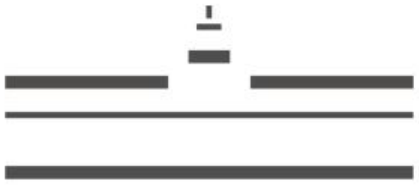


Westfälische
Wilhelms-Universität
Münster

Programmieren in Java

Einführung in die (imperative) Programmierung (Teil 2)

wissen.leben
WWU Münster

Name: Patrick Förster, Thorsten Hovenga

ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

Zusammengesetzte Datentypen

- Wie der Namen schon vermuten lässt, sind zusammengesetzten Datentypen solche, deren Werte sich aus einer Menge von Werten zusammensetzen
- Beispiele:
  - **Array**: Endliche Menge aus Werten eines bestimmten Typs

    $$A = \{a \mid a \in T\}$$

  - **Records**: Kartesisches Produkt aus Werten von bestimmten, aber durchaus verschiedenen Typen

    $$R = \{(r_1,...,r_n) \mid r_1 \in T_1,...,r_n \in T_n\}$$

  - **Union**: Ausgezeichnetes (Tag) kartesisches Produkt
    - Jeder Wert eines Union verfügt über eine zusätzliches Information (Tag), die den Wert eindeutig einem bestimmten Union zuordnet

      $R_1 = \{(x,y) \mid x,y \in \text{int}\}$ $R_2 = \{(c_1,c_2) \mid c_1,c_2 \in \text{int}\}$ $U_1 = \{(v_1,v_2,t) \mid v_1,v_2 \in \text{int}, t \in \{'R_1','R_2'\}$

    - Es gilt $(1,1) \in R_1 \wedge (1,1) \in R_2, \rightarrow (1,1) = (1,1)$, aber $(1,1,'R_1') \in U1 \wedge (1,1,'R_2') \in U1 \rightarrow (1,1,'R_1') \neq (1,1,'R_2')$

Arrays

• Jede Programmiersprache, die Arrays anbietet, benötigt eine Möglichkeit auf ein einzelnes Element aus dem Array zugreifen zu können:

$$m : I \to A = \{m \mid i \in I \Rightarrow m(i) \in A\}$$

• m ist eine Funktion, die ein Element des Datentyps I auf ein Element des Arrays A abbildet
• Die meisten Programmiersprachen bietet eine „Index-Funktion“ als Abbildung an:
    • Jedes Element ist eindeutig einem Wert aus dem Wertebereich **`int`** zugeordnet
    • Die Menge alle dieser Werte wird als Indexmenge I bezeichnet
    • I ist meist konsekutiv beginnend bei 0 für das erste Element
• Eine Abbildung deren Indexmenge nicht fortlaufend ist, wird als „assoziativ“ bezeichnet
• Bspw. könnte man Strings als Index verwenden

Arrays (Java)

- Java kennt nur konsekutive Arrays mit $I \subseteq +\text{int} = \{0...2147483647\}$
- Zum Erzeugen eines Arrays wird der **new**-Operator benutzt, der quasi den nötigen Speicherplatz für einen Array reserviert
- Der Zugriff auf ein Array-Element erfolgt über den **[]**-Operator
- **[]** wird zudem genutzt, um einen Array bestimmten Typs zu deklarieren

```
int[] array = new int[10];
System.out.println(array[5]);
for (int i = 0; i < array.length; i++) array[i] = i;
```

- Hier wird zuerst ein neuer Array vom Typ **int** der Größe **10** erzeugt
- Die Größe eines Arrays ist fest, d.h. sie muss bei der Deklaration über **[]** (rechts) angeben und kann im Nachhinein nicht mehr geändert werden
- Die Größe kann jederzeit über ***name*.length** abgefragt werden
- Nach der Initialisierung ist der Array noch „leer“ (der Zugriff in Zeile 2 liefert den Defaultwert **0**)
- Die **[]**-Operator kann auf der linken Seite einer Zuweisung genutzt werden, um ein Element mit der indizierten Stelle zu assoziieren

Arrays (Java)

- Man beachte: Das erste Element eines Arrays hat immer den Index **0**!
  - Die **for**-Schleife im Beispiel läuft daher von **0...9**
- Bei der Deklaration mit Initialisierung kann die Größe des Arrays weggelassen werden, wenn man explizit den Inhalt des Arrays angibt:

- Der Inhalt wird als Aufzählung in {}-Klammern angeben
- Das explizite Erzeugen per **new** fällt weg

```
int[] array = {0, 1, 2, 3, 4, 5, 6};
System.out.println(array.length);
➔ 7
```

- Achtung: der Versuch eines Zugriffes mit einem Index außerhalb der zulässigen Indexmenge wird mit einem Fehler (*Exception*, mehr dazu später) und dem Abbruch des Programms bestraft

```
System.out.println(array[-1]);
→Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsExcep
tion: -1
System.out.println(array[10]);
→…java.lang.ArrayIndexOutOfBoundsEx
ception: 10
```

Arrays (Java)

- Arrays können in Java auch mehrdimensional erzeugt werden
- Die Anzahl der **[]**-Operatoren bei der Deklaration, legt die Dimension fest

```
int[][] array = new int[10][5];
array[0][0] = 105;
```

- Dieses zwei-dimensionale Array hat 10*5=50 **int**-Einträge, die in 10 Zeilen und 5 Spalten gegliedert sind
- Ein Element wird dann analog zum 1-dimensional Array gesetzt
- Intern wird ein Array angelegt, dessen Elemente wiederum ein Array sind
- D.h. insbesondere, dass nur die Kardinalität der ersten Dimension bei der Initialisierung festgelegt werden muss:

```
int[][] array = new int[10][];
array[0] = new int[] {0, 1, 2, 3, 4, 5, 6};
array[1] = new int[] {0, 1, 2, 3};
```

- Man beachte, dass in diesem Beispiel der **{}**-Operator zum Erzeugen genutzt wird, allerdings ein **new** angeben werden muss
- Für den Compiler ist dies eine Zuweisung keine Initialisierung

Nachtrag Kontrollstrukturen: for(each)

• Java kennt noch eine alternative Variante der **`for`**-Schleife, die oft als **`foreach`** bezeichnet wird:

```
for ( Typ  Bezeichner : Aufzählung )
    Code
}
```

• In dieser Variante muss zu erst eine temporäre Variable (*Typ* + *Bezeichner*) definiert werden
• Diese Variable muss den gleichen Typ haben, wie alle Elemente in der *Aufzählung*
• Die *Aufzählung* ist entweder ein Array oder ein Objekt, das aufzählbar ist (mehr dazu später)

```
int[] array = {0, 1, 2, 3, 4, 5, 6};

for (int element : array) {
     System.out.println(element );
}
```

foreach

```
int[] array = {0, 1, 2, 3, 4, 5, 6};

for (int element = 0; element < array length; element++)
     System.out.println(element );
}
```

for

Speichermangement (Stack)

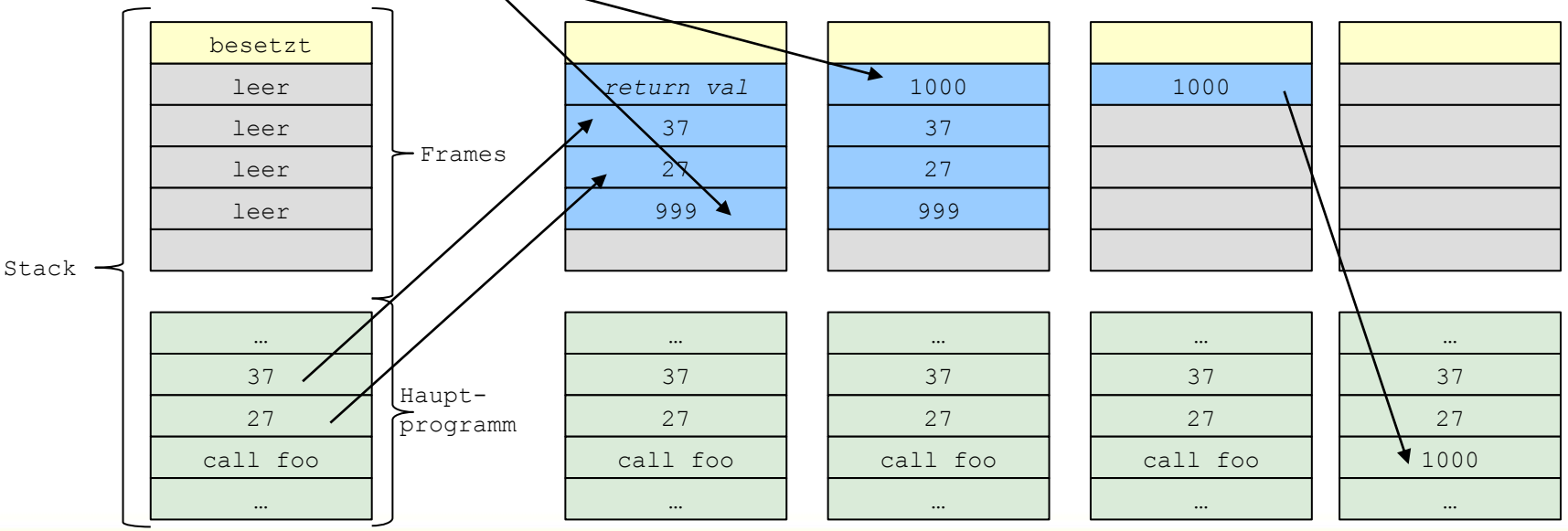
- Zur Laufzeit liegt das Programm irgendwo im flüchtigen Speicher
- Der Speicher ist aufgeteilt in Speicherzellen einer bestimmten (je nach System bspw. 32Bit) Größe
- Jede Zelle hat eine eindeutige Adresse
- Der einem Programm zugteilte Speicherbereich heißt *Programmstack*
- Für jeden Methodenaufruf wird ein gewisser Speicherplatz im Stack reserviert und nach dem Aufruf wieder freigegeben
- Die Größe des angeforderten Speicherbereiches richtet sich nach folgenden Kriterien:
    - Anzahl und Typ der Argumente
    - Lokale Variablen
    - Rückgabewert
- Beispiel: Die **`mult`**-Funktion erwartet zwei Argumente vom Typ **`int`**, keine lokalen Variablen und liefert einen Wert ebenfalls vom Typ **`int`**
- Es werden also mindestens 3 Speicherzellen für den Aufruf benötigt

