Prozeduren vs. Funktionen

- Mit der Formalisierung wird auch der Unterschied zwischen Prozeduren und Funktionen noch einmal klar.
- Der Aufruf beider Varianten bewirkt zunächst das Gleiche: die Eingabevariablen werden mit konkreten Werten belegt.
- Bei Funktionen wird anschließend der Wert des Rumpfes der Funktion (ein Ausdruck) ausgewertet; der Funktionsaufruf ist ja selbst wieder ein Ausdruck (mit einem Wert).
- Dabei wird die Variablenbelegung innerhalb des Rumpfes der Funktion nicht verändert, wir haben keine Zustandsänderung (daher hatten wir auch keinen Zustandsbegriff benötigt).

Prozeduren vs. Funktionen

- Bei Prozeduren können weitere Variablen und Konstanten (mit entsprechenden Belegungen) zu der Menge der Eingabevariablen hinzukommen.
- Die Anweisungen im Rumpf verändern ggf. deren Belegungen (das sind die vielbeschworenen Seiteneffekte), was wir mit dem Zustandsbegriff modelliert haben.
- Bei der (nicht seltenen) Mischform, Prozeduren die einen Rückgabewert vom Typ $\mathtt{T} \neq$ `void` besitzen, stellt der Ausdruck nach der `return`-Anweisung also nicht notwendigerweise den direkten Zusammenhang zwischen Eingabewerten und Ausgabe dar, da der Zustand am Ende des Methodenrumpfes typischerweise nicht mehr mit dem Startzustand (bei Aufruf) übereinstimmt.

Globale Größen

- Variablen und Konstanten, so wie wir sie bisher kennengelernt haben, sind *lokale Größen*, d.h. sie sind nur innerhalb des Blocks (auch Methodenrumpf), der sie verwendet, bekannt.
- Es gibt auch *globale* Variablen und Konstanten, die in mehreren Algorithmen (Methoden und sogar Klassen) bekannt sind.
- Diese globalen Größen sind z.B. für den Datenaustausch zwischen verschiedenen Algorithmen geeignet.
- Sie können in einem Modul (Klassenvereinbarung) spezifiziert werden und (wenn sie als `public` deklariert wurden) von den Methoden aller anderen Module (Klassen) verwendet werden.

Globale Größen

- Globale Variablen heißen in Java *Klassenvariablen*, die man üblicherweise am Beginn einer Klasse (das muss aber syntaktisch nicht sein) definiert.
- Wie erwähnt, gelten diese Variablen in der gesamten Klasse (im gesamten Modul) und ggf. auch darüber hinaus, stehen also für die Bildung von Ausdrücken zur Verfügung.
- Die Definition wird von den Schlüsselwörtern **`public`** und **`static`** eingeleitet (deren genaue Bedeutung wir immer noch erst später kennen lernen).
- Klassenvariablen kann man auch als Konstanten definieren, wie bei lokalen Konstanten dient hierzu das Schlüsselwort **`final`**

Globale Konstante – Beispiel

```
public class Kreis {
      /**
       * Die Konstante spezifiziert die Kreiszahl pi.
       */
      public static final double PI = 3.14159;

      /**
       * Berechnung des Umfangs eines Kreises mit gegebenem Radius.
       * @param radius Der Radius des Kreises.
       * @return Der Umfang des Kreises.
       */
      public static double kreisUmfang(double radius) {
            return 2 * PI * radius;
      }
}
```

Globale Variable – Beispiel

```
public class OhneGlobaleVar {

     public static void add(int x) {
          int sum = 0;
          sum = sum + x;
     }

     public static void main(String[] args) {
          int sum = 0;
          add(3);
          add(5);
          add(7);
          System.out.println("Summe: "+sum);
     }
}
```

Ausgabe: `Summe: 0`

```
public class MitGlobalerVar {

     public static int sum = 0;

     public static void add(int x) {
          sum = sum + x;
     }

     public static void main(String[] args) {
          int sum = 0;
          add(3);
          add(5);
          add(7);
          System.out.println("Summe: "+sum);
     }
}
```

Ausgabe: `Summe: 15`

Globale Größen

- Im Gegensatz zu lokalen Variablen muss man Klassenvariablen nicht explizit initialisieren.
- Sie werden dann automatisch mit ihren Standardwerten initialisiert:

Typ	Standardwert
`boolean`	`false`
`char`	`u0000`
`byte, short, int, long`	`0`
`float, double`	`0.0`

- Klassenkonstanten müssen dagegen explizit initialisiert werden.

Sichtbarkeit globaler Größen

- Lokale Variablen innerhalb einer Klasse können genauso heißen wie eine Klassenvariable.
- Beispiel:

```
public class Sichtbarkeit
{
     public static int variablenname;

     public static void main(String[] args) {
          boolean variablenname = true;
     }
}
```

- Das bedeutet: Während bei lokalen Variablen Sichtbarkeit und Gültigkeit zusammenfallen, muss man zwischen beiden Eigenschaften bei Klassenvariablen prinzipiell unterscheiden.

