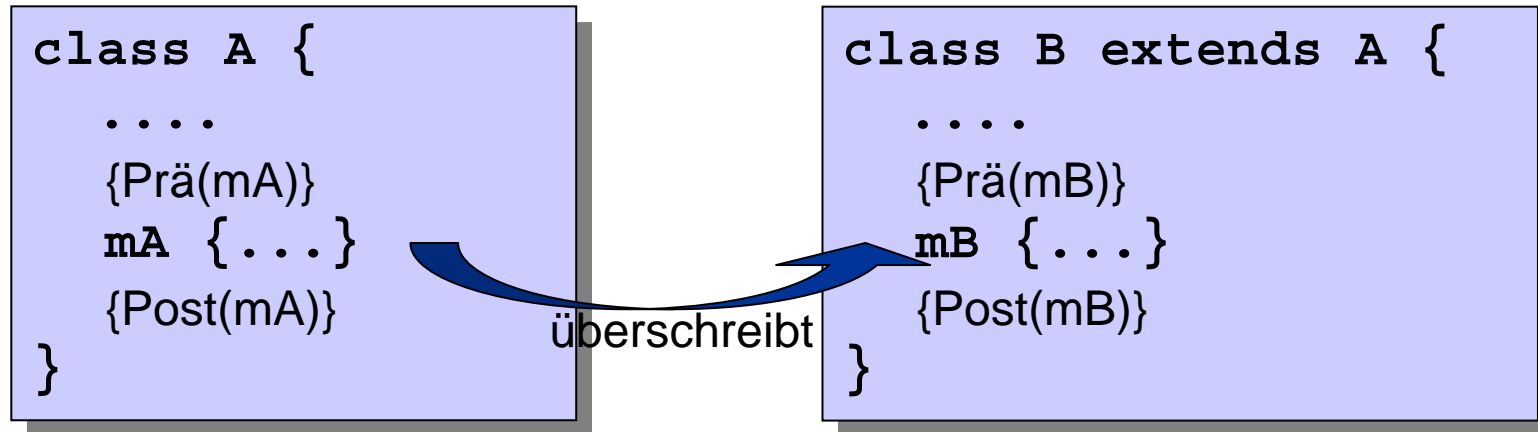


- Um nicht gegen das Ersetzungsprinzip zu verstoßen, muss man wissen, ob ein Referenzergebnis eines Ausdrucks einer Objektvariablen zugewiesen werden darf.
- Zu welcher Klasse ein konkretes Objekt gehört, lässt sich mit `instanceof` überprüfen:

```
public class UhrenKonfiguration {  
    public static Uhr konfigurieren (Uhr c) {  
        if (c instanceof AnalogUhr) {  
            AnalogUhr analog = (AnalogUhr) c;  
            analog.setzeZifferblattfarbe (java.awt.Color.blue);  
            return analog;  
        } else if (c instanceof DigitalUhr) {  
            DigitalUhr digital = (DigitalUhr) c;  
            digital.setzeSchriftgroesse (30);  
            return digital;  
        } else {  
            return c;  
        }  
    }  
}
```



Strenge Definition:

