Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

7. Vorlesungswoche

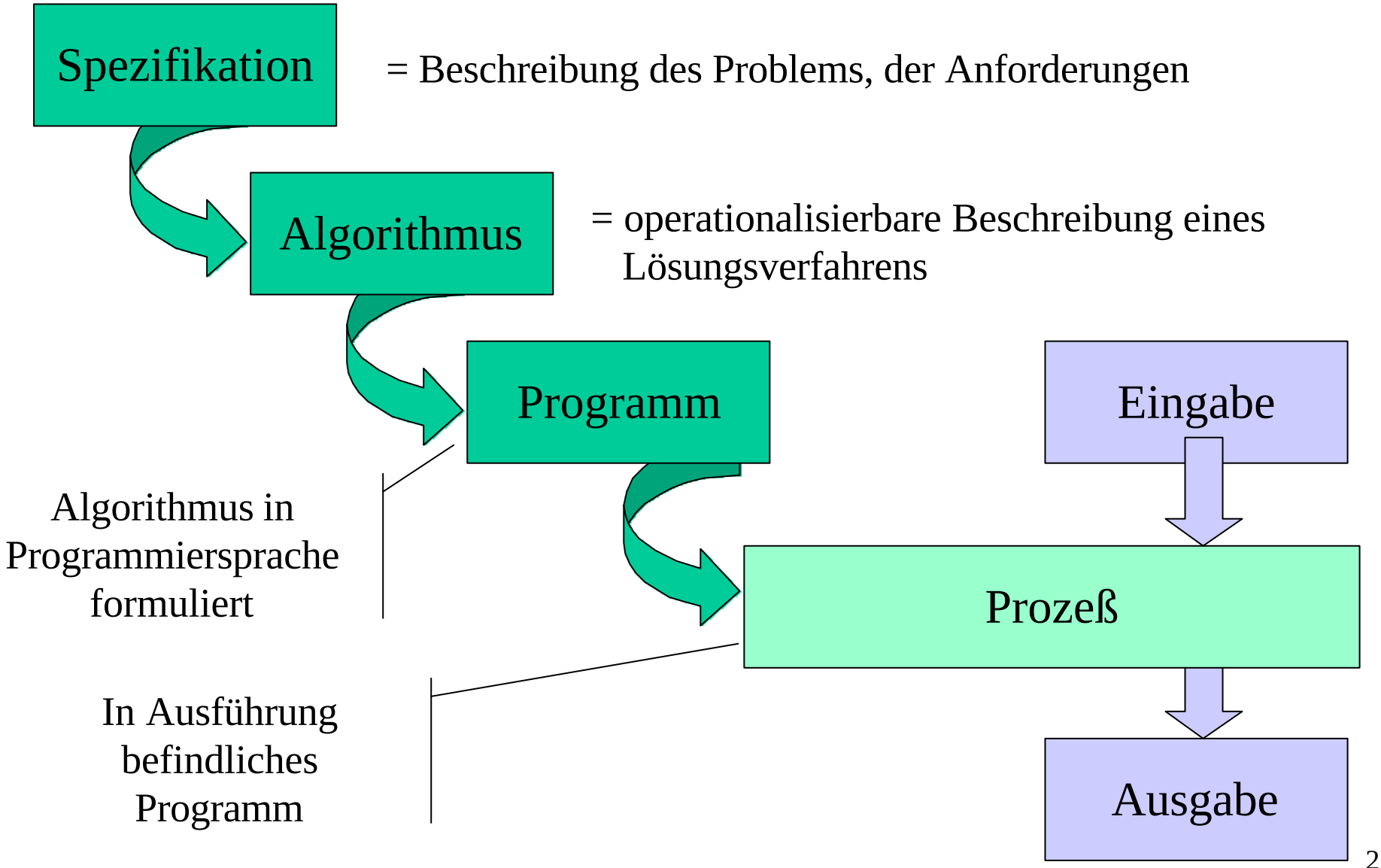
Objektorientierte Programmierung

Unterlagen:

Echtle, Goedicke; Einführung in die objektorientierte Programmierung mit Java, dpunkt-Verlag.

Doberkat, Dissmann; Einführung in die objektorientierte Programmierung mit Java, Oldenbourg-Verlag, 2. Auflage.

Stationen im Entwurf von Algorithmen und Programmen

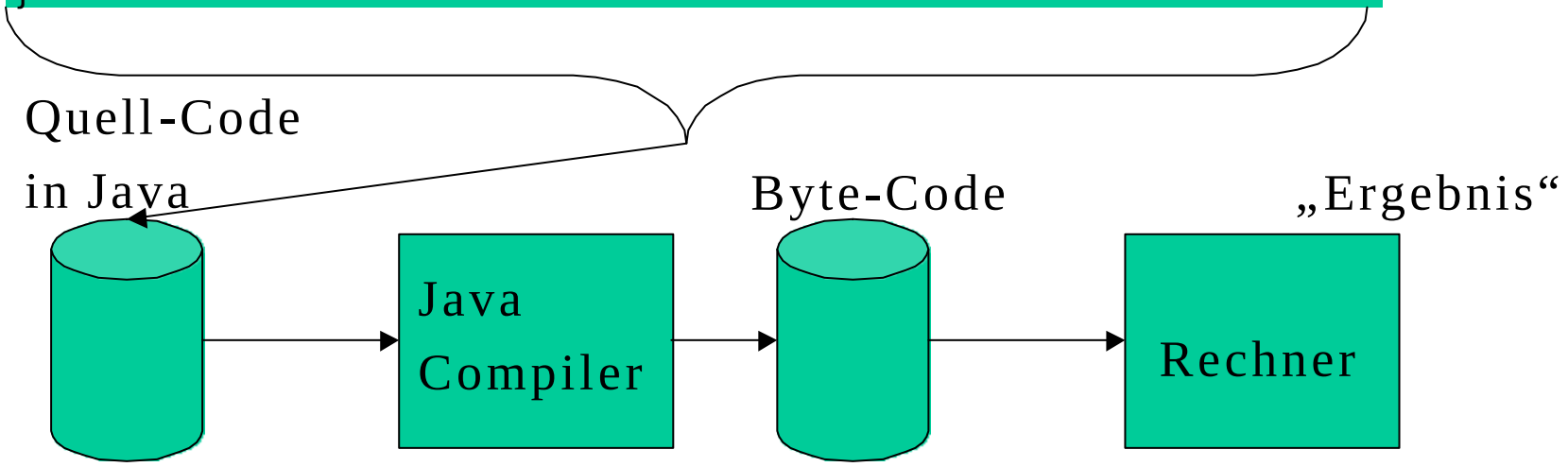


Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Ein Beispiel-Programm zeigt die wichtigsten Elemente

```
public class einfacheRechnung
{ public static void main (String [] Aufrufparameter)
  { int x, y;
    x = 10;
    y = 23 * 33 + 3 * 7 * (5 + 6);
    System.out.print (” Das Resultat lautet”);
    System.out.print (x + y);
    System.out.println (”.”);
  }
}
```



(©M. Goedicke, UGH Essen)

Der Aufbau eines Programms ...

