Praxis der Programmierung

Template-Funktionen und -Klassen

Einführung in Java

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Template-Funktionen

Minimumfunktion und offene Typen

- Aufruf `min(x,y)` sollte für möglichst alle Datentypen funktionieren, für die eine Ordnungsrelation definiert ist

- Keine Code-Verdopplung!
  $\rightsquigarrow$ Überladen der Funktion ungünstig
  $\rightsquigarrow$ Funktion mit offenem Typ definieren

- Template-Funktionen
  - verwenden offene Typen
  - definieren eine Schablone einer Funktion, die typunabhängiges Verhalten hat
  - Typen werden beim Aufruf der Funktion festgelegt
    $\rightsquigarrow$ Compiler erzeugt aus der Schablone eine zu den Typen passende Funktion

Beispiel: Minimumfunktion

```
template <typename T>   // eine Schablone mit dem Typparameter T
T min(T a, T b) {       // T kann jetzt wie ein Typ verwendet werden

     return (a < b) ? a : b;

}

int main() {
   int i = 19;
   int j = 66;
   int a = min(i,j);  // Funktion mit T = int wird jetzt erzeugt
}
```

- Statt `typename` kann synonym auch `class` verwendet werden.
- mehrere Typparameter durch Komma getrennt, *z.B.* `<class T, class S>`

Festlegung der Zieltypen

- entweder durch Parameterübergabe
- oder durch explizite Angabe der Zieltypen:

```
template <typename T1, typename T2, typename T3>
T3 foo(T1 x, T2 y) {
   // ...
}

int main() {
   cout << foo<short,long,int>(23,11);  // Rueckgabetyp int
}
```

swap als Template

```
template <typename T> void swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

Template-Klassen / generische Typen

Definition von Template-Klassen

```
// Schablone einer Klasse mit einem Typparameter T
template <typename T> class Cls {
   // jetzt kann T wie ein Datentyp verwendet werden, z.B.:
   T var;
   T foo(int n, T& t);
};

