Praktische Informatik 1

Imperative Programmierung und Objektorientierung

Karsten Hölscher und Jan Peleska

Wintersemester 2011/2012

Universität Bremen

Was bisher geschah ...

- Konzepte:
  - Klassen und ihre ...
  - ... Instanzen, die Objekte
  - Methoden und ihre Parameter
- Eine erste Programmierübung in Java: Ausfüllen von Methoden mit ...
  - Zuweisungen
  - if-else-Anweisungen
  - for-Schleifen
  - return-Anweisung
  - Operatoren +,-,<<,>>,>>> ...

Was bisher geschah ...

- Tutorieneinschreibe-Chaos
  - Sorry!
- Unklarheit bzgl. Aufgabenblatt 1
  - nochmal Sorry!

Was nun geschieht ...

- ABER alles wird besser
  - Abgabe Blatt 1
    - Donnerstag, 17. November 2011
  - Abgaben generell
    - Ausgabe Dienstag
    - Abgabe Donnerstag der Folgewoche

Welche Systematik steckt dahinter - Aufbau von Programmiersprachen

- **Strukturmechanismen**: Wie kann “sehr viel Code” in verständliche Komponenten zerlegt werden ?
- **Primitive Datentypen:** Welche Basis-Datentypen stellt die Programmiersprache zur Verfügung, um Variablen dieser Typen einzuführen?
- **Typerzeugungsmechanismen**: Wie kann man aus vorhandenen Datentypen neue, komplexere, bilden?
- **Deklaration und dynamische Erzeugung/Vernichtung**: Wie können Variablen eingeführt werden -- wie können Objekte zur Laufzeit dynamisch neu erzeugt oder vernichtet werden?

Aufbau von Programmiersprachen

- **Kausalordnung von Anweisungen:** Wie kann beschrieben werden, ob Anweisungen nacheinander oder nebenläufig (“parallel zu einander”) ausgeführt werden ?
- **Zuweisungen** (“Zuweisungs-Anweisung”): Wie lassen sich Werte zwischen Variablen übertragen?
  - Zuweisungen sind typisch für imperative Programmiersprachen; funktionale Programmiersprachen (z.B. Lisp) kommen ohne Zuweisungsbefehle aus

Aufbau von Programmiersprachen

- **Kontrollstrukturen:** Wie kann die sequenzielle oder nebenläufige Abarbeitung von Anweisungen unterbrochen werden, um die Programmausführung an anderer Stelle fortzusetzen?
- **Ausdrücke und Operatoren:** Wie lassen sich Variablen und Konstanten unter Zuhilfenahme von Operatoren zu komplexeren Ausdrücken zusammensetzen, um beispielsweise Berechnungen, verknüpfte logische Bedingungen oder das Zusammensetzen von Texten zu beschreiben?

Aufbau von Programmiersprachen

Sprachmerkmal	Wir haben in Java bereits dazu gesehen	Es gibt in Java dazu noch (werden wir später kennenlernen)
Strukturmechanismen	Dateien, Klassen, Methoden, Blöcke	Packages
Primitive Datentypen	int, float, boolean, char, ...	
Typerzeugungs-mechanismen	Klassen	Arrays, Enumerations (Aufzählungstypen)
Kausalordnung von Anweisungen	Sequenzielle Folge von Anweisungen “;”	Thread-Aktivierung (nebenläufig ausgeführte Folge von Anweisungen)