Prog. 2–2 *Erläuterung des Programms 2–1*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

(©M. Goedicke, UGH Essen)

Die syntaktische Struktur von (Java-) Programmen ist durch Schachteln von Klammern gegeben

- Generell bestehen solche Schachteln aus einem Kopf und einem Rumpf

```
public class einfacheRechnung
{ ...
}

public static void main ( ... )
{ ...
}
```

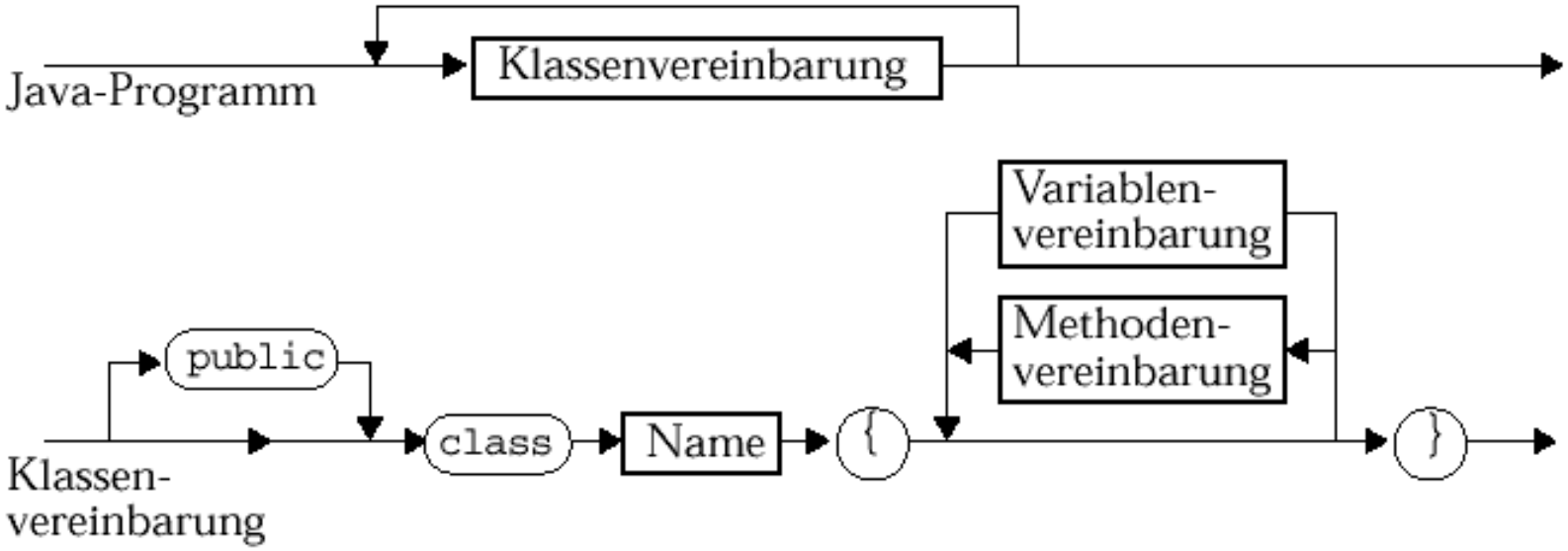
(©M. Goedicke, UGH Essen)

Java-Programme sind durch eine Folge von Klassendefinitionen gegeben

- Java-Programm ... Folge von Klassendefinitionen, die wiederum eine u.a. eine Folge von Methodendefinitionen enthalten
- Im Beispiel oben gab es
  - Die Definition einer Klasse `einfacheRechnung`
  - Innerhalb dieser Klasse die Definition einer Methode `main`
- `main` spielt eine besondere Rolle, da sie die Methode ist, die automatisch als erste aufgerufen wird, wenn man das Programm startet

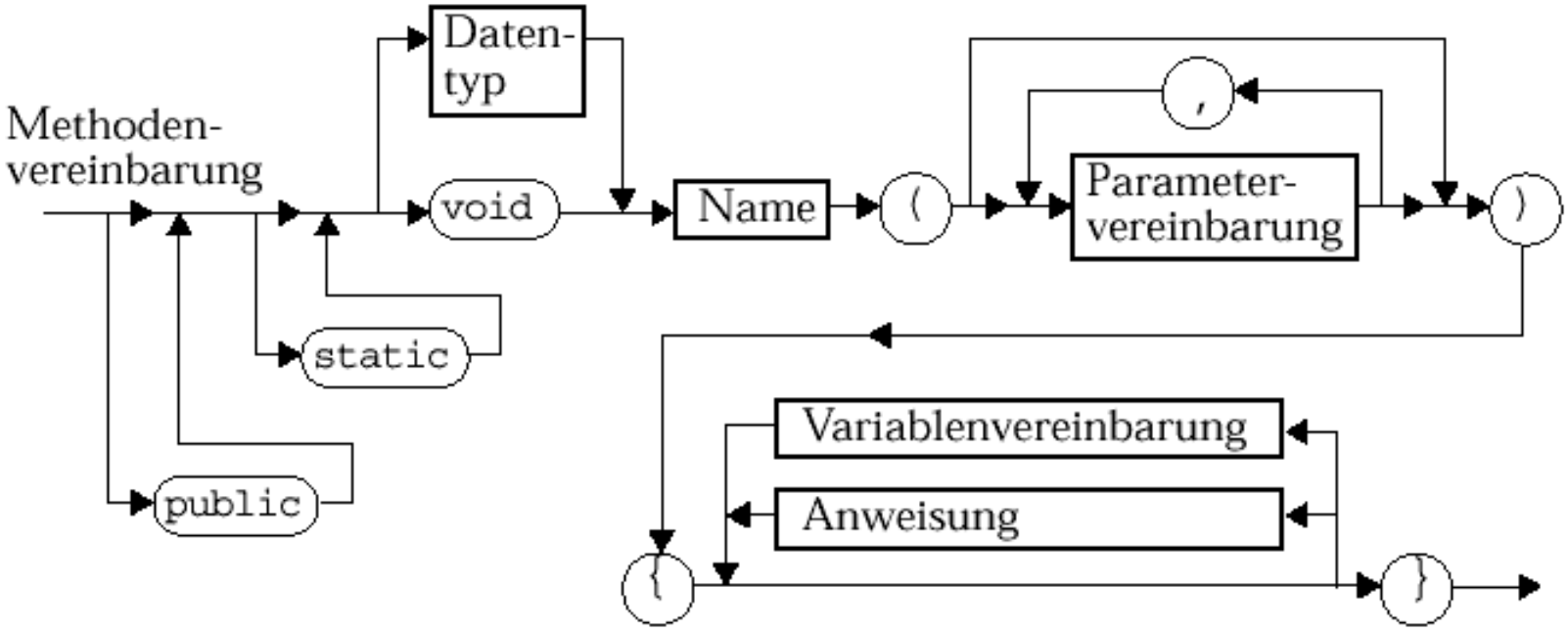
(©M. Goedicke, UGH Essen)

