Karlsruher Institut für Technologie

5. Tutorium

Konvertierung, Datenkapselung, Sichtbarkeit, Gültigkeitsbereiche, Listen

Tutorium 14

Péter Bohner | 30.11.2022

Inhaltsverzeichnis

- **Korrektur vom letzten Tutorium:**
  Wenn man die falsche Schleife im 2. ÜB verwendet (z.B. for statt for-each, while statt for), dann gibt es einen halben Punkt Abzug (Methodik).
- Meldet euch für die Präsenzübung im CAS bis zum 7.12. an!
- Dort müsst ihr eure (eingeteilte!) Tutoriumsnummer wissen (14)

Wiederholung

Was wisst ihr über Arrays? (`int[] a`)

- Haben immer eine feste Länge
- Werden in Java wie Objekte behandelt
- Können nur Instanzen eines Datentyps speichern (in diesem Beispiel ints)

Wie kann man auf die Länge eines Arrays zugreifen?

arrayName.length

Wiederholung

Was berechnet diese Methode?

```
int function(int[] a) {
    int result = Integer.MIN_VALUE;
    for (int value : a) {
        if (value > result) {
            result = value;
        }
    }
    return result;
}
```

Das Maximum des Arrays

Was müsste man an der obigen for-each-Schleife ändern, damit daraus eine normale for-Schleife wird?

```
int function(int[] a) {
    int result = Integer.MIN_VALUE;
    for (int i = 0; i < a.length; i++) {
        if (a[i] > result) {
            result = a[i];
        }
    }
    return result;
}
```

Konvertierung

Typen von Variablen, Konstanten und Methoden können in Java automatisch angepasst werden:

- Einschränkende Primitivkonvertierung (narrowing primitive conversion)
- Erweiternde Primitivkonvertierung (widening primitive conversion)
- String-Konvertierung (string conversion)
- Erzwungene Konvertierung (casting conversion)

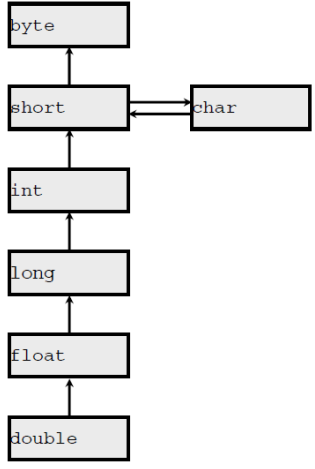
Narrowing Conversion

Überführung in **kleineren** Typ

- **Informationsverlust** bezüglich der **Genauigkeit** und **Größe**
- Wird erzwungen mit `cast`-Operator:

  (neuerTyp) alterWert
- Sehr Vorsichtig mit Narrowing Conversions umgehen!
- Bei ganzzahligen Werten werden die höherwertigen Bits abgeschnitten

```
int x = 500;
byte b = (byte) x; // b = -12;
```

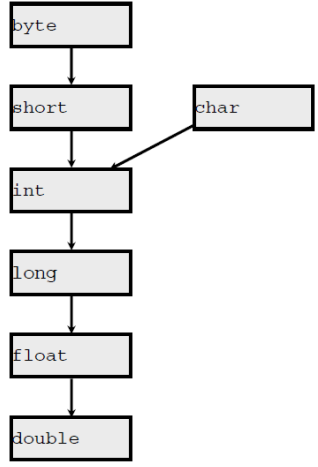


Widening Conversion

Überführung in **größeren** Typ

- Kein Informationsverlust bezüglich der Größe
- **Informationsverlust** bezüglich der **Genauigkeit**!
- Automatisch

```
int big = 52345678;
float approx = big;
int value = (int) approx; // value = 52345680;
```



Weitere Conversions

String Conversion

- Alle Datentypen in Java können zu `String` konvertiert werden
- Primitive Datentypen werden automatisch konvertiert

```
String s = primitive + "";
```

- Objekte können eigene `toString()`-Methode implementieren

```
objektname.toString();
```