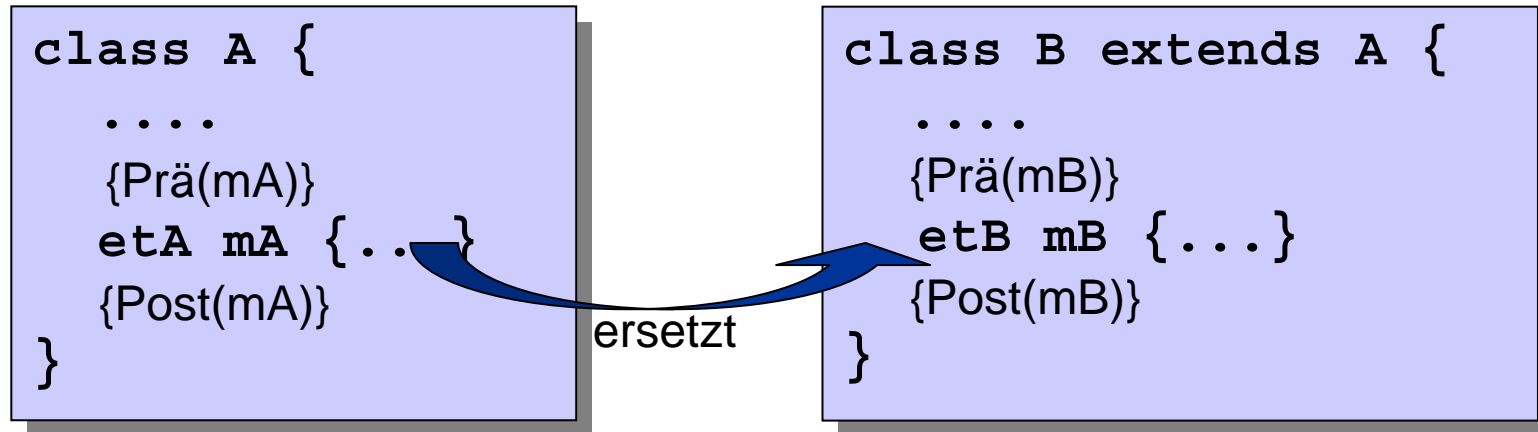


mB heißt verhaltensgleich zu mA falls gilt $\text{Prä}(mA) \Rightarrow \text{Prä}(mB)$ und $\text{Post}(mB) \Rightarrow \text{Post}(mA)$

- B ist verhaltensgleich zu A , wenn jede ererbte Methode von B verhaltensgleich zur entsprechenden Methode von A ist.
- Ist B verhaltensgleich zu A , so kann jederzeit ein Objekt der Klasse A durch ein Objekt der Klasse B ersetzt werden, ohne dass sich das Programmverhalten ändert.



Davon zu unterscheiden: (syntaktische) Konformität



- mB heißt konform zu mA , wenn
 - mB gleiche Parameteranzahl und Namen wie mA hat,
 - der Typ jedes expliziten Eingabeparameters von mB Oberklasse des entsprechenden Parametertyps von mA ist (Kontravarianz),
 - der Ergebnistyp von mB Unterklasse des Ergebnistyps von mA ist (Kovarianz).
- B ist konform zu A , wenn alle ererbten Methoden in B konform zu Methoden von A sind.
- Unter Konformität ist weiterhin Typsicherheit garantiert.

Abstrakte Klasse (abstract class)

- Oberklasse mit nur partieller Implementierung; die restliche Implementierung muss in einer Unterklasse nachgeholt werden
 - Mischung aus Klasse und Schnittstelle, daher sind keine Ausprägungen möglich.

Hat eine Klasse eine Schnittstelle als Obertyp und implementiert nicht alle Methoden der Schnittstelle, ist sie ebenfalls abstrakt.

```
abstract class Uhr {  
    ...  
    abstract void anzeigen ();    // Keine Implementierung  
}  
class AnalogUhr extends Uhr {  
    void anzeigen () {  
        ...  
    }  
}
```



- Generische Klassen parametrisieren bestimmte Argument- oder Ergebnistypen.
- Dies ist z.B. dann sinnvoll, wenn eine Klasse als Baumuster für eine Reihe von Klassen dienen soll, die ein selbes Verhalten auf unterschiedlichen Typen realisieren:
 - Zum Beispiel spielt bei Listen, Kellern, Warteschlangen etc. der Elementtyp keine Rolle

In Standard-Java wird in solch einem Fall `java.lang.Object` als Typ angegeben. Probleme hierbei:

- Typsicherheit kann dann nicht statisch überprüft werden.
- Zur Laufzeit ist Typumwandlung notwendig.

Es gibt Erweiterungen von Java (z.B. GenericJava), die generische Klassen unterstützen.



Standard-Java

```
public class Keller {  
    void push (Object element){  
        ...  
    }  
    Object pop () { ... }  
}
```

```
Keller k = new Keller ();  
k.push ( new Integer (5) );  
...  
Object o = k.pop();  
if (o instanceof Integer) {  
    Integer i = (Integer) o;  
}
```

GenericJava

```
public class Keller<A> {  
    void push (A element){  
        ...  
    }  
    A pop () { ... }  
}
```

```
Keller<Integer> k =  
    new Keller<Integer> ();  
k.push ( new Integer (5) );  
...  
Integer i = k.pop();
```



In Java sind Zeichenfolgen Objekte und als solche Ausprägungen der Klasse **String**.

