

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

Dr. Stefan Dissmann  
Fakultät für Informatik



## Lernziele

Nach erfolgreichem Abschluss des Moduls DAP 1 sollen die teilnehmenden Studierenden

- Programme in der Programmiersprache Java implementieren können,
- Java-Klassen und -Bibliotheken nutzen können,
- Java-Klassen und -Bibliotheken implementieren können,
- einige wichtige Algorithmen mit ihren Implementierungen kennen,
- einige wichtige Datenstrukturen mit ihren Implementierungen kennen,
- eigene Algorithmen konzipieren und implementieren können,
- eigene Datenstrukturen konzipieren und implementieren können,
- Algorithmen und Datenstrukturen bewerten können,
- objektorientierte Mechanismen anwenden können,
- einige Entwurfsmuster einsetzen können,
- einfache Ideen der Softwaretechnik einsetzen können.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 1.1. Einführung – Suche und Suchalgorithmen

Dr. Stefan Dissmann  
Fakultät für Informatik



## Lernziele 1. Einführung

Nach Durcharbeiten der Einführung sollen die teilnehmenden Studierenden

- ein erstes Verständnis von dem Begriff *Datenstruktur* besitzen
- ein erstes Verständnis von dem Begriff *Algorithmus* besitzen
- ein Beispiel für die Beziehung zwischen Datenstrukturen und Algorithmen angeben können
- den Verarbeitungsablauf bei der Erstellung von Java-Programmen kennen
- die Begriffe *Syntax* und *Semantik* unterscheiden und zuordnen können
- Syntaxdiagramme* lesen und entscheiden können, ob ein Wort zu einer gegebenen Sprache gehört
- Syntaxdiagramme für einfache Konstrukte selbst formulieren können
- einfache, nur aus einer *Klasse* bestehende Java-Programme mit ausschließlich *statischen Methoden* implementieren, übersetzen und ausführen lassen können
- Fehler den Phasen *Compilierung* oder *Ausführung* zuordnen können

**DAP**

=

**Datenstrukturen, Algorithmen  
und Programmierung**

## Datenstrukturen und Algorithmen

1. Beispiel für eine Datenstruktur: ungeordnete Tabelle von Matrikelnummern

1: 165321	2: 173048	3: 164213	4: 156736	5: 138916	6: 154701	7: 160200	8: 172727
9: 172002	10: 161823	11: 156807	12: 173559	13: 147644	14: 141902	15: 164671	16: 160097
17: 156941	18: 165411	19: 155467	20: 161634	21: 155593	22: 121993	23: 165997	24: 156019
25: 158904	26: 147219	27: 158603	28: 164340	29: 172409	30: 173011	31: 173105	32: 155711
33: 161901	34: 158815	35: 160644	36: 165781	37: 158485	38: 172523	39: 138321	40: 161202
41: 173114	42: 158731	43: 172230	44: 156739	45: 164478	46: 164510	47: 161361	48: 165401
49: 164093	50: 155319	51: 173191	52: 156702	53: 165782	54: 161422	55: 138005	56: 158216
57: 155245	58: 160791	59: 165903	60: 160519	61: 161517	62: 156541	63: 147008	64: 164733
65: 147358	66: 172196	67: 161776	68: 158514	69: 160427	70: 160371	71: 172692	72: 173671
73: 160817	74: 164181	75: 158604	76: 172341	77: 173801	78: 165443	79: 158332	80: 156113

## Datenstrukturen und Algorithmen

(Fortsetzung)

1. Beispiel für eine Datenstruktur: ungeordnete Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **166392**)

1: 165321	2: 173048	3: 164213	4: 156736	5: 138916	6: 154701	7: 160200	8: 172727
9: 172002	10: 161823	11: 156807	12: 173559	13: 147644	14: 141902	15: 164671	16: 160097
17: 156941	18: 165411	19: 155467	20: 161634	21: 155593	22: 121993	23: 165997	24: 156019
25: 158904	26: 147219	27: 158603	28: 164340	29: 172409	30: 173011	31: 173105	32: 155711
33: 161901	34: 158815	35: 160644	36: 165781	37: 158485	38: 172523	39: 138321	40: 161202
41: 173114	42: 158731	43: 172230	44: 156739	45: 164478	46: 164510	47: 161361	48: 165401
49: 164093	50: 155319	51: 173191	52: 156702	53: 165782	54: 161422	55: 138005	56: 158216
57: 155245	58: 160791	59: 165903	60: 160519	61: 161517	62: 156541	63: 147008	64: 164733
65: 147358	66: 172196	67: 161776	68: 158514	69: 160427	70: 160371	71: 172692	72: 173671
73: 160817	74: 164181	75: 158604	76: 172341	77: 173801	78: 165443	79: 158332	80: 156113

## Datenstrukturen und Algorithmen

(Fortsetzung)

1. Beispiel für eine Datenstruktur: ungeordnete Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **166392**)

**Algorithmus:** Überprüfe nacheinander solange Einträge, bis die Nummer gefunden wurde.

1: 165321	2: 173048	3: 164213	4: 156736	5: 138916	6: 154701	7: 160200	8: 172727
9: 172002	10: 161823	11: 156807	12: 173559	13: 147644	14: 141902	15: 164671	16: 160097
17: 156941	18: 165411	19: 155467	20: 161634	21: 155593	22: 121993	23: 165997	24: 156019
25: 158904	26: 147219	27: 158603	28: 164340	29: 172409	30: 173011	31: 173105	32: 155711
33: 161901	34: 158815	35: 160644	36: 165781	37: 158485	38: 172523	39: 138321	40: 161202
41: 173114	42: 158731	43: 172230	44: 156739	45: 164478	46: 164510	47: 161361	48: 165401
49: 164093	50: 155319	51: 173191	52: 156702	53: 165782	54: 161422	55: 138005	56: 158216
57: 155245	58: 160791	59: 165903	60: 160519	61: 161517	62: 156541	63: 147008	64: 164733
65: 147358	66: 172196	67: 161776	68: 158514	69: 160427	70: 160371	71: 172692	72: 173671
73: 160817	74: 164181	75: 158604	76: 172341	77: 173801	78: 165443	79: 158332	80: 156113

## Datenstrukturen und Algorithmen

(Fortsetzung)

1. Beispiel für eine Datenstruktur: ungeordnete Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **166392**)

**Algorithmus:** Überprüfe nacheinander solange Einträge, bis die Nummer gefunden wurde.

1. 165321	2. 173048	3. 164213	4. 156736	5. 138916	6. 154701	7. 160200	8. 172727
9. 172002	10. 161823	11. 156807	12. 173559	13. 147644	14. 141902	15. 164671	16. 160097
17. 156941	18. 165411	19. 155467	20. 161634	21. 155593	22. 121993	23. 165997	24. 156019
25. 158904	26. 147219	27. 158603	28. 164340	29. 172409	30. 173011	31. 173105	32. 155711
33. 161901	34. 158815	35. 160644	36. 165781	37. 158485	38. 172523	39. 138321	40. 161202
41. 173114	42. 158731	43. 172230	44. 156739	45. 164478	46. 164510	47. 161361	48. 165401
49. 164093	50. 155319	51. 173191	52. 156702	53. 165782	54. 161422	55. 138005	56. 158216
57. 155245	58. 160791	59. 165903	60. 160519	61. 161517	62. 156541	63. 147008	64. 164733
65. 147358	66. 172196	67. 161776	68. 158514	69. 160427	70. 160371	71. 172692	72. 173671
73. 160817	74. 164181	75. 158604	76. 172341	77. 173801	78. 165443	79. 158332	80. 156113

166392 ist nicht enthalten!

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: 156200	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: 165902	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: 156200	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: 165902	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**bekannter Algorithmus:** Überprüfe nacheinander Einträge, bis die Nummer gefunden wurde.

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: 156200	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 161127
57: 164130	58: 165291	59: 165296	60: 165304	61: 165902	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

Diese sequentielle Suche nutzt nicht die Sortierung!

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**intuitiver Algorithmus:** Suche zunächst die passende Zeile und dort nach der Nummer.

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: 156200	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: 165902	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**intuitiver Algorithmus:** Suche zunächst die passende Zeile und dort nach der Nummer.

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: 156200	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: 165902	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

Diese Suche nutzt die Sortierung!

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus: Suche weiter in der passenden Hälfte des zuletzt betrachteten Abschnitts.**

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: 156200	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: 165902	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus: Suche weiter in der passenden Hälfte des zuletzt betrachteten Bereichs.**

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: 156200	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: 165902	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus:** Suche weiter in der *passenden Hälften des zuletzt betrachteten Bereichs.*

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: <b>156200</b>	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: 165902	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

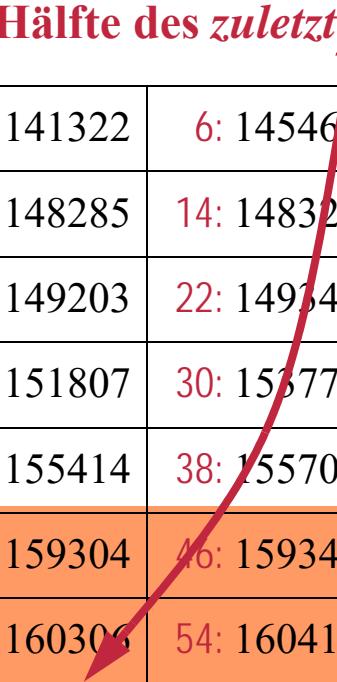
## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus:** Suche weiter in der *passenden* Hälfte des zuletzt betrachteten Bereichs.



1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: <b>156200</b>	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: <b>165902</b>	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus:** Suche weiter in der *passenden Hälften des zuletzt betrachteten Bereichs.*

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: <b>156200</b>	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: <b>165902</b>	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: 172351	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus:** Suche weiter in der *passenden* Hälfte des zuletzt betrachteten Bereichs.



1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: <b>156200</b>	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: <b>165902</b>	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: <b>172351</b>	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus:** Suche weiter in der *passenden Hälften des zuletzt betrachteten Bereichs.*

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: <b>156200</b>	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: <b>165902</b>	62: 165936	63: 165939	64: 165967
65: 170412	66: 170567	67: 170581	68: 171040	69: 172231	70: 172265	71: <b>172351</b>	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus:** Suche weiter in der *passenden Hälften des zuletzt betrachteten Bereichs.*

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: <b>156200</b>	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: <b>165902</b>	62: 165936	63: 165939	64: 165967
65: 170412	66: <b>170567</b>	67: 170581	68: 171040	69: 172231	70: 172265	71: <b>172351</b>	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

## Datenstrukturen und Algorithmen

(Fortsetzung)

2. Beispiel für eine Datenstruktur: **geordnete** Tabelle von Matrikelnummern

Aufgabe: Stelle fest, ob eine vorgegebene Nummer enthalten ist? (z.B. **170567**)

**weiterer Algorithmus: Suche weiter in der passenden Hälfte des zuletzt betrachteten Bereichs.**

1: 138749	2: 141013	3: 141082	4: 141092	5: 141322	6: 145461	7: 145532	8: 145671
9: 145981	10: 145982	11: 145990	12: 148273	13: 148285	14: 148321	15: 148451	16: 148469
17: 149020	18: 149021	19: 149114	20: 149145	21: 149203	22: 149341	23: 149415	24: 149441
25: 149709	26: 150106	27: 151445	28: 151491	29: 151807	30: 153771	31: 153901	32: 154122
33: 154136	34: 154209	35: 155241	36: 155351	37: 155414	38: 155701	39: 156106	40: 156109
41: <b>156200</b>	42: 156234	43: 156521	44: 156760	45: 159304	46: 159346	47: 159471	48: 159564
49: 159573	50: 159580	51: 160224	52: 160227	53: 160306	54: 160412	55: 160423	56: 164127
57: 164130	58: 165291	59: 165296	60: 165304	61: <b>165902</b>	62: 165936	63: 165939	64: 165967
65: 170412	66: <b>170567</b>	67: 170581	68: 171040	69: 172231	70: 172265	71: <b>172351</b>	72: 172396
73: 172457	74: 172471	75: 173100	76: 173204	77: 173217	78: 173244	79: 173501	80: 173785

# Datenstrukturen und Algorithmen

(Fortsetzung)

## erste Erkenntnisse:

- Die Gestaltung von Datenstrukturen und Algorithmen hängt zusammen.
- Die Gestaltung von Datenstrukturen und Algorithmen hat nichts mit Programmieren zu tun.
- Der zuletzt vorgestellte Algorithmus heißt *binäre Suche*, da der zu durchsuchende Bereich in jedem Schritt halbiert wird, also in zwei Teile zerlegt wird.
- Hat eine Datenstruktur bestimmte Eigenschaften, können Algorithmen diese nutzen.  
Im Beispiel:

Die binäre Suche nutzt die vorhandene Sortierung, um schneller zu einem Ergebnis zu kommen:  
– Es werden nur 4 Vergleiche bis zum Finden benötigt.  
– Die sequentielle Suche erfordert demgegenüber 66 Vergleiche.

- Allerdings hat das Erstellen der sortierten Datenstruktur Aufwand verursacht.  
Bei der Konzeption von Datenstrukturen und darauf arbeitenden Algorithmen muss daher der Gesamtaufwand betrachtet werden.
- Die binäre Suche setzt zudem voraus:
  - eine Möglichkeit zum Berechnen der gewünschten Position und
  - den unmittelbaren Zugriff auf den Wert an einer Position.

# Datenstrukturen und Algorithmen

(Fortsetzung)

## Algorithmus: Binäre Suche – Versuch einer Präzisierung

- gegeben sind
  - eine sortierte Folge von Einträgen
  - der zu suchende Wert  $w_s$
- bestimme den Wert  $w_m$  des Eintrags an der *jeweils mittleren* Position
- vergleiche  $w_s$  mit  $w_m$ 
  - falls  $w_s = w_m$ :  $w_s$  kommt in der Folge vor
  - falls  $w_s > w_m$ : wende das Vorgehen auf die *Hälfte* mit den größeren Werten an
  - falls  $w_s < w_m$ : wende das Vorgehen auf die *Hälfte* mit den kleineren Werten an
- wiederhole das Vorgehen so oft, bis
  - $w_s$  gefunden wurde oder
  - die zu betrachtende Hälfte keine Einträge enthält:  $w_s$  kommt nicht in der Folge vor

# Datenstrukturen und Algorithmen

(Fortsetzung)

## Algorithmus: Binäre Suche – Analyse

- Wie viele Vergleiche sind höchstens notwendig?  
entspricht: Wie häufig können Hälften wieder halbiert werden?

Annahme: Es liegen  $n$  Werte vor.

Fragestellung: Wie häufig kann  $n$  durch 2 dividiert werden, bis das Ergebnis 1 ist.  
entspricht: Wie oft muss 2 mit sich selbst multipliziert werden,  
bis  $n$  erreicht ist.

entspricht:  $2^x = n$ , also  $x = \log_2(n)$

- Beispiel: Es liegen 1.000.000 Werte vor,  
dann ist  $\log_2(1.000.000) \approx 20$ .  
Es werden maximal 21 Vergleiche benötigt, um festzustellen,  
ob ein gesuchter Wert vorkommt.

Zum Vergleich: Bei der sequentiellen Suche mit dem Ansehen aller Werte  
werden maximal  $n = 1.000.000$  Vergleiche benötigt.

# Datenstrukturen und Algorithmen

(Fortsetzung)

*Die Betrachtung von Datenstrukturen und Algorithmen  
ist ein zentraler Bestandteil der Informatik.*

aber:

Der Ablauf der binären Suche kann verbal nur ungenau beschrieben werden  
(- wie auch die vorangehenden Folien zeigen).

Es wird eine geeignete Notation zur Beschreibung von Algorithmen benötigt:

**Programmiersprache**

## Aufgaben von Programmiersprachen



Programmiersprache ist Hilfsmittel für die Kommunikation zwischen Mensch und Maschine.

### für Menschen

- mit vernünftigem Aufwand erlernbar
- nach Lernphase leicht lesbar und schreibbar
- erlaubt es, vollständige und präzise Handlungsanweisungen zu formulieren

### für Rechner

- ausführbar mit einer definierten Bedeutung
- ausführbar mit ökonomisch vertretbarem Aufwand

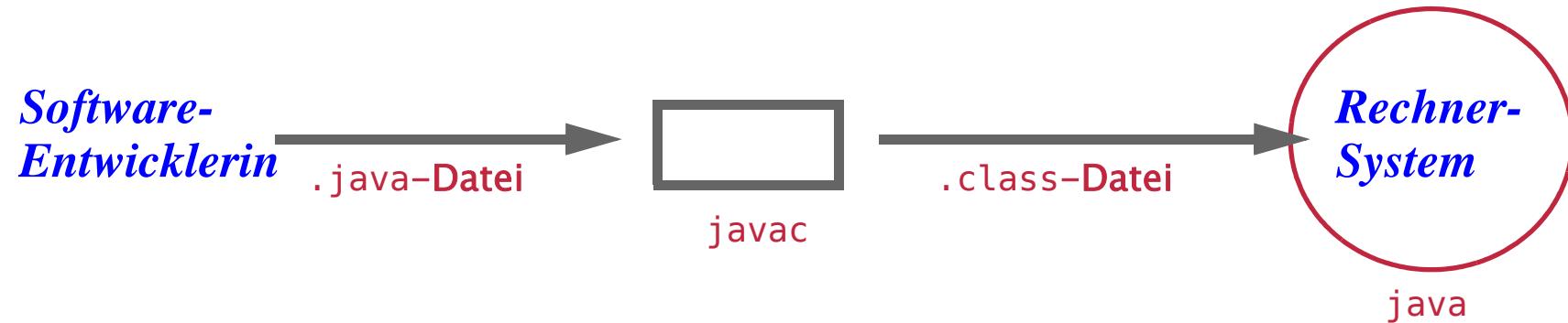
## Einsatz von Programmiersprachen



### Compiler

- ist selbst ein Programm, das auf einem Rechner ausgeführt wird,
- einen Text in einer Programmiersprache als Eingabe nimmt und
- einen Text in Maschinensprache als Ausgabe erzeugt,  
der auf Rechnern ausgeführt werden kann.
- Problem: Für eine Programmiersprache wird  
für jede Maschinensprache  
ein eigener Compiler benötigt.

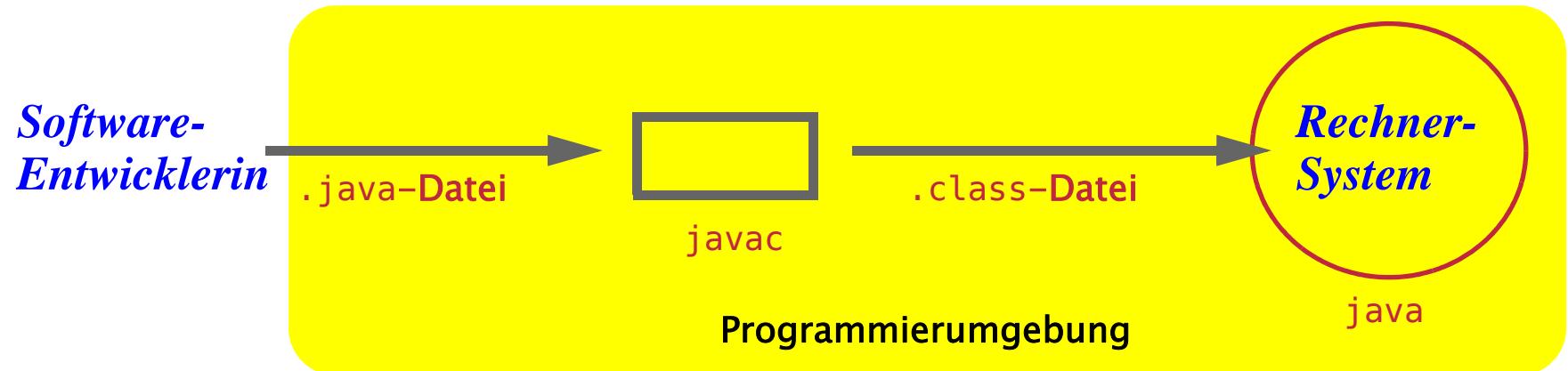
## Einsatz der Programmiersprache Java



- ❑ Java-Compiler: `javac`, ist Teil des Java Development Kit (JDK)
- ❑ erzeugt Datei mit Bytecode-Programm,
- ❑ das von der Virtuellen Maschine (`java`) ausgeführt wird,  
die Teil der Java Runtime Environment (JRE) ist.
  
