Lokale und globale Variablen

- Wir wollen eine Methode schreiben, die **bei jedem Aufruf** ihren Parameter `x` **zu einer Variablen** `sum` **addiert**
- Was gibt das neben-stehende Programm aus?

```
Summe: 0
```

- Ist dies das gewünschte Ergebnis?

```
class Programm{

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

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

}
```


Globale Variablen

- **Gewünscht** war wohl eher:

```
Summe: 15
```

- Eine Variable ist **global**, wenn sie außerhalb jeder Methode deklariert wurde
- Globale Variablen
  - werden vorerst immer als `static` deklariert
  - können in **allen Methoden** benutzt werden
- **Speicherplatz** für eine statische globale Variable
  - wird angelegt, **sobald** das **Programm gestartet** wird
  - existiert **solange** das Programm **läuft**

```
class GlobExample{
   static int sum = 0;

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

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


Lokale Variablen

- Lokale Variablen sind **außerhalb** einer Methode **nicht sichtbar**
- Der Speicherplatz für lokale Variablen wird **bei jedem Methodenaufruf neu** angelegt
- **Nach Beenden** einer Methode wird der **Speicherplatz wieder freigegeben**
- Lokale Variablen **leben also nur während der Ausführung** ihrer Methode

Ausgabe:
`20 kleiner 21`

```
class LocExample{
   static void minP10(int x, int y)
   {
      String s;
      x += 10;
      s=(x<y)?" kleiner ":" groesser ";
      System.out.println(x+s+y);
    }

   public static void main(String[] a)
   {
      int x = 10;
      minP10(x,21);
      System.out.println(x);

      System.out.println(s); // error

   }

}
```


Einsatz globaler Variablen

- **Globale** Variablen möglichst **vermeiden**
- Wenn globale Variable wirklich notwendig, dann nur einsetzen, 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 Nebeneffekten**
    - Lokale Variablen können nicht durch andere Methoden versehentlich überschrieben werden
- Erzeugen und Löschen **lokaler Variablen** ist in Java sehr **effizient** implementiert
  - Die meisten Rechner-Architekturen können auf **lokale** Variablen **effizienter** zugreifen als auf globale
    - Caching

Lokale und globale Konstanten

- **Globale und lokale Konstanten** können mit Hilfe des Schlüsselwortes `final` deklariert werden
- Beispiel:

```
class Program{
        static int a;

        // global constant (with static)
        static final float pi =3.14159265f;

        static void p(int x){
                // local constant (without static)
                final boolean debug = true;
                ...
        }

        public static void main(String[] args) {...}
}
```

- **Konstanten** werden üblicherweise von mehreren Methoden benutzt und sind daher **meist global**

Sichtbarkeitsbereich von Variablen

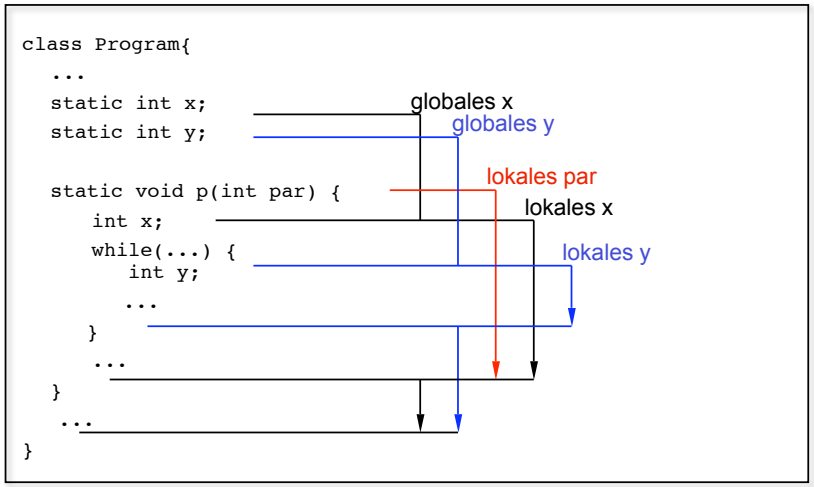
- **Der Sichtbarkeitsbereich** (Gültigkeitsbereich) einer Variablen
  - ist der Bereich, in dem auf die Variable **zugegriffen** werden kann
  - vom Punkt der **Deklaration bis zum Ende des Blocks** in dem die Variable deklariert wurde
    - **Lokale Variable**
      - Bis zum Ende der Methode
    - **Globale Variable**
      - Bis zum Programmende
- In **Blöcken** deklarierte Variablen sind bis Block-Ende sichtbar
  - Variablen-Namen im Block müssen sich von den Variablen-Namen der zugehörigen Methode **unterscheiden**
- **Lokale** Variablen **verdecken globale** Variablen

