

# Programmierungsmethodik 1

## Programmiertechnik

**Konstruktor, Sichtbarkeit,  
unveränderliche Klassen**

# Übung: Auto

- Schreiben Sie ein Klasse Auto. Ein Auto hat folgende Eigenschaften:
  - Name (Typ oder Fabrikat)
  - Tankkapazität (z.B. 32 Liter)
  - Tankinhalt (z.B. 24.5 Liter)
  - Verbrauch auf 100km (z.B. 8 Liter)
  - bislang gefahrene Strecke (in km)
- Zur Steuerung des Autos gibt es drei Methoden:
  - Fahren
  - Volltanken
  - Informationen ausgeben
- dies könnte hilfreich sein:
  - $\text{Verbrauchter Sprit} = \text{Spritverbrauch} * \text{Strecke} / 100$

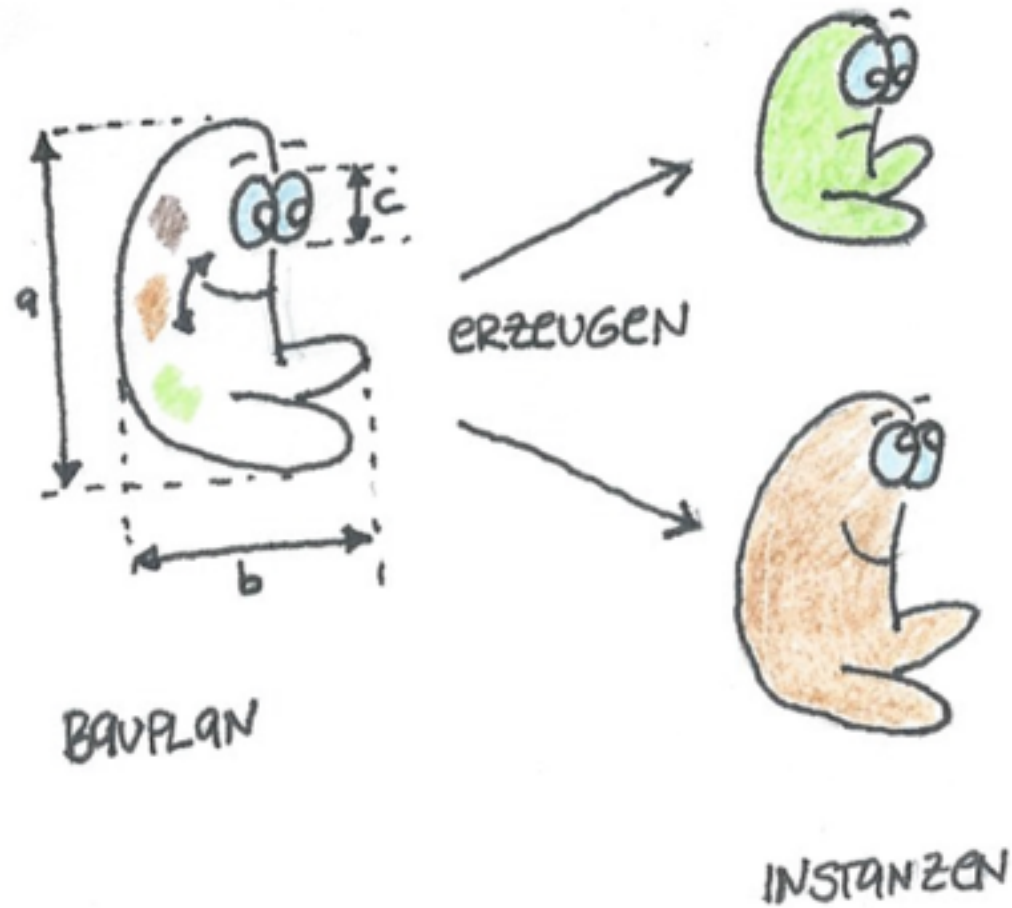
# Wiederholung

- Methoden
- Argumente und Parameter
- Überladen
- ErgebnISRückgabe
- UML

# Ausblick



# Worum gehts?



# Agenda

- Konstruktoren
- Sichtbarkeit
- UML
- Unveränderliche Klassen



# Konstruktor

# Konstruktoren

- Aufruf von new
  - Erzeugen der Objektes
  - Aufruf des Konstruktors
- Verwendung: Initialisierung des Zustandes (Objektvariablen)