- ❑ Vorteil: Bytecode-Programm ist auf jedem Rechner ausführbar,  
für den es eine Virtuelle Maschine gibt.
- ❑ Voraussetzung: Virtuelle Maschine darf nur wenige Anforderungen stellen,  
um auf vielen Rechnern realisiert werden zu können.

## Einsatz der Programmiersprache Java

(Fortsetzung)

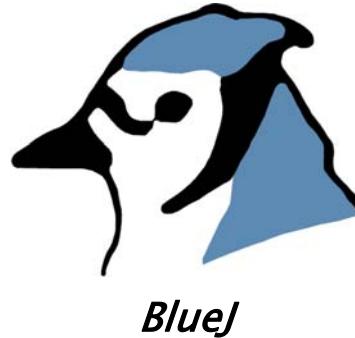


- Eine Programmierumgebung (Integrated Development Environment, IDE) ermöglicht das  
Eingeben, Übersetzen und Ausführen von Java-Programmen.
- professionelle Programmierumgebungen:  
Netbeans, Eclipse
- einfache Programmierumgebung für Java:  
**BlueJ**

## Einsatz der Programmiersprache Java

(Fortsetzung)

*Software-  
Entwicklerin*



DAP 1 ist eine Lehrveranstaltung:

Es interessiert nur die Entwicklung,  
nicht die Nutzung, die Pflege oder die Verbreitung  
der entwickelten Programme.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 1.2. Einführung – Sprachen in der Informatik

Dr. Stefan Dissmann  
Fakultät für Informatik



## Grammatik

Grammatik (im allgemeinen Sprachgebrauch)

ist die systematische Beschreibung der *Syntax* einer *Sprache*,  
also der Konstruktionsvorschriften für die zur Sprache gehörenden Zeichenfolgen.

*Grammatik* (Formalismus in der Informatik):

$$G = (\Sigma, N, P, s)$$

**mit**

einem *Alphabet*  $\Sigma$  (Menge der *Terminalsymbole*),  
einer Menge  $N$  von *Nichtterminalsymbolen*,  
einer Menge  $P$  von *Produktionsregeln*,  
einem *Startsymbol*  $s \in N$ .

## Grammatik

(Fortsetzung)

$$G = (\Sigma, N, P, s)$$

- Eine Produktionsregel  $p \in P$  ersetzt *genau ein* Nichtterminalsymbol  $n \in N$  durch *eine Folge* von Zeichen aus  $\Sigma \cup N$ . \*)
- Ein von  $G$  erzeugtes *Wort*  $w$  ist eine Folge von Terminalsymbolen, die ausgehend vom Startsymbol  $s$  durch das *wiederholte* Anwenden von Produktionsregeln abgeleitet werden kann.
- Die von  $G$  erzeugte *Sprache*  $L(G)$  ist die Menge *aller* Wörter, die von  $G$  erzeugt werden können.
- $G$  dient dazu, die große – meist unendliche – Menge der Wörter einer Sprache  $L(G)$  durch die deutlich kleineren Mengen  $\Sigma$ ,  $N$  und  $P$  zu beschreiben.

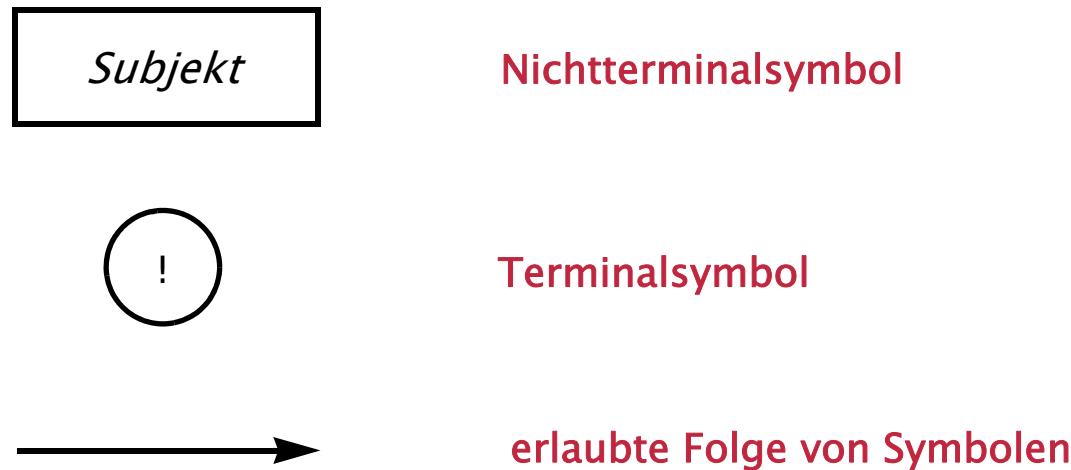
\*) Anmerkung: Durch diese Definition wird  $G$  zu einer *kontextfreien* Grammatik.

## Grammatik

(Fortsetzung)

$$G = (\Sigma, N, P, s)$$

Notation, um Produktionsregeln anzugeben: *Syntaxdiagramm*



## Grammatik

(Fortsetzung)

Beispiel 1:

$$G_1 = (\Sigma_1, N_1, P_1, s_1)$$

mit  $\Sigma_1 = \{ \text{Anna}, \text{Bob}, \text{Chelsea}, \text{kauft}, \text{trifft}, ! \}$

$N_1 = \{ \text{Aussage}, \text{Name}, \text{Aktion} \}$

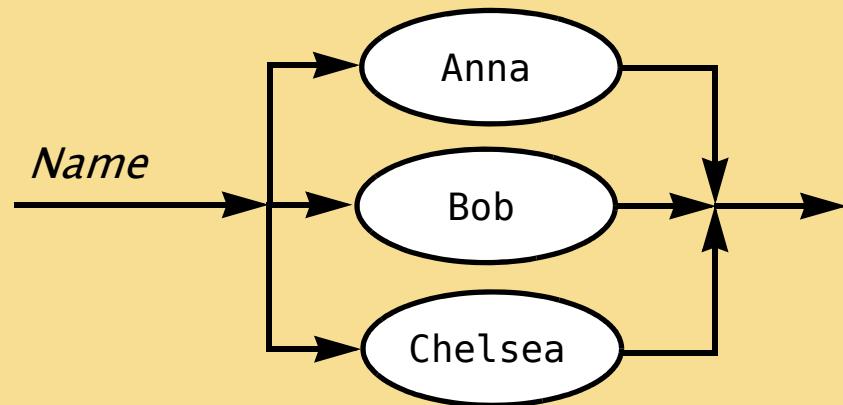
$s_1 = \text{Aussage}$

$P_1$  *siehe Diagramme auf den folgenden Folien*

## Grammatik

(Fortsetzung)

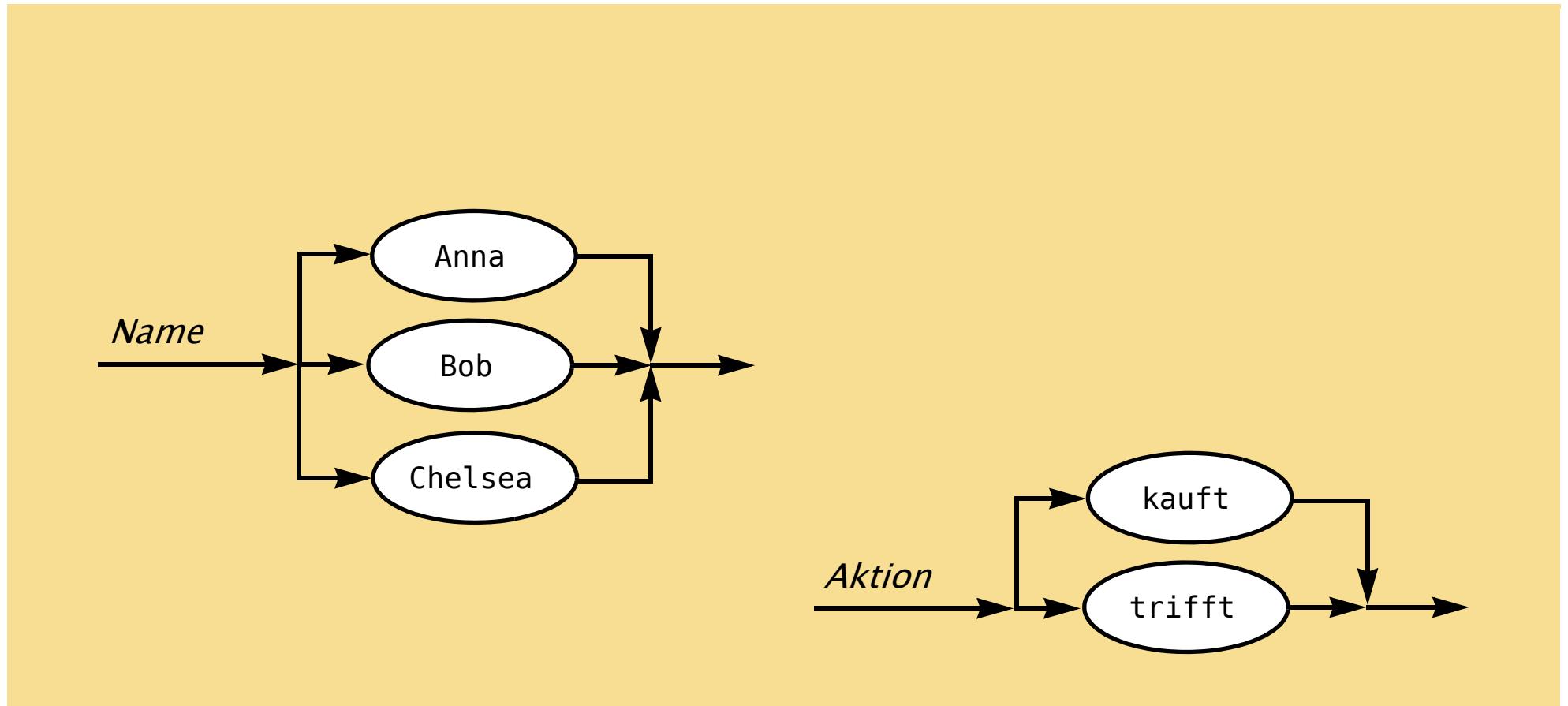
Beispiel 1: Produktionsregeln



## Grammatik

(Fortsetzung)

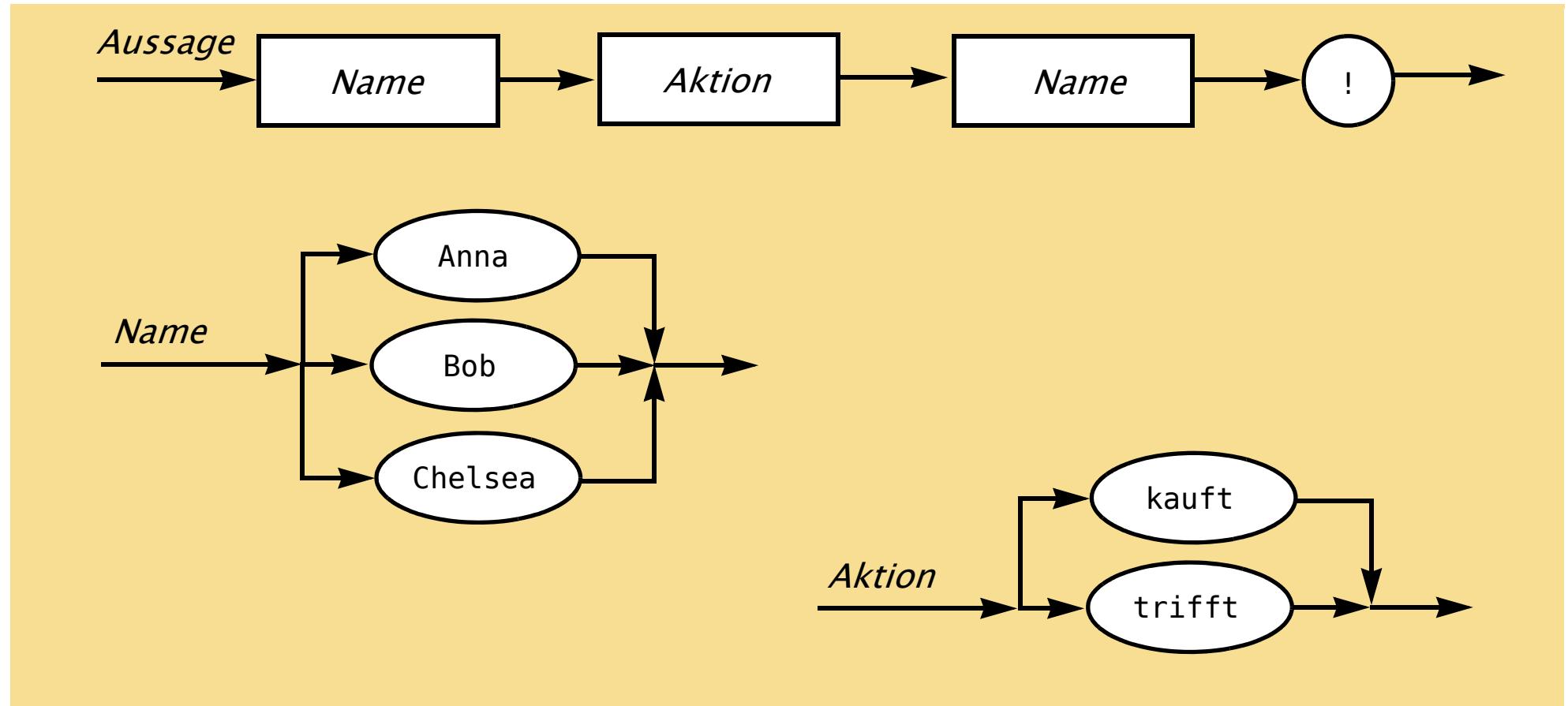
Beispiel 1: Produktionsregeln



## Grammatik

(Fortsetzung)

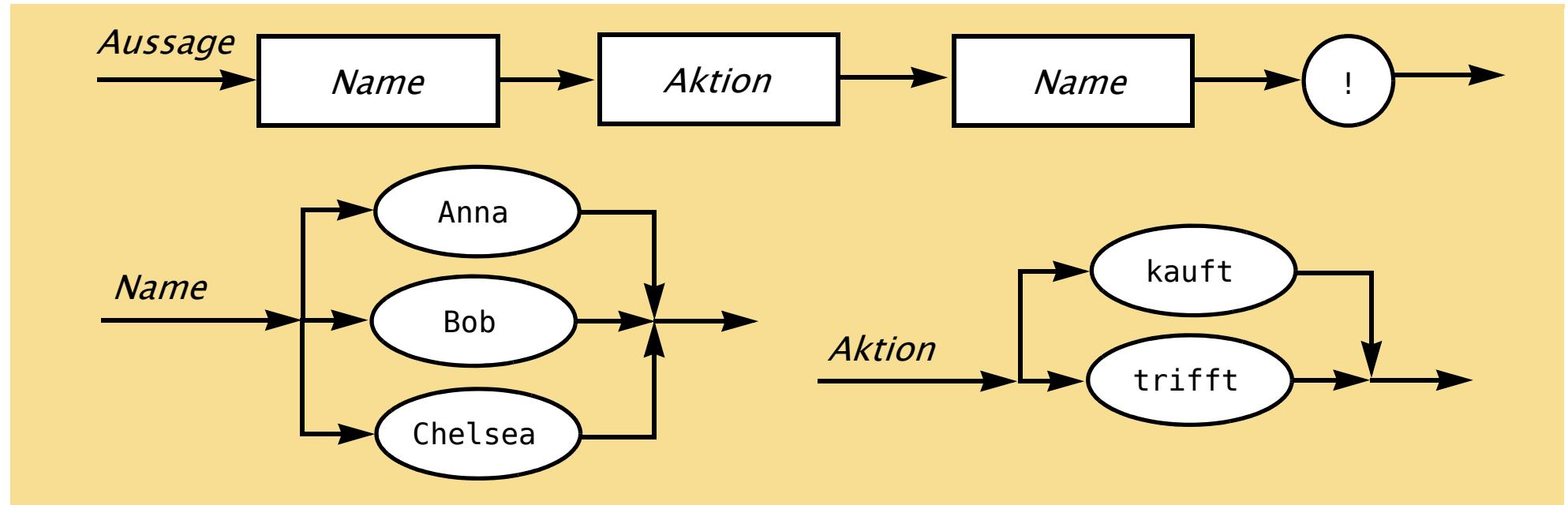
Beispiel 1: Produktionsregeln



## Grammatik

(Fortsetzung)

Beispiel 1: Produktionsregeln



einige Wörter der von dieser Grammatik erzeugten Sprache sind:

Anna kauft Bob!

Anna kauft Anna!

Bob kauft Chelsea!

Bob trifft Chelsea!

Bob trifft Bob!

Bob kauft Anna!

Chelsea kauft Chelsea!

Anna kauft Chelsea!

## Grammatik

(Fortsetzung)

- ❑ Eine Grammatik macht Aussagen zum Aufbau der Wörter der von ihr erzeugten Sprache: *Syntax* der Sprache
- ❑ Eine Grammatik macht *keine* Aussagen zur Bedeutung der erzeugten Wörter: *Semantik* der Sprache
- ❑ Ein syntaktisch korrektes Wort hat nicht immer eine Bedeutung.
- ❑ Die Semantik eines Wortes erschließt sich häufig erst durch den Kontext:
  - Bob kauft Bob!  
hat dann eine sinnvolle Semantik, falls Bob ein Mensch und Bob ein Sportgerät ist.
  - Bob kauft Anna!  
hat dann eine sinnvolle Semantik, falls Bob ein Mensch und Anna ein Wellensittich ist.
  - Bob kauft Chelsea!  
hat keine sinnvolle Semantik, falls Bob ein Sportgerät ist.
- ❑ Wird ein syntaktisch korrektes Wort in einem semantisch falschen Kontext verwendet, so kann ihm keine Bedeutung zugeordnet werden.
- ❑ Ein Wort ist syntaktisch falsch, wenn es nicht durch die Grammatik erzeugt werden kann:  
Bob Anna trifft trifft
- ❑ Ein syntaktisch falsches Wort hat keine Semantik.