Casting Conversion

- Typkonvertierung erzwingen
- `int a = 1000; byte b = (byte) a; // b: -24`

Eigene `toString()`-Methode

Beispiel:

```
class Book {
    final int id;
    int year;
    String title;
    String author;
    ...
    @Override
    String toString() {
        return title + ": " + author;
    }
}
```

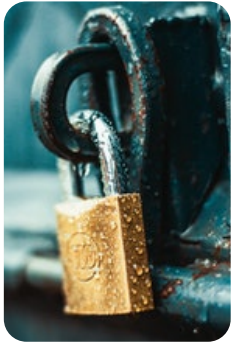
- Erlaubt es, eigene String-Repräsentationen eines Objekts zu erzeugen
- Beispiel `Book`: Man kann sich eine Buchliste einfach und übersichtlich ausgeben lassen
- `@Override` sagt dem Compiler, dass die standardmäßige `toString()`-Methode ersetzt wird

```
Book[] books; // fill array
for (Book book : books) {
    System.out.println(book.toString());
}
```

Anwendung

```java
byte b = 10;                          // no conversion
int i = b;                            // widening conversion
double d = b * 1.5;                   // widening conversion
float f = (float) d;                  // narrowing conversion + cast conversion
String output = f + "";               // string conversion
```

Datenkapselung



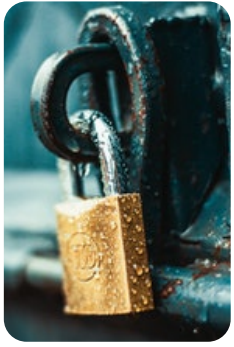
Was ist an diesem Code schlecht?

```
class Account {
    int balance;
    ...
}
```

Andere Klasse kann ohne Probleme von einem Objekt `account` der Klasse Account den Kontostand ändern.

$\Rightarrow$ Daten müssen gekapselt werden!

Datenkapselung



Was heißt das auf **Java** bezogen?

- Variablen dürfen von außen nicht sichtbar und veränderbar sein
  - Kontostände könnten einfach geändert werden
- Deshalb werden Variablen kontrolliert durch Methoden verändert
- „lose gekoppelte“ Programmkomponenten werden ermöglicht
- Interne Strukturen bleiben verborgen ⇒ nur Schnittstelle bekannt

Beispiel

Schlecht:

```
class Account {
    int balance;
    ...
}
account.balance = 1000;
```

Besser:

```
class Account {
    private int balance;
    public int getBalance() {}
    ...
}
```

Schlüsselwort `private` **verhindert direkten Zugriff** auf Attribut!

⇒ **Ab sofort modelliert ihr immer so!**

Attribute zurückgeben

Verwendung von `gettern`:

getter

- Kontrolliertes zurückgeben der Attribute
- Nicht für alle Attribute sind `getter` sinnvoll

```
public int getBalance() {
    return this.balance;
}
```

Attribute verändern

Verwendung von `settern`:

setter

- Kontrolliertes ändern von Attributen
- Nicht für alle Attribute sind `setter` sinnvoll

```
public void setBalance(int amount) {
    int newBalance = this.balance + amount;
    if (newBalance >= 0) {
        this.balance = newBalance;
    }
}
```

Pakete

Erklärung

- Dienen der Strukturierung des Quelltextes
- Vermeidung von Namenskonflikten
- Innerhalb eines Paketes dürfen Klassen **nicht gleich** heißen
- Es kann also in zwei verschiedenen Paketen jeweils eine Klasse mit gleichem Namen geben

Anwendung

- Pakete werden umgekehrt zu ihrer Domain benannt
- `com.java.util` (Kleinschreibung beachten)
- Deklaration `package pfad.zur.klasse`
- Ordnerstruktur entspricht der Benennung des Paketes

  $\Rightarrow$ Eine Klasse ist in einem Paket, wenn sie im Ordner mit dem Paketnamen gespeichert wurde

Sichtbarkeit

Modifikator	Sichtbarkeit			
	Klasse	Paket	Unterklasse	„Welt“
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	-
-	✓	✓	müssen im Paket sein	-
`private`	✓	-	-	-