# Konstruktor

- Definition eines Konstruktors wie normale Methode
- aber:
  - derselbe Name wie die Klasse (Großbuchstabe am Anfang!)
  - kein Vorsatz von void (oder anderem Ergebnistyp)
  - keine ErgebnISRückgabe (return unzulässig)
- abgesehen davon
  - Kopf, Parameterliste, Rumpf wie andere Methoden
- mehrere Konstruktoren für eine Klasse definierbar
  - Überladen von Methoden möglich wie gehabt

# Default-Konstruktor

- Konstruktor mit leerer Parameterliste
  - "Default-Konstruktor"
  - wird automatisch generiert, wenn kein eigener Konstruktor definiert ist

# Konstruktor

- eigener Default-Konstruktor von Bruch:

```
Bruch() {  
    zaehler = 0;  
    nenner = 1;  
}
```

- new ruft automatisch den Konstruktor auf:

```
Bruch bruch = new Bruch(); // Aufruf von Bruch()  
bruch.print(); // gibt 0/1 aus
```

# Konstruktor

- Initialisierung von Objektvariablen: Defaultwerte
- lokale Variablen starten nicht initialisiert (ohne Wert)
- Gegensatz: Objektvariablen werden automatisch mit einem Defaultwert initialisiert
  - abhängig vom Typ
- Beispiel: Ohne Konstruktor starten Rational-Objekte mit 0/0
  - unbrauchbar wegen 0 im Nenner

# Defaultwerte

Typ	Defaultwert
byte, short, int, long	0
float, double	0.0
boolean	false
char	\u0000
Referenztypen	null

# Konstruktor

- Initialisierung von Objektvariablen: Explizit
- auch Objektvariablen können bei der Deklaration explizit initialisiert werden

```
class Bruch {  
    int numerator = 0;  
    int denominator = 1;  
    ...  
}
```

- in einfachen Fällen Ersatz für Konstruktor
- explizite Initialisierung überschreibt Defaultwerte
- explizite Initialisierung zeitlich vor Konstruktoraufrufen
  - in Konstruktoren sind Objektvariablen bereits initialisiert

# Konstruktor

- Initialisierung von Objektvariablen
- Konstruktor mit nicht-leerer Parameterliste ist erlaubt

```
Bruch(int zaehler, int nenner) {  
    this.zaehler = zaehler;  
    this.nenner = nenner;  
}
```

- im new-Aufruf müssen passende Argumente angegeben werden:

```
Bruch bruch = new Bruch(2, 3);  
bruch.print();
```

# Konstruktor

- Automatische Definition
- Klasse ohne explizit definierten eigenen Konstruktor
  - Compiler erzeugt automatisch Default-Konstruktor
- Rumpf eines automatisch erzeugten Default-Konstruktors ist leer
- Beispiel
  - Klasse Rational hat nur eigenen Konstruktor Bruch(int, int)
    - keinen Default-Konstruktor
  - Aufruf des (nicht definierten) Default-Konstruktors scheitert:  

```
new Bruch(2, 3);  
new Bruch(); // Fehler
```
- Fazit
  - jede Klasse hat immer mindestens einen Konstruktor



# Konstruktor

- Kopier-Konstruktor
- erzeugt eine Kopie eines bereits existierenden Objekts
  - Kopie = Objektvariablen haben identische Werte
- Vorlage (= Original-Objekt) wird als Parameter übergeben

```
Bruch(Bruch andererBruch) {  
    zaehler = andererBruch.zaehler;  
    nenner = andererBruch.nenner;  
}
```