## Grammatik

(Fortsetzung)

Beispiel 2:

$$G_2 = (\Sigma_2, N_2, P_2, s_2)$$

mit  $\Sigma_2 = \{ \text{vor!}, \text{langsam}, \text{stopp}, - \}$

$N_2 = \{ \text{Einparken}, \text{Antreiben}, \text{Abbremsen} \}$

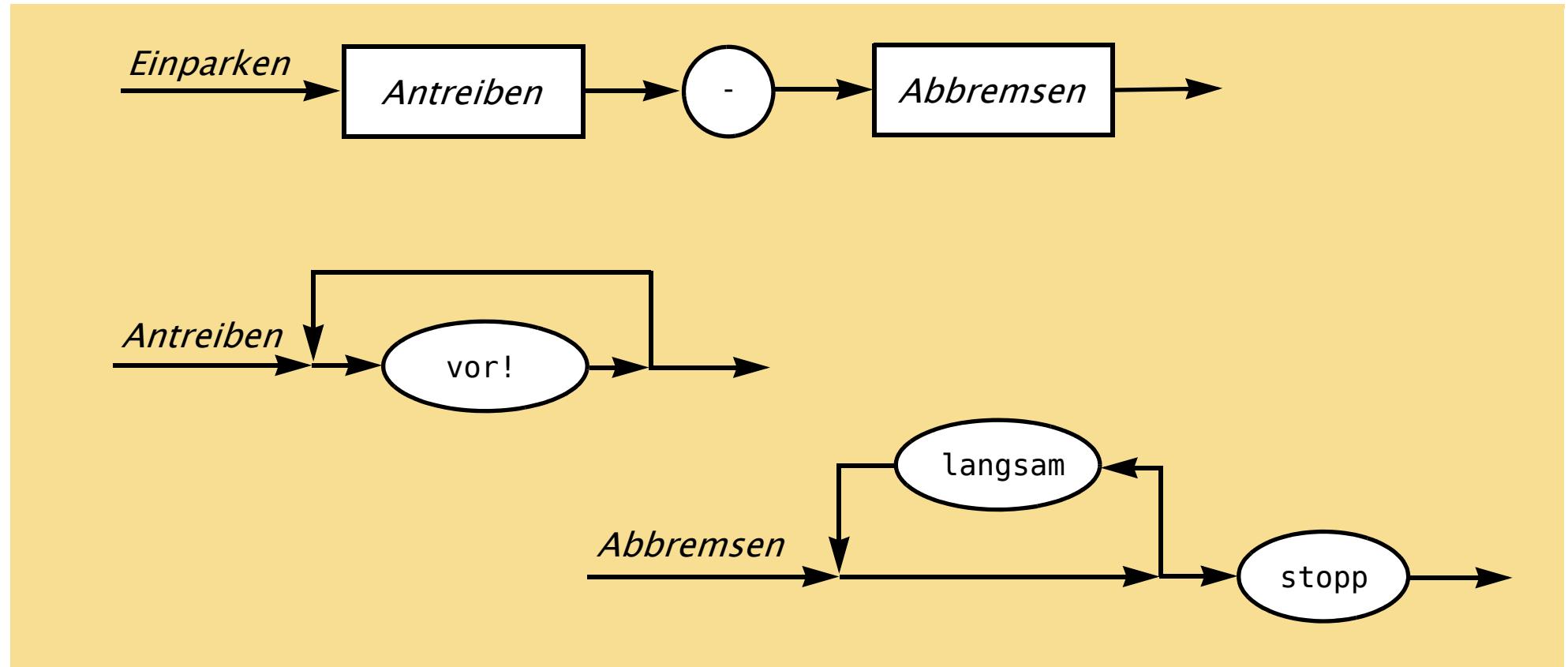
$s_2 = \text{Einparken}$

$P_2$  *siehe Diagramme auf den folgenden Folien*

## Grammatik

(Fortsetzung)

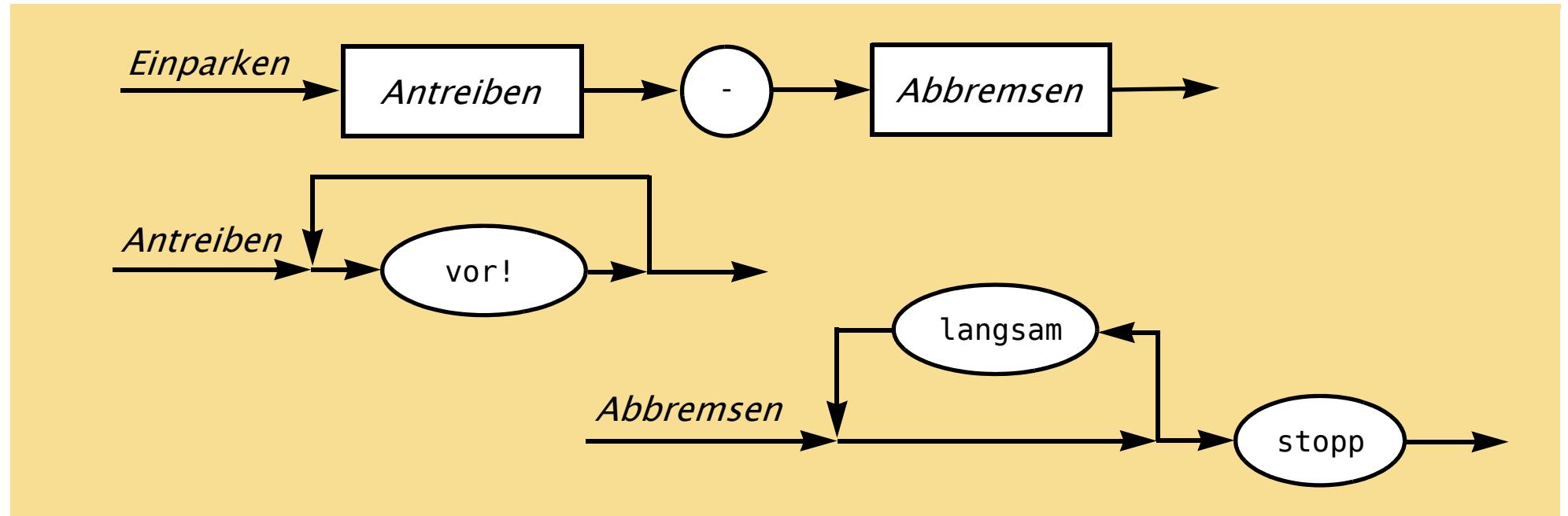
Beispiel 2: Produktionsregeln



## Grammatik

(Fortsetzung)

Beispiel 2: Produktionsregeln



mögliche Wörter der von dieser Grammatik erzeugten Sprache sind:

vor! - stopp

vor! - langsam stopp

vor! vor! vor! - stopp

vor! vor! - langsam langsam stopp

vor! - langsam langsam langsam stopp

## Grammatik

(Fortsetzung)

Beispiel 3:

Gesucht wird eine Grammatik  $G_3 = (\Sigma_3, N_3, P_3, s_3)$

für folgende informelle Beschreibung der gültigen Wörter:

- Ein *Ausdruck* darf die folgenden Zeichen enthalten: **x y +**
- In einem *Ausdruck* dürfen die Zeichen beliebig häufig auftreten.
- Das Zeichen **+** darf nicht am Anfang eines *Ausdrucks* stehen.
- Auf das Zeichen **+** muss immer unmittelbar das Zeichen **x** folgen.

## Grammatik

(Fortsetzung)

Beispiel 3:

Gesucht wird eine Grammatik  $G_3 = (\Sigma_3, N_3, P_3, s_3)$   
für folgende informelle Beschreibung der gültigen Wörter:

- Ein *Ausdruck* darf die folgenden Zeichen enthalten: **x y +**
- In einem *Ausdruck* dürfen die Zeichen beliebig häufig auftreten.
- Das Zeichen **+** darf nicht am Anfang eines *Ausdrucks* stehen.
- Auf das Zeichen **+** muss immer unmittelbar das Zeichen **x** folgen.

$$\Sigma_3 = \{ x, y, + \}$$

## Grammatik

(Fortsetzung)

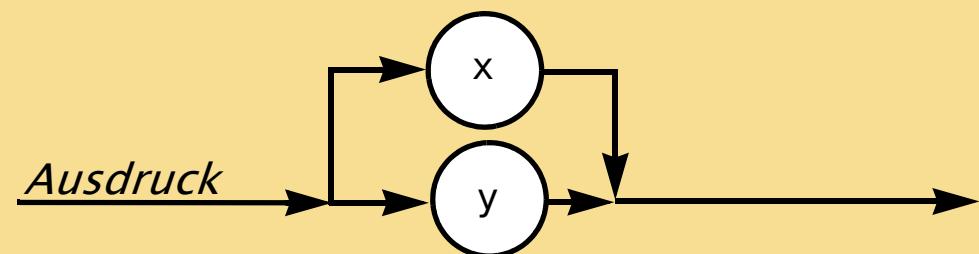
Beispiel 3:

Gesucht wird eine Grammatik  $G_3 = (\Sigma_3, N_3, P_3, s_3)$   
für folgende informelle Beschreibung der gültigen Wörter:

- Ein *Ausdruck* darf die folgenden Zeichen enthalten: **x y +**
- In einem *Ausdruck* dürfen die Zeichen beliebig häufig auftreten.
- Das Zeichen **+** darf nicht am Anfang eines *Ausdrucks* stehen.
- Auf das Zeichen **+** muss immer unmittelbar das Zeichen **x** folgen.

$$\Sigma_3 = \{ x, y, + \}$$

$P_3$



## Grammatik

(Fortsetzung)

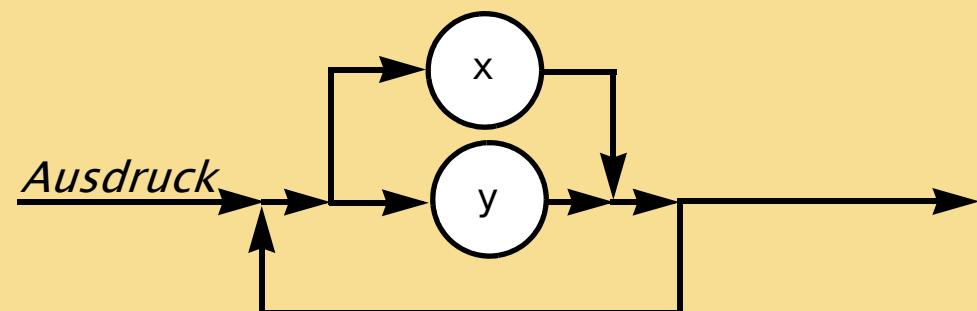
Beispiel 3:

Gesucht wird eine Grammatik  $G_3 = (\Sigma_3, N_3, P_3, s_3)$   
für folgende informelle Beschreibung der gültigen Wörter:

- Ein *Ausdruck* darf die folgenden Zeichen enthalten: **x y +**
- In einem *Ausdruck* dürfen die Zeichen beliebig häufig auftreten.
- Das Zeichen **+** darf nicht am Anfang eines *Ausdrucks* stehen.
- Auf das Zeichen **+** muss immer unmittelbar das Zeichen **x** folgen.

$$\Sigma_3 = \{ x, y, + \}$$

$P_3$



## Grammatik

(Fortsetzung)

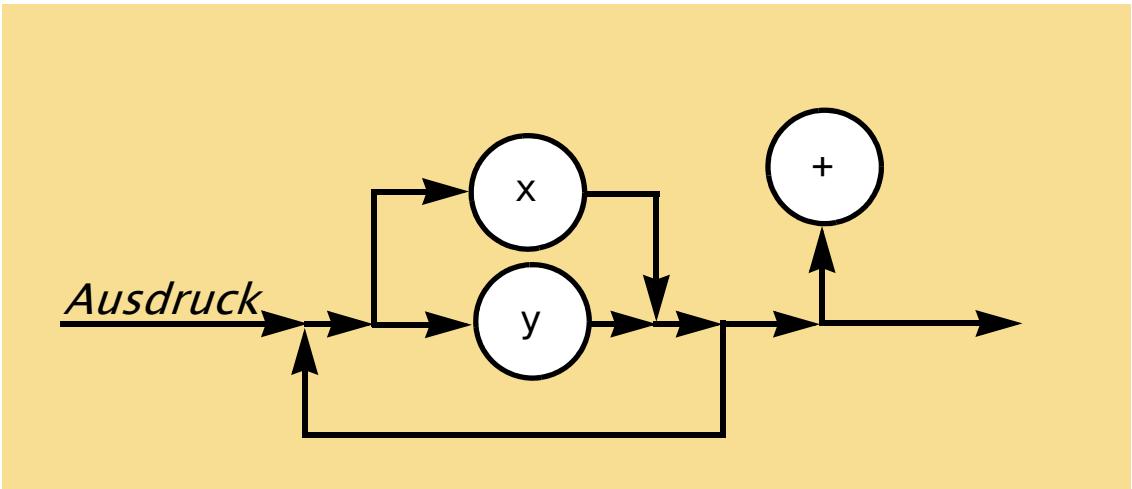
Beispiel 3:

Gesucht wird eine Grammatik  $G_3 = (\Sigma_3, N_3, P_3, s_3)$   
für folgende informelle Beschreibung der gültigen Wörter:

- Ein *Ausdruck* darf die folgenden Zeichen enthalten: **x y +**
- In einem *Ausdruck* dürfen die Zeichen beliebig häufig auftreten.
- Das Zeichen **+** darf nicht am Anfang eines *Ausdrucks* stehen.
- Auf das Zeichen **+** muss immer unmittelbar das Zeichen **x** folgen.

$$\Sigma_3 = \{ x, y, + \}$$

$P_3$



## Grammatik

(Fortsetzung)

Beispiel 3:

Gesucht wird eine Grammatik  $G_3 = (\Sigma_3, N_3, P_3, s_3)$   
für folgende informelle Beschreibung der gültigen Wörter:

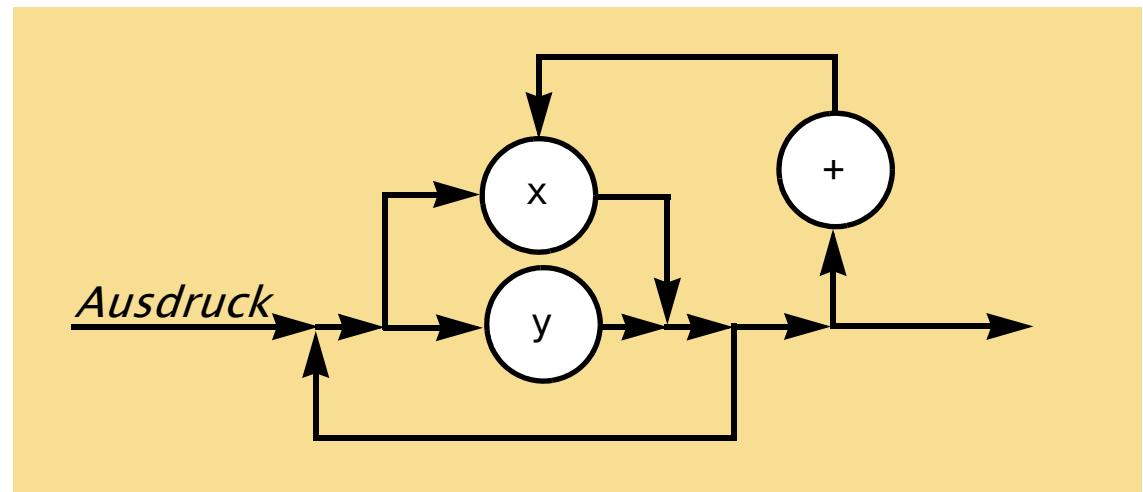
- Ein *Ausdruck* darf die folgenden Zeichen enthalten: **x y +**
- In einem *Ausdruck* dürfen die Zeichen beliebig häufig auftreten.
- Das Zeichen **+** darf nicht am Anfang eines *Ausdrucks* stehen.
- Auf das Zeichen **+** muss immer unmittelbar das Zeichen **x** folgen.

$$\Sigma_3 = \{ x, y, + \}$$

$$N_3 = \{ \textit{Ausdruck} \}$$

$$s_3 = \textit{Ausdruck}$$

$$P_3$$

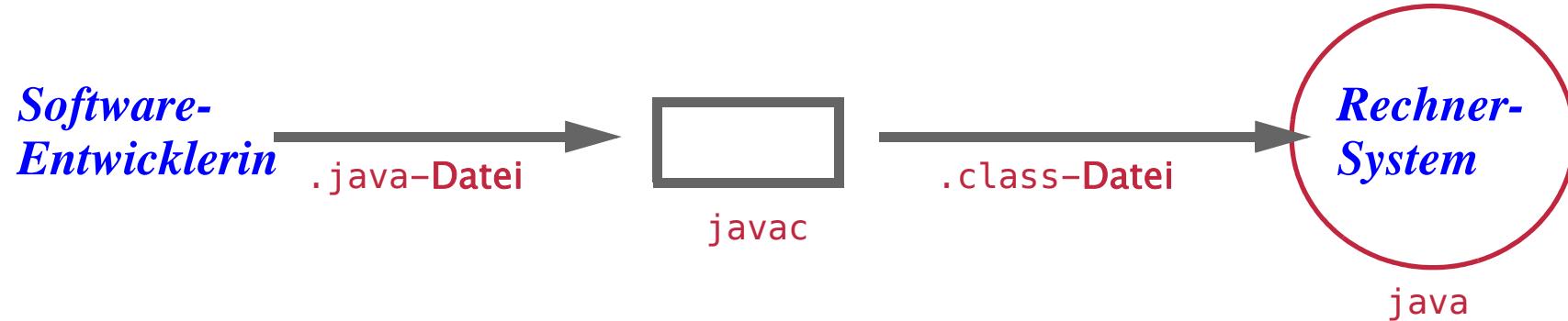


## Grammatik

(Fortsetzung)

- ❑ Syntaxdiagramme werden in dieser Vorlesung dazu benutzt, um den Aufbau der Konstrukte von Java zu verdeutlichen.
- ❑ Die Syntax eines Programms wird immer vom Compiler überprüft, wenn das Programm übersetzt wird.
- ❑ Die Semantik der Konstrukte von Java wird in dieser Vorlesung informell anhand von Beispielprogrammen beschrieben.
- ❑ Die Semantik von Programmen kann der Compiler nur sehr eingeschränkt prüfen:
  - Falls der Compiler Bob und Anna als Menschen kennt, dann kann er für Bob kauft Anna! einen semantischen Fehler melden.
  - Falls der Compiler kein Wissen zu Bob und Anna hat, wird er Bob kauft Anna! auch dann akzeptieren, wenn es sich um einen Fehler der Entwicklerin handelt.

## Einsatz der Programmiersprache Java



Der Java-Compiler prüft Syntax und Semantik strikt und formal.

Konsequenzen:

- Der gleiche Programmtext führt immer zum gleichen Ergebnis des Compilers.
- Syntaktisch falschen Programmtext kann der Compiler nicht in Bytecode übersetzen.
- Syntaktisch falscher Programmtext kann daher nie zu einem ausführbaren Programm führen.
- Der Compiler prüft Semantik nur sehr eingeschränkt.
- Der Compiler kennt die von der Entwicklerin gewünschte Semantik nicht.  
Für den Compiler bestimmt der Java-Programmtext die Semantik.
- Ein compilierbarer Programmtext muss daher nicht die gewünschte Semantik haben.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 1.3. Einführung – Erste Java-Programme

Dr. Stefan Dissmann  
Fakultät für Informatik



## Programmiersprache Java – ein erstes Programm

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

- Dieses Programm tut sehr wenig:

Wird es ausgeführt, zeigt es den Text `hello world` auf dem Bildschirm an:



hello world

- Um das zu erreichen, werden allerdings schon ziemlich viele Konzepte von Java benötigt.

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```



Text-Literal

- "hello world" ist ein Text-Literal:  
eine explizit angegebene Folge von Zeichen.
- Zwischen "..." können fast beliebige Zeichen stehen, die zusammen den Text bilden.
- *aber*: "..." muss in einer Zeile stehen.

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

- ❑ `System.out.println` ist eine Operation, die den Text auf dem Bildschirm anzeigt.
- ❑ In Java wird eine solche Operation als *Methode* bezeichnet.
- ❑ In `( ... )` eingeschlossen ist ein *Argument*, mit dem die Methode arbeitet.
- ❑ In diesem Beispiel ist das Argument der Text `hello World`,  
der als Text-Literal angegeben wird.
- ❑ `System.out.println` ist eine Methode, die Java bereits kennt.