template <typename T> T Cls<T>::foo(int n, T& t) {
   // ...
}

int main() {
   Cls<int> instance;  // Cls<int> ist ein generischer Typ
                       // Compiler legt generischen Typ erst bei der
   // ...              // Definition eines Objekts an
}
```

Einführung in Java

Java

Applikation: eigenständiges Programm, das in Java geschrieben wurde und ohne Browser ausgeführt werden kann

Applet : Java-Programm, das in eine HTML-Seite eingebunden wird und in einem Browser ausgeführt werden kann

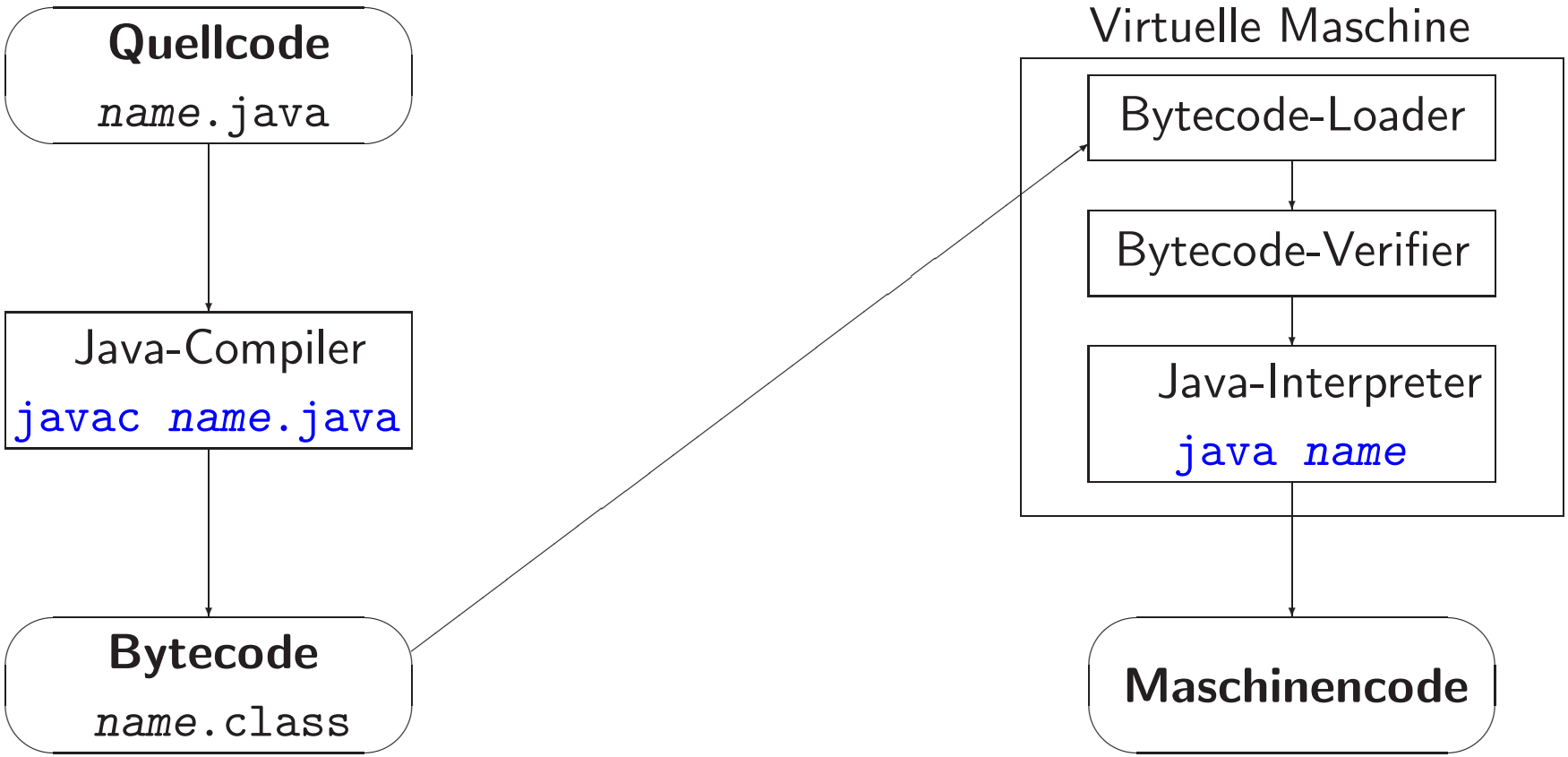
Java und das Internet

- *Idee:* Übertragung von Programmcode vom Server zum Client, der im Internet-Browser ausgeführt wird

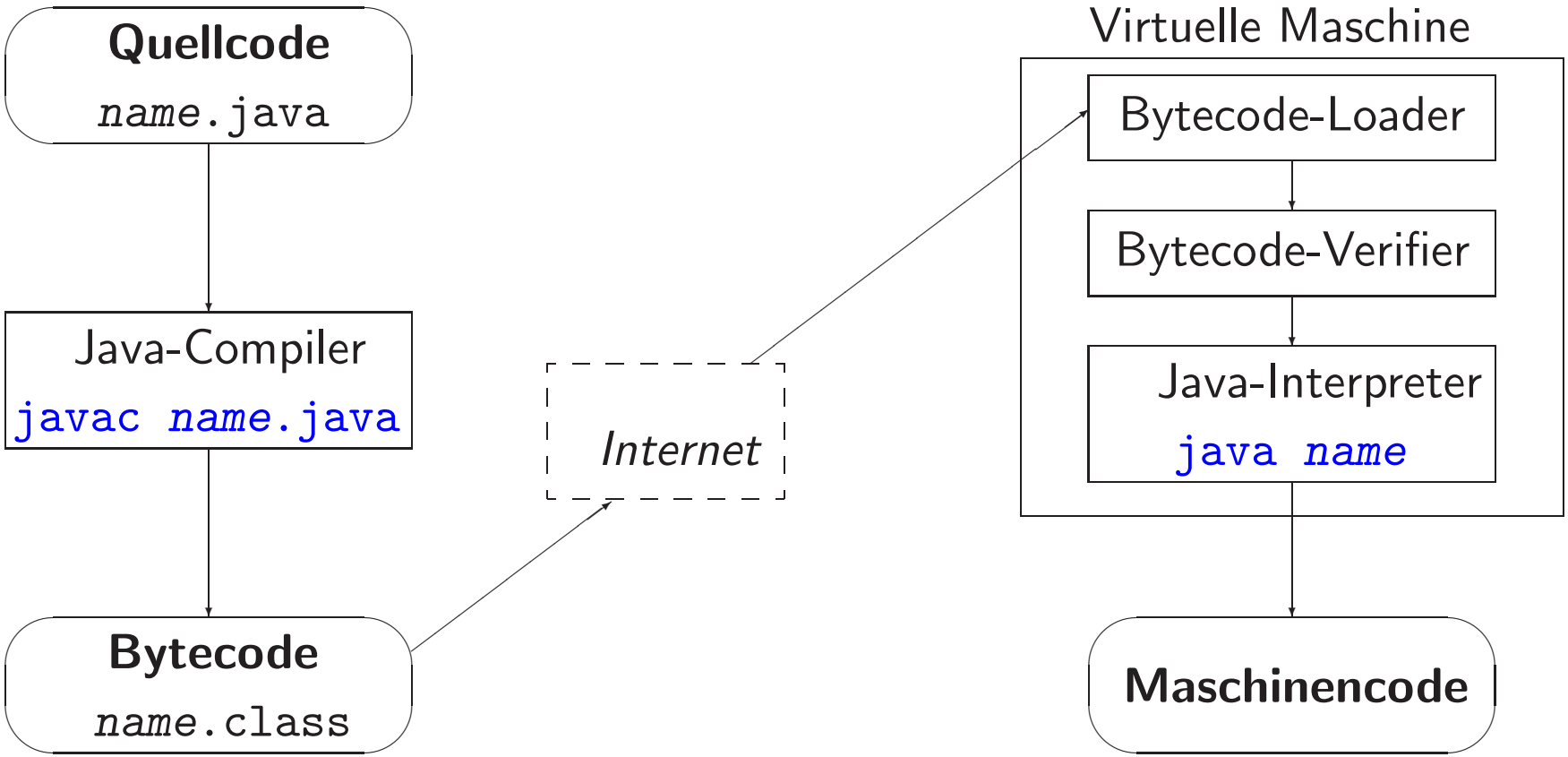
$\Longrightarrow$ **Anforderungen an die Programmiersprache:**

- plattformunabhängiger Programmcode (*Portabilität*)
- gegen einfache Änderungen der Ablaufumgebung *robuster* Programmcode
- *Sicherheit* z.B. vor Viren sowie vor dem Zugriff auf das Dateisystem, die Hardware und den Browser
- Vielfältigkeit/Universalität
- modernes Programmier-Paradigma

Quellcode — Bytecode — Maschinencode



Applets im Internet