Syntax Diagramme für einfache Java-Programme (1)



(©M. Goedicke, UGH Essen)

Syntax Diagramme für einfache Java-Programme (2)



(©M. Goedicke, UGH Essen)

Elemente von höheren Programmiersprachen ...

- Einfache Anweisungen
- Zusammenfassung von mehreren Anweisungen zu einer Methode
  - auch Unterprogramm, Funktion, Prozedur, ...
- Einfache Datentypen
  - Einzelne Werte werden selten betrachtet; daher: ganze Wertemengen
  - Grundvorrat an einfachen (primitiven) Wertemengen vorgegeben:
    - ein Abschnitt der ganzen Zahlen
    - eine Teilmenge der rationalen Zahlen
    - einfache Zeichen und Zeichenketten
    - Wahrheitswerte (wahr, falsch)
  - Dazu werden die üblichen Operationen auf diesen Mengen zur Verfügung gestellt
    → Wertemengen + Operationen = Datentyp
- Primitive Wertemengen werden als Basis für komplexe Strukturen benutzt
- Vorschriften, wie einzelne einfache Daten zu komplexen Gebilden geformt werden
  → komplexe Datenstrukturen: Feld / Array, Struct / Record, ...

Höhere Programmiersprachen bieten benutzerdefinierte Typen

- C,C++,Pascal,Java, alle bieten Unterstützung für
  - Erzeugen mehrerer gleichartiger Variable: Felder / Arrays
  - Zusammenfassung von bestehenden Datentypen: Structs/ Records
    - z.B. zur Definition von Personendaten:
      Person =
      { Vorname, Nachname, Adresse, Geb-Datum}
      und darin enthaltenen Komponenten wie der Adresse als
      Adresse= { Straße, PLZ, Stadt}
      deren Komponenten letztendlich primitive Datentypen sein müssen

Was macht also Objektorientierung aus ?

- Objekte: Vereinigung von Algorithmus und Datenstrukturen
- Verallgemeinerung und Abstraktion: Klassen von Objekten

neue Begriffe in diesem Umfeld:

- Objekt, Instanz, Klasse, Methode, Vererbung, Interface, ...

Objekte

- Die Trennung von Verfahrenvorschrift (Algorithmus) und Datenstrukturen hat sich als hinderlich erwiesen.
- Daher: Besonders wichtig für das Verständnis von Programmen ist die Zusammenfassung der zulässigen Verfahren und der dazugehörigen Datenstrukturen
- Beispiel:
  Person = {Vorname, Nachname, Adresse,
  Geb-Datum}
  + {Ändere_Nachname (…), Ändere_Adresse(…)}
- Damit können auf einzelne Objekte zugeschnittene Verfahren formuliert werden, die spezifische Eigenschaften eines Objektes sicherstellen

(©M. Goedicke, UGH Essen)

Klassen von Objekten

- In der Regel ist es mühsam alle betrachteten Objekte einzeln zu beschreiben ....
  Man ist nicht an einer einzelnen Person, sondern an allen Personen interessiert
- Daher: Zusammenfassung aller gleichartigen Objekte zu einer Klasse von Objekten
- Üblicherweise werden dann in Programmen nur die Definition von Klassen und Verfahren angegeben, wie einzelne Exemplare (Objekte, Instanzen) geschaffen werden können
- Beispiel:
  class Person = {Vorname, ...} + {Ändere (...) }
  Person Erika_Mustermann

(©M. Goedicke, UGH Essen)

Klassen: Gleichartige Objekte

```
public class Girokonto {
  private int kontostand;     // in Pfennigen gerechnet

  public Girokonto() {
    kontostand = 0;
  }

  /* Einzahlen und Abheben */
  public void zahle (int pfennige){
    kontostand = kontostand + pfennige;
  }

  public int holeKontostand() {
    return (kontostand);
  }
}
```

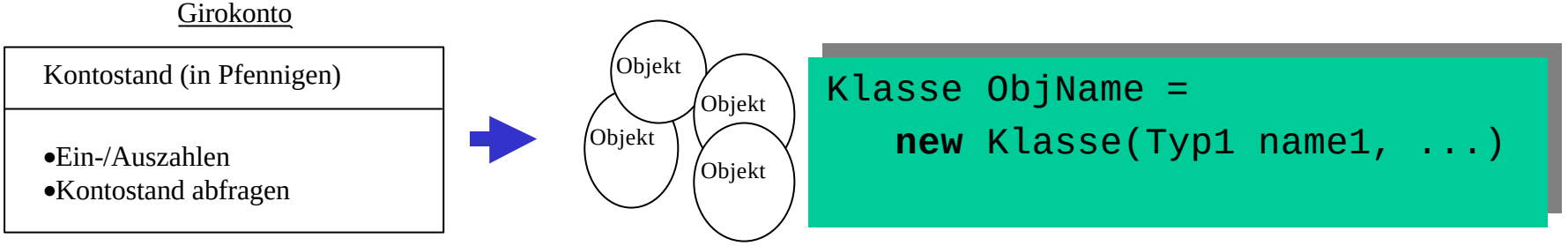
```
Modifier class
   Klassenname {
Attribut-
   Deklarationen
   Methoden
}
```

```
//    Kommentar
/*    Kommentar  */
```

(©V.Gruhn, Uni Do)

Von Klassen zu Objekten

- Exemplare aus einer Klassen-Schablone:



  - Erzeugen

```
Girokonto einKonto= new Girokonto();
```

  - Löschen
    - in Java: wenn Objekt nirgendwo mehr benutzt

      -> automatische Garbage-Collection
    - in anderen Sprachen auch explizit: delete / free
  - Zustand eines Objekts verändern (→ Attribute)
    Nachrichten an ein Objekt senden (→ Methoden)

