

# Se greit Eidn! recap

Hat jemand schon die Pingu-Gang informiert?

Dieses *Recap* liefert *keine Garantie auf Vollständigkeit*. Ich konzentriere mich bewusst auf einige wenige aber wichtige *Kernthemen*, die in der Hinsicht — zumindest auf den Folien — auch leicht simplifiziert dargestellt sind.

*Fröhliche Weihnachten*  
Flo

# 3

1. Algorithmenkonstruktion
2. Programmkonstrukte (Namen & Programmfluss)
3. Arrays und Iterationen
4. Unterprogramme
5. Objektorientierte Programmierung
6. Dynamische Datenstrukturen
7. Weiterführende Konzepte OOP
8. Rekursion
9. Laufzeitkomplexität
10. Suchen und Sortieren

# 4

## What can we say about an Algorithm?

### Totale Korrektheit

#### 1. Termination

Der Algorithmus endet nach endlich vielen Schritten für jede Eingabe.

#### 2. Partielle Korrektheit

Wenn der Algorithmus terminiert, ist er korrekt.



# 5

## But what is an Algorithm?

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems.

- › Endlich viele wohldefinierte Einzelschritte.
- › Kann grafisch, textuell, ... erfolgen.
- › **Fordert gemeinsames Sprachverständnis.**



# 6

## Discussing an Algorithm

Problem ————— ? —————→ Lösung

- › Problemspezifikation                      Was meinen Sie mit „schnell“?
- › Problemabstraktion                      Was ist gegeben, was ist gesucht?
- › Algorithmenentwurf                      Wie kommen wir von gegeben zu gesucht?
- › Korrektheitsnachweis                      Löst unser Ansatz das Problem?
- › Aufwandsanalyse                      Wie verhält er sich?

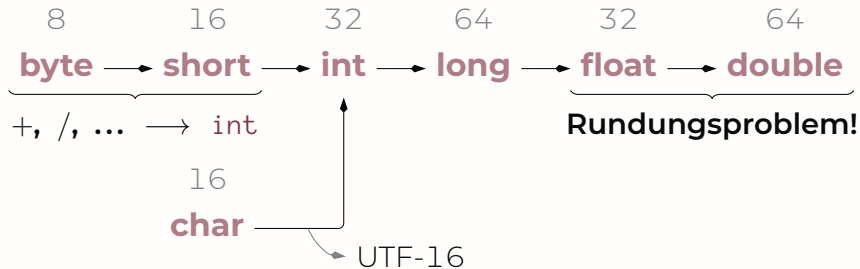


- › Totale Korrektheit
  - › *Terminiertheit*: Endliche Schritte für jede Eingabe.
  - › *Partielle Korrektheit*: Wenn terminiert, dann korrekt.
- › Abstraktion ist wichtig.

# 8

## Primitive Datentypen

### boolean





# 9

## Präzedenzregeln

> Eine Auswahl:

$a++, a-- \rightarrow !a, -a, ++a, --a \rightarrow a * b, a / b, a \% b$   
 $\rightarrow a + b, a - b \quad \rightarrow a == b, a < b, \dots$   
 $\rightarrow a \wedge b \quad \rightarrow a \&\& b$   
 $\rightarrow a || b$

# 10

## Variablendefinition

**Deklaration:** Es gibt etwas mit diesem Namen (und Typ):

```
int x; char w;
```

**Zuweisung:** Wertzuweisung, bzw. Ersetzung des Wertes:

```
x = 12; x = 9; w = 'k';
```

**Initialisierung:** *Erste* Wertzuweisung zu einer Variable:

```
int x = 12; char w; w = 'x';
```

**final** erzwingt, dass eine Variable nur Initialisiert und dann nicht erneut zugewiesen werden darf.



# 11

## Standardwerte

- › Standardwerte für:
  - › Klassen-Variablen (**static**)
  - › Instanz-Variablen
  - › Array-Komponenten (wie bei **new int[]**)
- › Der Standardwert ist „Null“:
  - › **byte, short, int, long** → 0
  - › **float, double** → 0.0
  - › **char** → `'\u0000'` („Unicodesymbol mit Wert 0“)
  - › **boolean** → **false**
  - › Komplexe Datentypen → **null**

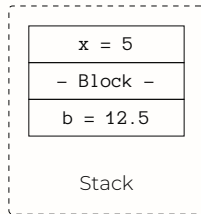


## 12

## Blöcke

- › Methodenaufrufe erzeugen „Stack-Frames“.
- › Code-Blöcke arbeiten vergleichbar.
- › Beim Verlassen wird alles davon vom Stack geschmissen.

```
int x = 5;  
{  
    double b = 12.5;  
    x = x + (int) b;  
}  
int y = x;
```