Dementsprechend ist die Behandlung:

- Stringvariablen sind Referenzvariablen, denen über Konstruktoren Literale zugewiesen werden.
- Besonders herausgehoben (auch syntaktisch) ist die Konkatination "+"
- Alle anderen Operationen werden in der üblichen Weise in Form von Methoden realisiert. Die Klasse enthält Methoden für die Positionsbestimmung innerhalb einer Zeichenkette, Vergleichen von Zeichenfolgen und Konvertierungen.
- String-Objekte können nicht unabhängig verändert werden, da Kopie als Ergebnis einer Methode zurückgegeben wird.

Sollen die einzelnen Zeichen einer Zeichenfolge bearbeitet werden, ist die Klasse **StringBuffer** geeigneter.



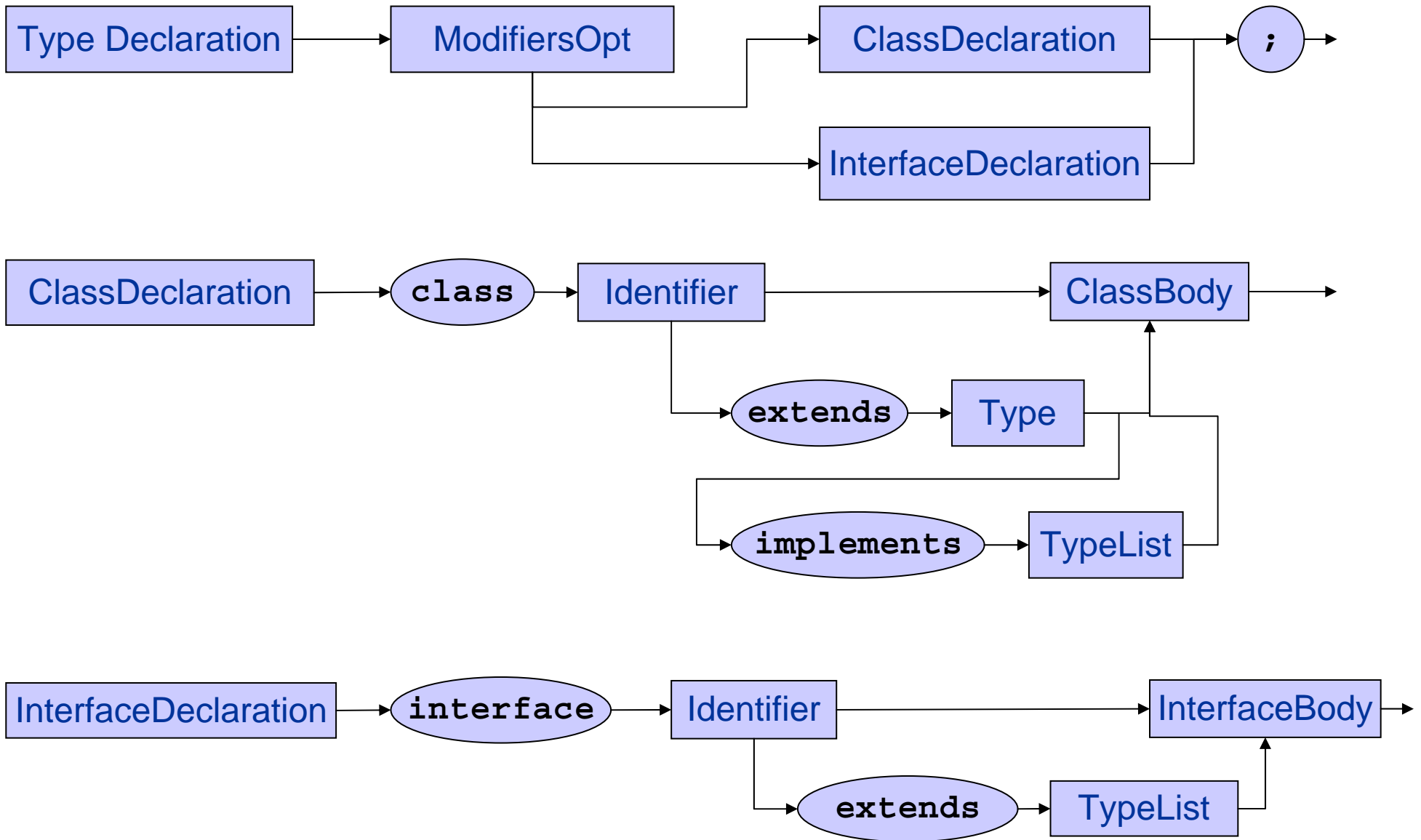
- Reihungen sind Objekte einer anonymen Klasse, die durch den Typ der Elemente bestimmt wird. Sie kommen mit einem konstanten Attribut `length`, welches die Anzahl der Elemente angibt.
- Reihungsvariablen sind daher Objektvariablen, und Reihungen werden durch `new` bereitgestellt, wobei die gewünschte Größe angegeben wird.