```
class Programm{
  static int x=10;
  static int y=5;

   static void method(int x){
      int z = 0;
      while(z<10) {
        x = 10 * z;
        int y;
        y = 20 * z++;
      }
      System.out.println("2. x y: "+x+" "+y);
   }

   public static void main(String[] args)
   {
     System.out.println("1. x y: "+x+" "+y);
     {
       int y = 10 ;
       method(y);
       System.out.println("3. x y: "+x+" "+y);
     }
     System.out.println("4. x y: "+x+" "+y);
   }
}
```

Ausgabe:
```
1. x y: 10 5
2. x y: 90 5
3. x y: 10 10
4. x y: 10 5
```


Sichtbarkeit von Variablen (Beispiel)

- **Beispiel**



Zeichen (Charakter)

- Häufig vorkommende **Steuerzeichen**
  - `'\n'` **new line (LF)**
  - `'\r'` **return (CR)**
  - `'\t'` **Tabulatorsprung**

- Beispiele für **Unicode-Nummernbereiche**
  - `\u0000 - \u007f` **ASCII Zeichen**
  - `\u0080 - \u024f` **Umlaute, Akzente, Sonderzeichen**
    - `\u0034` **ä** `\u00c4` **Ä**
    - `\u00df` **ß**
    - `\u00f6` **ö** `\u00d6` **Ö**
    - `\u00fc` **ü** `\u00dc` **Ü**
  - `\u0370 - \u03ff` **griechische Zeichen** (z.B. \u03c0 = π)
  - `\u0600 - \u06ff` **arabische Zeichen**

- **Vollständige Definition** des Unicodes unter `www.unicode.org`

Standardoperationen für Zeichen

- **Vollständig** unter
  `http://java.sun.com/j2se/1.5.0/docs/api`
- `Character.isLetter(ch)`
  - Ist `ch` ein **Unicode-Buchstabe**?
- `Character.isDigit(ch)`
  - Ist `ch` eine **Ziffer**?
- `Character.isLetterOrDigit(ch)`
  - Ist `ch` ein **Unicode-Buchstabe oder** eine **Ziffer**?
- `Character.toUpperCase(ch);`
  - **Umwandeln** von `ch` in einen **Großbuchstaben**
  - Ist ch kein Kleinbuchstabe passiert nix
- `Character.toLowerCase(ch);`
  - **Umwandeln** von ch in einen **Kleinbuchstaben**
  - Ist `ch` kein Großbuchstabe passiert nix

```
class Programm{
    public static void main(String[] args)
    {
      char cA = 'A';
      char cB = '1';
      char cC = '+';
      System.out.
      println(Character.isDigit(cA));
      System.out.
      println(Character.isDigit(cB));
      System.out.
      println(Character.isLetterOrDigit(cC));
      System.out.
      println(Character.toLowerCase(cA));
      System.out.
      println(Character.toLowerCase(cC));
    }
}
```

Ausgabe:
```
false
true
false
a
+
```


Beispiel für Zeichen-Arrays

- Finde das **erste Auftauchen** des Zeichenarrays `pat` in einem Zeichenarray `s`

```
char s[] = {'E','i','n','f','u','e','h','r','u','n','g',' ','i','n',' ',
'd','i','e',' ','P','r','o','g','r','a','m','m','i','e','r','u',n','g'};
char pat[] = {'i','e'};
```

```
boolean found = false;
int i,j;
int last = s.length - pat.length; // last possible position
for (i=0; i<=last; i++) {
   if(s[i]==pat[0]) { // first char of pat matches
      j=1;
      while(j<pat.length && pat[j]==s[i+j]) j++;
      if(j==pat.length) {
         found = true;
         break;
      }
   }
}
if(found) System.out.println("pattern found on position " + i);
else      System.out.println("pattern not found");
```


Zeichenketten (Strings)

- **Eigener Datentyp** für **Zeichenketten** (*Strings*), da häufig verwendet
- String-Konstanten
  - Zeichenketten zwischen doppelten Hochkommata
  - Am Bildschirm nicht darstellbare Zeichen bzw. nicht auf der Tastatur zu findende Zeichen, können durch **Escape-Sequenzen** ausgedrückt werden
    - Beispiel '`\n`' oder '`\u03c0`'
    - Siehe Unicode-Tabelle

Unterscheide
- `"x"`
  - Ist ein **String**
- `'x'`
  - Ist ein **Character**