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

Aufruf einer Methode

Terminologie:

- "Das Argument wird übergeben."
- "Die Methode wird ausgeführt."
- "Die Methode wird aufgerufen."

Voraussetzung für Ausführung  
Vorgang des Abarbeitens der Methode  
Der Programmtext sieht das Ausführen der Methode vor.

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

Anweisung

- Eine *Anweisung* ist ein einzeln ausführbarer Arbeitsschritt.
- Eine Anweisung endet (meist) mit dem Zeichen ;
- Eine Anweisung steht immer in einer Methode, die Arbeitsschritte zusammenfasst.
- Hier besteht die Anweisung aus genau dem Aufruf einer Methode.

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

Methodenkopf

bilden zusammen:  
Deklaration einer Methode

Methodenrumpf

- ❑ Methoden können auch selbst angelegt werden:  
Das wird als *Deklaration* einer Methode bezeichnet.
- ❑ `main` ist eine Methode mit dem *Parameter* `args`, die an dieser Stelle deklariert wird.
- ❑ Der Name `main` besitzt in Java eine besondere Bedeutung:  
Mit dieser Methode startet die Ausführung eines Programms.
- ❑ Die Deklaration einer Methode besteht aus der Angabe eines Methodenkopfs und eines Methodenrumpfs in `{...}`
- ❑ Die Wörter `public static void` werden von Java vorgegeben (Terminalsymbole).  
Ihre Bedeutung kann hier noch nicht erklärt werden.

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

← **Klassenkopf**

**bilden zusammen:**  
**Deklaration einer Klasse**

← **Klassenrumpf**

- FirstProgram ist eine *Klasse*.
- Die Deklaration einer Klasse besteht besteht aus der Angabe eines Klassenkopfs und eines Klassenrumpfs in {....}.
- Öffentliche Klassen (**public**) sind eigenständige Programmeinheiten, die in einer Datei mit gleichem Namen gespeichert werden müssen: FirstProgram.java
- Klassen enthalten Deklarationen von Methoden,  
sind also eine Sammlung von Deklarationen ausführbarer Einheiten.

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
/*
 *  author Stefan Dissmann
 *  version 1.0
 */

public class FirstProgram
{
    public static void main( String[] args )
    {
        // show one line
        System.out.println( "hello world" );
    }
}
```

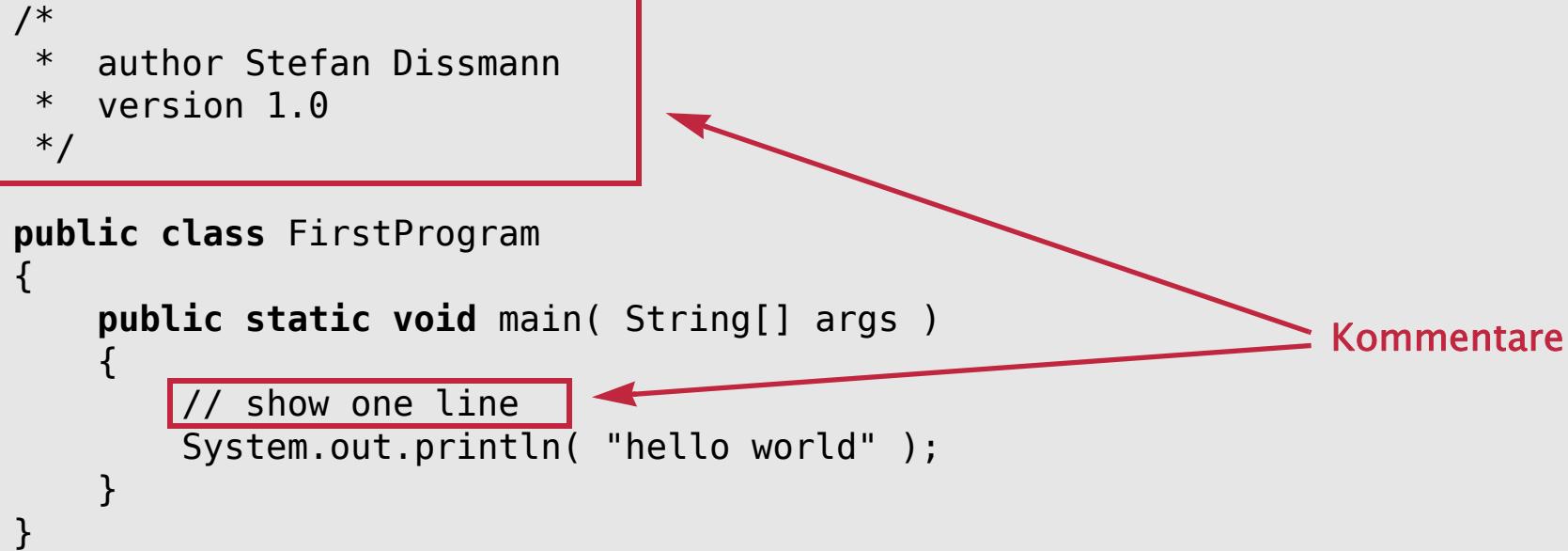
- ❑ *Kommentare* verbessern die Verständlichkeit des Programms für die Entwickler oder enthalten zusätzliche Informationen für die Entwickler.
  
- ❑ Anmerkung: Auch das strukturierte Einrücken der verschiedenen Bestandteile des Programms dient der Verbesserung der Lesbarkeit.

# Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
/*
 *  author Stefan Dissmann
 *  version 1.0
 */

public class FirstProgram
{
    public static void main( String[] args )
    {
        // show one line
        System.out.println( "hello world" );
    }
}
```



Kommentare

- Ein *Kommentar* hat keine Bedeutung für die Übersetzung und Ausführung des Programms.
- /\* ... \*/*-Kommentar kann sich über mehrere Zeilen erstrecken.
- //*-Kommentar endet immer am Zeilenende.

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
public class FirstProgram{public static void main(String[]args){  
System.out.println("hello world");}}
```

Die Ausgabe bleibt unverändert.

Anmerkungen:

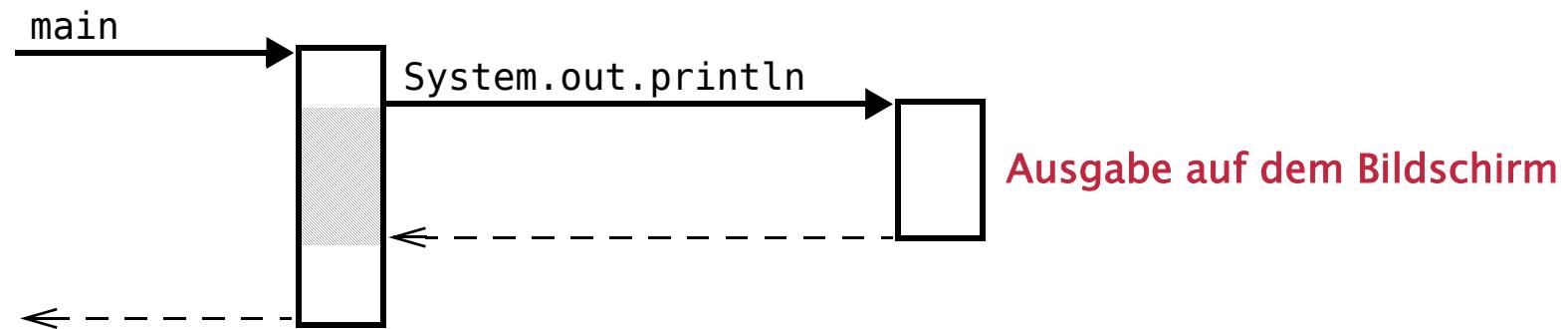
- ❑ Leerzeichen, Zeilenumbrüche und Kommentare dienen – mit wenigen Ausnahmen – nur zur Verbesserung der Lesbarkeit für die Entwickler.
- ❑ Leerzeichen sind aber notwendig, um Wörter zu trennen:  
**public class ist nicht publicclass**
- ❑ Einzelne Zeichen mit besonderer Bedeutung *müssen nicht* durch Leerzeichen abgeteilt werden:  
    `);}}` oder   `FirstProgram{public` sind syntaktisch korrekt.
- ❑ Statt eines Leerzeichens können fast überall auch mehrere Leerzeichen eingesetzt werden, um beispielsweise sichtbare Einrückungen zu erzeugen
- ❑ *Die Lesbarkeit ist wichtig für die menschlichen Entwickler, nicht für den Compiler.*

## Programmiersprache Java – ein erstes Programm

(Fortsetzung)

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

Ausführen des Programms durch einen Rechner bedeutet, dass die Methode `main` ausgeführt wird. Dabei passiert Folgendes:



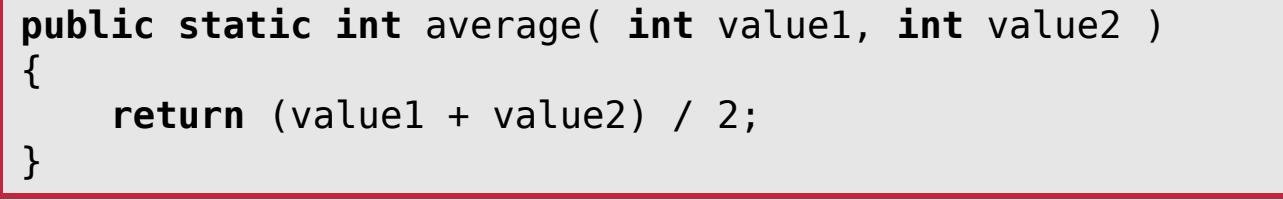
## Programmiersprache Java – ein zweites Programm

```
public class SecondProgram
{
    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```



public static int average( int value1, int value2 )  
{  
 return (value1 + value2) / 2;  
}

Methode

- average ist eine Methode, die in der Klasse SecondProgram deklariert wird.
- average ist ein vom Entwickler gewählter Name.
- average hat zwei *Parameter* value1 und value2 mit selbst gewählten Namen.
- Ein Parameter dient beim Aufruf der Methode dazu, ein Argument zu übernehmen.
- Der Wert des Arguments wird dann innerhalb der Methode über den Parameter bereitgestellt.
- average berechnet einen Wert und stellt diesen im Programm bereit.

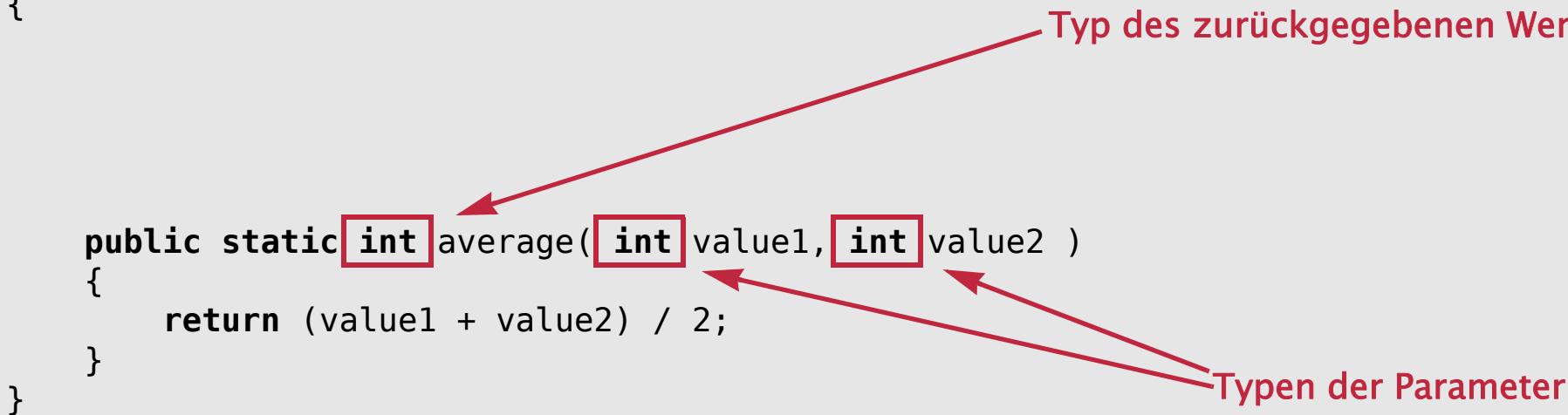
## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

Typ des zurückgegebenen Wertes

Typen der Parameter



- **int** (für *integer*) bezeichnet einen Teilbereich der ganzen Zahlen:  
–2 147 483 648 ... 2 147 483 647
- **int** ist die Angabe eines Wertebereichs und wird als *Typ* bezeichnet
- **int** schränkt die möglichen Werte ein, die bei der Bearbeitung von `average` auftreten:
  - die Werte der beim Aufruf an die Parameter übergebenen Argumente
  - den Wert, der nach der Ausführung der Methode zurückgegeben wird

## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

Typ des zurückgegebenen Wertes

Anweisung:  
Zurückgeben eines Wertes

- Die **return**-Anweisung bestimmt, welcher Wert beim Aufruf der Methode `average` zurückgegeben werden soll.

## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

Zurückgeben eines Wertes

int-Literal

Ausdruck

- $(value1 + value2) / 2$  ist ein mathematischer Ausdruck der eine **int**-Zahl berechnet:  
+ ist die *ganzzahlige* Addition, / ist die *ganzzahlige* Division.
- Die Klammern ( ... ) bestimmen die Auswertungsreihenfolge,  
ohne Klammerung gilt Punkt- vor Strichrechnung.
- 2 ist ein Literal des Typs **int** mit dem Wert 2 (*trivial*).

## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 20: " + average( 15, 20 ) );
    }

    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

- ❑ `main` ist die Methode in der Klasse `SecondProgram`, mit der die Ausführung des zweiten Programms beginnt.
- ❑ `String[] args` ist der Parameter der Methode `main`. Die Bedeutung kann hier noch nicht erklärt werden.

## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 20: " + average( 15, 20 ) );
    }

    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

Typ des zurückgegebenen Wertes

- void** (deutsch "leer") kennzeichnet eine Methode, die keinen Wert zurückgibt.  
Die nicht benötigte Typangabe wird durch **void** ersetzt.
- Eine durch **void** gekennzeichnete Methode muss keine **return**-Anweisung enthalten.
- Eine Methode, die wie die Methode `average` einen Wert zurückgibt,  
muss immer (mindestens) eine **return**-Anweisung enthalten.

## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 20: " + average( 15, 20 ) );
    }

    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

Ausdruck



- "average of 15 and 20: " + average( 15, 20 ) ist ein Ausdruck,  
der einen Text festlegt.

## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 20: " + average( 15, 20 ) );
    }

    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

Konkatenation

Aufruf

- ❑ `average( 15, 20 )` ist ein Aufruf der Methode `average`, bei dem die Werte `15` und `20` als Argumente übergeben werden.
- ❑ Da `/` die *ganzzahlige* Division ist, liefert `average( 15, 20 )` als Ergebnis `17`.
- ❑ `+` ist an dieser Stelle die *Konkatenation* (Verkettung) von zwei Texten, das Ergebnis des Aufrufs von `average` wird implizit von `int` in Text umgewandelt.
- ❑ `System.out.println( "average of 15 and 20: " + average( 15, 20 ) )` ist also der Aufruf von `System.out.println` mit dem Argument `"average of 15 and 20: 17"`

## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 20: " + average( 15, 20 ) );
    }

    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

- Die Vorgabe in der Deklaration der Methode heißt: *Parameter*
- Der übergebene Wert beim Aufruf der Methode heißt: *Argument*
- Während der Ausführung nehmen die Parameter die Werte der Argumente an:  
value1 erhält den Wert 15, value2 erhält den Wert 20.
- Argumente können auch unmittelbar für die Übergabe berechnet werden:  
average( 15, 20 ) für Aufruf von System.out.println.

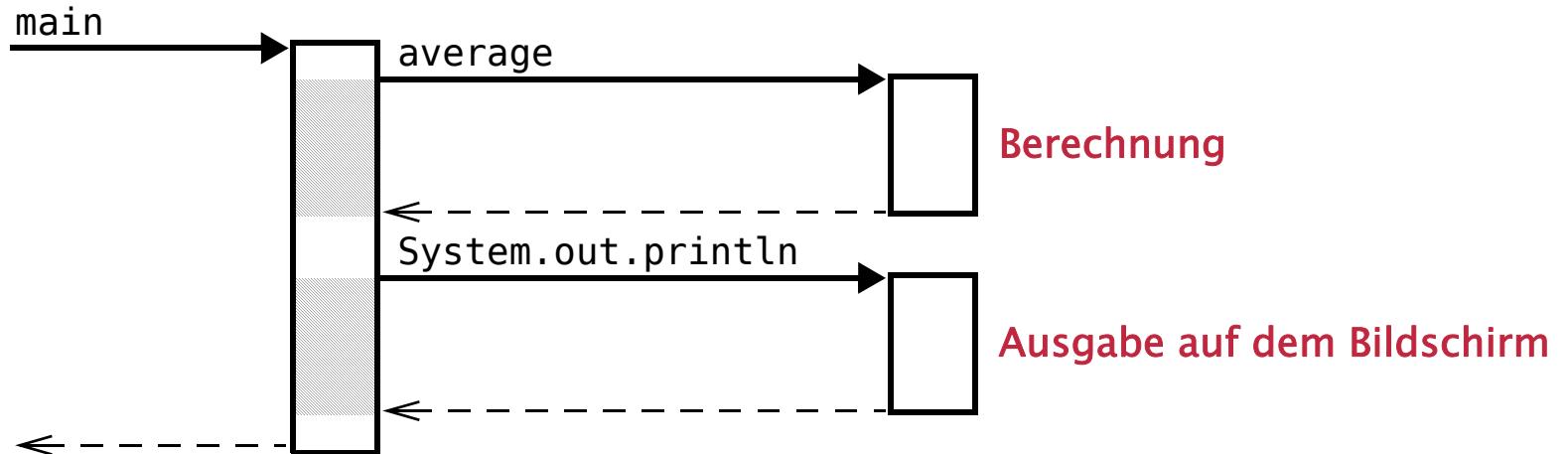
## Programmiersprache Java – ein zweites Programm

(Fortsetzung)

```
public class SecondProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 20: " + average( 15, 20 ) );
    }

    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }
}
```

Ausführung:



## Programmiersprache Java – ein drittes Programm

```
public class ThirdProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 5: " + average( 15, five() ) );
    }

    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }

    public static int five()
    {
        return 5;
    }
}
```

## Programmiersprache Java – ein drittes Programm

(Fortsetzung)

```
public class ThirdProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 5: " + average( 15, five() ) );
    }

    public static int average( int value1, int value2 )
    {
        return (value1 + value2) / 2;
    }

    public static int five()
    {
        return 5;
    }
}
```

Aufruf ohne Argument

Deklaration ohne Parameter

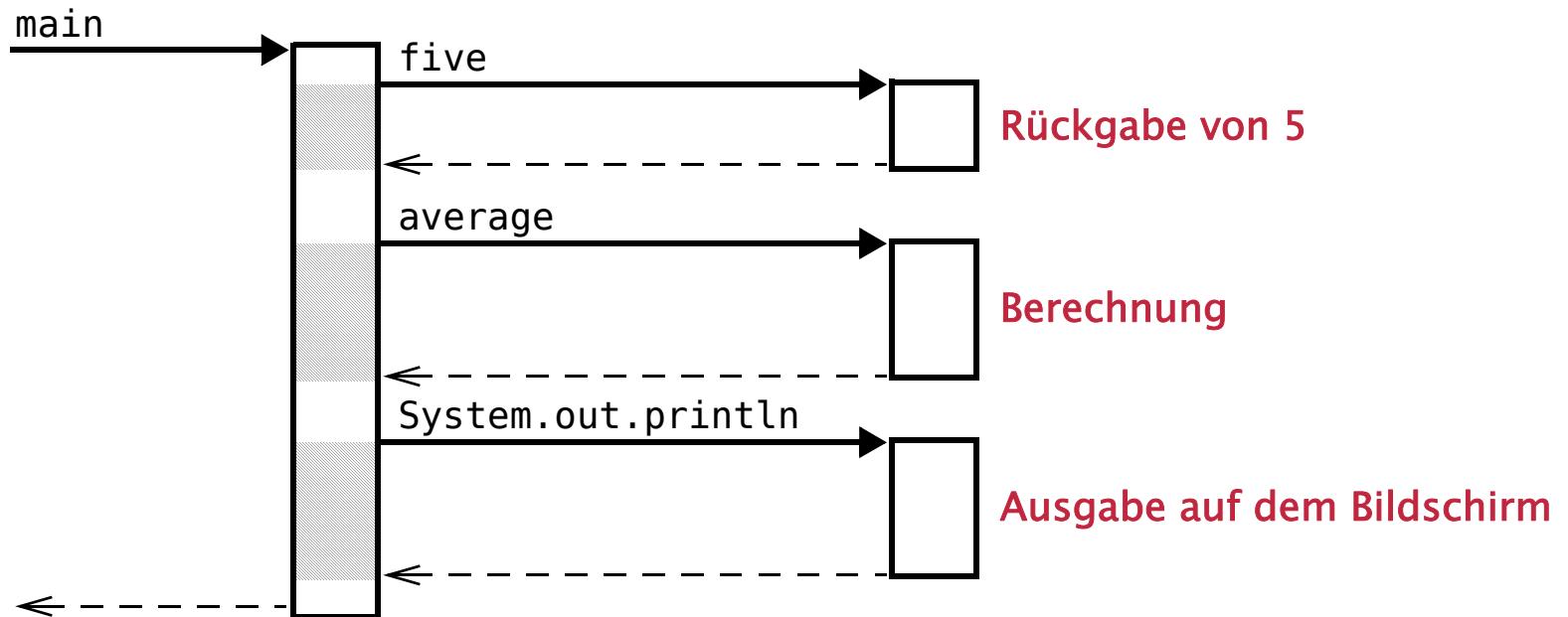
- ❑ `five()` ist eine Methode *ohne* Parameter, die immer nur den Wert 5 zurückgibt:  
() sind bei der Deklaration und beim Aufruf der Methode `five` notwendig!
- ❑ `average( 15, five() )`  
Die Berechnung von Argumenten kann beliebig geschachtelt werden.