Eigenschaften von Java

- portabel
  - virtuelle Maschine
  - plattformunabhängige Datentypen
- robust
  - keine Pointer, Garbage Collector
  - stenge Objektorientiertheit, relativ strikte Typenprüfung
- sicher
  - virtuelle Machine, Bytecode-Verifizierung
  - digitale Signaturen
- multithreaded
  - parallele, kommunizierende Prozesse
- verteilt
  - Remote Method Invocation (RMI)
- streng objektorientiert

Quellen

- Java-Entwicklungsumgebung:

  `http://www.oracle.com/technetwork/java/index.html`
  `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
  `http://www.oracle.com/`

- Literatur:

  http://docs.oracle.com/javase/8/docs/api/index.html
  http://www.javabuch.de/
  http://openbook.galileocomputing.de/javainsel/

Einfache Datentypen

Datentyp	mögliche Werte	Wrapper-Klasse
boolean	true, false	Boolean
byte	-128..127 (8 Bit)	Byte
short	-32768..32767 (16 Bit)	Short
int	-2147483648..2147483647 (32 Bit)	Integer
long	-9223372036854775808..922$\cdots$807 (64 Bit)	Long
float	Gleitkommazahl (32 Bit)	Float
double	Gleitkommazahl (64 Bit)	Double
char	einzelne Unicode-Zeichen (16 Bit)	Character

Quellcode einer Java-Applikation

