Wozu Methoden?

- Strukturierung
  - Langen Programmcode übersichtlich machen
  - Aufteilung in kleine, gut verständliche Funktionseinheiten
- Vermeidung von Codeverdopplung
  - Gleicher Programmcode wird einmal als Methode implementiert
  - Kann öfters verwendet werden

Parameterlose Methoden

Beispiel: Ausgabe einer Überschrift

```
class Sample {

   static void PrintHeader()                  // Methodenkopf
   {
      Console.WriteLine("Artikelliste");  // Methodenrumpf
      Console.WriteLine("------------");
   }

   static void Main (String[] args)
   {
      PrintHeader();                          // Aufruf
      ...
      PrintHeader();
      ...
   }
}
```

Vorteile:

- Methode PrintHeader wird öfter verwendet
- Ausgabe des Headers funktioniert immer gleich
  - Will man etwas ändern, muss es nur einmal geändert werden – in der Methode!
- Strukturierung des Programms

Namenskonvention Methoden

- CamelCase mit großem Anfangsbuchstaben
- Beispiele:
  - Math.**Sqrt**
  - Console.**WriteLine**
  - Console.**ReadLine**
  - Convert.**ToInt32**
  - **PrintHeader**
  - **FindMaximum**

Parameter (Wertparameter -> call by value)

Werte, die vom Rufer an die Methode übergeben werden

```
class Sample {
   static void PrintMax (int x, int y) {
      if (x > y) Console.Write(x);
      else Console.Write(y);
   }

   static void Main () {
      ...
      PrintMax(100, 2 * i);
   }
}
```

formale Parameter
- im Methodenkopf (hier x, y)
- sind Variablen der Methode

aktuelle Parameter
- an der Aufrufstelle (hier 100, 2*i)
- können Ausdrücke sein

Parameterübergabe

Aktuelle Parameter werden den entsprechenden formalen Parametern zugewiesen

x = 100; y = 2 * i;

⇒ aktuelle Parameter müssen mit formalen zuweisungskompatibel sein

⇒ formale Parameter enthalten Kopien der aktuellen Parameter

Funktionen

Methoden, die einen Ergebniswert an den Rufer liefern

```
class Sample {

   static int Max (int x, int y) {
      if (x > y) return x; else return y;
   }

   static void Main (string[] arg) {
      ...
      Console.WriteLine(Max(100, i + j) + 1);
   }
}
```

- haben Funktionstyp (z.B. *int*) statt void (= kein Typ)
- liefern Ergebnis mittels return-Anweisung an den Rufer
  (*x* muss zuweisungskompatibel mit *int* sein)
- Werden wie Operanden in einem Ausdruck benutzt

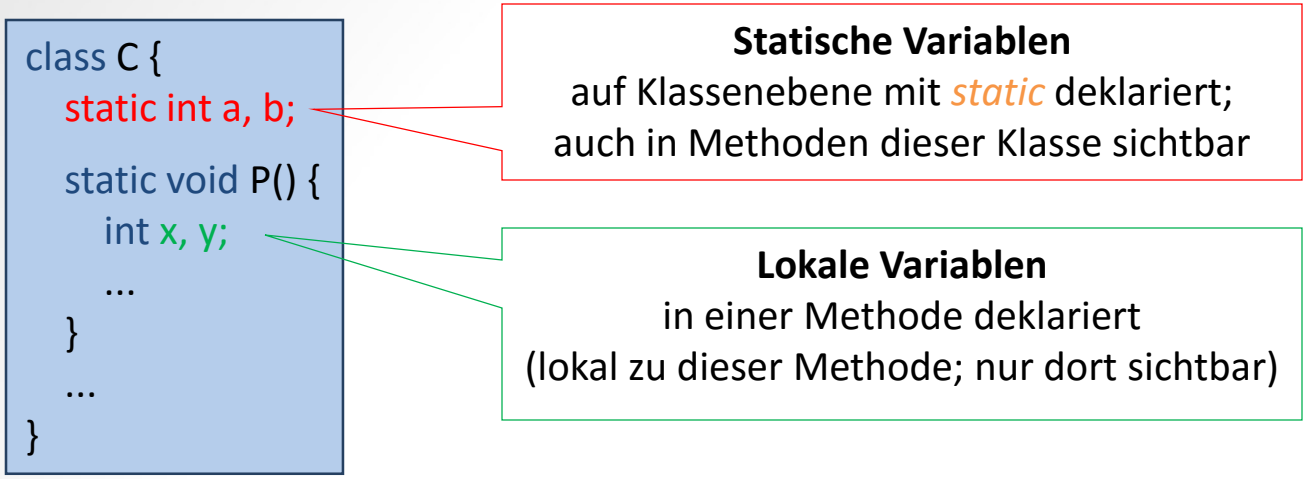
Funktionen Methoden mit Rückgabewert
Prozeduren Methoden ohne Rückgabewert

Weiteres Beispiel

Ganzzahliger Zweierlogarithmus

```
class Sample {

   static int Log2 (int x)
   { // assert: x >= 0
      int expo= 0;
      while (x > 1) {x = x / 2; expo++;}
      return expo;
   }

   static void Main ()
   {
      int x = Log2(17);  // x == 4
      ...
   }
}
```

Lokale und statische Variablen



Reservieren und Freigeben von Speicherplatz

Statische Variablen
am Programmbeginn angelegt
am Programmende wieder freigegeben

Lokale Variablen
bei jedem Aufruf der Methode neu angelegt
am Ende der Methode wieder freigegeben

Beispiel: Summe einer Zahlenfolge

Semantisch falsch!

```
class Wrong {

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

   static void Main() {
      Add(1); Add(2); Add(3);
      Console.WriteLine("sum = " + sum);
   }
}
```

Semantisch richtig!