Wichtig:

- Nutzung von Klassen aus Paketen mit `import packagename.classname;`
- Auch Nutzung von Sichtbarkeitsmodifikatoren bei Klassen

Zugriffsrechte

Es ergibt sich folgende Richtlinie:

1. Attribute immer `private`
2. Zugriffe und Modifikationen mit `gettern` und `settern`
3. Klassenkonstanten können `private` oder `public` sein
4. Hilfsmethoden sind `private`

Gültigkeit

Gültigkeitsbereiche

- **Attribute** sind in der gesamten **Klasse** verfügbar
- **Parameter** sind innerhalb der **Methode** verfügbar
- **Lokale Variablen** sind innerhalb ihres **Blocks** verfügbar

Überschattung (bei Variablen mit gleichem Namen)

- Java verbietet gleiche Namen bei mehreren lokale Variablen/Parameter
- Aber Lokale Variablen/Parameter können heißen wie Attribut
- Lokale Variablen haben **Vorrang**
- Attribute werden „überschattet“, aber durch das Schlüsselwort `this` bleibt der Zugriff möglich

Aufgabe

```
1  class Person {
2      private int age;
3      public int increaseAge(int newAge) {
4          if (newAge < age) {
5              age = 10;
6          }
7          else {
8              int age = newAge;
9          }
10         return age;
11     }
12 }
```

Gebt für jede Variable ihren **Gültigkeitsbereich** an.
An welchen Stellen im Code erkennt ihr **Überschattung**?

Zeile 2: `age` → Attribut, in der ganzen Klasse gültig

Zeile 3: `newAge` → Parameter, in der Methode gültig

Zeile 8: `age` → lokale Variable, nur im Rumpf von else gültig
Zeile 8: `age` → überschattet Attribut `this.age`

Was gibt die Methode zurück?

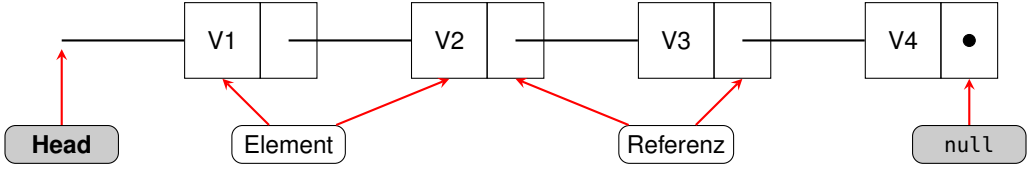
`newAge < age`: 10
sonst: Das, was vorher im Attribut `age` gespeichert war

Abstrakter Datentyp

- Geheimnisprinzip: Eigentliche Implementierung (interner Aufbau) bleibt verborgen
- Abstrakte Sicht wird nach außen weitergegeben (Nutzersicht)
- Zugriff nur über eine **Schnittstelle**

$\Rightarrow$ eine Datenstruktur realisiert einen zusammengesetzten abstrakten Datentyp (z.B. rekursive Datenstruktur)

Listen (einfach verkettet)



Listen - Implementierung

Knoten (Element):

```
class ListNode {
    private int element;
    private ListNode next;

    public ListNode(int element, ListNode next) {
        this.element = element;
        this.next = next;
    }
}
```

Liste:

```
class List {
    private ListNode head;

    public List() {
        this.head = null;
    }

    // list operations
}
```

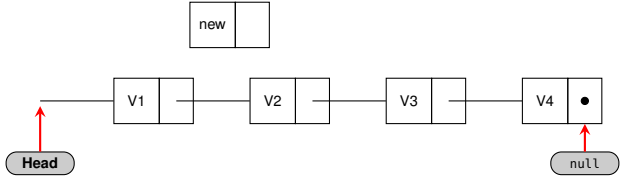
Listen - Operationen

Es gibt verschiedene Operationen, um Listen zu modifizieren:

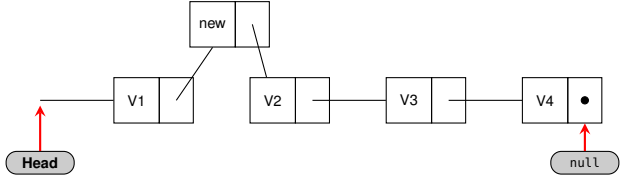
- `void addFirst()`
- `void addLast()`
- `void remove()`
- `boolean contains()`
- `int size()`
- `boolean isEmpty()`

Listen - Einfügen

Ausgangssituation:



Nach Einfügen von new:



Listen - Einfügen

Element vorne an der Liste anhängen:

```
public void addFirst(int value) {

    ListNode newHead = new ListNode(value, this.head);
    this.head = newHead;

}
```

Listen - Einfügen

Element hinten an der Liste anhängen:

```
public void addLast(int value) {
    // your code
}
```

Jetzt ihr!

Wir vervollständigen die Methode gemeinsam, sodass ein neuer Listenknoten am Ende der Liste eingefügt wird.

Listen - Element hinten einfügen

Was passiert, wenn die Liste leer ist?

1. Ist die Liste leer? (Prüfe `Head` auf `null`)
   - Ja: Füge Element an erster Stelle ein, breche ab
   - Sonst: Mache weiter

Wie kommt ihr an den letzten Knoten?

2. Neue Referenz auf aktuellen `Head` bilden
3. Bis zum letzen Knoten iterieren

Was muss noch gemacht werden?

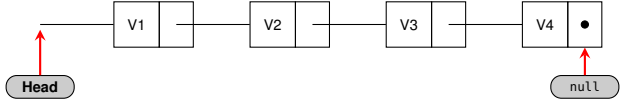
4. Neuen Knoten anhängen

```
public void addLast(int value) {
    if (this.head == null) {
        addFirst(value);
        return;
    }

    ListNode current = this.head;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new ListNode(value, null);
}
```

Listen - Entfernen

Ausgangssituation:



Nach Entfernen von `V2`:

Listen - Entfernen

Implementierung

Schaut Euch das in den Vorlesungsfolien an
Fragen könnt ihr gerne im Tutorium stellen
Analoges Vorgehen: `contains`-Methode

⇒ Es ist wichtig, dass ihr das Prinzip verstanden habt!

Listen vs. Arrays

Wann benutze ich was?

Arrays

- Einfacher Zugriff auf Elemente
- Verwendung ist intuitiv (z.B. `for-each`-Schleife)
- Größe ist fest
- Umstrukturierung aufwendig

Listen

- Länge dynamisch
- Einfügen und Löschen an gewünschten Stellen einfach
- Zugriff auf einzelne Elemente erschwert

LinkedList

Listen sind in Java bereits implementiert und müssen nur noch importiert werden:

- `import java.util.LinkedList`
- `LinkedList<Type> name = new LinkedList<Type>();`
- `Type` darf nicht primitiv sein

⇒ Nur bei Übungsaufgaben verwenden, wenn es auch erlaubt ist!

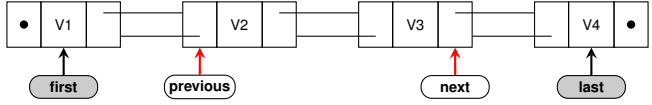
LinkedList

Welche Methoden sind bereits nutzbar?

- `void add(Type element)`
- `Type remove(int index)` / `boolean remove(Type element)`
- `Type get(int index)` / `Type getFirst()` / `Type getLast()`
- `int size()`

`Link zur Dokumentation`

Doppelt-verkettete Liste



- Knoten kennt den vorherigen und den darauffolgenden Knoten
- Etwas flexibler als einfach-verkettete Listen
- Bei der Modifikation aufpassen, da jetzt mehrere Zeiger verändert werden müssen

Bis zum nächsten Tutorium am 07.12.2022!

Nicht vergessen: **Donnerstag** um 06:00 Uhr Abgabeende!
Nächstes Übungsblatt gibt es **morgen**.