```
class Program {
  public static void main(String[] args)
  {
    String S1 = "fruit flies like bananas";
    String S2 = "Alice\t2000\nBob\t1000";
    String S3 = "Otto contains \"tt\"";
    String S4 = "2\u03c0 is the
         circumference of the unit circle";

    System.out.println(S1);
    System.out.println(S2);
    System.out.println(S3);
    System.out.println(S4);
  }
}
```

Ausgabe
```
fruit flies like bananas
Alice   2000
Bob     1000
Otto contains "tt"
2π is the circumference of the unit circle
```


String

- String ist eine **Klasse** (Verallgemeinerung von Datentyp)
  - **Typ einer Variablen** kann ein Datentyp oder eine Klasse sein
  - Datentypen haben Werte, Klassen haben **Instanzen**
  - Für den Umgang mit Instanzen von Klassen stehen **Methoden** zur Verfügung
- Ist der Typ einer Variablen eine Klasse, so wird nicht die Instanz (der Wert), sondern eine **Referenz auf die Instanz** in der Variablen gespeichert

Beispiel

```
String s = "fruit flies like bananas";
```

Deklaration von String-Variablen

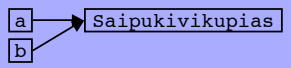
- **Deklaration** von String-Variablen

```
String s;
```

- Einer String-Variablen können **String-Konstanten** oder andere **String-Variablen** zugewiesen werden
- **Analog zu den Feld-Variablen** (Arrays) enthalten String-Variable Referenzen von Stringobjekten, nicht die Stringobjekte selbst

Beispiel:

```
String a, b;
a = "Saipukivikupias";
b = a;
```

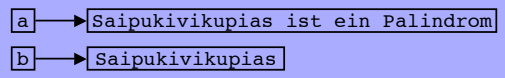


- Mann kann auf die **einzelnen Zeichen** eines Strings **nicht mittels Indizierung zugreifen**

Stringobjekte sind nicht veränderbar

- Stringobjekte sind **nicht veränderbar**
- Beispiel

```
String a, b;
a = "Saipukivikupias";
b = a;
a = a + " ist ein Palindrom";
```



– Es wird ein **neues Stringobjekt** erzeugt, auf das anschließend `a` verweist

– `b` zeigt immer noch auf das alte Stringobjekt

Verkettung von Strings

- Strings können mit dem "+" Operator **verkettet** werden
- Ist in einem Verkettungsausdruck einer der Operanden nicht vom Typ String, so wird dieser **automatisch in die Stringdarstellung umgewandelt** (Typcast)

```
// Welchen Wert hat s?

boolean b = true;
String s = "\u03c0 is approx. ";
s += 3.14 + ": "+ b;
```

`s` hat den Wert:

"π is approx. 3.14: true"

Vergleichen von Strings

- Der String-**Vergleich**

```
String s = "Hallo";
if(s=="Hallo") System.out.println(true);
else System.out.println(false);
```

liefert `false`

- **Vergleich der Referenzen**, kein Wertevergleich
  - `s` zeigt auf Stringobjekt mit Wert "Hallo"
  - Stringkonstante "Hallo" ist **anderes Objekt** (gleicher Inhalt)
  - Die beiden Objekte haben **verschiedene Adressen**
- **Wertevergleich** mit Hilfe der Methode `equals(...)`
  Beispiel