Speichermangement (Stack)

- Beispiel (sehr abstrahiert)

```
static int foo(int n, int m) {
     int temp = n * m;

     return temp + 1;
}
```



Speichermangement (Heap)

- Lokale Variablen überleben den Aufruf einer Methode nicht
- Was wenn man Daten innerhalb einer Methode erzeugen möchte, die auch nach dem Aufruf noch leben, allerdings nicht als Rückgabewert dienen sollen
- Für ein solches Speichermanagement ist der Stack ungeeignet
- Neben dem Stack existiert noch der sogenannte Heap
  - Der Speicher ist nicht wie der Stack sequentiell aufgebaut
  - Jede beliebige Speicherzelle kann gelesen und beschrieben werden
  - Daten, die in diesem Speicher abgelegt werden, leben losgelöst von Methoden solange bis sie explizit wieder gelöscht werden

Der **new**-Operator und Zeiger

- Im Umgang mit Arrays wurde der **new**-Operator vorgestellt:

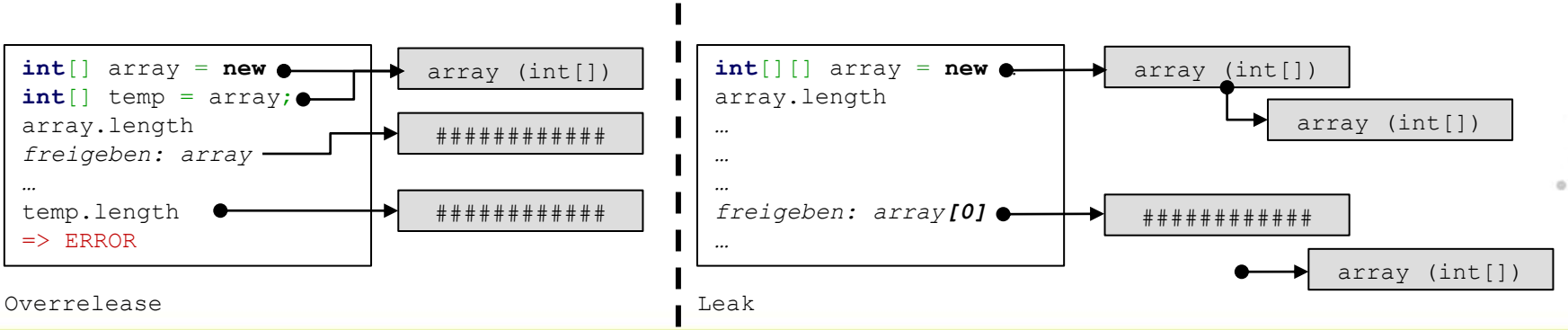
```
int[] array = new int[10];
```

- Dieser ist kein Array-spezifischer Operator, sondern veranlasst, dass Speicher im Heap für ein bestimmtes Datenobjekt angelegt werden soll
- **new** kann nur in Verbindung mit Objekten benutzt werden (siehe OOP)
- D.h. insbesondere, dass Daten eines primitiven Typs wie **int**, **float** oder **boolean** nicht per **new** erzeugt werden können
- Als Rückgabe des **new**-Operators erhält man einen *Zeiger (Pointer)* auf den reservierten Speicherbereich
- Zusammen mit dem **.**-Operator, kann auf das dort abgelegte Objekt zugegriffen werden
- Ohne Zeiger kein Zugriff!
- Bspw. um Attribute des Objektes abzufragen, oder Methoden aufzurufen:

```
int[] array = new …
if (array.length > 0)System.out.println(array.toString());
```

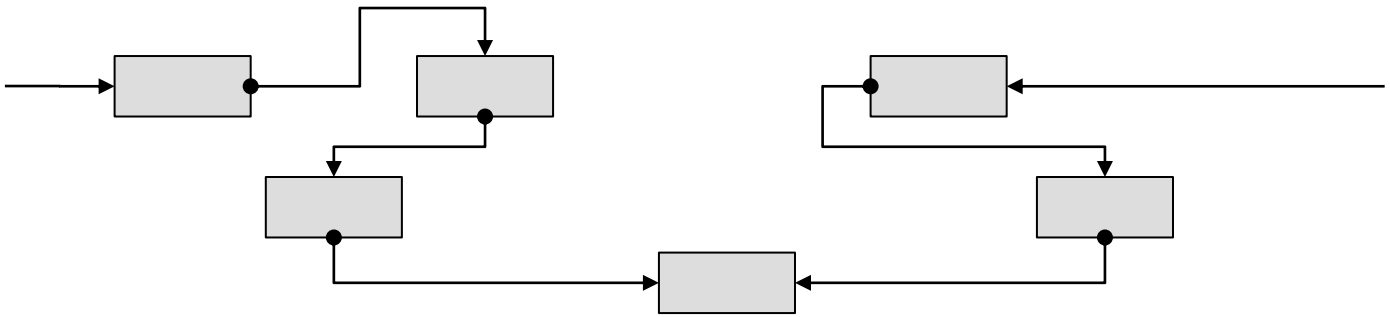
Speichermanagement (Leak/Overrelease)