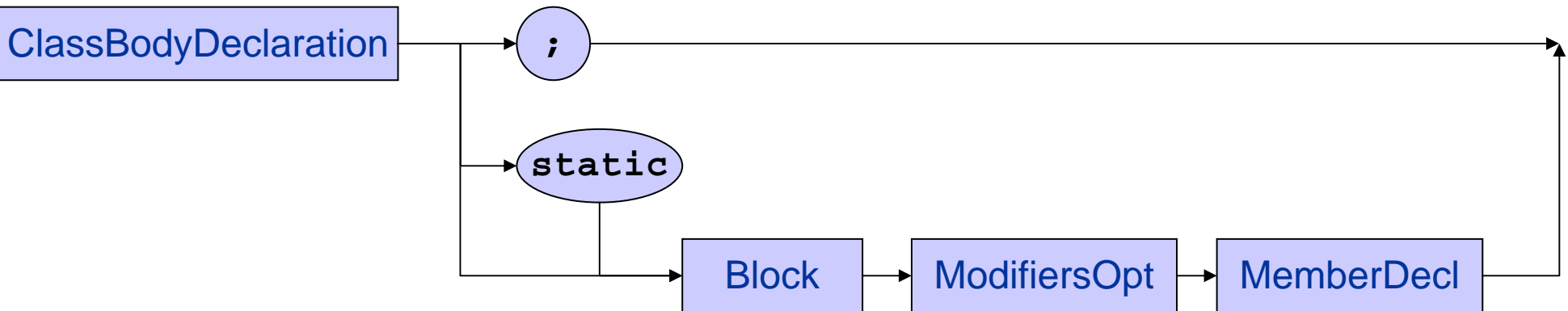
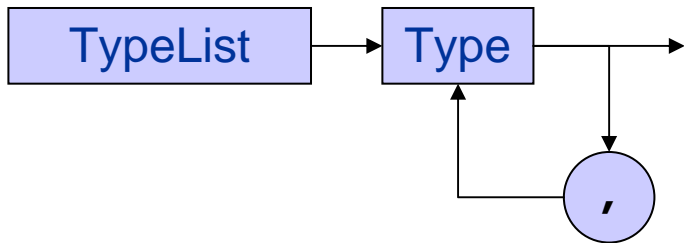
```
Uhr[] uhren_liste = new Uhr[5];           // 5 Uhren
uhren_liste[0] = new AnalogUhr ();
uhren_liste[1] = new DigitalUhr ();

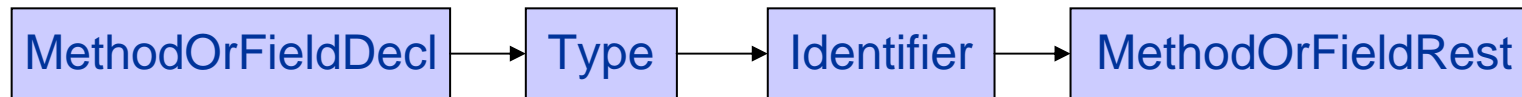