Sichtbarkeit globaler Größen

- Das ist kein Widerspruch zum Verbot, den gleichen Namen innerhalb des Gültigkeitsbereichs einer Variable nochmal zu verwenden, denn genau genommen heißt die Klassenvariable anders.
- Zu ihrem Namen gehört der vollständige Klassenname, in dem die Variable/Konstante definiert wurde (übrigens inklusive des Package-Namens, was ein Package ist, lernen wir aber — Sie ahnen es — erst später).
- Unter dem vollständigen Namen ist eine globale Größe auch dann sichtbar, wenn der innerhalb der Klasse geltende Name durch den identisch gewählten Namen einer lokalen Variable verdeckt ist.

Sichtbarkeit globaler Größen

- I.Ü. gilt diese Namensregel analog für unsere (statischen) Methoden; dadurch können auch Methoden, die gleich benannt sind (und sogar die selbe Signatur haben) aber in unterschiedlichen Klassen vereinbart wurden, eindeutig unterschieden werden.
- Das ist nicht ganz unwichtig schließlich könnte es in einem großen Software-Kosmos dann nur jeweils eine einzige Methode mit einer speziellen Signatur `type m(inputType inputParam)` geben.
- Vorsicht, das hat nix — äh tschuldigung: nichts — mit dem Konzept des Überladens zu tun!

Namen statischer Elemente

- Beispiel:
  - Der (vorläufig) vollständige Name der Konstanten `PI` aus der Klasse `Kreis` ist also:

    ```
    Kreis.PI
    ```

    und stellt die Kreiszahl allen Modulen zur Verfügung.
  - Das uns bereits bekannte Modul `Math` stellt zwei Konstanten zur Verfügung:
    - die Eulersche Zahl `Math.E`
      (The double value that is closer than any other to e, the base of the natural logarithms)
    - und ebenfalls die Kreiszahl `Math.PI`
      (The double value that is closer than any other to π, the ratio of the circumference of a circle to its diameter)

Namen statischer Elemente

- Beispiel mit Methoden
  Die beiden Methoden `add` der Klassen `OhneGlobaleVar` und `MitGlobalerVar` haben auf den ersten Blick exakt dieselbe Signatur, können aber anhand des (vorläufig) vollständigen Namens eindeutig unterschieden werden:

  `OhneGlobaleVar.add` `MitGlobalerVar.add`

- Statische Methoden und statische (globale) Variablen/Konstanten heißen auch Klassenmethoden bzw. Klassenvariablen/-konstanten.

Zustände globaler Größen

- Wie passen globale Größen in unser Zustandsmodell?
- Wir müssen nur die Menge der Namen $N(\mathcal{S})$ um die globalen Variablen und Konstanten erweitern, ansonsten werden sie behandelt wie lokale Variablen, d.h. die Semantik der Zustandsübergänge gilt analog.
- Einziger Unterschied: sie sind wirklich überall sichtbar/gültig, auch innerhalb des Rumpfes einer aufgerufenen Methode.

Zustände globaler Größen

- Sind wirklich alle globalen Variablen/Konstanten immer bekannt (d.h. die entsprechenden Zettel existieren)?
- Es würde wenig Sinn machen alle Variablen/Konstanten aller Module, die es auf der Welt gibt, in $N(\mathcal{S})$ aufzunehmen (was für eine Papierverschwendung!!!).
- Eine ähnliche Problematik ergibt sich übrigens bei den Methoden anderer Klassen.

Zustände globaler Größen

- Tatsächlich muss man globale Größen und Methoden anderer Module (Klassen) explizit *laden*.
- Dies funktioniert in Java mit der Anweisung **`import`**, die wir in Zusammenhang mit Packages noch kennen lernen werden.
- Nur schon mal soviel: es gibt Module (aus der Standardbibliothek), die automatisch geladen werden (z.B. `Math`), sodass diese nicht explizit geladen werden müssen (d.h. deren Klassenvariablen/-konstanten stehen implizit zur Verfügung bzw. sind immer Teil von $N(\mathcal{S})$).

Vorsicht mit globalen Größen

- *Achtung*: Globale Variablen möglichst vermeiden, ansonsten nur, wenn
  - sie in mehreren Methoden verwendet werden müssen
  - ihr Wert zwischen Methodenaufrufen erhalten bleiben muss
- Wann immer möglich, lokale Variablen verwenden
  - Einfachere Namenswahl
  - Bessere Lesbarkeit: Deklaration und Verwendung der Variablen liegen nahe beieinander
  - Keine Nebeneffekte: Lokale Variablen können nicht durch andere Methoden versehentlich überschrieben werden