(©V.Gruhn, Uni Do)

Attribute

- Teile des Gesamtzustands eines Objekts

```
private int kontostand;
private Girokonto meinKonto;
```

```
Modifier Datentyp attributname
```

- Verändern von Attributen

```
kontostand = kontostand + pfennige;
```

- Einkapseln von Attributen (→ Ziel: möglichst alle Attribute einkapseln)
  - privat d.h. nur innerhalb des Objekts sichtbar `(private)`
  - öffentlich d.h. auch außerhalb des Objekts sichtbar `(public)`
  - Zugriffe auf gekapselte Attribute nur durch einzelne Methoden des Objektes,
    - einfachste Variante: get / set Operationen holen / setzen Inhalte

Methoden (1/2)

- Teile des möglichen Verhaltens eines Objekts
  (d.h. Nachrichten, die vom Objekt verarbeitet werden können)

Spezieller Rückgabewert: **`void`**

```
public void zahle (int pfennige){
    kontostand = kontostand + pfennige;
}

public int holeKontostand() {
    return (kontostand);
}
```

```
Modifier Rückgabetyp Methodenname (Typ1 ParamName1,Typ2 Param...){
  Liste von Anweisungen;
  [ return (Rückgabewert) ]
}
```

(©V.Gruhn, Uni Do)

Methoden (2/2)

- Aufruf von Methoden eines Objektes
  - Punkt-Notation, qualifizierter Zugriff

```
konto.holeKontostand();
konto.zahle(+100);
```

- Übergeben von Parametern
  - statt komplette Objekte zu übergeben, nur Zeiger darauf („by reference“)
  - bei einfachen Datentypen als Wert kopiert („by value“)

- Einkapseln von Methoden
  - privat, d.h. nur als interne Hilfs-Methode zugreifbar (`private`)
  - öffentlich, d.h. extern sichtbar (`public`)

(©V.Gruhn, Uni Do)

Spezielle Methode "main"

- Einsprungpunkt bei "Ausführen" einer Applikation
  - Übergeben von Argumenten aus der Kommandozeile möglich
  - für Testen von Klassen-Implementierungen geeignet

```
public class TestGirokonto {
     ...
     public static void main(String[] args) {
        ... // Hier Code zum Testen einfügen
     }
     ...
}
```

(©V.Gruhn, Uni Do)

Erweiterung des Beispiels: Klasse Girokonto

- Motivation
- Klasse erweitern um Attribute, Methoden
- Erzeugen von Objekten und Nachrichtenaustausch (Methoden) zwischen Objekten

- Aufgabe:
- Ergänzung der Funktionalität der Klasse "Girokonto„:
  - zum Sperren eines Kontos: boolesches Attribut `istGesperrt` mit den Methoden `void sperre(), void entsperre()` und `boolean istGesperrt()`
  - zum Überschreiben des aktuellen Kontostandes mit einem übergebenen Betrag die Methode `setzeKontostand()`
- Testen der Klasse "Girokonto" (siehe: Methode "main" in der Klasse "TestGirokonto")
  - Erzeugen eines Girokonto-Objekts
  - Nacheinander Einzahlen von 100 Pfennigen, Abheben von 20 Pfennigen und Einzahlen von 30 Pfennigen
  - Anschließend Kontostand ausgeben

Erweiterungen

```
public class Girokonto {
   ...
   private boolean istGesperrt;
    ...
   public void sperre() {
        istGesperrt=true;
   }

   public void entsperre(){
        istGesperrt=false;
   }

   public boolean istGesperrt (){
        return (istGesperrt);
   }
   ...
}
```

(©V.Gruhn, Uni Do)

Testklasse:

Einsprung von der Kommandozeile
[optional für jede Klasse]

```
public class TestGirokonto {
  public static void main(String[] args) {

    // erzeuge neues Konto
    Girokonto konto = new Girokonto();
    // führe Transaktionen durch
    konto.zahle(+100);
    konto.zahle(+30);
    konto.zahle(-20);
    // gib Kontostand aus
    int aktuellerStand = konto.holeKontostand();
    Bildschirm.gibAus(aktuellerStand);

  }
}
```

Klasse „Bildschirm“

(©V.Gruhn, Uni Do)

Hilfsklasse „Bildschirm“

- **Benutzereingaben und Bildschirmausgaben ganz abstrakt aus Anwendersicht: "Welche Methoden gibt es?"**

– **komfortable Ausgabe auf den Bildschirm**

```
Bildschirm.gibAus (text)
```

– **komfortable Eingabe eines Textes von Tastatur**

```
eingabetext = Bildschirm.gibEin()
```

– **Warten auf Tastendruck**

```
Bildschirm.warteAufTastendruck()
```

(©V.Gruhn, Uni Do)

Zwischenstand: Objektorientierte Programmierung

- Programm = Mehrere Objekte, die Nachrichten austauschen
- Klassen: Schablonen für Objekte
  - Attribute
  - Methoden
- Objekte
  - Erzeugen
  - Zerstören
- Nachrichtenaustausch
  - Aufruf von Methoden eines Objektes (Punkt-Notation)
    z.B. `konto.holeKontostand()`

- Was ist anders als an bisheriger, imperativer Programmierung ?
  - Objekte: Attribute & Methoden zusammen
    - vorher in Structs/Records mit Zeigern auf Funktionen auch möglich, aber unüblich
  - Klassen
    - vorher als Module, in denen Funktionen für bestimmte Datenstrukturen gesammelt wurden,

    -> möglich

  Wesentlich ist eine andere (objektorientierte) Sicht auf eine Aufgabenstellung, die durch Sprachkonstrukte unterstützt wird.

Beispiel: Girokontenverwaltung

- Imperative Programmierung:
  - verwalte k Girokonten in Feld mit Struct {Personendaten, Kontostand}
  - definiere Funktionen zum Einfügen, Ändern, Löschen von Girokonten
  - Ändern beinhalte die Ausgabe des Kontostandes & Ein/Auszahlungen
  - Funktionen haben typischerweise Kontonummer als Identifier
- Objektorientierte Programmierung:
  - Klasse Girokonto mit Funktionen Erzeugen, Get/Set, Freigeben
  - Klasse Girokontenverwaltung mit Datenstruktur zur Verwaltung einer Menge von k Elementen mit Einfügen, Ändern, Löschen

Was fällt auf ?