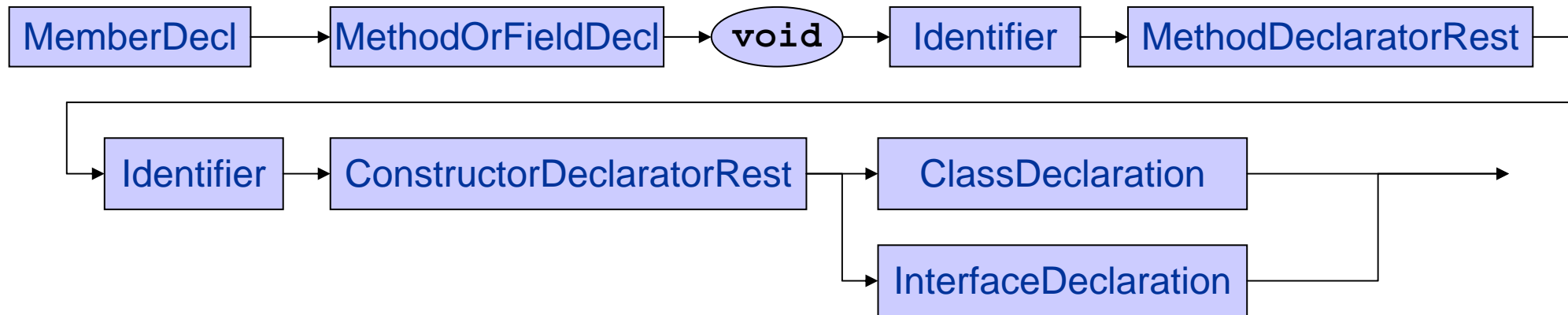
int l = uhren_liste.length;               // l ist 5

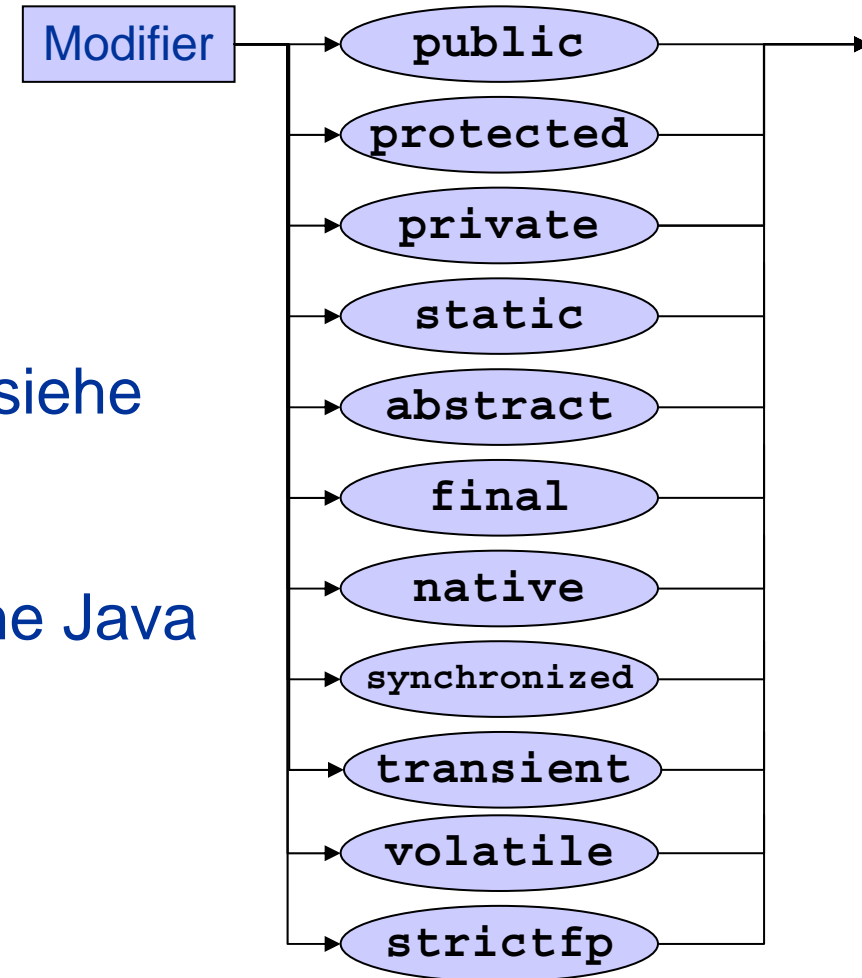
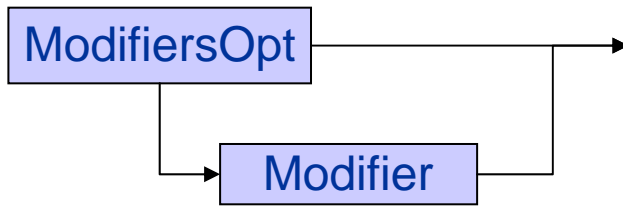
// Mehrdimensionale Reihung mit 2 x 3 Elementen
int[][] matrix;
matrix = new int[2][3];
```