- Aufruf mit existierendem Objekt als Parameter

```
Bruch original = new Bruch(2, 7);  
Bruch kopie = new Bruch(original);  
kopie.print(); // gibt 2/7 aus
```

# Konstruktor

- Konstrukturen können manchmal aufwändig sein
  - z.B. Tests und Vorverarbeitung von Parametern, Protokollausgaben, ...
- falls mehrere überladene Konstrukturen definiert werden:
  - Idee: Kopien des gleichen Codes in jedem Konstruktor
- besser: Code nur in einem Konstruktor, von allen anderen mitbenutzen
  - Konstruktor-Verkettung (engl. constructor chaining)
  - Aufruf eines anderen Konstruktors der gleichen Klasse
- Syntax
  - this als Repräsentant des eigenen Objektes
- Einschränkungen verketteter Konstruktoraufrufe:
  - this(...) muss erste Anweisung im Konstruktorrumpf sein
  - nur ein Aufruf von this(...) ist erlaubt

# Konstruktor

- Beispiel: Verkettete Konstruktoren

```
public Bruch(int zaehler, int nenner) {  
    this.zaehler = zaehler;  
    this.nenner = nenner;  
}
```

```
public Bruch(Bruch andererBruch) {  
    this(andererBruch.zaehler, andererBruch.nenner);  
}
```

```
public Bruch(int zahl) {  
    this(zahl, 1);  
}
```

## Übung: Konstruktor

- Erstellen Sie einen weiteren Konstruktor für die Klasse Bruch, der den Bruch mit einer Ganzzahl initialisiert, also einen int-Parameter hat.
- Der Konstruktor soll den Code eines bereits bestehenden Konstruktors wiederverwenden.



# Sichtbarkeit

# Einführung

- Nicht jede Methode solle von überall her aufgerufen werden können
- Beispiel: Berechnung des GGT hat nichts mit einem Bruch im Allgemeinen zu tun
  - interne Hilfsmethode
  - sollte nur intern verwendet werden können
- Lösung in Java: Sichtbarkeitsmodifizierer
  - private, public, protected

# Sichtbarkeit/Zugriffsschutz

- Modifier private für Objektvariablen/Methoden:
  - Zugriff (Sichtbarkeit) beschränkt auf die eigene Klasse
- Ohne Modifier:
  - Zugriff nur für Klassen im gleichen Package möglich
- Modifier public für Klassen/Objektvariablen/Methoden:
  - Zugriff aus beliebigen anderen Klassen und Packages erlaubt
- Modifier protected:
  - später (Vererbung: Zugriff nur für die eigene und abgeleitete Klassen)!
- Daumenregel für kleine Projekte
  - immer public oder private

# Sichtbarkeit/Zugriffsschutz

- Anpassung von Bruch

```
class Bruch {  
    private int zaehler;  
    private int nenner;  
    ...  
}
```

- Zugriff nur aus der Klasse Rational selber heraus möglich
- nicht von Anwendung (allgemein: außen) aus

```
public class BruchAnwendung{  
    public static void main(String[] args) {  
        bruch.zaehler = 23; // Fehler!  
        ...  
    }  
}
```



# Zugriffsmethoden

- Getter
- private-Objektvariablen sind von außen nicht erreichbar
- Möglichkeit, Wert dennoch zugänglich machen
  - Auskunftsmethode (engl. getter) anbieten
  - liefert den aktuellen Wert einer Variablen liefert!
- Definition einer Getter-Methode zu einer Objektvariablen  
`private <Typ> <Name>;`
- nach dem Muster (als Konvention, keine Compiler-Prüfung):  

```
public <Typ> get<Name>() {  
    return <Name>;  
}
```
- Beispiel:
  - `public int getZaehler()` für Objektvariable `zaehler`

# Zugriffsmethoden

- Setter
- private-Objektvariablen in veränderlichen Klassen
  - von außen kein Zugriff, keine Veränderung möglich
- um Modifikation zulassen
  - Änderungsmethode
  - engl. setter
  - Wertzuweisung
- Definition einer Setter-Methode zu einer Objektvariablen

```
private <Typ> <Name>;
```