```
if (s.equals("Hallo")) System.out.println(true);
else System.out.println(false);
```

Methoden von String (kein Parameter)

- **Vollständig unter** `http://java.sun.com/j2se/1.5.0/docs/api`
- `l=s.length();`
  - **Länge** l eines Strings `s`
  - Unterschied zur Länge eines Arrays: `a.length`
- `s2 = s.trim();`
  - **Entfernen von Leerzeichen** an den String-Enden von `s`
  - Es wird ein neuer String erzeugt
- `s2 = s.toUpperCase();`
  - **Umwandlung in Großbuchstaben**
  - Es wird ein neuer String erzeugt
- `s2 = s.toLowerCase();`
  - **Umwandlung in Kleinbuchstaben**
  - Es wird ein neuer String erzeugt

Methoden von String (1 Parameter)

- **Vollständig unter** `http://java.sun.com/j2se/1.5.0/docs/api`
- `ch = s.charAt(i);`
  - **Zeichen** mit Index i aus String `s` (Indizierung beginnt bei 0)
- `i = s.indexOf(pat);`
  - **Position** des **ersten Vorkommens** von `pat` (oder -1)
- `s2 = s.substring(pos);`
  - **Teilstring** von `s` ab Position `pos` (erzeugt neuen String)
- `if (s.startsWith(pat))...`
  - **Vergleicht** den **Anfang** eines Strings `s` mit `pat`:
- `if (s.endsWith(pat)) ...`
  - **Vergleicht** das **Ende** eines Strings `s` mit `pat:`
- `i = s.compareTo(s2);`
  - **Alphabetischer** Vergleich zwischen String `s` und `s2`.
  - Rückgabewert:
    - `i<0` für `s<s2`
    - `i>0` für `s >s2`
    - `i=0` für `s =s2`

Methoden von String (2 Parameter)

- **Vollständig unter** `http://java.sun.com/j2se/1.5.0/docs/api`
- `s2 = s.substring(pos, end);`
  - **Teilstring** von `s` ab Position `pos` bis (ausschließlich) `end`
- `i = s.indexOf(pat,pos);`
  - **Position** des **ersten Vorkommens** von `pat` (oder -1).
  - **Begin** der Suche ab Position `pos`
- `i = s.lastIndexOf(pat);`
  - **Position** des **letzten Vorkommens** von `pat` (oder -1)
- `s2 = s.replace(ch1, ch2);`
  - J**edes Vorkommen** des Zeichens `ch1` durch das Zeichen `ch2` **ersetzen**
  - Erzeugt den neuen String `s2`

Schrittweiser Aufbau von Strings

- **Stringobjekte sind konstant,**
  - eignen sich **nicht zum schrittweisen Aufbau** einer Zeichenkette
- Möglichkeiten der **String-Erzeugung.**
  - String wird in seiner **endgültigen Form erzeugt**
  - String wird in einem `char`-**Array** aufgebaut und **anschließend** in ein Stringobjekt **umgewandelt**
  - String wird in einem `SringBuffer` **Objekt** aufgebaut und **anschließend** in ein Stringobjekt **umgewandelt**

Erzeugen von Strings aus einem `char`-**Array**

```
...
char[] c = new char[26];
for(int i=0; i<26; ++i) c[i] =(char)('A'+i);
String s1 = new String(c);
String s2 = new String(c,23,3);
```

Inhalt von `s1, s2`

```
s1 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
s2 = "XYZ"
```

StringBuffer

- Der Typ `StringBuffer` verhält sich wie `String`, **kann** aber **editiert werden**
- `StringBuffer` wird verwendet um eine **Zeichenkette aufzubauen** und sie anschließend in einen `String` zu konvertieren

Erzeugen von Strings aus einem `StringBuffer`

```
StringBuffer b = new StringBuffer();
```

- **Erzeugt ein leeres** `StringBuffer`-Objekt

  und weist dessen Adresse der Variablen `b` zu