Aufbau von Programmiersprachen

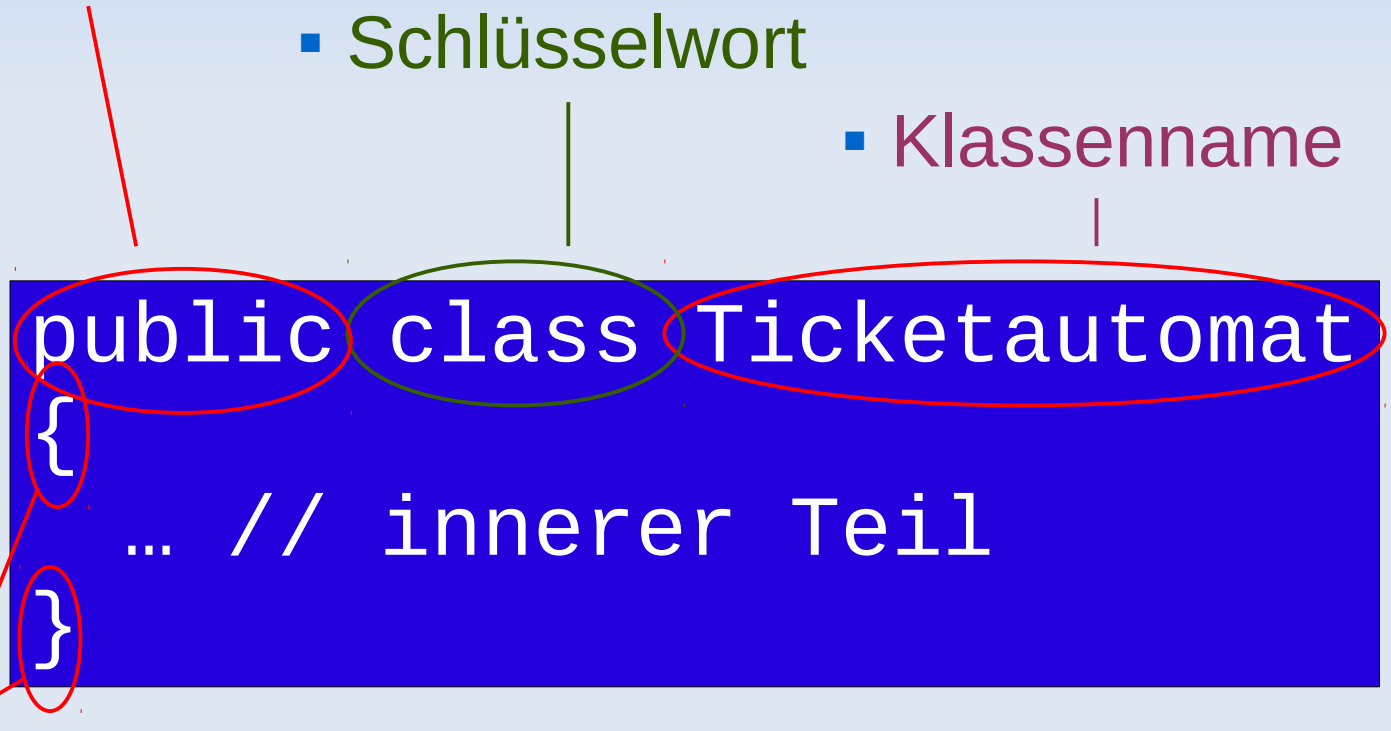
Sprachmerkmal	Wir haben in Java bereits dazu gesehen	Es gibt in Java dazu noch (werden wir später kennenlernen)
Zuweisungen	<variable> = <Ausdruck>	
Kontrollstrukturen	if, for, return	while, do-while, goto, switch
Ausdrücke und Operatoren	+,-,<<,>>,>>>, ~....	%,/,*,!,^, ...
Deklaration und dynamische Erzeugung/Vernichtung	<Typ> <Variablenname>; <Klasse> <Var> = new <Klasse>();	

naiver Ticketautomat

- zu modellieren:
  - Ticketautomat mit einer Ticketart
  - Kunde kann Geld einwerfen
  - Automat summiert eingeworfenes Geld
  - Kunde kann Ticket anfordern, sobald korrekter Betrag erreicht ist