```
/* Name.java
*/

public class Name { // muss mit dem Dateinamen uebereinstimmen
     public static void main (String[] args) {
          Anweisung;
          Anweisung;
          ...
          Anweisung;
     }
}

// Name
```

Beispiel Hello World

```
public class HelloWorld {
    public static void main(String[] args) {
       System.out.println("Hello World!");
    }
}
```

- `System.out.println(`s`);` zur Stringausgabe mit Zeilenvorschub
- `System.out.print(`s`);` erzeugt die Stringausgabe ohne Zeilenvorschub.

Java-Klassen und Applikationen

- Es gibt nur Klassen. (streng objektorientiert; nur wenige Spezialfälle)
- Alle Funktionen sind Methoden (ggf. statisch),
  alle globalen Variablen sind statische Datenelemente.
- Es gibt keine Header-Dateien.
  (Um Schnittstellen festzulegen, werden Interfaces benutzt.)
- Eine Java-Applikation benutzt genau eine Java-Klasse mit einer `main`-Methode.
- Es gibt nur die `main`-Methode mit einem String-Array als Parameter:
  `public static void main(String[] args)`.
  `String` ist eine Klasse der Standardbibliothek.

Parameterübergabe

Der Parameter der `main`-Methode

```
String[] args
```

wird beim Programmstart mit jenen Zeichenketten initialisiert, die auf der Kommandozeile als Argumente nach dem Namen der Applikation angegeben sind.

```
java BspApplikation das sind 4 Parameter
```

bewirkt die Initialisierung

```
args[0] = "das"     args[1] = "sind"
args[2] = "4"       args[3] = "Parameter"
```

Definieren und Initialisieren

- wie in **C++**, *aber*:
- Modifikatoren für die Sichtbarkeit stehen in jeder Definition als erstes Schlüsselwort

```
private int x;
public static void methode() { }
```

- Konstruktoren werden *immer* mit `new` aufgerufen
  $\rightsquigarrow$ Objekte werden *immer* auf dem Heap angelegt
- *Alle* Variablen von einem Klassentyp sind Referenzvariablen.
  (Ohne dass Sie mit `&` als solche vereinbart werden müssen!)
- Es gibt keine Pointer!

Standardinitialisierung von Datenelementen

Alle Datenelemente, die nicht durch die Parameter des Konstruktors initialisiert werden, erhalten standardmäßige Initialwerte wie folgt:

`byte, short, int`	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
Verweistypen	null

Aufruf überladener Konstruktoren

- Mittels `this(`*`Parameterliste`*`)` können andere, überladene Konstruktoren (mit passender Parameterliste) aufgerufen werden.
- Vermeidung von Code-Verdopplung

```
public Class Cls {
    // 400 Datenelemente
    int number;

    Cls() { /* Initialisierung der 400 Datenelemente */ }

    Cls(int number) {
       this();
       this.number = number;
    }
}
```

Nullpointer

- Standardinitialisierung

  ⇝ Vermeidung von logischen Fehlern durch fehlende Initialisierung von Datenelementen

  ⇝ Es entstehen Nullpointer!

- Es entstehem leicht Laufzeitfehler: `NullPointerException`
  - bei Zugriff auf Datenelemente von Variablen mit Wert `null`
  - bei Aufruf von Methoden mit Variablen mit Wert `null`

Zerstören von Objekten

- Es gibt keine Destruktoren.
- Objekte auf dem Heap, auf die keine Referenz mehr existiert, werden automatisch (von Zeit zu Zeit) vom Garbage Collector gelöscht. Der Speicherbereich ist dann wieder freigegeben.

  $\rightsquigarrow$ Vermeidung von Memory Leaks und Dangling Pointers

Datentypen

einfache Datentypen

```
boolean
byte, short, int, long
float, double
char
```

Wrapper-Klassen:

```
Boolean
Byte, Short, Integer, Long
Float, Double
Character
```

Referenz-/Verweisdatentypen

```
String
Point
Integer
...
```

haben KEINE Wrapper-Klassen

einfache Datentypen	**Referenz-/Verweisdatentypen**

Variablendeklaration

```
int num1, num2;
boolean b1, b2;
```

```
Point p1, p2;
String str1, str2;
```

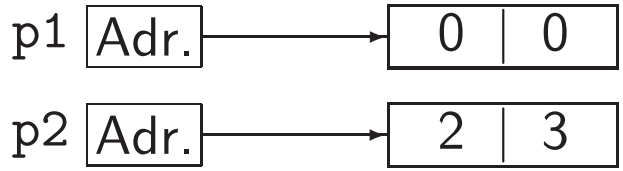
Variableninitialisierung