1. Lokalisierung, Kapselung von Daten und Methoden auf einen kleinen Bereich, den sie auch betreffen
2. stärkere Abstraktion: Kontenverwaltung verwaltet Elemente unabhängig von deren internen Besonderheiten (Sparbuch,Girokonto, ...)
3. Bei Methodenaufrufen werden nur Referenzen auf Objekte übergeben, stärkt die Vorstellung von „Dingen“, die als solches existieren und ein Eigenleben führen.

im folgenden etwas genauer: Aufruf von Objekten

- Konstruktoren: Aktivitäten bei Objekterzeugung
- Referenzen als Verweise auf Objekte
- Garbage Collection
- Übergabe von Parametern an Methoden
- Klassenattribute / Klassenmethoden
- Namensraum
  - Überladen von Methoden
  - Überdecken von Attributen
- Vertiefung OO-Programm = Mehrere Objekte + Nachrichtenaustausch
  - Aufgaben verteilen („Spezialisten“)

Konstruktor

- Methode, die automatisch bei Erzeugung eines Objektes aufgerufen wird
- Wird in der Regel benutzt, um Attribute zu initialisieren
- Dadurch charakterisiert, daß der Methodenname mit dem Klassennamen übereinstimmt
- Konstruktoren besitzen keinen Rückgabewert (auch nicht `void`)
- Überladen von Konstruktoren ist möglich
- Methoden für das Durchführen von Arbeiten direkt bei der Objekt-Erzeugung:
  - Methodenname = Klassenname
  - Folge von Anweisungen, die abzuarbeiten sind
  - kein Rückgabewert

```
class Girokonto {
   ...
   public Girokonto() {
        kontostand = 0;
        ...
   }
}
```

(©V.Gruhn, Uni Do)

Konstruktor

- Argumente der Konstruktormethode werden bei Objekterzeugung durch **`new`** *`Klassenname`*`(`*`Argument1, Argument2, ...`*`)` an den Konstruktor übergeben,
- Anweisungen im Konstruktor werden auf neu allokiertem Speicher ausgeführt
- Ohne Angabe eines Konstruktors ist nur der leere Konstruktor ohne Argumente definiert
- Ist ein (nicht leerer) Konstruktor definiert, muß der leere Konstruktor explizit definiert werden, um noch benutzt werden zu dürfen

Referenzen auf Objekte

- Bei primitiven Datentypen enthält eine Variable direkt den Inhalt (z. B. einen Int-Zahlenwert)
- Bei Objekten von Klassen symbolisiert die Variable nur eine Referenz (einen Verweis) auf das Objekt
- Es können auch mehrere Referenzen auf ein Objekt zeigen
- Ein Referenz kann auch leer sein, `null` ist das Schlüsselwort für die leere Referenz
- Beispiel: `Girokonto k = null;`

Referenzen auf Objekte

- Mit dem == Operator kann man zwei Referenzen auf Gleichheit testen:

  Beispiel:

```
Girokonto k1 = new Girokonto();
...
Girokonto k2 = k1;
...
if (k1 == k2)
   System.out.println( ”k1,k2 verweisen auf
                                        dasselbe Objekt");
```

- Mit dem == Operator kann man ebenso abfragen, ob eine Referenz leer ist:

  Beispiel:

```
Girokonto k;
...
if (k == null)
   System.out.println(”k ist leere Referenz");
```

(©V.Gruhn, Uni Do)

Wie werden Objekte freigegeben / zerstört ?

- In vielen Sprachen durch explizite Statements im Programm
  - `free`- oder `delete`-Methoden

  Funktioniert, ist aber häufige Fehlerquelle
  - x,y verweisen auf 1 Objekt,
  - free(x) gibt Speicher frei
  - y wird zu späterem Zeitpunkt verwendet
  - Problem: Speicheradresse existiert, Speicherinhalt kann beliebig sein

Alternative in Java: Garbage Collection

- Ist die letzte Referenz auf ein Objekt verschwunden da
  - die Referenzen den Gültigkeitsbereich verlassen haben, oder da
  - `null`-Referenzen zugeweisen wurden

  wird das Objekt der Garbage Collection zugeführt

  => es sind keine `free`- oder `delete`-Methoden nötig
- Funktioniert auch bei zyklischen Referenzen der Objekte untereinander
- Mit `System.gc();` wird eine Empfehlung an die Garbage Collection ausgegeben, aktiv zu werden, in der Regel ist dies aber nicht nötig

Klassenattribute und -methoden

Attribute und Methoden gehören zu Objekten (also Instanzen) einer Klasse

Aber: es gibt Klassenattribute und -methoden, die nicht zu Instanzen (also Objekten) gehören, sondern zu den **Klassen** selbst

Klassenvariablen

- Attribute, die für jedes Objekt neue Instanzen bilden, heißen Instanzvariablen (Standardfall)
- Werden Attribute mit `static` gekennzeichnet, handelt es sich um **Klassenvariablen**, die für die gesamte Klasse nur eine Instanz besitzen
- Klassenvariablen existieren auch ohne die Existenz eines Objektes. Auf sie kann durch *Klassenname*`.`*Attributname* zugegriffen werden.

Klassenmethoden

- Methoden, die ausschließlich auf Klassenvariablen zurückgreifen, dürfen mit `static` gekennzeichnet werden, sie heißen **Klassenmethoden**. Klassenmethoden dürfen auch nur Klassenmethoden benutzen
- Klassenmethoden können auch ohne Existenz eines Objektes mit *Klassenname.Methodenname*`(...)` aufgerufen werden.
- Die `main()`-Methode ist eine Klassenmethode, da zu Beginn noch keine Objekte erzeugt wurden

Beispiel

```
class Demo {
   int a;
   static int b;

   public static void main
                  (String[] args) {
      a = 1;        // nicht erlaubt
      b = 1;        // erlaubt
      test();       // nicht erlaubt
      new Demo().test(); // erlaubt
      test2();           // erlaubt
      new Demo().test2();// erlaubt
   }

   void test() {
      a = 2;        // erlaubt
      b = 2;        // erlaubt
   }

   static void test2() {
      b = 3;
   }
}
```

```
...
Demo d = new Demo();
d.a = 1;          // erlaubt
d.b = 1;          // erlaubt
Demo.a = 1;       // nicht erlaubt
Demo.b = 1;       // erlaubt
Demo.test();      // nicht erlaubt
Demo.test2();     // erlaubt
...
```

Überladen von Methoden

- Wunsch für Methoden/Attribute: Gleiche Namen für gleiche Zwecke
  - z.B. Girokonto erzeugen ohne/mit einem initialem Kontostand