## Programmiersprache Java – ein drittes Programm

(Fortsetzung)

```
public class ThirdProgram
{
    public static void main( String[] args )
    {
        System.out.println( "average of 15 and 5: " + average( 15, five() ) );
    }
    ...
}
```

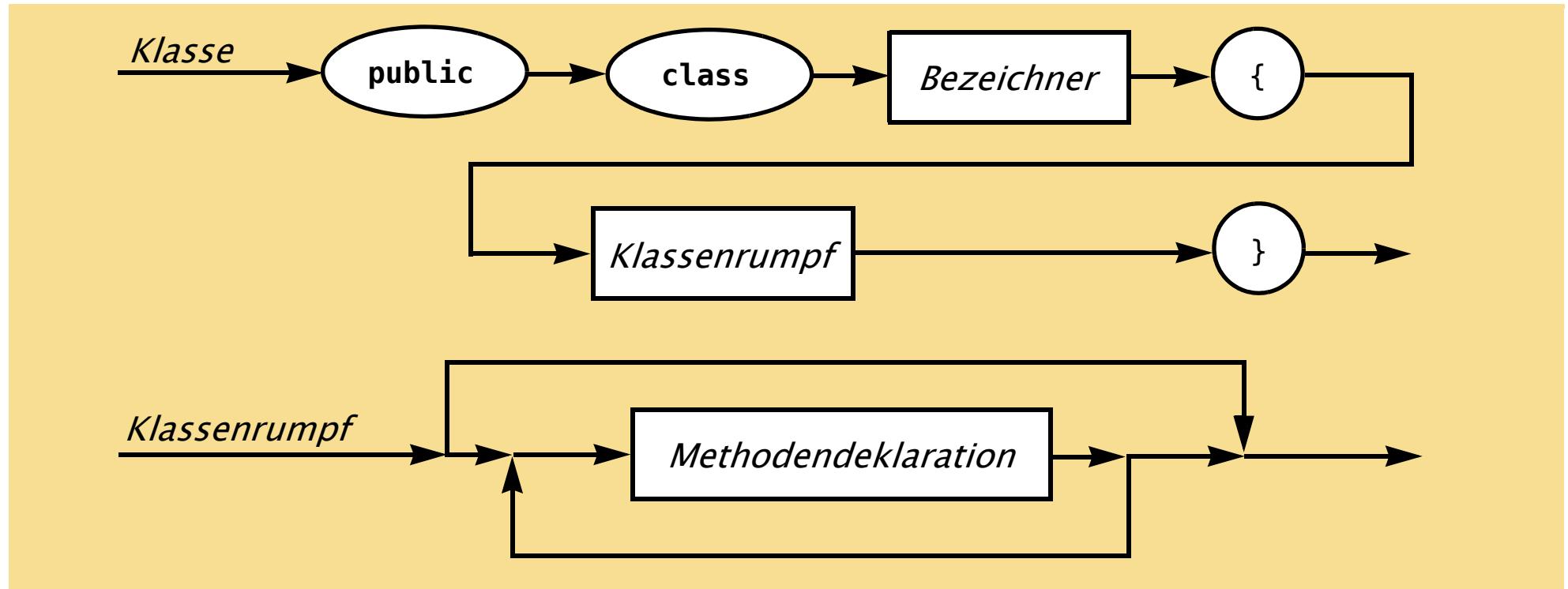
Ausführung:



## Zusammenfassung

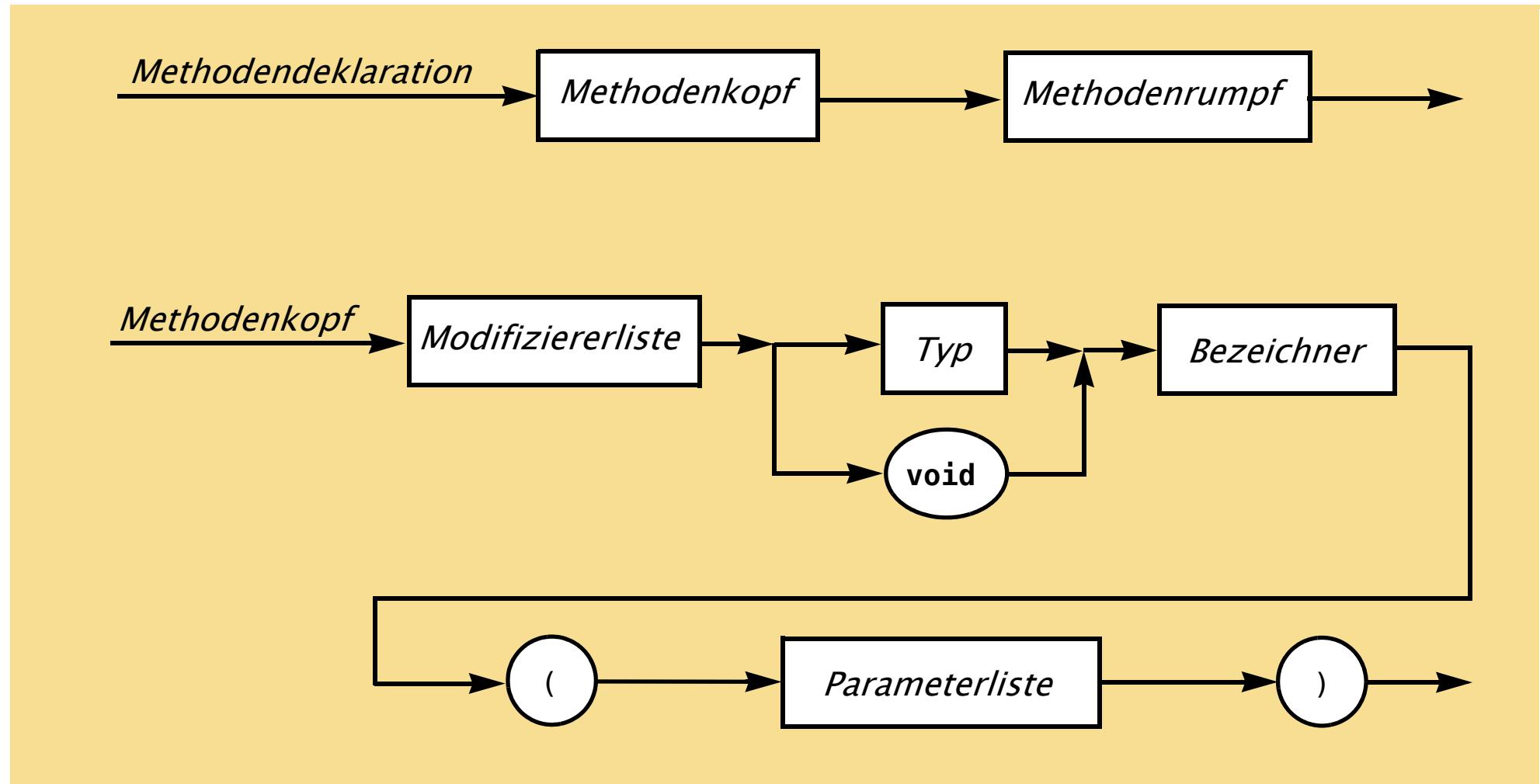
- ❑ Eine *Klasse* bildet die äußere strukturelle Einheit eines Java-Programms.
- ❑ Eine *Methode* ist eine Folge von Handlungsschritten (Anweisungen), die unter einem Namen zusammengefasst werden.
- ❑ Methoden werden innerhalb von Klassen deklariert.
- ❑ Es gibt besondere Methoden, z.B.: `main`, `System.out.println`
- ❑ Eine Methode kann *Parameter* besitzen, die im Methodenkopf deklariert werden.
- ❑ Eine Methode kann einen *Rückgabewert* liefern, dessen Typ im Methodenkopf deklariert wird.
- ❑ Eine Methode kann aufgerufen und ausgeführt werden:  
Dann werden Werte als *Argumente* an die Parameter übergeben und der Rückgabewert wird berechnet.
- ❑ Eine *Anweisung* ist ein Handlungsschritt in einer Methode:
  - Methodenaufruf
  - **return**-Anweisung
- ❑ Ein *Ausdruck* ist eine Berechnung, die einen Wert eines bestimmten *Typs* liefert.
- ❑ Ein *Typ* bezeichnet einen Wertebereich, der die Werte für Argumente und in Ausdrücken beschränkt.

## Syntaxdiagramm zu *Klasse* \*)



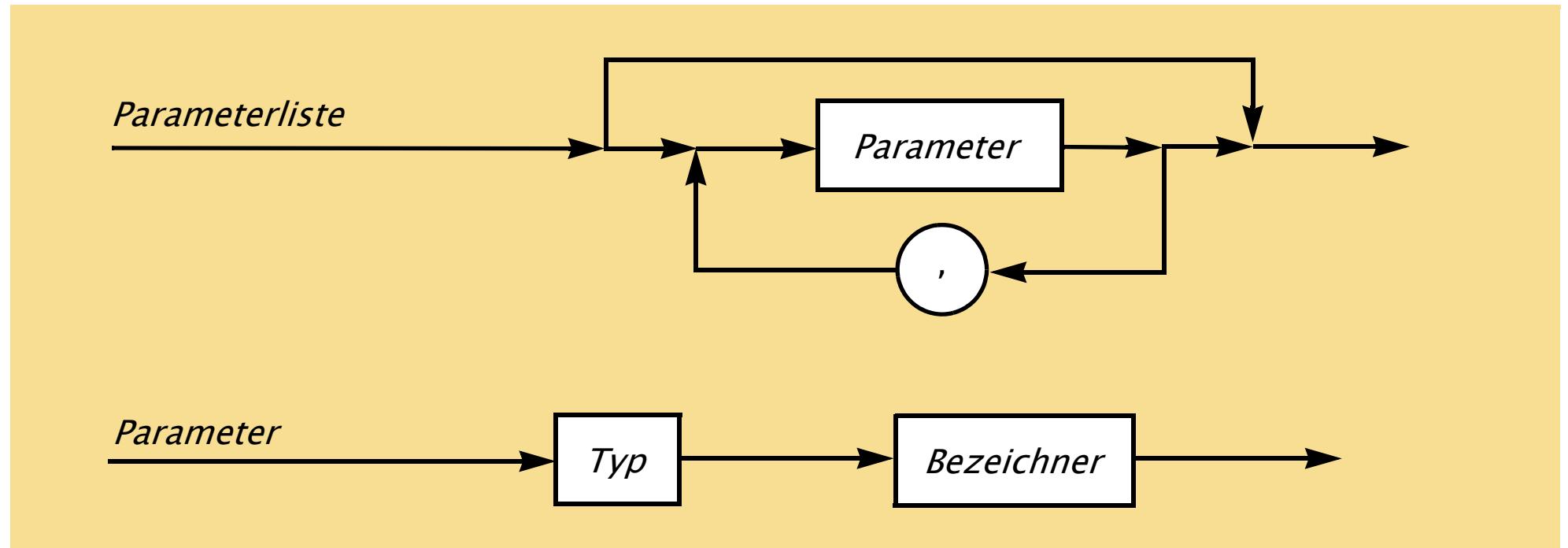
- ❑ Das Nichtterminalsymbol *Bezeichner* wird in den Übungen näher betrachtet.
  - ❑ Diese Diagramme werden in späteren Versionen noch erweitert.
- \*) Ein Teil der Syntaxdiagramme wird im Laufe der Vorlesung weiter ergänzt, so dass jeweils nur die zuletzt vorgestellte Version die vollständige Beschreibung der Syntax liefert.

## Syntaxdiagramm zu *Methode*



## Syntaxdiagramm zu *Methode*

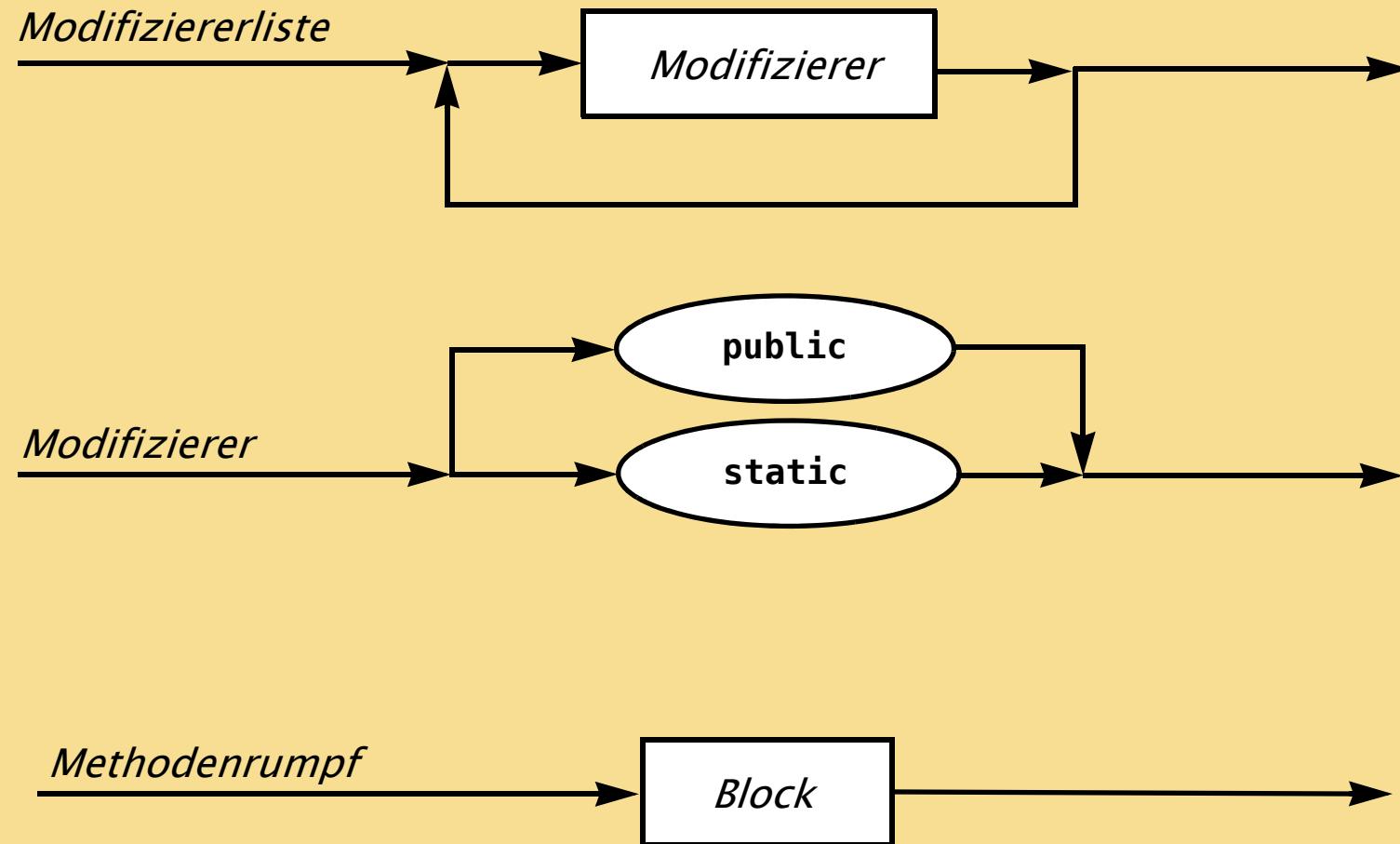
(Fortsetzung)



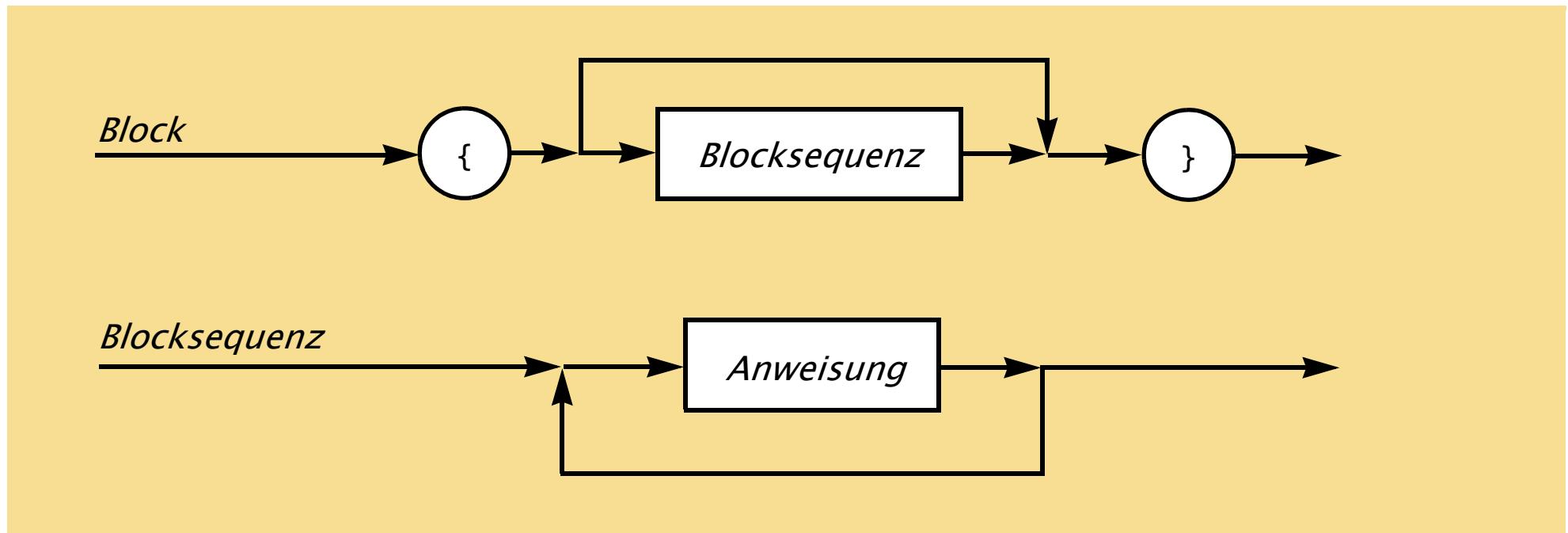
- ❑ Insbesondere kann eine *Parameterliste* leer sein, also keine Parameter enthalten.
- ❑ Das Syntaxdiagramm *Methodenkopf* erzwingt aber immer die () .
- ❑ Bisher ist als *Typ*-Angabe nur das Schlüsselwort **int** bekannt.