```
b.append("Java");
```

- **Speichert** `"Java"` in `b`

```
String c = new String(b);
```

- **Erzeugt eine String** `c` mit dem Inhalt `"Java"`

Methoden von StringBuffer

- **Vollständig unter** `http://java.sun.com/j2se/1.5.0/docs/api`
- `i = b.length();`
  - **Anzahl** der **Zeichen** in einem StringBuffer `b`:
- `b.append(x);`
  - **Anhängen** von `x` an einen StringBuffer `b`:
  - Mögliche Typen von x
    - `char`, `int`, `long`, `float`, `double`, `boolean`, `String` und `char[]`
- `b.insert(pos, x);`
  - **Einfügen** von `x` an der Stelle `pos` in einem StringBuffer `b`
  - Mögliche Typen von x
    - `char`, `int`, `long`, `float`, `double`, `boolean`, `String` und `char[]`
- `b.delete(from,to);`
  - **Löschen** der Zeichen `b[from]` bis `b[to-1]` eines StringBuffers b

Methoden von StringBuffer

- **Vollständig unter** `http://java.sun.com/j2se/1.5.0/docs/api`
- `b.replace(from,to,pat);`
  - **Ersetzen** von `b[from]` bis `b[to-1]` durch `pat`
- `s = b.substring(from, to);`
  - `b[from]` bis `b[to-1]` als **String zurückgeben**
- `ch = b.charAt(i);`
  - Das **Zeichen** mit Index `i` aus `b` **zurückliefern**
- `b.setChar(i,c);`
  - Das Zeichen mit Index `i` in `b` duch **Charakter** `c` **ersetzen**
- `s = b.toString();`
  - **String** mit selbem Inhalt wie `b` **zurückgeben**
- Die Methoden `append`, `insert`, `delete` und `replace` sind Funktionen, die den veränderten `StringBuffer` **zurückgeben**
  - Zulässige Schreibweise: `a.append(x).append(y).delete(5,10);`

Stringkonvertierung

- **Umwandlung** von Werten verschiedener Typen (`int`, `float`, `char[]`) **in String und umgekehrt**
- Erzeugen einer **Zahl aus einem String**
  - `int  i = Integer.parseInt("12345");`
  - `short  h = Short.parseShort("12345");`
  - `long  l = Long.parseLong("12345");`
  - `float  f = Float.parseFloat("3.14159");`
  - `double d = Double.parseDouble("3.14159");`
- Erzeugen eines **Strings aus** `x`**-Wert**
  - `String s = String.valueOf(x);`
  - **Typ von** `x` kann sein
    - `char, int, long, float, double, boolean` oder `char[]`
- Erzeugen eines `char`-Arrays mit dem Inhalt des Strings `s`
  - `char[] a = s.toCharArray();`

Aufzählungen

- **Aufzählung** = kleine Menge mit konstantem Wertevorrat
- Variablen sollen **nur zuvor vereinbarte Werte** annehmen
- Bis inklusive Java 1.4
  - **Modellierung über ganzzahlige Werte** (Konstanten)
  - **Nachteil**
    - Einfache Implementierung **nicht typsicher**
    - Typsichere Implementierung aufwändig (Konstanten als Objekte einer Klasse)
- **Ab Java 1.5**
  - **Definition von Aufzählungen** mit `enum`

```
enum Typname { Wert1, Wert2, ... }
```

```
public class Wochentag {
   public static final int MONTAG = 0;
   public static final int DIENSTAG = 1;
   public static final int MITTWOCH = 2;
   public static final int DONNERSTAG = 3;
   public static final int FREITAG = 4;
   public static final int SAMSTAG = 5;
   public static final int SONNTAG = 6;
}

public class AufzaehlungsBeispiel {
   public static void main(String[] args)
   {
      int tag = Wochentag.FREITAG;
      tag = 17; // Wird nicht gecheckt!!!
      if (tag == Wochentag.SONNTAG)
          System.out.println("Ruhen.");
   }
}
```


Der Aufzählungstyp `enum` (ab Java 5)

- **Definition außerhalb** von Methoden
- **Werte** können
  - mit `System.out.println()` im **Klartext ausgegeben** werden
  - Mit `equals` **auf Gleichheit geprüft** werden
  - in `switch`-Anweisungen verwendet werden
- **Typname** besitzt eine **Methode** `values`, die ein **Array** (vom Typ `String[]`) **aller Werte** liefert

```
Ausgabe: MONTAG
         DIENSTAG
         MITTWOCH
         DONNERSTAG
         FREITAG
         SAMSTAG
         SONNTAG
```

```
public class BeispielMitEnum {
   enum Wochentag {MONTAG, DIENSTAG, MITTWOCH,
          DONNERSTAG, FREITAG, SAMSTAG, SONNTAG}
   public static void main(String[] args)
   {
      Wochentag tag = Wochentag.FREITAG;

      tag = 17; // Fehlermeldung vom Compiler

      if (tag == Wochentag.SONNTAG)
         System.out.println("Ruhen.");
      switch(tag) {
      case SAMSTAG: case SONNTAG:
           System.out.println("Feiern!!");
           break;
      default: System.out.println("Arbeiten.");
      }

      // Durch alle Werte laufen
      System.out.println("Alle Wochentage:");
      for(Wochentag t: Wochentag.values())
         System.out.println(t);
   }
}
```