Definition von Type, Block und Identifier siehe erste Vorlesung, Details zu **MethodDeclaratorRest** und **ConstructorDeclaratorRest** siehe Java Spezifikation

```
package uhren;

import java.util.Date;
import uhren.Taktgeber;
import uhren.TimeZone;

public abstract class Uhr implements Taktgeber {
    private static long    zaehler = 0;
    private long           seriennummer;
    private long           minuten;
    private long           stunden;
    private TimeZone       zeitzone = new TimeZone ("UTC");
    private final Date     herstellungsdatum = new Date ();

    public Uhr () {
        seriennummer = zaehler;
        zaehler++;
    }
    ...
}
```



```
...  
public Uhr ( long min ) {  
    this ();  
    stunden = min / 60;  
    minuten = min % 60;  
}  
  
public abstract void anzeigen ();  
  
public void setzeZeit (long std, long min) {  
    stunden = std; minuten = min;  
}  
public void setzeZeit (long std) {  
    setzeZeit (std, 0);  
}  
public long leseZeitInSekunden () {  
    return (stunden * 3600) + (minuten * 60);  
}  
...
```



```
...  
public void setzeZeitZone (TimeZone tz) {  
    if ( tz == null ) { return; }  
    zeitzone = tz;  
}  
public TimeZone leseZeitZone () {  
    return zeitzone;  
}  
}
```



```
package uhren;

import java.awt.Color;
import uhren.Uhr;

public class AnalogUhr extends Uhr {
    private Color zifferblattfarbe;

    public AnalogUhr (long min) {
        super (min);
    }

    public void anzeigen() { ... }

    public void setzeZifferblattfarbe (Color c) {
        zifferblattfarbe = (c == null) ? Color.blue : c;
    }
}
```

```
package uhren;

import uhren.Uhr;

public class DigitalUhr extends Uhr {
    private int schriftgroesse;

    public DigitalUhr (long min) {
        super (min);
    }

    public void anzeigen() { ... }

    public void setzeSchriftgroesse (int groesse) {
        schriftgroesse = (groesse <= 0) ? 10 : groesse;
    }
}
```

```
package uhren;  
  
interface Taktgeber {  
    long leseZeitInSekunden ();  
}
```

```
package uhren;  
  
public class TimeZone {  
    private String identifikator;  
  
    public TimeZone (String id) {  
        setzeIdentifikator (id);  
    }  
    public void setzeIdentifikator (String id) {  
        identifikator = id;  
    }  
}
```

```
package uhren;

import uhren.*;
import java.awt.Color;

public class UhrenKonfiguration {

    public static Uhr konfigurieren (Uhr c) {
        if (c instanceof AnalogUhr) {
            AnalogUhr analog = (AnalogUhr) c;
            analog.setzeZifferblattfarbe (Color.blue);
            return analog;
        } else if (c instanceof DigitalUhr) {
            DigitalUhr digital = (DigitalUhr) c;
            digital.setzeSchriftgroesse (30);
            return digital;
        } else {
            return c;
        }
    }
}
```