- Benutzt werden sollen Methoden mit gleichen Namen
- Problem: Wie diese Methoden auseinanderhalten?
- Idee: unterscheide anhand Parameter-Typen:
  Methoden gleichen Namens müssen sich also im Typ von mindestens einem Parameter oder in der Anzahl der Parameter unterscheiden

```
zahle (int betrag)
zahle (int betrag, String verwendungszweck)
```

Überladen von Methoden

- Unterschiedliche Methoden müssen sich im Methodennamen oder in der Übergabeparameterliste (oder in beidem) unterscheiden
- Hat eine Klasse mehrere Methoden mit identischem Namen nennt man diese Methode überladen
- In unterschiedlichen Klassen dürfen auch Methoden mit identischem Namen und identischen Übergabeparameterlisten deklariert werden

Überdecken von Attributen

Verwenden von Variablen mit bereits benutzten Namen

Zugreifen auf überdeckte Attribute über `this`

```
public class Girokonto {
   private int kontostand;    // in Pfennigen gerechnet
   ...

   public void setzeKontostand(int kontostand) {
      this.kontostand = kontostand;
   }

   ...
}
```

- Variablendeklarationen in Klassenmethoden überdecken die Attribute der Klasse
- Die Attribute sind nur überdeckt, nicht überschrieben
- Auf Attribute der Klasse kann dann über das Schlüsselwort `this` zugegriffen werden
- `this` ist eine Referenz auf das zur Methode zugehörende Objekt

Beispiel: Überladen von Konstruktoren, Methoden

- Erweitern einer Klasse Girokonto um Attribute und Methoden
- Ziel: Übersicht behalten durch sinnvolles, sparsames Vergeben von Methodennamen (Namen mehrfach in einer Klasse vergeben)

Anforderung:

- Implementieren einer zweiten Konstruktor-Methode für die Klasse „Girokonto“. Sie soll es ermöglichen, dass schon bei der Erzeugung eines Girokonto-Exemplars ein Wert für den initialen Kontostand übergeben werden kann. Welchen Namen muss diese Methode tragen? Der übergebene Parameter soll `kontostand` heißen. Es gibt in der Klasse "Girokonto" bereits ein Attribut mit diesem Namen.

Beispiel: Überladen einer Konstruktormethode für „Girokonto“

- Lösungsmuster:
  - Konstruktor-Methode -> Name identisch mit Klassenname
  - Überdecken des Namens `kontostand` (als Attributname und Variablenname)

```
public class Girokonto {
     …
     public Girokonto(int kontostand) {
        this.kontostand = kontostand;
     }
     …
}
```

Zusammenspiel zwischen Objekten

- In OO-System kooperieren Objekte, um gemeinsam eine Aufgabe zu lösen
- Jede Klasse kann etwas besonders gut (Spezialist)
  - z.B. Klasse "Girokonto" für Umgang mit Girokonten
- Beim Erstellen eines OO-Systems müssen Spezialisten ausgemacht werden
  - z.B. Erweiterung um Währungsrechnen

Am Beispiel des Girokontos:

- Weg von Anforderungstext zu einer neuen Klasse
- Ändern einer bestehenden Klasse
- Aufgabe:
- Entwerfen einer neue Klasse „Waehrungsrechner“ für das Umrechnen von EuroCent in Pfennige (100 EuroCent ≈ 200 Pfennige)
  - Muss diese Klasse Attribute besitzen? Welche?
  - Erforderlich Methode "int wandeleInPfennig (int eurocent)" und "int wandeleInEuroCent (int pfennige)“.
- Anpassen der Klasse "Girokonto" :
  - Girokonto-Objekt erhält bei Erzeugung "individuellen" Währungsrechner
  - Kontostand sei intern stets in EuroCent gespeichert, Wert in jeder andere Währung wird über Konvertierung berechnet, Speicherungsart sei geheim („Information-Hiding“).

Beispiel: Zusammenspiel von Objekten

```
public class Waehrungsrechner {
  private int euroFaktor;

  public Waehrungsrechner(){
     euroFaktor = 2;
  }
  public int wandeleInEuroCent (int pfennige){
    int euroCent = pfennige / euroFaktor;
    return (euroCent);
  }
  public int wandeleInPfennig (int euroCent){
   int pfennige = euroCent * euroFaktor;
   return (pfennige);
  }
}
```

→

Beispiel: Änderungen in „Girokonto“

```
public class Girokonto {
  private int kontostand = 0;
  private Waehrungsrechner wr;

  public Girokonto(Waehrungsrechner wr) {
    this.wr = wr;
  }

  public Girokonto(Waehrungsrechner wr, int kontostand){
    this.wr = wr;
    // 1. Umrechnen des Betrags in EuroCent
    int euroCent = wr.wandeleInEuroCent(kontostand);
    // 2. Konstand aktualisieren
    this.kontostand = euroCent;
  }
```

→

Beispiel: Änderungen in „Girokonto“

```
public void zahle (int pfennige){
  // 1. Umrechnen des zahlbetrags in EuroCent
  int euroCent = wr.wandeleInEuroCent(pfennige);

  // 2. Konto aktualisieren
  kontostand = kontostand + euroCent;
}

public int holeKontostand(){
  // 1. Umrechnen des aktuellen Kontostandes in Pfennige
  int pfennige = wr.wandeleInPfennig(kontostand);

  // 2. Kontostand in Pfennigen zurückgeben
  return (pfennige);
}
```

Zwischenstand

- Objektorientierte Programmierung
  - Programm = Mehrere Objekte, die Nachrichten austauschen
  - Klassen: Schablonen für Objekte (Attribute & Methoden)
  - Objekt:
    - Lebenszyklus in Java: Konstruktur, Methodenaufrufe / Nachrichtenübermittlung, Garbage Collection
    - Lebenszyklus in C++: Konstruktur sofern vorhanden, Methodenaufrufe, Destruktor, delete
  - starke Lokalisierung der Betrachtung auf einzelne „Dinge“, deren Zustand (Attribute) und Zustandsänderungen (Methoden),
    - erinnert an endliche Automaten
    - „Kommunikation“ zwischen Objekten durch Nachrichtenaustausch im Sinne von Methodenaufrufen
- Einzelheiten zu: Konstruktoren, Referenzen, Garbage Collection, Übergabe von Parametern an Methoden, Klassenattribute / Klassenmethoden
- Namensraum
  - Überladen von Methoden
  - Überdecken von Attributen
- Vertiefung OO-Programm=Mehrere Objekte + Nachrichtenaustausch
  - Aufgaben verteilen („Spezialisten“)
  - Klasse „Waehrungsrechner“

Klassen: Nicht-triviale Beispiele

- String
- StringBuffer
- Object

Klasse `String`

- Zeichenketten werden durch Objekte der Klasse **`String`** dargestellt.
- **`String`** ist kein primitiver Typ, sondern eine vordefinierte Klasse.
- **`String`** ist eine eigene Klasse, da Literale (konstante Zeichenketten) und Operatoren (+ zur Konkatenation) definiert sind.
- In enger Verbindung mit der Klasse **`String`** steht die Klasse **`StringBuffer.`**

Klasse `String`

Warum zwei String Klassen?