## Syntaxdiagramm zu *Methode*

(Fortsetzung)

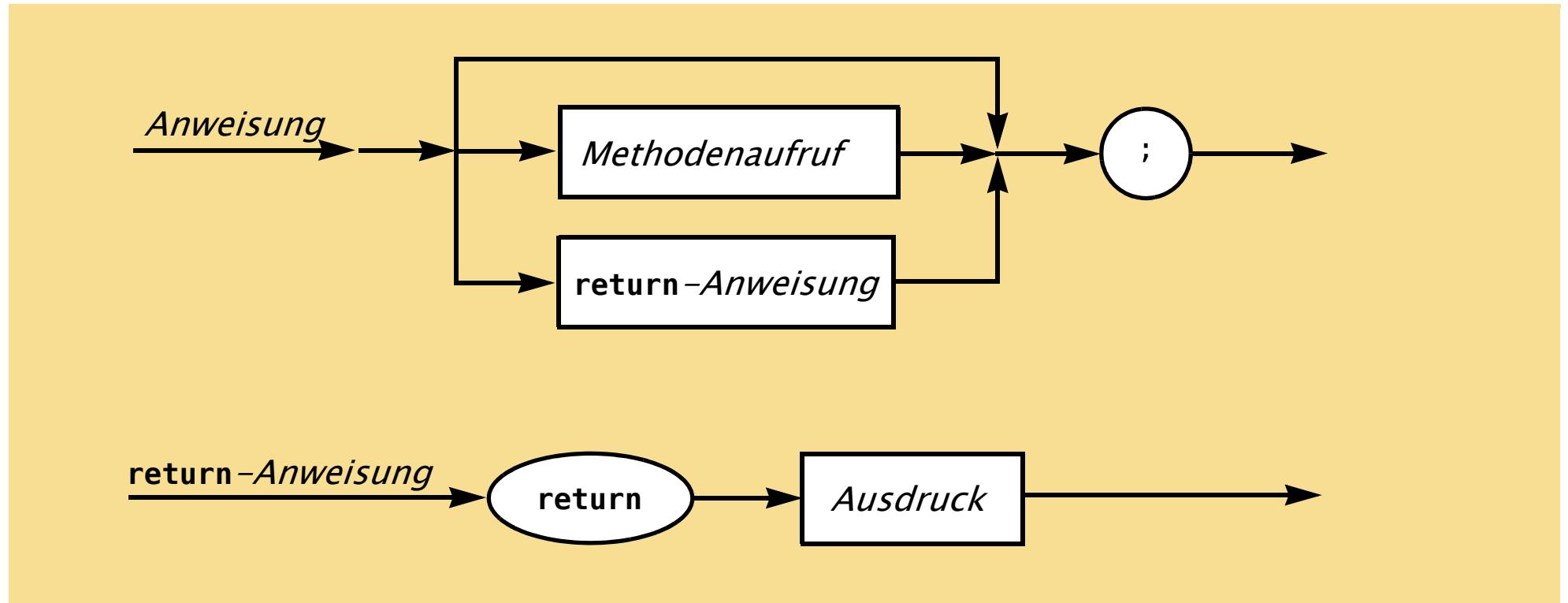


## Syntaxdiagramm zu *Block*



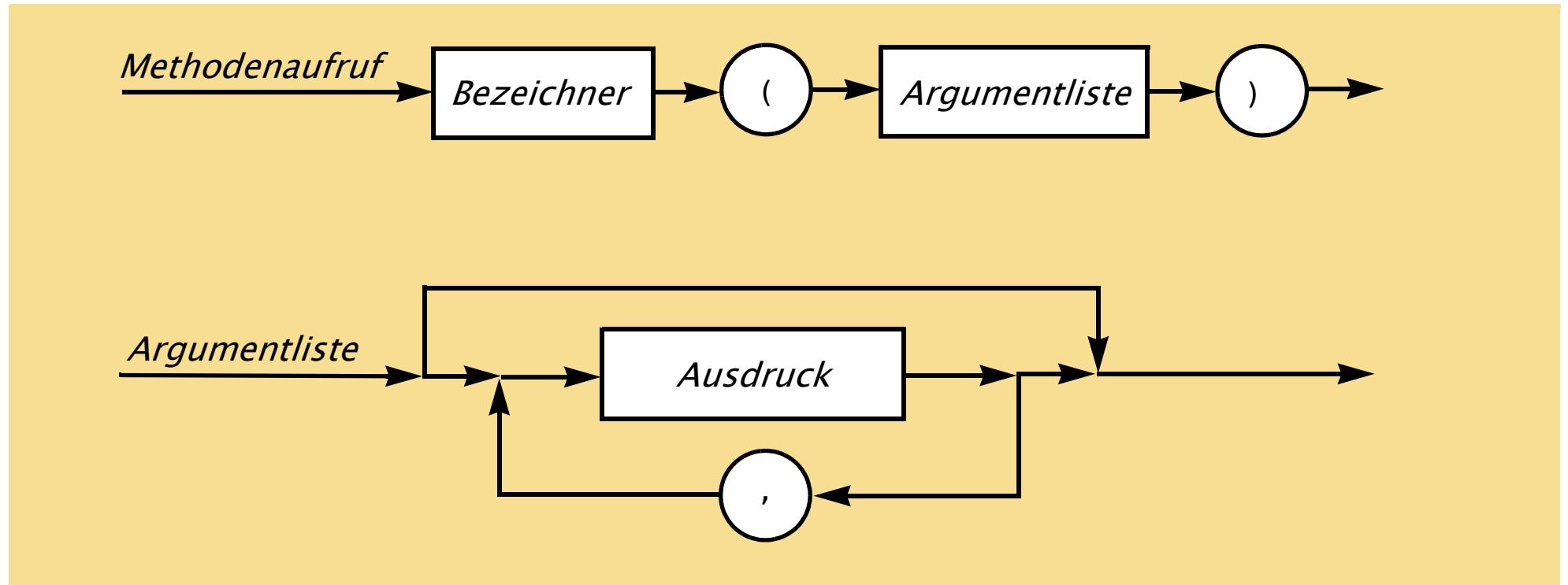
- Das Nichtterminal *Block* wird später auch noch an anderer Stelle der Java-Grammatik verwendet.

## Syntaxdiagramm zu *Anweisung* und *return-Anweisung*



- ❑ Eine *Anweisung* kann auch eine *leere Anweisung* sein:  
Folgen von `;;;` sind erlaubt.

## Syntaxdiagramm zu *Methodenaufruf*



- ❑ Da Methoden ohne Parameter deklariert werden können, erlaubt *Argumentliste* auch den Aufruf von Methoden ohne Parameter.
- ❑ Beim Aufruf einer Methode müssen die Anzahl und die Typen der Argumente immer zu der Parameterliste passen.  
*Diese Übereinstimmung lässt sich aber nicht durch Produktionsregeln erzwingen.*

## Syntaxdiagramme

Hinweise:

- Einige Nichtterminalsymbole werden hier nicht als Diagramme veranschaulicht, weil sie
  - noch nicht ausreichend anhand von Beispielen eingeführt worden sind,
  - selbsterklärend sind oder
  - in den Übungen vertieft werden.
- Es gibt semantische Abhängigkeiten zwischen Nichtterminalsymbolen, die sich nicht auf der Ebene der Syntax(-diagramme) darstellen lassen:
  - Zu jeder aufgerufenen Methode muss es eine passende Deklaration geben.
  - Ein als Argument übergebener Wert muss zum Typ der zugehörigen Parameterdeklaration passen.
  - Der Wert, den der Ausdruck in einer **return**-Anweisung liefert, muss zur Typangabe im Kopf der umgebenden Methode passen.
- Begriff:  
Die *Signatur* einer Methode sind die zum Aufruf notwendigen Informationen:

*der Name der Methode sowie die Anzahl und Folge der Typen der Parameter*

## Symbole in Java

bisher kennengelernt:

- *Schlüsselwörter* sind Terminalsymbole, die aus Buchstaben gebildet werden

Beispiele:      **public**      **int**      **return**

- *Sonderzeichen* sind Terminalsymbole, die aus Nicht-Buchstaben gebildet werden

Beispiele:      ;      {      }      /      //      /\*      \*/

Anmerkung:

Die korrekte (*Klein-)*Schreibung ist wichtig bei  
Schlüsselwörtern und Sonderzeichen.

Insbesondere dürfen keine Leerzeichen eingeschoben werden.

## Symbole in Java

(Fortsetzung)

bisher wurden vorgestellt:

- **Literale** sind meist Nichtterminalsymbole, die als Folge aus einer vorgegebenen Menge von Terminalsymbolen gebildet werden:
  - Literale des Typs `int` bestehen aus einzelnen Ziffern: 2431
  - Literale des Typs `String` bestehen aus fast beliebigen Zeichen: "Ergebnis"
- **Bezeichner** ist ein Nichtterminalsymbol, das eine Folge aus einer vorgegebene Menge von Terminalsymbolen definiert:

value1      average      FirstProgram

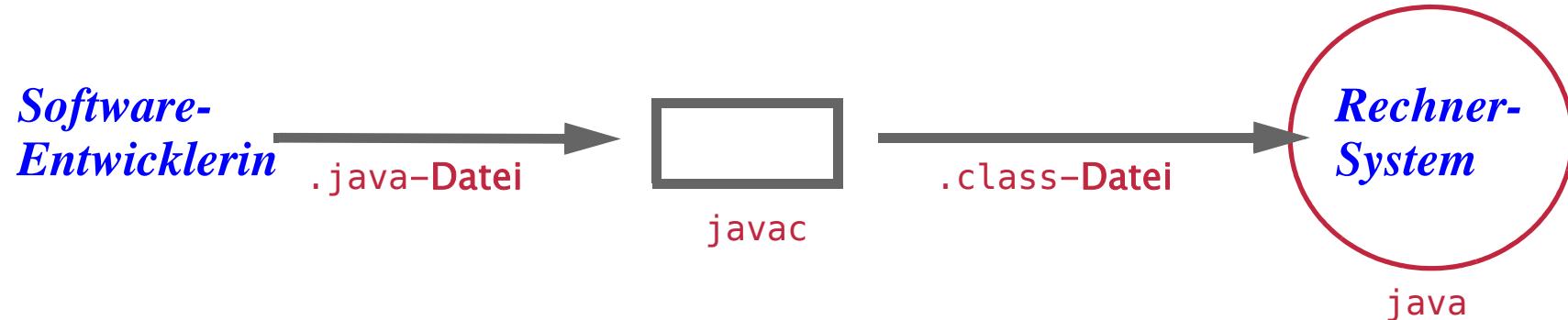
In Java werden Bezeichner üblicherweise folgendermaßen aufgebaut:

- Parameter- und Methodennamen beginnen mit einem Kleinbuchstaben,
- Klassennamen beginnen mit einem Großbuchstaben,
- bei aus mehreren Wörtern zusammengesetzten Namen werden das zweite und alle folgenden Wörter mit einem Großbuchstaben begonnen (*CamelCase*).

Anmerkungen:

- Die korrekte (*Klein-)***Schreibung** ist bei Bezeichnern wichtig!
- Schlüsselwörter können *nicht* als Bezeichner verwendet werden.

## Compiler – Überprüfen von Programmeigenschaften

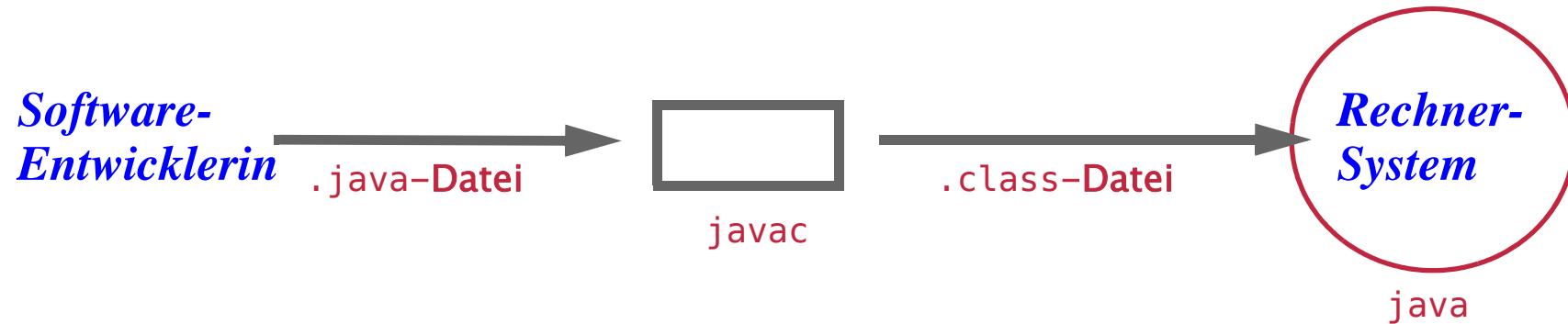


### Überprüfungen

- ❑ Der Compiler (javac) kennt die Grammatik von Java und überprüft, ob das zu übersetzende Programm (. java) mit dieser Grammatik erzeugt werden kann.
- ❑ Der Compiler kennt zusätzliche Eigenschaften, die für Java-Programme gelten müssen, wie zum Beispiel:
  - Aufgerufene Methoden müssen auch deklariert sein.
  - Für Methodenaufrufe müssen die Anzahl der deklarierten Parameter und der übergebenen Argumente übereinstimmen.
  - Für Methodenaufrufe müssen die Typen der deklarierten Parameter und der übergebenen Argumente übereinstimmen.
  - Für Operatoren müssen die Typen der Operanden zulässig sein:  
5 / 2 ist als Division von zwei **int**-Werten zulässig, "fuenf" / 2 ist unzulässig.

## Compiler – Überprüfen von Programmeigenschaften

(Fortsetzung)

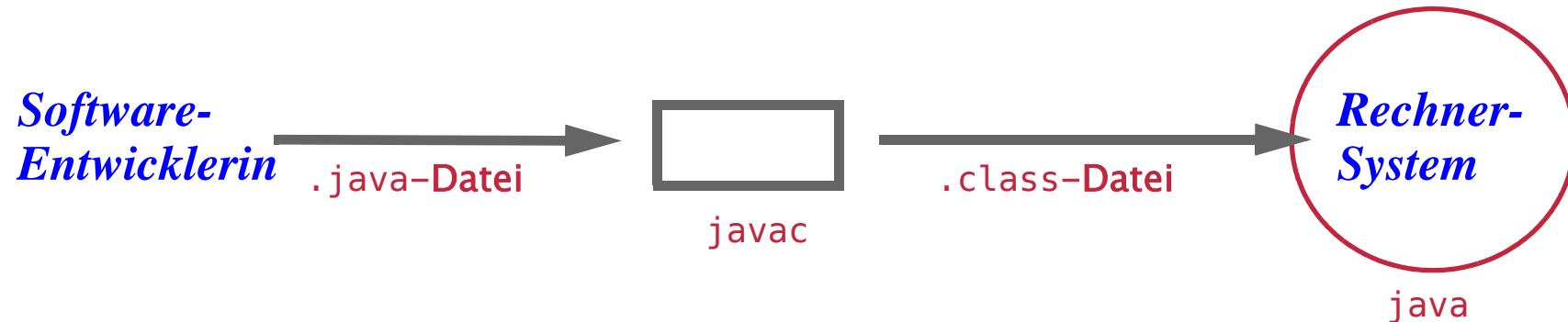


### Überprüfungen

- Falls der Compiler einen Fehler feststellt,
  - wird dem Entwickler eine entsprechende Fehlermeldung angezeigt und
  - es wird *keine* Bytecode-Datei (.class) erzeugt.
- Falls der Compiler keinen Fehler feststellt,  
wird eine Bytecode-Datei (.class) erzeugt, die ausgeführt werden kann.

## Compiler – Überprüfen von Programmeigenschaften

(Fortsetzung)



### Überprüfungen

- Falls der Compiler einen Fehler feststellt,
  - wird dem Entwickler eine entsprechende Fehlermeldung angezeigt und
  - es wird *keine* Bytecode-Datei (.class) erzeugt.
- Falls der Compiler keinen Fehler feststellt,  
wird eine Bytecode-Datei (.class) erzeugt, die ausgeführt werden kann.

aber:

Der Compiler weiß *nicht*, welche Funktionalität der Entwickler umsetzen wollte und welche Werte verarbeitet werden sollen.

## Compiler – Überprüfen von Programmeigenschaften

(Fortsetzung)

### Aufgabe:

Erstelle eine Methode, die den Wert ihres Parameters verdoppelt und den berechneten Wert als Ergebnis zurückgibt.

# Compiler – Überprüfen von Programmeigenschaften

(Fortsetzung)

**Aufgabe:**

Erstelle eine Methode, die den Wert ihres Parameters verdoppelt und den berechneten Wert als Ergebnis zurückgibt.

**programmierte Lösung:**

```
public static int doubleValue( int value )
{
    return 2 / value;
}
```

## Compiler – Überprüfen von Programmeigenschaften

(Fortsetzung)

### Aufgabe:

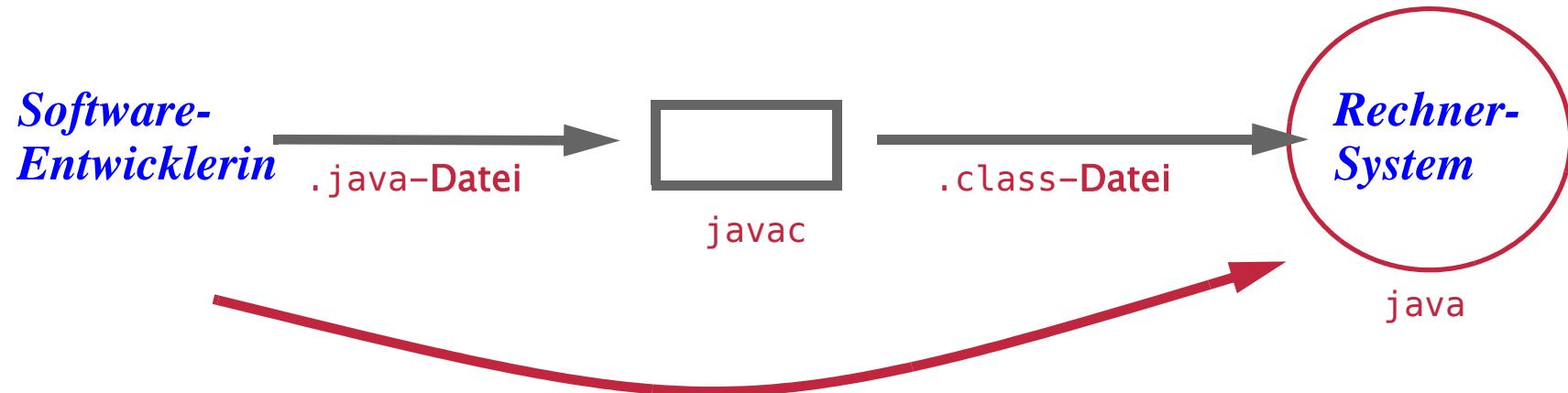
Erstelle eine Methode, die den Wert ihres Parameters verdoppelt und den berechneten Wert als Ergebnis zurückgibt.