- Die Ressourcen eines Heaps sind naturgemäß begrenzt
- Mit **new** reservierter Speicherbereich, der nicht mehr genutzt wird, sollte daher wieder freigegeben werden
- Dabei kann es zu zwei möglichen Problemen kommen:
  - Wird ein Speicherbereich zu früh freigegeben, ist das Verhalten beim nächsten Zugriff undefiniert (genannt: *Overrelease*)
  - Wenn ein Speicherbereich freigegeben wird, der seinerseits wieder auf einen anderen Bereich verweist, diesen aber nicht freigibt, wird dieser niemals wieder freigegeben (genannt: *Leak*)



Speichermanagement (Garbage Collection)

- In vielen „älteren“ Programmiersprachen war es Aufgabe des Programmierers sich um den Management des Heap-Speicher zu kümmern
- Jedes mit **`new`** erzeugte Datenobjekte muss durch einen entsprechenden inversen Befehl (**`destroy`**, **`release`**, o.Ä) wieder zerstört werden
- In Java gibt keinen Befehl zum Freigaben von Speicher
- Das JRE überprüft zyklisch und eigenständig den Heap auf Datenobjekte, die nicht mehr benötigt werden
- Wird ein solches Objekt gefunden, wird es entfernt
- Dieses Verfahren wird als *Garbage Collection* bezeichnet



Strings

- Obwohl es den Anschein erregen mag, ist **`String`** in Java *kein* primitiver Datentypen
- Als Literal angeben wird eine Zeichenketten wie `"Hello World"` zu einem Objekt auf dem Heap
- String ist also nach Array der zweite vorgestellte „zusammengesetzte Datentyp“
- Vielmehr noch ist **`String`** eine eigene Klasse (siehe OOP)
- **`String`** ist der einzige nicht numerische und nicht primitive Datentyp für den der Operator **`+`** definiert ist:

```
String hello = "Hello";
String world = "World";

System.out.println(hello + " " + world);
→ Hello World
```