The Java development environment provides two classes that store and manipulate character data:

- *String, for constant strings, and StringBuffer, for strings that can change.*
- *You use Strings when you don't want the value of the string to change. For example, if you write a method that requires string data and the method is not going to modify the string in any way, use a String object. Typically, you'll want to use Strings to pass character data into methods and return character data from methods.*
- *Because they are constants, Strings are typically cheaper than StringBuffers and they can be shared.*

So it's important to use Strings when they're appropriate.“

(aus dem Java Tutorial)

Methoden der Klasse `String` (Ausschnitt)

Konstruktoren:

- **`public String(char chars[])`**
  - Konvertiert Array von Zeichen in einen String
- **`public String(byte bytes[])`**
  - Konvertiert Array von Bytes in einen String
  - Plattformspezifische Zeicheninterpretation wird benutzt
- **`public String(StringBuffer buffer)`**
  - Konvertiert StringBuffer nach String

- Beispiel:
- **`char[] c = {'S', 'c', 'h', 'u', 'l', 'z', 'e'};`**
- **`String s = new String(c);`**
- **`System.out.println(s);`**

String-Objekte

- Objekte für den Umgang mit Zeichenketten
- → Aufruf von Methoden für Erzeugen, Aneinanderhängen, Längenmessung, Vergleich, ...
- Erzeugen
  - **`String vorname= new String ("Max")`**
  - **`String nachname= "Mustermann"`** (Kurz-Form mit Literal)
- Messen der Länge

  `int laenge= name.length()` **® 14**
- Aneinanderhängen (Konkatenation)

  `String name = vorname + nachname` (neues String-Objekt)

String-Konkatenation

- Der +-Operator kann verwendet werden, um zwei Objekte der Klasse String zu verknüpfen
- Ist nur einer der Operanden ein Objekt der Klasse String, und ein anderer ein primitiver Datentyp, wird letzterer nach String gewandelt

Beispiel:

```
int alter = 27;
System.out.println("Alter :  " + alter + " Jahre");
```

Ausgabe:

```
Alter: 27 Jahre
```

String-Objekte

Vergleichen (mit einem anderem Objekt):

`public boolean equals(Object anObject)`

- Vergleicht String mit einem anderen Object auf Gleichheit
- Typ Object heißt, daß Objekte aller Klassen als Argument übergeben werden dürfen (später mehr)

.

```
String vorname2= "Max"
boolean istGleich = vorname.equals(vorname2)        ® true
```

Operator "==" vergleicht nur, ob die Referenz identisch ist, also ob es sich um dasselbe Objekt handelt, und nicht, ob es inhaltlich identisch ist.

also: == vergleicht Referenzen beider Seiten miteinander!

```
boolean istGleicheReferenz = (vorname == vorname2)
                                                    ® false
```

Methoden der Klasse `String` (Ausschnitt)

Beispiel:

```
• String s1 = "Test", s2=s1;
• String s3="Te", s4="st", s5;
• s5 = s3 + s4;
• System.out.println((s1 == s2)+"/"+s1.equals(s2));
• System.out.println((s1 == s3)+"/"+s1.equals(s3));
• System.out.println((s1 == s5)+"/"+s1.equals(s5));
```

Ausgabe:

```
• true / true
• false / false
• false / true
```

Methoden der Klasse `String` (Ausschnitt)

- **`public int length()`**
  - Gibt Länge des Strings zurück
- **`public byte[] getBytes()`**
  - Konvertiert String in Array von Bytes
  - Plattformspezifische Zeicheninterpretation wird benutzt
- **`public int compareTo(String anotherString)`**
  - vergleicht zwei Zeichenketten lexikographisch basierend auf Unicode-Nummern (Telefonbuchordnung)
  - Ergebnis ist gleich 0, falls Strings identisch sind,
  - kleiner 0, falls der String lexikographisch kleiner ist, als das Argument,
  - größer 0, falls der String lexikographisch größer ist, als das Argument

Methoden der Klasse `String` (Ausschnitt)

- **`public String toLowerCase()`**
- **`public String toUpperCase()`**
  - Wandelt String in Kleinbuchstaben bzw. Großbuchstaben
  - Funktioniert auch mit deutschen Umlauten
  - "ß" wird bei `toUpperCase` nach "SS" gewandelt, Rückwandlung funktioniert hier natürlich nicht

Beispiel:

- **`String a = "Schoßhündchen"`**
- **`System.out.println(a.toLowerCase());`**
  - Ausgabe: **`schoßhündchen`**
- **`System.out.println(a.toUpperCase());`**
  - Ausgabe: **`SCHOSSHÜNDCHEN`**
- **`System.out.println(a.toUpperCase().toLowerCase());`**
  - Ausgabe: **`schosshündchen`**

`String` und `StringBuffer`

- **`String`** ist nur geeignet, um nicht änderbare Zeichenketten zu verarbeiten, denn z.B. bei Stringkonkatenation durch "+" wird ein neues Objekt erzeugt und die kompletten alten Inhalte kopiert.
- Müssen in eine Zeichenkette laufend kleine Teilzeichenketten angefügt werden, so ist **`StringBuffer`** besser geeignet

Methoden der Klasse `StringBuffer` (Ausschnitt)

- **`StringBuffer`** hat ein initiales Fassungsvermögen
- Wird mehr benötigt, wird es automatisch erweitert
- Bei anfänglicher realistischer Voreinschätzung der benötigten Länge läßt sich die Effizienz steigern
- Konstruktoren:
- **`public StringBuffer()`**
  - Erzeugt Leerstring mit initialem Fassungsvermögen von 16 Zeichen
- **`public StringBuffer(int length)`**
  - Erzeugt Leerstring mit spezifiziertem initialen Fassungsvermögen
- **`public StringBuffer(String str)`**
  - Initialisiert **`StringBuffer`** mit vorgegebenem **`String`**.