# 13

## Überschatten



- › Instanz-/Klassenvariable mit gleichem Bezeichner wie lokale Variable.
- › Der lokale Bezeichner überschattet den „globalen“.
- › So kann auch der Typ verändert werden.

```
class LetThereBeDarkness {  
    static int happiness = 7;  
    public static void main(String[] args) {  
        System.out.println(happiness);  
        String happiness = "Maunz";  
        System.out.println(happiness);  
    }  
}
```

Ausgabe:

```
7  
Maunz
```



# 14

## Fallunterscheidungen

- › Wenn „X“ dann „Y“ sonst „Z“

`if(X) Y else Z`

- › Der Sonst-Fall (`else Z`) ist optional.
- › Wir können `Y` und `Z` durch einen Code-Block aus mehreren Zeilen bestehen lassen.
- › Wir können die Ausdrücke verschachteln.

- › Es gibt eine ternäre Kurzform:

$$X \text{ ? } Y \text{ : } Z$$

- › Der Sonst-Fall ( : Z ) ist **nicht** optional.
- › Y und Z müssen zu einem Wert evaluieren.
- › Wir können die Ausdrücke verschachteln.
- › Anders als **if-else** ist der Operator kein Statement! Wir müssen ihn also eingebettet verwenden.

- › *Implizit:* **byte** → **short** → **int** → **long** → **float** → **double**  
Zahlen von klein zu groß, sowie: **char** → **int**.
- › *Präzedenzregeln:*  
Post vor Prä, sonst wie Arithmetik & Logik.
- › *Default-Werte:*  
Zahlen und Zeichen 0, Boolean **false**, Rest **null**.
- › *Überschatten:* Lokal über Global.



# 17

## Arrays

- › Sind komplexe Datentypen (→ Heap).
- › Jeder Datentyp kann ein (eindimensionales) Array werden:

```
int[] = new int[12];
```

- › Mehrdimensionale Arrays, sind Arrays von Arrays von...
- › Zugriff: `<array>[<index>]`
  - › Liefert `<index>`-Element in `<array>`.
  - › Wenn Array von Array, ist `<array>[<index>]` wieder ein Array.

# 18

## Schleifen

- > Drei Schleifenarten: **for**, **while**, **do-while**
- > Sie sind alle gleichmächtig

```
for(int i = 3; i < 8; i++) {  
    // Miau  
}
```

*Anzahl bekannt*

```
int i = 3;  
while(i < 8) {  
    // Miau  
    i += 1;  
}
```

*Kein Maximum*

```
int i = 3;  
do {  
    // Miau  
    i += 1;  
} while(i < 8);
```

*Mindestens 1*

- > Bei **for** sind alle drei Blöcke optional.
- > Bei **do-while** aufpassen:
  - > Semikolon!
  - > Wird es wirklich mindestens ein mal durchlaufen?

- › Arrays sind komplexe Datentypen.
- › Mehrdimensionale Arrays sind eindimensionale Arrays von eindimensionalen Arrays von...
- › Die drei Schleifenarten sind gleichmächtig.
  - › Maximum bekannt: **for**
  - › Mindestens ein mal: **do-while**
  - › Sonst: **while**

- › Ein Unterprogramm lagert einen Teil des Programms aus.
- › In Java: **Methoden**

```
⟨Modifikatoren⟩ ⟨Rückgabetyp⟩ ⟨Name⟩(⟨Parameterliste⟩) {  
    ⟨Inhalt⟩  
}
```

- › Modifikatoren beeinflussen beispielsweise die Sichtbarkeit.

## 21

## Ein Beispiel

```
⟨Modifikatoren⟩ ⟨Rückgabetyp⟩ ⟨Name⟩(⟨Parameterliste⟩) {  
    ⟨Inhalt⟩  
}
```

	Modifikatoren	R-Typ	Name	Parameterliste
	<code>public static</code>	<code>String</code>	<code>giveMeTheMiau</code>	<code>(int times, double cuteness)</code>
Inhalt	{	<code>String out = "";</code>	<code>{</code>	
		<code>for(int i = 0; i &lt; times * cuteness; i++) {</code>		
		<code>    out += "Miau_";</code>		
		<code>}</code>		
	<code>return out;</code>			
	<code>}</code>			

**Signatur:** Name & Parametertypen  
`giveMeTheMiau(int, double)`

- › `void` kennzeichnet, dass die Methode nichts zurückgibt.
- › Sonst: jeder Ausführungspfad muss einen Wert vom angegebenen Typ liefern.

```
public static int iHaveCatPics(String where) {  
    if(where.equals("MIAFF"))  
        return 42;  
}
```

- › **Verboten**, da nicht jeder Ausführungspfad `int` liefert.