Klassenaufbau

- **äußerer Teil**
  - Zugriffsmodifikator
  - Schlüsselwort
  - Klassenname



  - Klammern

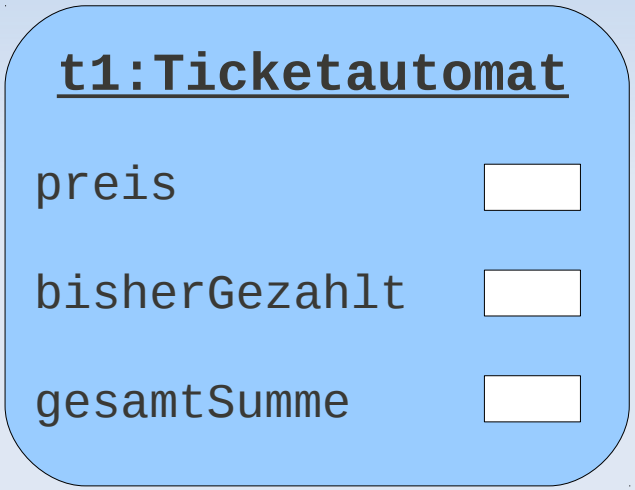
Klassenaufbau

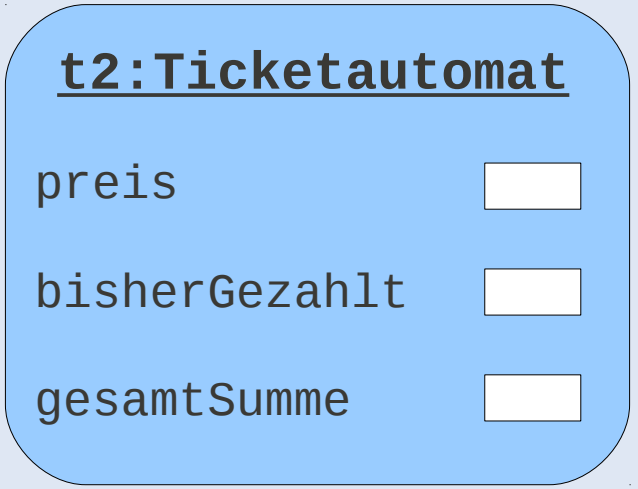
- **innerer Teil**
  - Attribute
  - Konstruktoren
  - Methoden

```
public class Klassenname
{
  Attribute
  Konstruktoren
  Methoden
}
```

Attribute

- jedes Objekt reserviert Platz für Attributwerte





- Attribute sind *Variablen*

Kommentare

- Erläuterungen für menschliche Leser
- kein Einfluss auf Funktionalität
  - einzeilig

von hier bis zum Zeilenende!

```
// Der Ticketpreis
private int preis;
```

  - mehrzeilig

```
/* Diese Klasse modelliert
   einen einfachen Ticketautomaten.
   Der Preis für ...
 */
```

Semikolon

- Semikolon schließt Anweisungen ab
  - Attributdeklaration ist Anweisung:

```
Zugriffsmodifikator   Typ   Attributname;
```

```
                private   int   preis;
```

  - Vereinbarung: Attribute vorerst *immer* `private`!

Konstruktor

- Konstruktor dient der *Initialisierung*
  - d.h. Objekt wird in gültigen Zustand versetzt
  - *Konstruktorkopf* hat keinen Rückgabetyp:

```
Zugriffsmodifikator   Klassenname( Parameterliste )
```

```
                public Ticketautomat( int ticketpreis )
```

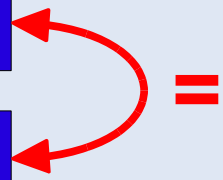
  - genereller Aufbau:

```
Konstruktorkopf
{
   // Anweisungen
}
```

Konstruktor

- Klasse kann mehrere Konstruktoren haben:
  - unterschiedliche Parameterlisten
    - Typreihenfolge!

```
public Kreis(int xkoor, int ykoor)
```




```
public Kreis(int xkoor, int radius)
```

```
public Kreis(int radius, String farbe)
```

  - kein Konstruktor definiert ⇒ Standardkonstruktor:

```
public Klassenname() {}
```