- nach dem Muster (als Konvention, keine Compiler-Prüfung):

```
public void set<Name>(<Typ> <Name>) {  
    this.<Name> = <Name>;  
}
```

- Beispiel

- public void setZaehler( int neuerWert ) für Objektvariable zaehler

# Einsatz von Gettern und Settern

- Getter/Setter sind besser als public-Membervariablen
- Schutz
- logische Kapselung
- Möglichkeit zur Kontrolle der Werte
  - z.B. Test des Nenners auf 0 vor Wertzuweisung
- Möglichkeit der Fehlersuche
  - z.B. zusätzliches `System.out.println(..)` bei Wertänderungen
- abweichende Implementierung
  - Wert kann dynamisch berechnet werden, ohne dass überhaupt eine Objektvariable existiert, z.B. `getWert()`
- verborgene Optimierungen
  - z.B. Kürzen eines Bruchs erst bei Ausgabe

# Einsatz von Gettern und Settern

- aber
  - Methoden, die Operationen auf den Daten durchführen, sind besser als "dumme" Getter und Setter
- Beispiel: Erweitern von Brüchen
  - mit `void erweitere( int faktor );`
  - statt: `setZaehler( getZaehler() * faktor );`

# Fazit Sichtbarkeit

- ab sofort: Sichtbarkeitsmodifizierer für jede
  - Objektvariable
  - Methode
- Objektvariablen (quasi) immer private
  - falls Zugriff notwendig (nur dann!): Getter und/oder Setter

## **public bei Klassendefinition**

- verzichtet man bei der Deklaration einer Klasse auf public, ist sie nur im gleichen Package sichtbar
- gleiches Verhalten wie default-Sichtbarkeit (kein Sichtbarkeitsmodifikator) bei Objektvariablen und Methoden

# Übung: Sichtbarkeitsmodifizierer

- Geben Sie für jede Objektvariable und Methode an, ob Sie public oder private sein sollte:
  - int zaehler;
  - int nenner;
  - Bruch()
  - Bruch(int zaehler, int nenner)
  - Bruch(Bruch andererBruch)
  - void initialisiere(int zaehler, int nenner)
  - void print()
  - void vereinfache()
  - void erweitere(int faktor)
  - int berechneGgt(int zahl1, int zahl2)
  - void addiereDazu(Bruch andererBruch)
  - double getWert()
  - void subtrahiereDavon(Bruch andererBruch)
  - int getZaehler()
  - int getNenner()
  - void setZaehler(int neuerWert)
  - void setNenner(int neuerWert)



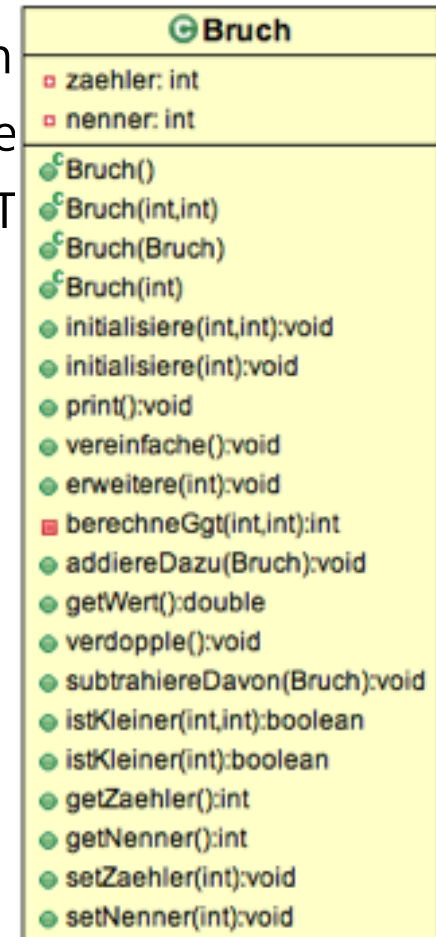
UML



# UML

- public-Elemente
  - Modifier „+“ verwenden
- private-Elemente
  - Modifier „-“ verwenden
  - oder private Variablen / Methoden weglassen (nicht Teil der Schnittstelle)