### programmierte Lösung:

```
public static int doubleValue( int value )
{
    return 2 / value;
}
```

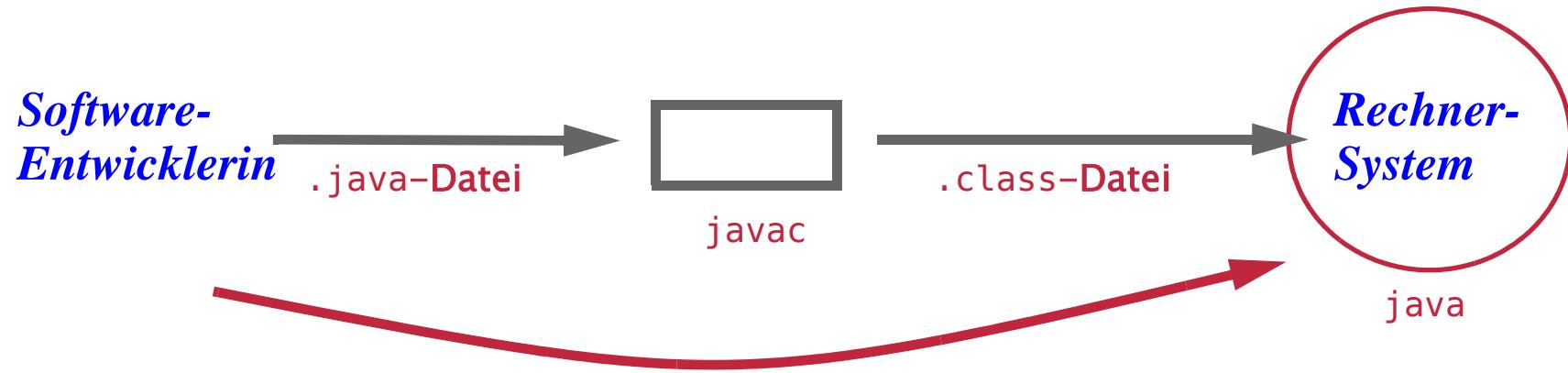
- Der Compiler findet keinen Fehler, da die Methode syntaktisch korrekt ist.
- Trotzdem ist die Lösung fehlerhaft, da die gestellte Aufgabe *nicht* gelöst wird.

## Testen durch den Entwickler



Ein Test ist das Ausführen des Programms mit Werten,  
um die richtige (oder falsche) Umsetzung der Aufgabe zu erkennen:  
Durch einen Test erfolgt eine (partielle) Prüfung der Semantik des Programms.

## Testen durch den Entwickler



Ein Test ist das Ausführen des Programms mit Werten, um die richtige (oder falsche) Umsetzung der Aufgabe zu erkennen:  
Durch einen Test erfolgt eine (partielle) Prüfung der Semantik des Programms.

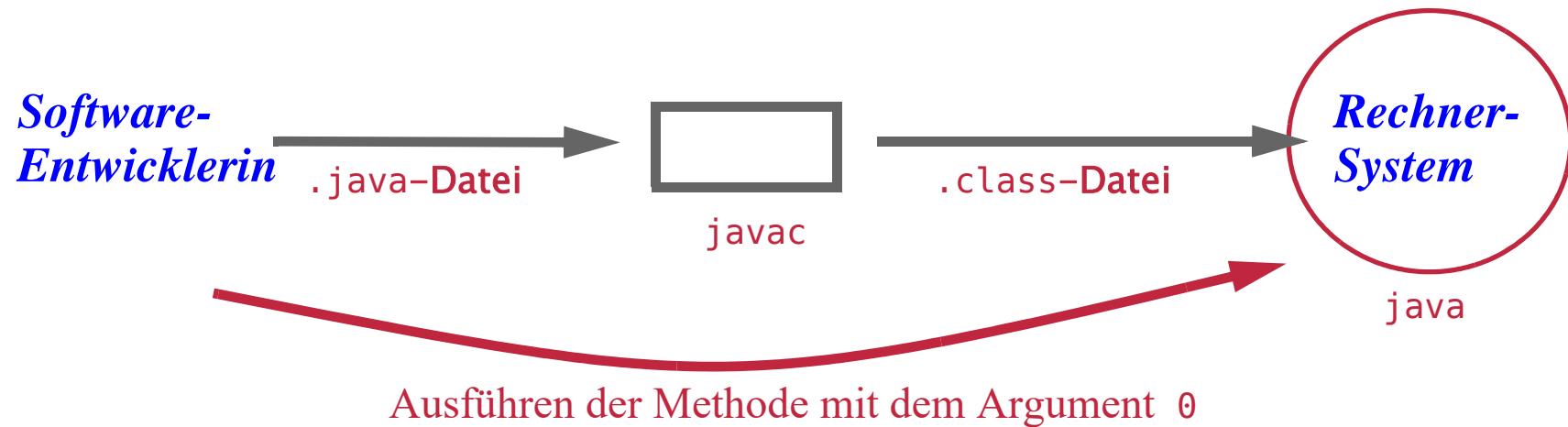
Anmerkung:

Auch die Zahl und die Auswahl der zum Testen benutzen Werte ist wichtig:  
`doubleValue( 1 )` würde das erwartete Ergebnis 2 liefern.

```
public static int doubleValue( int value )
{
    return 2 / value;
}
```

## Testen durch den Entwickler

(Fortsetzung)

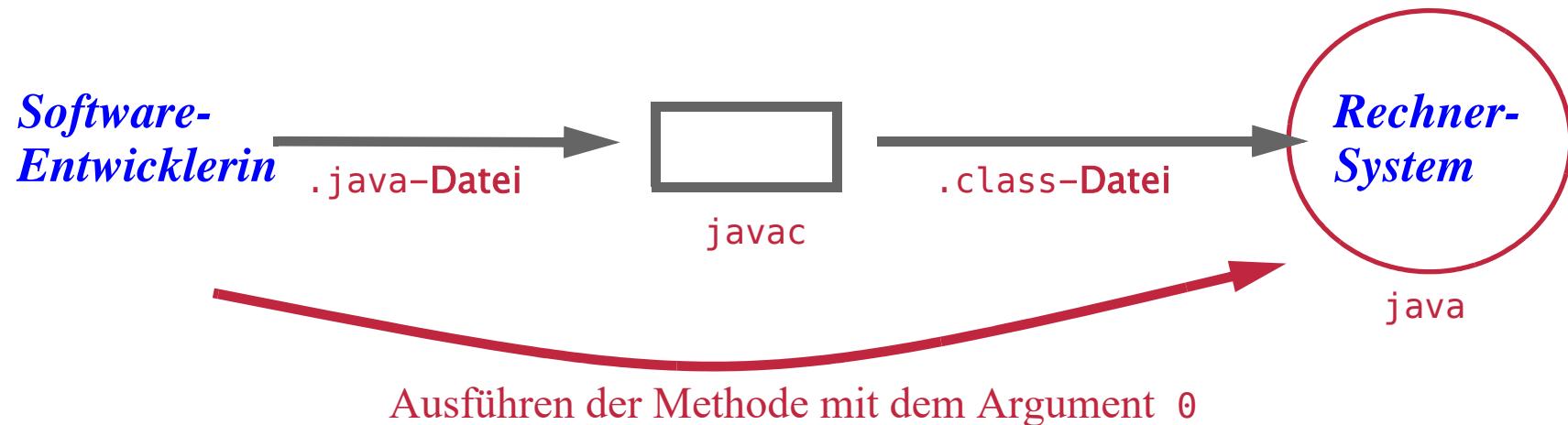


- Der Aufruf `doubleValue( 0 )` erzeugt ein mathematisches Problem:  
Die Division durch 0 besitzt kein gültiges Ergebnis in `int`.

```
public static int doubleValue( int value )
{
    return 2 / value;
}
```

## Testen durch den Entwickler

(Fortsetzung)



- Der Aufruf `doubleValue( 0 )` führt zu:

```
java.lang.ArithmetricException: / by zero
at ClassName.doubleValue(ZeroDivision.java:5)
```

- Abbruch der Ausführung der Methode `doubleValue`
- Anzeige  
dass eine *Ausnahme*(-situation) aufgetreten ist:  
welcher Art die *Ausnahme* ist:  
wo die *Ausnahme* aufgetreten ist:

*ArithmetricException*  
*/ by zero*  
*at ClassName.doubleValue*

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 2.1. Einfache Algorithmen - Berechnung einer Summe

Dr. Stefan Dissmann  
Fakultät für Informatik



## Lernziele des Kapitels 2. Einfache Algorithmen

Nach Durcharbeiten des Kapitels 2. Einfache Algorithmen sollen die teilnehmenden Studierenden

- die *Typen boolean, int, long, float* und **double** kennen
- Operatoren und Ausdrücke kennen und deren Auswertungsreihenfolgen bestimmen können
- Variablen* deklarieren und initialisieren können
- Werte an Variablen zuweisen können
- Bedingungen formulieren und auswerten können
- bedingte Anweisungen* und ihre Verwendung kennen
- for**- und **while-Schleifen** und ihre Verwendung kennen
- Feld*-Variable deklarieren und nutzen können
- den Ablauf des Algorithmus zur *Primzahlenbestimmung* nach Eratosthenes skizzieren und nachvollziehen können

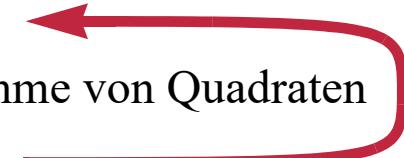
## Problemstellung «Summe der Quadrate»

Aufgabe:

Berechne die Summe der Quadrate aller natürlichen Zahlen bis zu einer vorgegebenen Obergrenze  $n$ .

oder mathematisch:  $1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2$

*Lösungsidee:*

- berechne das Quadrat einer Zahl
  - addiere das berechnete Quadrat zu der eventuell schon berechneten Summe von Quadrate
  - prüfe, ob noch Quadrate berechnet werden müssen: *dann gehe zurück*
  - die berechnete Summe ist das Ergebnis
- 

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

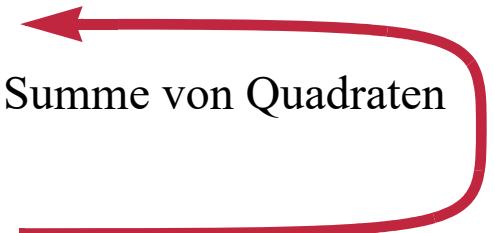
Aufgabe:

Berechne die Summe der Quadrate aller natürlichen Zahlen bis zu einer vorgegebenen Obergrenze  $n$ .

oder mathematisch:  $1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2$

*Lösungsidee «mit etwas mehr Ordnung»:*

- beginne mit der Zahl 1
- berechne das Quadrat der gewählten Zahl
- addiere das berechnete Quadrat zu der eventuell schon berechneten Summe von Quadraten
- erhöhe den Wert der gewählten Zahl um 1
- prüfe, ob  $n$  noch nicht überschritten ist: *dann gehe zurück*
- die berechnete Summe ist das Ergebnis



## Problemstellung «Summe der Quadrate»

(Fortsetzung)

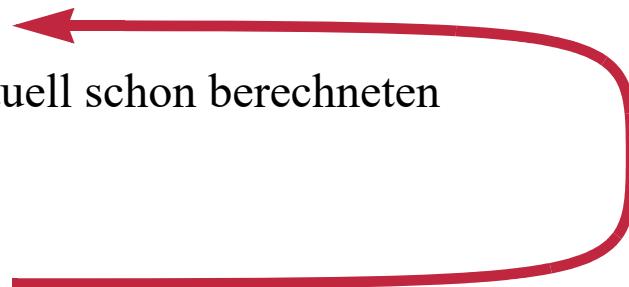
Aufgabe:

Berechne die Summe der Quadrate aller natürlichen Zahlen bis zu einer vorgegebenen Obergrenze  $n$ .

oder mathematisch:  $1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2$

*Lösungsidee mit «Namen» für Werte:*

- beginne mit der Zahl `value = 1`
- berechne das Quadrat von `value`
- addiere das berechnete Quadrat zu der eventuell schon berechneten Summe (`subtotal`) von Quadraten
- erhöhe den Wert von `value` um 1
- prüfe, ob `value <= n`: *dann gehe zurück*
- `subtotal` ist das Ergebnis



## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### Aufgabe:

Berechne die Summe der Quadrate aller natürlichen Zahlen bis zu einer vorgegebenen Obergrenze  $n$ .

oder mathematisch:  $1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2$

### 1. Lösungsansatz:

- initialisiere einen Zähler für die natürliche Zahlen
- initialisiere die zu berechnende Zwischensumme
- bestimme die neue Zwischensumme als
- erhöhe den Wert von `value` um 1
- prüfe, ob `value <= n`: *dann gehe zurück*
- gib `subtotal` als Ergebnis zurück

`value = 1`

`subtotal = 0`

`subtotalneu = subtotalalt + value * value`

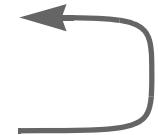
`valueneu = valuealt + 1`

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

- 1. Lösungsansatz:
- initialisiere einen Zähler für die natürliche Zahlen
- initialisiere die zu berechnende Zwischensumme
- bestimme die neue Zwischensumme als
- erhöhe den Wert von `value` um 1
- prüfe, ob `value <= n`: dann gehe zurück
- gib `subtotal` als Ergebnis zurück

```
value = 1
subtotal = 0
subtotalneu = subtotalalt + value*value
valueneu = valuealt + 1
```

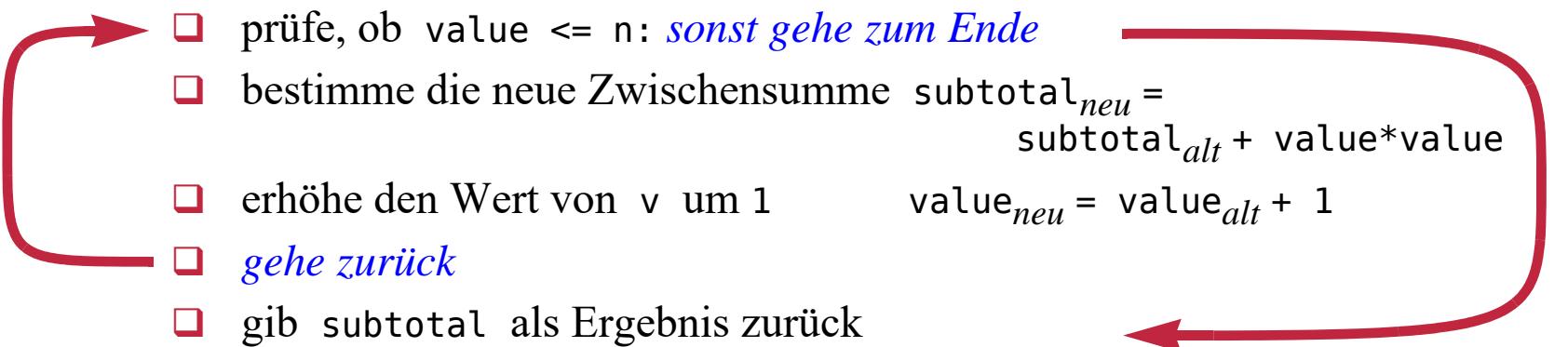


### 2. Lösungsansatz:

- initialisiere einen Zähler für die natürliche Zahlen
- initialisiere die zu berechnende Zwischensumme
- prüfe, ob `value <= n`: *sonst gehe zum Ende*
- bestimme die neue Zwischensumme `subtotalneu = subtotalalt + value*value`
- erhöhe den Wert von `v` um 1
- gehe zurück*
- gib `subtotal` als Ergebnis zurück

`value = 1`  
`subtotal = 0`

`subtotalneu = subtotalalt + value*value`  
`valueneu = valuealt + 1`



## Problemstellung «Summe der Quadrate»

### 1. Implementierung

```
public static int sumOfSquares( int n )
{
    int value = 1;
    int subtotal = 0;
    while ( value <= n )
    {
        subtotal = subtotal + value * value;
        value = value + 1;
    }
    return subtotal;
}
```

(Fortsetzung)

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

```
public static int sumOfSquares( int n )
{
    int value = 1;
    int subtotal = 0;
    while ( value <= n )
    {
        subtotal = subtotal + value * value;
        value = value + 1;
    }
    return subtotal;
}
```

Deklaration

- Eine *Variable* speichert einen Wert des bei ihrer Deklaration vorgegeben Typs:
  - **int** `value` deklariert die Variable `value`, die **int**-Werte speichern kann
  - **int** `subtotal` deklariert die Variable `subtotal`, die ebenfalls **int**-Werte speichern kann

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

```
public static int sumOfSquares( int n )
{
    int value = 1;           ← Initialisierung
    int subtotal = 0;         ← Initialisierung
    while ( value <= n )
    {
        subtotal = subtotal + value * value;
        value = value + 1;
    }
    return subtotal;
}
```

- Eine Variable kann bei ihrer Deklaration mit einem Wert *initialisiert* werden:
  - **int** value = 1 initialisiert die Variable value mit dem (zulässigen) Wert 1
  - **int** subtotal = 0 initialisiert die Variable subtotal mit dem (zulässigen) Wert 0

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

```
public static int sumOfSquares( int n )
{
    int value = 1;
    int subtotal = 0;
    while ( value <= n ) ← Zugriff
    {
        subtotal = subtotal + value * value; ← Zuweisung
        value = value + 1;
    }
    return subtotal;
}
```

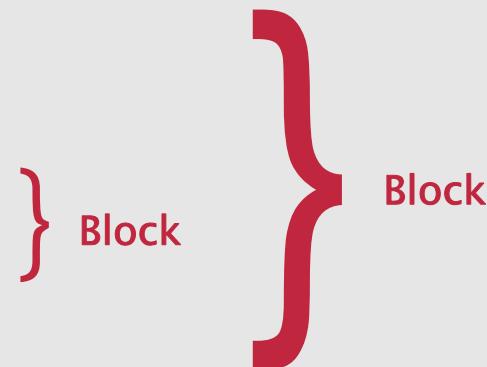
- Auf eine Variable kann über ihren Namen zugegriffen werden:  
So kann ihr sowohl ein Wert zugewiesen werden als auch ihr Wert abgerufen werden.
- Einer Variablen kann während der Ausführung ein neuer Wert zugewiesen werden,  
eine *Zuweisung* = erfolgt immer von rechts nach links:  
Daher erhöht `value = value + 1` den Wert von `value` um 1.

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

```
public static int sumOfSquares( int n )
{
    int value = 1;
    int subtotal = 0;
    while ( value <= n )
    {
        subtotal = subtotal + value * value;
        value = value + 1;
    }
    return subtotal;
}
```



- Ein Block wird immer durch `{...}` eingeschlossen.
- Ein Block begrenzt den Bereich, in dem deklarierte Variablen bekannt sind.
- Ein Block fasst Anweisungen zusammen, die gemeinsam behandelt werden.
- Blöcke können geschachtelt werden.

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

Anmerkungen zu Variablen:

- ❑ Eine Variable muss *vor* ihrer Nutzung (Zugriff) deklariert werden.
- ❑ Eine Variable ist nur innerhalb des Blocks bekannt, in dem sie deklariert ist.
- ❑ Eine Variable ist auch in tiefer geschachtelten Blöcken bekannt.
- ❑ In einem Block darf es *keine* zwei Variablen mit gleichem Namen geben.  
Insbesondere darf zwischen umgebenden und inneren Blöcken kein Konflikt auftreten.
- ❑ Deklarationen von Variablen des gleichen Typs können zusammengefasst werden.