- › Methoden mit **static** gehören der Klasse.
  - › Wir benötigen kein Objekt.
  - › In ihnen „gibt“ es *kein* **this**.
- › Java sucht beim Aufruf eine Methode mit entsprechender Signatur.
  - › Implizite Typkonvertierung!
- › **Überladung**: Gleicher Name, unterschiedliche Signatur.



- › Aufruf: „Stack-Einträge“ der Parameter werden *kopiert*.
  - › Primitive Datentypen: *call-by-value*
  - › Komplexe Datentypen: *call-by-reference*
- › Genau genommen hat Java nur „call-by-value“
- › **Seiteneffekt:** Wenn nach dem Aufruf mehr als der Rückgabewert verbleibt.
  - › Instanzvariable verändert sich.
  - › Durch Referenz wird ein Array modifiziert.
  - › Eine Nachricht fliegt durchs Internet.
  - › ...



## 25

## Varargs

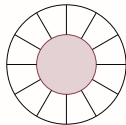
- › Der letzte Parameter darf in Java ein „Vararg“ sein.
- › Drei Punkte: erlaubt beliebig viele Argumente des Typs.
- › Kann in der Methode als Array verwendet werden.

```
static double[] funFunFun(int factor, double... arguments) {  
    for(int i = 0; i < arguments.length; i++) {  
        arguments[i] *= factor;  
    }  
    return arguments;  
}
```

- › *Überladung*: Gleicher Name, andere Signatur.
  - › *Signatur*: Name & Parametertypliste
  - › Müssen zudem in selber Klasse sein (später: Vererbung)
- › Beim Aufruf macht Java call-by-value:
  - › Alle Parameter werden kopiert (Stack).

- › Objekte abstrahieren Daten und Verhalten.
- › Ermöglicht Wiederverwendung von Komponenten.
- › Klassen definieren eine Blaupause:
  - › Welche Eigenschaften haben die Daten?
  - › Was kann mit den Daten gemacht werden?
- › Die Attribute definieren den Zustand
- › Die Methoden definieren das Verhalten

**Attribute**  
**Methoden**





- › Klasse ist eine Blaupause
- › Objekte weisen Attribute konkrete Werte zu

```
class Penguin {  
    double cute;  
    int age;  
    String name;  
  
    void walk(int snow) { ... }  
    void peep() { ... }  
    ...  
}
```

Penguin udbert

cute	5.7
age	12
name	„Udi“

Penguin josie

cute	6.1
age	10
name	„Josie“

Penguin peter

cute	6
age	10
name	„Petaa“

Penguin saphira

cute	8
age	11
name	„Saph“

- › Erlaubt für Werte beim Erzeugen („initialer Zustand“).
- › Ein Konstruktor ist *keine* Methode.
- › Unterschiede:
  - › Muss heißen wie die Klasse
  - › Hat keinen Rückgabotyp
  - › Kann nichts zurückgeben
  - › Benötigt **new** für Aufruf
  - › ...
- › Gemeinsamkeiten:
  - › Hat Parameter
  - › Kann überladen werden
  - › Kann Methoden aufrufen
  - › Kann Objekte erzeugen
  - › ...
- › Java erzeugt *genau dann* einen leeren Konstruktor, wenn kein expliziter Konstruktor existiert.

```
public class Supa {  
    int x;  
}
```

```
public class Mega {  
    String name;  
    public Mega(String name) {  
        this.name = name;  
    }  
}
```

> `Supa s = new Supa();`

Erlaubt, durch den leeren Standardkonstruktor. `s.x` hat den Standardwert `0`.

> `Mega m = new Mega();`

Nicht erlaubt. Existiert ein expliziter Konstruktor, erstellt Java keinen leeren mehr!

> `Mega x = new Mega("Huhu");`


Erlaubt. `x.name` ist "Huhu".

# 31

## Einen Konstruktor überladen

- > Konstruktoren können einander mit **this** aufrufen. Dies muss dann das *erste* Statement sein. Keine Rekursion.

```
public class Car {  
    int topSpeed;  
    final boolean pengu;  
    public Car(int top, boolean p) {  
        this.topSpeed = top;  
        this.pengu = p;  
    }  
    public Car(int top) {  
        this(top, true);  
        System.out.println("Piep");  
    }  
    public Car() { this(20); }  
}
```

A curved arrow points from the `this(20);` line in the `Car()` constructor to the `this(top, true);` line in the `Car(int top)` constructor, indicating a recursive call.