Sichtbarkeit (engl. *Scope*)

- wo im Quelltext ist Variable zugreifbar
- Parameter sind Variablen!
- Sichtbarkeit
  - formale Parameter
    - nur im Rumpf der Methode
  - Attribute
    - gesamte Klassendefinition

Lebensdauer

- wie lange existiert Variable
- Lebensdauer
  - formaler Parameter
    - auf Ausführungszeit der Methode beschränkt
    - nach Ausführung sind Werte verloren!
  - Attribute
    - Lebensdauer des Objekts

Zuweisung

- Werte werden mittels Zuweisungen in Variablen gespeichert:

```
variable = ausdruck
```

Zuweisungsoperator

```
preis = ticketpreis;
```

- Wert des Ausdrucks rechts wird in Variable links gespeichert

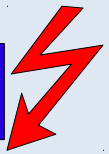
- Variablen speichern immer nur einen Wert
  ⟹ vorheriger Wert geht verloren!

Zuweisung

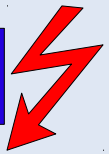
- Eine Zuweisung ist eine Anweisung
  ⟹ Semikolon!
- Typ des Ausdrucks rechts muss zum Typ der Variable links passen:

```
int preis = 500;
```

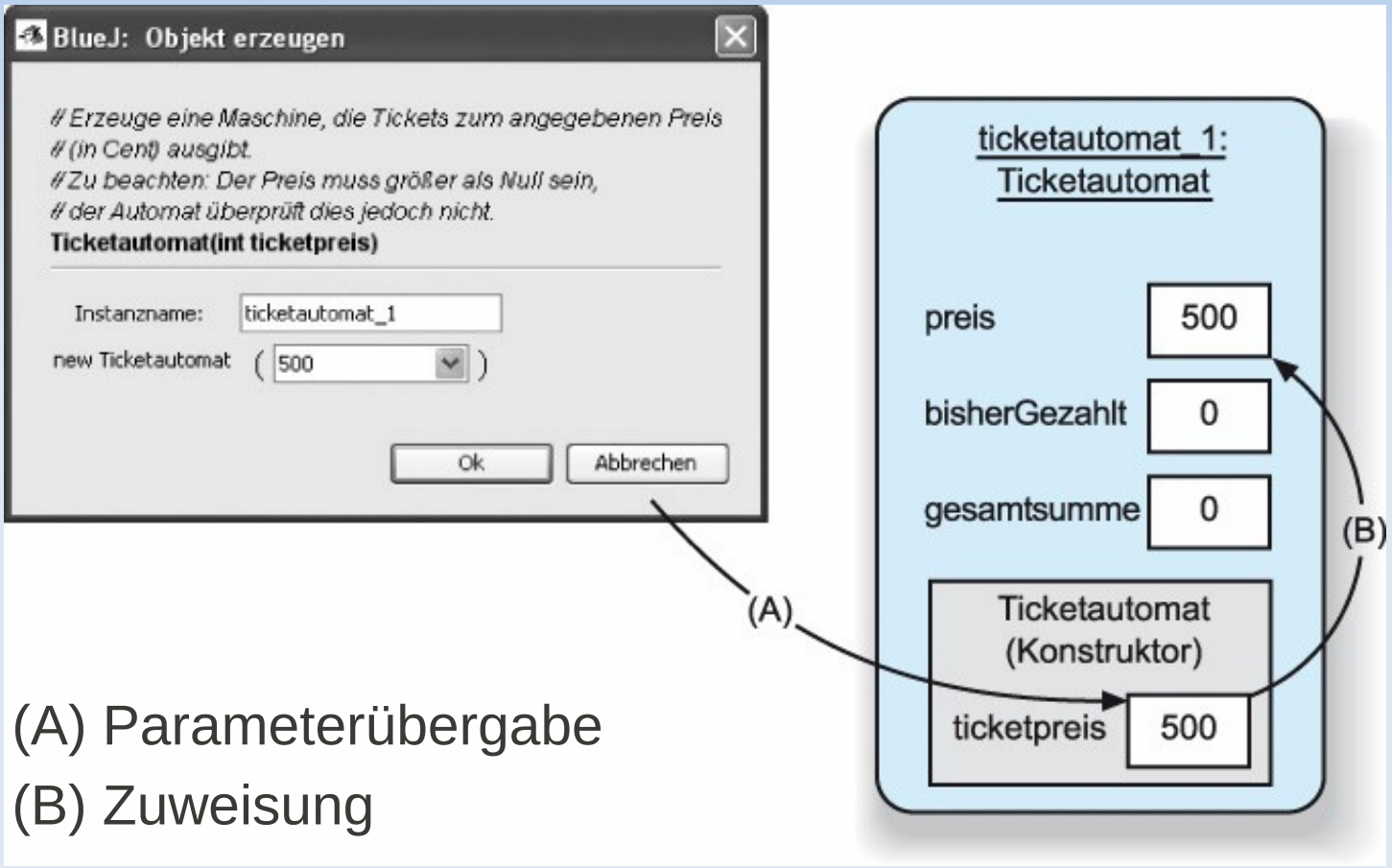
```
int preis = “hello”;
```



```
int preis = false;
```