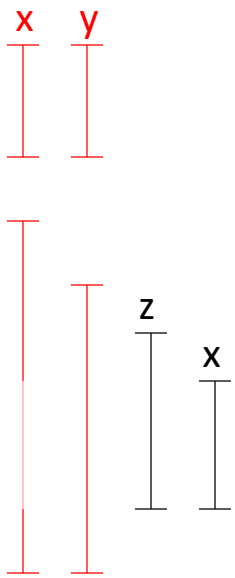
```
class Correct {

   static int sum = 0;

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

   static void Main() {
      Add(1); Add(2); Add(3);
      Console.WriteLine("sum = " + sum);
   }
}
```

- *sum* ist in *Main* nicht sichtbar
- *sum* wird bei jedem Aufruf von *Add* neu angelegt (alter Wert geht verloren)

Sichtbarkeitsbereich von Namen

Programmstück, in dem auf diesen Namen zugegriffen werden kann
(auch *Gültigkeitsbereich* oder *Scope* des Namens genannt)

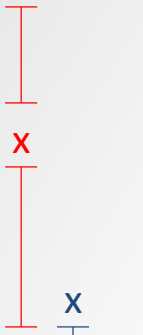
```
class Sample {
   static void P() {
      ...
   }
   static int x;
   static int y;
   static void Q(int z) {
      int x;
      ...
   }
}
```



Regeln

1. Ein Name darf in einem Block nicht mehrmals deklariert werden (auch nicht in geschachtelten Anweisungsblöcken).
2. Lokale Namen verdecken Namen, die auf Klassenebene deklariert sind.
3. Der Sichtbarkeitsbereich eines lokalen Namens beginnt bei seiner Deklaration und geht bis zum Ende der Methode.
4. Auf Klassenebene deklarierte Namen sind in allen Methoden der Klasse sichtbar.

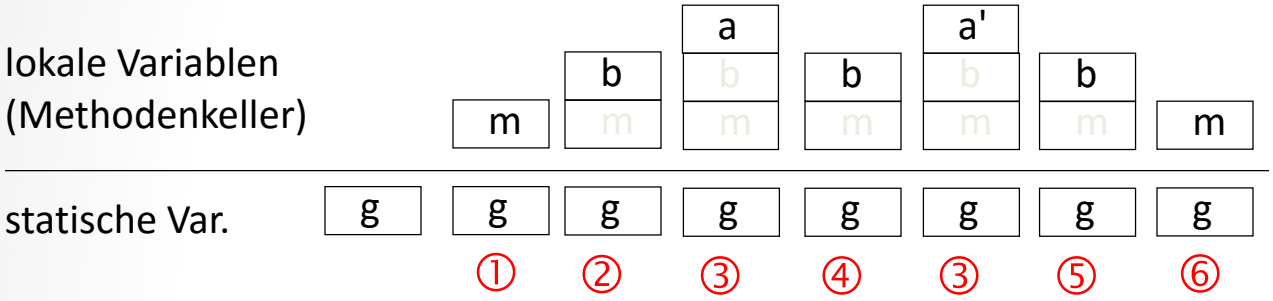
Beispiel zu Sichtbarkeitsregeln




```
class Sample {

   static void P() {
      Console.WriteLine(x);          // gibt 0 aus
   }

   static int x = 0;

   static void Main() {
      Console.WriteLine(x);          // gibt 0 aus
      int x = 1;                     // verdeckt statisches x
      Console.WriteLine(x);          // gibt 1 aus
      P();
      if (x > 0) {
         int x;                      // Fehler: x ist in main bereits deklariert
         int y;
         ...
      } else {
         int y;                      // ok, kein Konflikt mit y im then-Zweig
         ...
      }
      for (int i = 0; ...) {...}
      for (int i = 1; ...) {...}     // ok, kein Konflikt mit i aus letzter Schleife
   }
}
```

Lebensdauer von Variablen

```
class LifenessDemo {
   static int g;
   static void A() {
      int a;
   ③...
   }
   static void B() {
      int b;
   ②... A(); ④... A(); ...⑤
   }
   static void Main() {
      int m;
   ①... B(); ...⑥
   }
}
```



HTL LEONDING

Lokalitätsprinzip

Variablen möglichst lokal deklarieren, nicht als statische Variablen!

Vorteile

- Übersichtlichkeit
  Deklaration und Benutzung nahe beisammen
- Sicherheit
  Lokale Variablen können nicht durch andere Methoden zerstört werden
- Effizienz
  Zugriff auf lokale Variable ist oft schneller als auf statische Variable