- > `new Car()`: topSpeed ist 20, pengu ist true. Mit Piep!
- > `new Car(14)`: topSpeed ist 14, pengu ist true. Mit Piep!
- > `new Car(2, false)`: topSpeed ist 2, pengu ist false. Kein Piep!

- › Statische Variablen & Methoden sind an die Klasse gebunden.
  - › Kein Objekt notwendig um Sie aufzurufen/abzugreifen.
  - › Keine Instanz- sondern Klassenvariablen.
- › Die Frage lautet: Ist diese Eigenschaft allen Objekten gemein?



## Sichtbarkeit & Gültigkeit

- › Klassen, Methoden & Attribute haben **Sichtbarkeiten**.
  - › Sie können nur verwendet werden, wo sie sichtbar sind.
  - › Diese Sichtbarkeit wird durch **public**, **private**, ... kontrolliert.
- › Variablen, ... haben einen **Gültigkeitsbereich** (Scope).
  - › So gibt es den Gültigkeitsbereich der Klasse und
  - › Den lokalen Gültigkeitsbereich in Methoden, Blöcken, ...
- › Eine Variable kann z.B. sichtbar, aber gerade durch Überschattung nicht gültig sein.

- › Java nutzt **Packages** als Namensräume
  - › Ordner der dann alle zugehörigen Java-Dateien enthält.
  - › Können über **import** <package>; sichtbar gemacht werden.
  - › Dabei wird `java.lang` immer importiert.
- › Sichtbarkeiten interagieren auch mit Vererbung.

- Java erlaubt vier verschiedene Sichtbarkeitsmodifikatoren:
  - *private*: Nur innerhalb der Klasse.
  - *protected*: Im gesamten Paket sowie allen Unterklassen.
  - *public*: Überall sichtbar wo Paket sichtbar.
  - „*default*“: Überall im Paket.
- *Hinweis*: Objekte einer Klasse können auf private Elemente anderer Objekte der gleichen Klasse zugreifen.

# 36

## Ein Scope Beispiel

- › Es gibt vier Sichtbarkeitsbereiche für Variablen.

```
public class Penguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse
    protected int watschelIndex; // 3. geschützt
    public void watschel() { /* ... */ } // 4. (erneut: global)
    void piepsen() { /* ... */ } // 5. Standard: Paket
}
```

- › Die Klasse **Tiger** sei im selben, **Auto** sei in einem anderen Paket, **Felspingu** erbe von **Pinguin** aber sei in einem anderem Paket:

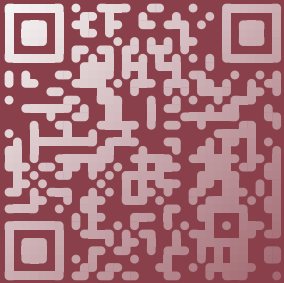
	1	2	3	4	5
Pinguin	✓	✓	✓	✓	✓
Tiger	✓	□	✓	✓	✓

	1	2	3	4	5
Auto	✓	□	□	✓	□
Felspingu	✓	□	✓	✓	□

- › Wenn irgendetwas schief geht, wirft Java eine Ausnahme.
- › Wir unterscheiden drei Arten:
  - › **Error**: Katastrophe, zum Beispiel kein Speicher. Kaum behandelbar.
  - › **RuntimeException**: Vorhersehbar, aber nun schwer behandelbar (wie ein negativer Array-Index, ...).
  - › **Exception**: Schwer vorhersehbare Ausnahme (wie Datei nicht gefunden, ...).
- › Normale „Exceptions“ *müssen* in Java:
  - › Behandelt werden (**try-catch**), oder
  - › weitergeleitet werden (**throws**).

- › Eine Ausnahme unterbricht den Programmfluss.
- › Wird sie nicht direkt behandelt, eskaliert die Ausnahme den „Call-Stack“ hinauf.
- › Was über die `main`-Methode eskaliert wird von der JVM abgefangen und beendet das Programm.
- › Ausnahmen sollten in Java aber nicht zur Werterückgabe verwendet werden.

- › Eine Klasse definiert die Blaupause für Objekte.
  - › Attribute definieren den Zustand.
  - › Methoden definieren den Verhalten.
- › Der Konstruktor baut den initialen Zustand
  - › **Instanziierung**: Erzeugen eines neuen Objektes.
  - › Wenn keiner: erzeugt Java den leeren Standardkonstruktor.
  - › **this** erlaubt Aufruf von Überladungen.
- › Klassen, Methoden, ...: **Sichtbarkeit** (**public**, ...)
- › **Gültigkeitsbereich**: Wo die Variablen „deklariert sind“.



Sources & current version  
on GitHub

**Florian Sihler**

22.12.2021