```
num1 = -12;
num2 = 4;
b1 = true;
b2 = false;
```

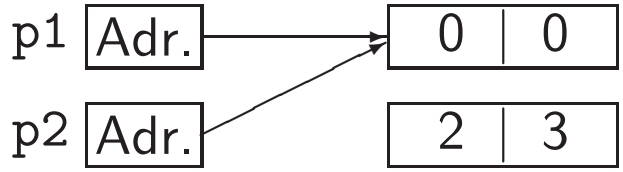
```
p1 = null;
p2 = new Point();
str1 = new String("Hallo!");
str2 = "Hallo?";
```

im Hauptspeicher:

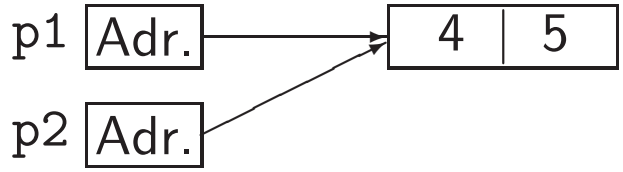
```
p1 = new Point();
p2.moveTo(2,3);
```




```
p2 = p1;
```




```
p1.moveTo(4,5);
```




```
num2 = num1;
```


```
num1 = 99;
```

Arrays (1)

- Arrays sind Verweisdatentypen.
- Arrays stehen alle Methoden zur Verfügung, die von allen Objekten (Instanzen beliebeiger Java-Klassen) benutzt werden können. (Arrays erben von der Klasse `java.lang.Object`.)
- Array-Elemente werden (wie die Datenelemente von Objekten) automatisch initialisiert.

Aber:

- Es gibt keine Klasse, von der Arrays Instanzen sind.
- Arrays haben keine Konstruktoren. Statt dessen gibt es eine spezielle Syntax des `new`-Operators.

Arrays (2)

- Arrays sind immer **eindimensional**, können aber geschachtelt werden (d.h. Arrays als Elemente enthalten).

- Deklaration:

```
int[] bsp;
int bsp[];
String[][] aStr;
String aStr[][];
```

Die Anzahl der Elemente wird erst bei der Initialisierung angegeben.

Initialisierung von Arrays

- direkte Initialisierung (gleichzeitig mit der Deklaration):

```
int[] bsp = {5, 4, 3, 2, 1};
String[][] aStr = {{"ja", "nein"}, {"yes", "no"}, {"?"}};
```

- nachträgliche Initialisierung mit dem `new`-Operator:

```
bsp = new int[5];   // Standardinitialisierung der Elemente
aStr = new String[3][2];
```

Danach ist z.B. möglich:

```
bsp[0] = 5; bsp[1] = 4; bsp[2] = 3; bsp[3] = 2; bsp[4] = 1;
int laenge = bsp.length;    // ergibt 5
```

Schlüsselwörter zur Zugriffsmodifikation

- Datenelemente/Methoden/Konstruktoren mit dem Modifier

`public`	sind für alle Klassen sichtbar;
`private`	sind nur für die Klasse sichtbar, in der sie vereinbart sind;
ohne Modifier	sind für Klassen aus demselben Paket (Verzeichnis) sichtbar
`protected`	für alle Unterklassen und Klassen aus demselben Paket sichtbar

- Der Modifier ist das *erste* Schlüsselwort in der Definition/ Signatur.
- Klassen dürfen auch Modifier haben. (`public class ...`)
- Schlüsselwort `final` zur Vereinbarung von Konstanten

Vererbung

- keine Mehrfachvererbung
- Syntax: `public class Unterklasse extends Oberklasse`
- alle Methoden sind implizit virtuell
- Aufruf von Konstruktoren der Oberklasse: `super(...)`
- Aufruf von Methodenimplementierungen der Oberklasse (beim Überschreiben): `super.method(...);`
- alle Klassen erben (automatisch) von `java.lang.Object`
- abstrakte Klassen und Methoden mit Schlüsselwort `abstract` definieren, z.B.: `public abstract class { public abstract void method(); }`

Quellcodes einer Unterklasse (Prinzip)

```
public class Unterklasse extends Oberklasse {

      // neue Datenelemente

      // Konstruktoren
      public Unterklasse(...)  {
          super(...);  // optional
          // weitere Anweisungen;
      }

      // neue Methoden

      // überschriebene Methoden
      public void methode(...)  {
          super.methode(...);  // optional
          // weitere Anweisungen
      }

  }
```