Methoden der Klasse `StringBuffer` (Ausschnitt)

- **`public StringBuffer append(String str)`**
  - Hängt eine Zeichenkette an den String an
  - Fassungsvermögen wird dynamisch an die benötigten Gegebenheiten angepaßt
- **`public int length()`**
  - Gibt die Länge der aktuellen Zeichenkette zurück (nicht des aktuellen Fassungsvermögens)
- **`public String toString()`**
  - Wandlung in eine Zeichenkette vom Typ **`String`**

Einfügen von Zeichen in StringBuffer-Objekte:

```
StringBuffer sb = new StringBuffer("Drink Java!");
sb.insert(6, "Hot ");
System.out.println(sb.toString());
```

Ausgabe für Beispiel: Drink Hot Java!

Ersetzen von Zeichen an bestimmten Positionen:

- **`setCharAt`** ersetzt ein Zeichen an einer bestimmten Position innerhalb des StringBuffer-Objektes.

Methoden der Klasse `StringBuffer` (Ausschnitt)

Anhängen am Ende eines StringBuffer-Objekts:

```
StringBuffer sb = new StringBuffer("Drink Hot");
sb.append("!");
System.out.println(sb.toString());
```

Ausgabe: Drink Hot!

Reversieren eines StringBuffer-Objektes:

```
class ReverseString {
    public static String reverseIt(String source) {
      int i, len = source.length();
      StringBuffer dest = new StringBuffer(len);
      for (i = (len - 1); i >= 0; i--) {
                dest.append(source.charAt(i));
      }
      return dest.toString();
    }
}
```

Methoden der Klasse `StringBuffer` (Ausschnitt)

Erläuterung zum Reversieren:

- Die Anweisung **`StringBuffer dest`** legt fest, daß ein StringBuffer-Objekt verwendet werden soll.
- Hierfür wird Speicherplatz allokiert.
- **`StringBuffer(len)`** initialisiert das Objekt.
- Durch das Anhängen eines Zeichens an ein StringBuffer-Objekt kann prinzipiell neue Speicherallokation nötig sein. Dies ist teuer. Deshalb wird im Reversieren ein passend langes StringBuffer-Objekt auf einmal allokiert.

Objektorientierte Programmierung

zwei Beispiele für nicht-triviale Klassen: String, StringBuffer

Was fällt auf ?

- in der Benutzung:
  - Unterscheidung muss vom Programmierer berücksichtigt werden
  - Funktionalität erlaubt automatische Prüfung durch Compiler/Interpreter, dass String Objekte nicht verändert werden
  - Anwendungsbereich für String
    - Übergabe von Zeichenketten an andere Objekte, wobei die Zeichenkette nicht verändert werden darf
    - Ein/Ausgabe, geteilte Nutzung fester Zeichenketten
    - durch Konkatenation und Objekterzeugung gewisse Flexibilität
  - Anwendungsbereich für StringBuffer
    - Zeichenketten, die in signifikantem Umfang / Häufigkeit manipuliert werden
- in der Implementierung:
  - unbekannt, Vermutung: Klassen intern sicherlich unterschiedlich implementiert
  - String: Effizienzgewinn durch feste Größe
  - StringBuffer: reichhaltigere Funktionalität durch variable Länge
- Vgl: „C“: Zeichenketten sind als Zeiger/Referenz auf Character Arrays realisiert, deren Länge durch „\n“ angezeigt wird.
  - welche Fehlermöglichkeiten bestehen dort ???

zum Abschluss noch eine allg. Klasse: `Object`

- Die Klasse **`Object`** ist die Oberklasse aller Klassen, und stellt einige Methoden zur Verfügung:
- **`public String toString()`**
  - Die **`toString()`**-Methode gibt eine (etwas kryptische) Wandlung eines Objektes in einen String, sie sollte von Klassen überdeckt werden
  - Die **`System.out.println`**-Methode akzeptiert beliebige Objekte als argument, sie benutzt die (eventuell selbst überschriebene) **`toString()`**-Methode
  - Strings können mit beliebigen Objekten durch den `+`-Operator verknüpft werden. Dazu wird implizit von den Objekten die **`toString()`**-Methode aufgerufen, um die String-Darstellung zu erhalten

Die Klasse `Object`

- **`public boolean equals(Object obj)`**
  - Das Objekt wird mit dem Argument auf Gleichheit überprüft
  - In der Standardimplementierung wird nur die Referenz verglichen (also, ob es sich im dasselbe Objekt handelt)
  - Eine neue Methode kann die `equals`-Methode überdecken, um einen inhaltlichen Vergleich zu ermöglichen
  - Beispiel: Für einen Kunden soll abgefragt werden, ob er bereits Kunde ist. Für den Vergleich spielt der Name, der Geburtsort und das Geburtsdatum eine Rolle, nicht jedoch die Adresse.
- **`public Object clone()`**
  - returniert eine wertegleiche Kopie des Objektes.
  - Hinweis: Wenn in dem zu klonenden Objekt Referenzen auf andere Objekte (z.B. Arrays) vorkommen, so werden die Referenzen geklont, nicht aber die referenzierten Objekte. Auf diese Weise kommt es zu “shared data”, geteilten Objekten.

    Ob das jeweils beabsichtigt ist, muss sorgfältig geprüft werden.
- **`public final Class getclass()`**
  - returniert das Objekt, das die Klasse des Objektes repräsentiert, dessen Methode aufgerufen wird.

Zusammenfassung

- Objektorientierte Programmierung
  - Programm = Mehrere Objekte, die Nachrichten austauschen
  - Klassen: Schablonen für Objekte (Attribute & Methoden)
  - Objekt:
    - Lebenszyklus in Java: Konstruktur, Methodenaufrufe / Nachrichtenübermittlung, Garbage Collection
    - Lebenszyklus in C++: Konstruktur sofern vorhanden, Methodenaufrufe, Destruktor, delete
  - starke Lokalisierung der Betrachtung auf einzelne „Dinge“, deren Zustand (Attribute) und Zustandsänderungen (Methoden),
    - erinnert an endliche Automaten
    - Kapselung, kontrollierte Zugriffsmöglichkeiten, „Kommunikation“ zwischen Objekten durch Nachrichtenaustausch im Sinne von Methodenaufrufen
- Einzelheiten zu: Konstruktoren, Referenzen, Garbage Collection, Übergabe von Parametern an Methoden, Klassenattribute / Klassenmethoden
- Namensraum: Überladen von Methoden, Überdecken von Attributen
- OO-Programm=Mehrere Objekte + Nachrichtenaustausch
  - Aufgaben verteilen („Spezialisten“)
- Konkrete Klassen: String, StringBuffer, Object
  - String, StringBuffer: Versuch per Struktur Fehlerquellen zu vermeiden, z.B. konstante Strings können mangels Zugriffsmöglichkeiten nicht verändert werden.