Klassen und Konstruktoren in Java

```
class Circle {
    int r;
    Circle(int r) { this.r = r; }        // 1
    Circle(Circle c) { this.r = c.r; }   // 2
    Circle() { r = 1; }                  // 3
    ...
}


    ...
    Circle a = new Circle(2);  // Konstruktor 1
    Circle b = new Circle(a);  // Konstruktor 2
    Circle c = new Circle();   // Konstruktor 3
```

Konstruktoren in Java

- Default-Konstruktor, falls kein expliziter Konstruktor:

```
class Classname {
    // public Classname() {};
    ...
}
```

- `super(...)` führt Konstruktor der Oberklasse aus
- `this(...)` führt anderen Konstruktor derselben Klasse aus
- Konstruktor beginnt nicht mit `super(...)` oder `this(...)`: `super()` implizit am Anfang des Konstruktors eingefügt

Verdecken versus Überschreiben

Variablen gleichen Namens in Ober- und Unterklasse:

- Variable in Unterklasse verdeckt Variable in Oberklasse
- verdeckte Variable zugreifbar: `super.var`
  `((Oberklasse)this).var`

Methoden gleichen Namens in Ober- und Unterklasse:

- Unterklassenmethode überschreibt Oberklassenmethode
- überschriebene Methode zugreifbar: `super.method(...)`
- kein Zugriff über `((Oberklasse)this).method(...)`

Final

Überschreiben einer Methode verhinderbar:

```
final method ( ... ) { ... }
```

final Methoden sollen meist vermieden werden

Ableitung einer Unterklasse verhinderbar:

```
final class FinalClass { ... }
final class FinClass extends NonFinalClass { ... }
```

in einigen oo Programmierstilen sind final Klassen häufig:

- Instanzen werden nur von final Klassen erzeugt
- dadurch Ersetzbarkeit einfacher zuzusichern

Pakete

- eine public Klasse pro Datei
- Paket umfaßt alle Dateien bzw. Klassen im selben Ordner
- explizite Paketdeklaration: `package paketName;`
- Aufruf von `foo()` in der Datei `myclasses/test/AClass.java`:

  ```
  myclasses.test.AClass.foo()
  ```

- Kürzer durch Import-Deklaration (am Dateianfang)

```
import myclasses.test;          ... test.AClass.foo() ...
import myclasses.test.AClass;       ... AClass.foo() ...
import myclasses.test.*;            ... AClass.foo() ...
```

Sichtbarkeit

	`public`	`protected`	default	`private`
lokal sichtbar	ja	ja	ja	nein
global sichtbar	ja	nein	nein	nein
lokal vererbbar	ja	ja	ja	nein
global vererbbar	ja	ja	nein	nein

lokal = im selben Paket, auch außerhalb der Klasse

global = auch außerhalb des Pakets

Anwendung der Sichtbarkeit

Public: Bei der Verwendung der Klasse und deren Instanzen benötigte Methoden, Konstanten und (selten) Variablen

Private: Methoden und Variablen, die nur innerhalb der Klasse verwendet werden sollen — vor allem, wenn Bedeutung außerhalb der Klasse nicht klar

Protected: Methoden und Variablen für Verwendung nicht benötigt, aber hilfreich bei Erweiterungen der Klasse

Default: Eng zusammenarbeitende Klassen im selben Paket sollen auf Methoden und Variablen zugreifen können, die sonst privat wären

Interfaces

```
interface X {
    static final double PI = 3.14159;
    double fooX();
}

interface Y {
    double fooY();
}

interface Z extends X, Y {
    double fooZ();
}
```

Unterschiede zu abstrakten Klassen

- Schlüsselwort `interface` statt `abstract class`
- alles implizit public — daher `public` nicht nötig
- alle Methoden abstrakt — daher `abstract` nicht nötig
- keine Variablen, nur Konstanten (`static final ...`)
- `static` und `final` in Methodendeklarationen verboten
- nach `extends` können mehrere Namen von Interfaces stehen — Mehrfachvererbung

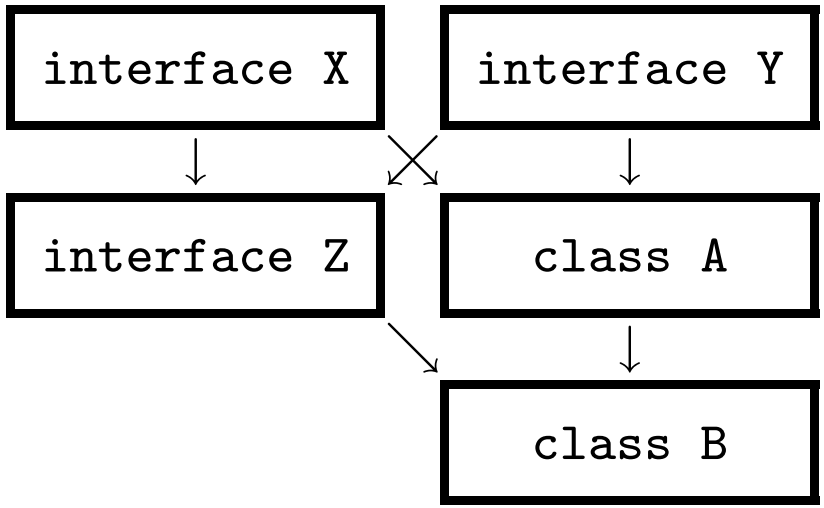
Klassen implementieren Interfaces

```
class A implements X, Y {
    double factor = 2.0;
    public double foo() { return PI; }
    public double fooX() { return factor * PI; }
    public double fooY() { return factor * fooX(); }
}

class B extends A implements Z {
    public double fooY() { return 3.3 * foo(); }
    public double fooZ() { return factor / fooX(); }
}
```

Interfaces und Untertyprelationen

Interfaces für Untertyprelationen gut geeignet



Abstrakte Klassen gegenüber Interfaces nur dann zu bevorzugen, wenn Code vererbt werden soll

Zusicherungen auf abstrakten Methoden (Interfaces) wichtig!