Datenübergabe mit Parameter



(A) Parameterübergabe

(B) Zuweisung

Methoden

- Methoden implementieren das Verhalten von Objekten
- Methoden bestehen aus *Kopf* und *Rumpf*
  - Kopf: Zugriffsmodifikator und Signatur

```
public int gibPreis()
```

  - Rumpf: Deklarationen und Anweisungen

```
{
   return preis;
}
```

Block

“getter”-Methoden

- sondieren den Objektstatus
  - d.h. liefern Informationen über den Zustand
  - Rückgabeanweisung:

```
return preis;
```

  - Ausführung einer Methode endet mit Rückgabe!
  - Ergebnistyp muss mit Rückgabeanweisung übereinstimmen!

```
public int gibs() {
   return “hello”;
}
```

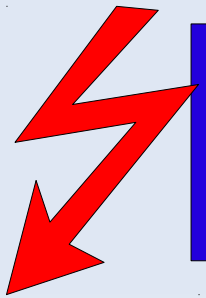
```
public int gibs() {
   return 500;
}
```

“getter”-Methoden

- sondieren den Objektstatus
  - d.h. liefern Informationen über den Zustand
  - Rückgabeanweisung:

```
return preis;
```

  - Ausführung einer Methode endet mit Rückgabe!
  - Ergebnistyp muss mit Rückgabeanweisung übereinstimmen!



```
public int gibs() {
   return “hello”;
}
```

```
public int gibs() {
   return 500;
}
```

“setter”-Methoden

- ändern den Objektstatus
  - d.h. verändern den Wert von Attributen

```
bisherGezahlt = bisherGezahlt + betrag;
```

  - typischerweise ohne Rückgabe

```
public void geldEinwerfen(int betrag) {…}
```

  - optional spezielle Anweisung

```
  return;
```

    beendet Ausführung der Methode ohne Rückgabe

Operator

- arithmetischer Operator

```
bisherGezahlt = bisherGezahlt + betrag;
```

- Operatoren sind Methoden in anderer Schreibweise

`a + b` entspricht `int plus(int a, int b)`

- Operatoren haben einen Typ

```
+ : int x int → int
- : int x int → int
* : int x int → int
/ : int x int → int
% : int x int → int
```

Ausgabe

- `System.out.println()` gibt Parameter auf Konsole aus

```
System.out.println(“hello world!”);
```

gibt aus:

Stringkonkatenation

- **+** im Kontext von Zeichenketten bewirkt Konkatenation (**+** ist *überladen*)

```
System.out.println(5 + “ cent”);
```

gibt aus:

- Quiz: Was ist die Ausgabe von

```
System.out.println(5 + 6 + “ cent”);
```