- Textebene 1
  - Textebenen
  - Textebenen
  - Textebenen



hier: grün/rot  
anstelle von +/-



# Unveränderliche Klassen

# Unveränderliche Klassen

- Idee
- früheres Beispiel (böswillige Version von addiere()) zeigt Problem:
  - Methode kann unerkannt fremdes Objekt manipulieren
  - Seiteneffekt
- Ursache
  - freier Zugriff auf Objektvariablen
- Lösungsidee
  - unveränderliche Klassen
  - engl. immutable classes
  - lassen keine externen Änderungen an Objektvariablen zu
- Folge
  - Lesen von Informationen aus Objekten ok
  - Änderung nicht möglich

# Unveränderliche Klassen

- Modifizierer final für Objektvariablen blockiert Änderungen
  - wie bei lokalen Variablen
- Zuweisung der Werte üblicherweise im Konstruktor
  - dann: nach Objekterzeugung mit new nicht mehr änderbar
- Änderungen notwendig
  - Erzeugen neuer Objekte statt Veränderung

## Beispiel: Unveränderlicher Parameter

```
public Bruch addiereDazuFinal(final Bruch andererBruch) {  
    return new Bruch(zaehler * andererBruch.nenner + nenner  
        * andererBruch.zaehler, nenner * andererBruch.nenner);  
}
```

```
Bruch bruch1 = new Bruch(2, 7);  
Bruch bruch2 = new Bruch(3, 2);  
Bruch ergebnis = bruch1.addiereDazuFinal(bruch2);  
ergebnis.print(); // Ausgabe: 25,14
```

# Unveränderliche Klassen

- Werte- und Referenzsemantik
- Wertesemantik
  - engl. value semantics
  - Methoden lassen existierende Objekte unverändert
    - this und Parameter-Objekte
  - Verhalten entspricht dem von primitiven Typen
    - Operationen lassen Operanden unberührt
- Gegensatz: Referenzsemantik
  - engl. reference semantics
  - Operationen können this oder Parameter-Objekte verändern

# Werte- vs. Referenzsemantik

Primitive Typen	Wertesemantik
Unveränderliche Klassen	Wertesemantik
Veränderliche Klassen	Referenzsemantik

# Unveränderliche Klassen

- Wertesemantik vorziehen, wenn möglich
- Beispiel: Suche nach der Ursache für fehlerhaftes Objekt
  - Referenzsemantik
    - Jeder Methodenaufruf kommt in Betracht, muss überprüft werden
    - Beispiel: `bruch1.addiereDazu(bruch2);`
      - könnte a oder b oder keines oder beide verändern
  - Wertesemantik
    - nur Konstruktoraufrufe sind zu prüfen
    - Methodenaufrufe können ein korrekt erzeugtes Objekt nicht mehr stören
- Definieren Sie Klassen mit Wertesemantik (= unveränderlich), wo immer möglich!



# Unveränderliche Klassen

- trotz Wertesemantik
  - Modifier final eventuell zu rigoros
- Beispiel
  - Brüche  $6/8$  und  $3/4$  logisch (mathematisch) gleichwertig
  - Kürzen eines Bruch-Objektes zulässig, „Wert“ bleibt unverändert
- technisch
  - kontrollierte Veränderungen am Objekt zulassen
  - dennoch logisch Wertesemantik einhalten!

# Übung: Unveränderliche Klassen

- Nehmen wir an, die Klasse Bruch ist unveränderlich umgesetzt, also zaehler und nenner sind als final deklariert.
- Geben Sie neue Methoden für
  - erweitere(int faktor) und
  - vereinfache()in dieser Klasse an.

# Zusammenfassung

- Konstruktoren
- Sichtbarkeit
- UML
- Unveränderliche Klassen