- Der Operator kann beliebig oft hintereinander geschaltet werden und liefert als Ergebnis die Konkatenation der einzelnen Strings

Strings

- Aufgepasst bei der Erzeugung:

```
String string1, string2 = string1 = new String("Hello World");
String string3, string4 = string3 = "Hello World";

System.out.println(string1 == string2);                    → false
System.out.println(string1.equals(string2));               → true
System.out.println(string3 == string4);                    → true
System.out.println(string3.equals(string4));               → true
```

- Strings, die mit mit **new** erzeugt werden, liefern immer ein neues Objekt
- Strings, die per Literal erzeugt werden, liefern nur ein neues Objekt, wenn nicht schon ein String per Literal mit gleichen Wert erzeugt wurde!

- *Coercion* bei der Konkatenation:
  - Ist ein Argument des **+** Operators vom Typ **String**, so wandelt der Compiler automatisch alle anderen Argumente zu String um

```
int intValue1 = 27;
int intValue2 = 37;
System.out.println(intValue1 + " * " + intValue2 + " + 1 = " + (intValue1 * intValue2 + 1));
→ 27 * 37 + 1 = 1000
```

String-Operationen

- Test ob ein String einen bestimmten Teil-String enthält

```
String string = "Hello World";
if (string.contains("Hello"))
            System.out.println("\"Hello\" gefunden");
```

  - Um das Zeichen " innerhalb eines String verwenden zu können, muss dies mit einem vorangestellten Backslash „escaped“ werden

- Test ob Teil-String am Anfang oder Ende

```
if (string.startsWith("Hello")) System.out.println(string + " startet mit \"Hello\"");
if (string.endsWith("World")) System.out.println(string + " ended mit \"World\"");
```

- Umwandeln in Klein- bzw. Großbuchstaben:

```
System.out.println(string.toUpperCase()); → "HELLO WORLD"
System.out.println(string.toLowerCase()); → "hello world"
```

- Länge eines Strings:

```
String string = "Hello World";
if (string.length() == 0) System.out.println ("Der String ist leer");
```

String-Operationen

- Index eines Teil-Strings ermitteln:

```
String string = "Hello World, Hello World";

System.out.println(string.indexOf("World"));          → 6
System.out.println(string.lastIndexOf("World"));      → 19
```

- Teil-String kopieren:

```
String string = "Hello World, Hello World";

int index = string.indexOf("World");
int lastindex = string.lastIndexOf("World");
System.out.println(string.substring(index, lastindex));     → World, Hello
```

- Teil-Strings ersetzen:

```
String string = "Hello World, Hello World";

String newString = string.replaceAll("World", "Java");
System.out.println(newString);                              → Hello Java, Hello Java
```

  - **`replaceAll`** liefert einen neuen String und ändert nicht den alten!

String-Operationen

- Einen String aufsplitten:

```
String string = "Hello World, Hello World";

String[] fragments = string.split(" ");
for (String fragment : fragments) System.out.println(fragment);

→ Hello
→ World,
→ Hello
→ World
```

  - Achtung: **`replaceAll`** und **`split`** erwarten nicht wie es hier den Anschein erwecken mag einen einfachen String als erstes Argument, sondern einen regulären Ausdruck: http://tinyurl.com/boh9atc

- Unnötige *Whitespaces* am Anfang und Ende entfernen:

```
String string = "          Hello World          ";

System.out.println(string);                    →           Hello World
System.out.println(string.trim());             → Hello World
```

Aufgabe

- Es soll ein Taschenrechner geschrieben werden, der die Operationen +, -, *, /, % auf dem Wertebereich **int** ausführen kann
- Die Eingabe findet dabei „interaktiv“ über die Konsole statt
- Der Rechner soll eine Eingabe analysieren und auswerten (der Rechner ist also ein Interpreter!)
- Der Rechner soll über die Möglichkeit verfügen Variablen mit einem Initialwert anzulegen
- Name und Wert sollen dafür als String kodiert in einem Array gespeichert werden (dieser Array sei als „Kontext“ bezeichnet)
- Definierte Variablen sollen wieder aus dem Kontext löschbar sein
- Ein Code-Rahmen für den Einstieg wird bereitgestellt

- Anlegen einer Variablen: **`assign VARIABLENNAME VARIABLENWERT`**
- Löschen einer Variablen: **`remove VARIABLENNAME`**
- Rechnen: **`add|sub|mult|div|mod WERT_A|VAR WERT_B|VAR`**
- Abbrechen: **`exit`**