Beispiele:

```
int v1, v2, v3;  
int v4, v5 = 2, v6;  
int value = 0, subtotal = 0;
```

- ❑ Anmerkung:

Ein Parameter kann als eine Variable betrachtet werden, die

- im Block der Methode bekannt ist und
- beim Aufruf der Methode den Wert des Arguments zugewiesen bekommt.

## Problemstellung "Summe der Quadrate"

(Fortsetzung)

### 1. Implementierung

Anmerkungen zu Initialisierung und Zuweisung:

- ❑ Initialisierung und Zuweisung sind *unterschiedliche* Konzepte. \*)
- ❑ Eine Variable ohne Initialisierung besitzt zunächst keinen Wert.  
Auf eine Variable ohne Wert darf nicht zugegriffen werden.  
Ihr kann aber ein Wert zugewiesen werden.
- ❑ Eine Zuweisung ist eine Anweisung, bei der zunächst der Ausdruck rechts von `=` ausgewertet wird und dann der ermittelte Wert in der Variablen gespeichert wird.

\*) Der Unterschied kann hier aber noch nicht demonstriert werden.

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

```
public static int sumOfSquares( int n )
{
    int value = 1;
    int subtotal = 0;
    while ( value <= n )
    {
        subtotal = subtotal + value * value;      } Block
        value = value + 1;
    }
    return subtotal;
}
```

**Schleife**

- Eine *Schleife* ermöglicht die wiederholte Ausführung der Anweisungen innerhalb des nachfolgenden *Blocks* zwischen `{...}`.
- Eine Schleife beginnt mit dem Schlüsselwort `while`. Es folgen eine Bedingung und ein Block.
- Eine Schleife ist eine Anweisung.

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

```

public static int sumOfSquares( int n )
{
    int value = 1;
    int subtotal = 0;
    while ( value <= n ) ←
    {
        subtotal = subtotal + value * value; } Block
        value = value + 1;
    }
    return subtotal;
}

```

Schleife

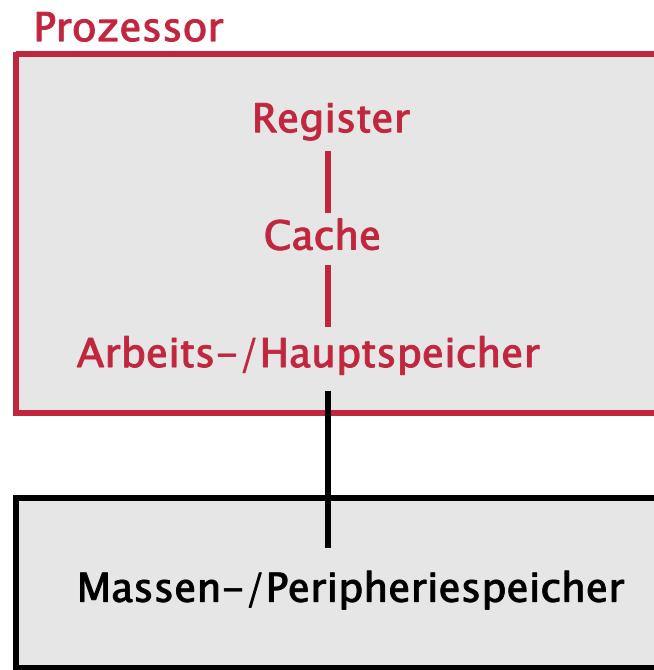
Bedingung

Block

- Die Bedingung kontrolliert die Ausführung der Anweisungen in dem nachfolgenden *Block*.
- Eine *Bedingung* ist ein logischer Ausdruck, der einen der beiden Werte **true** (*wahr*) oder **false** (*falsch*) liefert.
- Ergibt die Auswertung der Bedingung den Wert **true**, so wird der Block ausgeführt.
- Nach Ausführung aller Anweisungen des Blocks wird *erneut* die Bedingung ausgewertet.
- Ergibt die Auswertung der Bedingung den Wert **false**, so wird *hinter* dem Block fortgefahrene.  
Dadurch werden die Anweisungen des Blocks solange wiederholt, bis die Auswertung der Bedingung erstmals **false** ergibt.

## Exkurs: einfaches Modell der Speicherhierarchie

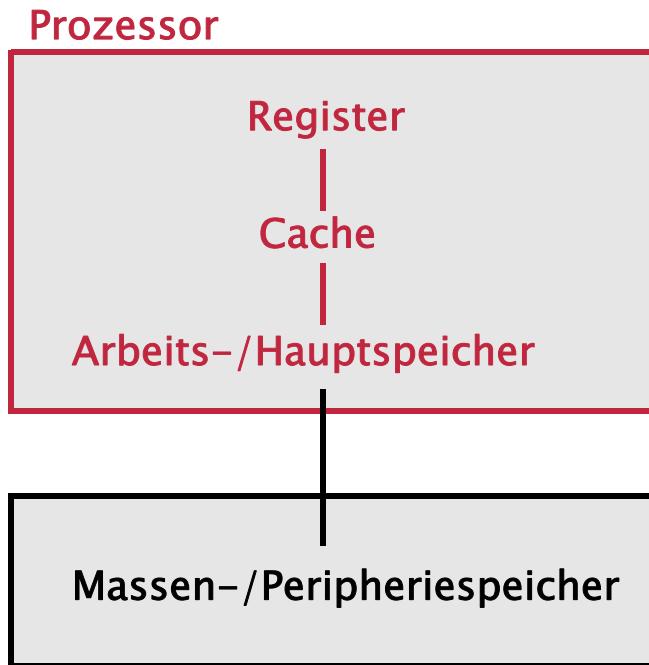
Hardware:



## Exkurs: einfaches Modell der Speicherhierarchie

(Fortsetzung)

Hardware:



Software:

**Variable, Parameter**

existiert nur während der Programmausführung

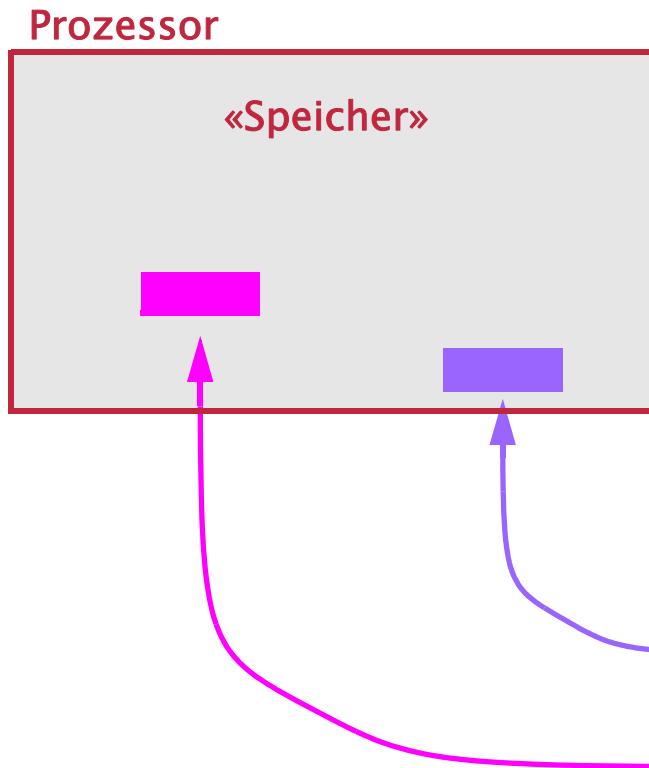
**Datei**

existiert dauerhaft, evt. auch nur auf einem Medium

## Exkurs: einfaches Modell der Speicherhierarchie

(Fortsetzung)

Hardware:



Software:

Eine *Variable* ist ein Name für einen Bereich im Speicher, der genau die Größe besitzt, die benötigt wird, um *einen* Wert des Typs der Variablen abzulegen.

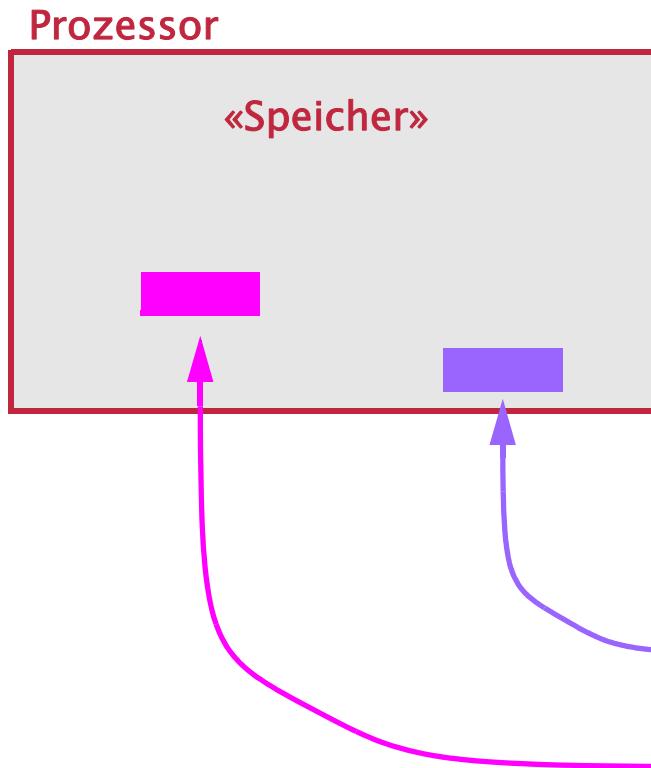
`int v`

`int subtotal`

## Exkurs: einfaches Modell der Speicherhierarchie

(Fortsetzung)

Hardware:



Software:

Der Typ der Variablen wird im Programm angegeben. Daher kann der Compiler bereits *vor der Ausführung* des Programms den entsprechenden Speicherbereich festlegen.

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

Die Bedingung `value <= n` führt einen Vergleich von zwei Werten durch.

Vergleichsoperatoren:

- ❑ Ein Vergleichsoperator vergleicht zwei Werte eines Zahlentyps wie `int` und liefert immer ein Ergebnis des Typs `boolean`.
- ❑ Der Typ `boolean` besitzt nur zwei Werte: `true` und `false`.
- ❑ Vergleichsoperatoren:

<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
==	gleich
!=	ungleich

Hinweis: `=` ist eine Zuweisung (Folie 109)

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

Anmerkungen:

- ❑ Vergleichsoperatoren können auch auf den Typ **boolean** angewendet werden:

```
boolean b1, b2;  
...  
while ( b1 == b2 ) ...
```

- ❑ *aber* statt **while ( b1 == true ) ...**  
reicht **while ( b1 ) ...**  
da **b1 == true** genau dann den Wert **true** liefert,  
wenn **b1** den Wert **true** besitzt.

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

#### Zuweisung

- ❑ Zuweisungsoperator (bereits bekannt)  
dient auch zur Veränderung der Werte von Variablen auf der Basis ihres alten Wertes:

`v = v + 4;`

`v = v * 5;`

- ❑ Verbundoperatoren

verkürzen (nur) die Schreibweise für diesen Fall:

`int v = 3, w = 5;`

`v += 4;`

`v *= w + 3;`

entspricht

entspricht

`v = v + 4;`

`v = v * (w + 3);`

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

Inkrement-/Dekrement-Operatoren

verkürzen die Schreibweise für den Fall einer Erhöhung/Verminderung um 1.

#### Inkrement-/Dekrement-Operatoren

##### – Präfixoperatoren

`++v;`

entspricht

`v = v + 1;`

`--v;`

entspricht

`v = v - 1;`

Der Ausdruck `++v` liefert den Wert von `v` *nach* der Zuweisung.

#### - Postfixoperatoren

`v++;`

entspricht

`v = v + 1;`

`v--;`

entspricht

`v = v - 1;`

*Aber* der Ausdruck `v++` liefert den (*alten*) Wert von `v` *vor* der Veränderung.

`int v = 0, w = 3;`

`v = w++;`

führt zu:

`w` wird erhöht auf 4, `v` erhält aber den Wert von `w` vor der Erhöhung, also 3, zugewiesen.

`v = ++w;`

führt zu:

`w` wird jetzt erhöht auf 5, `v` erhält den neuen Wert 5 zugewiesen.

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 1. Implementierung

```
public static int sumOfSquares( int n )
{
    int value = 1;
    int subtotal = 0;
    while ( value <= n )
    {
        subtotal += value * value;
        value++;
    }
    return subtotal;
}
```

The code is annotated with two red arrows pointing from text labels to specific operators:

- A red arrow points from the label "Verbundoperator" to the assignment operator `+=` in the line `subtotal += value * value;`.
- A red arrow points from the label "Postfixoperator" to the increment operator `value++` in the line `value++;`.

## Problemstellung «Summe der Quadrate»

### 2. Implementierung

(Fortsetzung)

Das Ergebnis lässt sich auch *direkt* mit einer Formel berechnen:

$$s = \frac{n \cdot (n + 1) \cdot (2 \cdot n + 1)}{6}$$

```
public static int sumOfSquares( int n )
{
    return n * (n + 1) * (2 * n + 1) / 6;
```

## Problemstellung «Summe der Quadrate»

(Fortsetzung)

### 2. Implementierung

Das Ergebnis lässt sich auch direkt mit einer Formel berechnen:

```
public static int sumOfSquares( int n )
{
    return n * (n + 1) * (2 * n + 1) / 6;
}
```

Das ändert aber *nicht* den Aufruf der Methode `sumOfSquares`:

```
public class Summation
{
    public static int sumOfSquares( int n )
    {
        // unknown implementation ...
    }

    public static void main( String[] args )
    {
        System.out.println( "Ergebnis für n=34: " + sumOfSquares( 34 ) );
    }
}
```

*Das Verbergen von Abläufen in Methoden verbessert offensichtlich die Änderungsfreundlichkeit des Programms.*

## Problemstellung «Summe der Quadrate»

Vergleich der beiden Implementierungen

(Fortsetzung)

Ziel: Überprüfen der Korrektheit

```
public static void main(String[] args)
{
    int sum1 = 0, sum2 = 0; int value = 1;
    while ( value <=100000 & sum1 == sum2 )
    {
        sum1 = SummationWithLoop.sumOfSquares( value );
        sum2 = SummationWithFormula.sumOfSquares( value );
        value++;
    }
    System.out.println( "value is: " + value );
    System.out.println( "sum (computed by loop) is: " + sum1 );
    System.out.println( "sum (computed by formula) is: " + sum2 );
}
```

- ❑ Bis maximal zum Wert 100000 werden die Summen der Quadrate jeweils mit beiden Methoden (Algorithmen) berechnet und die Ergebnisse verglichen.
- ❑ Die zuletzt berechneten Werte werden ausgegeben.

## Problemstellung «Summe der Quadrate»

Vergleich der beiden Implementierungen

(Fortsetzung)

Ziel: Überprüfen der Korrektheit

```

public static void main(String[] args)
{
    int sum1 = 0, sum2 = 0; int value = 1;
    while ( value <=100000 & sum1 == sum2 )
    {
        sum1 = SummationWithLoop.sum0fSquares( value );
        sum2 = SummationWithFormula.sum0fSquares( value );
        value++;
    }
    System.out.println( "value is: " + value );
    System.out.println( "sum (computed by loop) is: " + sum1 );
    System.out.println( "sum (computed by formula) is: " + sum2 );
}

```

logischer Operator

Zugriff auf andere Klasse

- ❑ Operator & liefert eine logische UND-Verknüpfung (mathematisch:  $\wedge$ )
- ❑ `SummationWithLoop.sum0fSquares( value )` bezeichnet den Aufruf der Methode `sum0fSquares` aus der Klasse `SummationWithLoop`.

## Problemstellung «Summe der Quadrate»

Vergleich der beiden Implementierungen

(Fortsetzung)

Ausgabe:

```
value is: 1025
sum (computed by loop) is: 358438400
sum (computed by formula) is: -357389482
```

negativer Wert  
nicht möglich!

Ist die Formel falsch?

Ist die Implementierung der Formel falsch?

Wird die Summe falsch berechnet?

## Problemstellung «Summe der Quadrate»

Vergleich der beiden Implementierungen

(Fortsetzung)

Ausgabe:

```
value is: 1025
sum (computed by loop) is: 358438400
sum (computed by formula) is: -357389482
```

Ist die Formel falsch?

Ist die Implementierung der Formel falsch?

Wird die Summe falsch berechnet?

- Problem liegt im *Wertebereich* des Typs **int**:    -2 147 483 648 ... 2 147 483 647
- In der Formel wird  $n*(n+1)*(2*n+1)$  berechnet und dann durch 6 geteilt.  
Für  $n = 1025$  wird daher zunächst **2 150 630 400** berechnet und bereits *vor* der Division der Wertebereich des Typs **int** verlassen.
- Es gibt *keine* Fehlermeldung, aber das Ergebnis ist fehlerhaft.  
*Ein Entwickler muss ein solches mögliche Überschreiten des Wertebereichs beachten und eventuell geeignete Maßnahmen ergreifen.*

## Exkurs: Binärzahlenformat

interne Darstellung von *positiven* und *negativen* Zahlen als Binärzahl

- **Zweierkomplement**

Vorteil: verzichtet auf die Unterscheidung von Vorzeichen und Zahl

- Wertebereich:

- positive Zahlen beginnen mit einer 0
- negative Zahlen beginnen mit einer 1

- Beispiel:

$$\begin{aligned} \text{(übliches) Dezimalsystem: } (37)_{10} &= 7 \times 10^{1-1} + 3 \times 10^{2-1} \\ &= 7 \times 1 + 3 \times 10 \end{aligned}$$

Binärsystem:

$$\begin{aligned} (100101)_2 &= (1 \times 2^{1-1} + 0 \times 2^{2-1} + 1 \times 2^{3-1} + \\ &\quad 0 \times 2^{4-1} + 0 \times 2^{5-1} + 1 \times 2^{6-1})_{10} \\ &= (1 \times 1 + 1 \times 4 + 1 \times 32)_{10} \\ &= (37)_{10} \end{aligned}$$

## Exkurs: Binärzahlenformat

(Fortsetzung)

interne Darstellung von *positiven* und *negativen* Zahlen als Binärzahl

### *Zweierkomplement*

Vorteil: verzichtet auf die Unterscheidung von Vorzeichen und Zahl

### Wertebereich:

- positive Zahlen beginnen mit einer **0**: **00001001** entspricht  $(9)_{10}$
- Null ist **00000000**
- negative Zahlen beginnen mit einer **1**: **11110111** entspricht  $(-9)_{10}$

### Vorzeichenwechsel (Umwandlung positive in negative Zahl):

- invertiere jede einzelne Ziffern
- addiere **1** hinzu
- Beispiel:

$$\begin{array}{r} \mathbf{00001001} \text{ wird zu } \mathbf{11110110} \text{ und weiter zu } \mathbf{11110111} \\ \text{mit} \quad \mathbf{00001001} \\ + \quad \mathbf{11110111} \\ = \quad \mathbf{100000000} \end{array}$$

## Exkurs: Binärzahlenformat

(Fortsetzung)

### Zweierkomplement – Problem des Überlaufs

Beispiel:

$$\begin{array}{r} \mathbf{01101001} & \text{entspricht } (105)_{10} \\ + \mathbf{01110001} & \text{entspricht } (113)_{10} \\ = \mathbf{11011010} & \text{entspricht } (-38)_{10} \end{array}$$

*Überlauf in den Bereich der negativen Zahlen*

## Einfache Typen

### Zahlentypen

mit anderen Wertebereichen sind:

- Typ **long**  
ganze Zahlen  
im Wertebereich -9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807
- Rechnungen mit den ganzzahligen Typen **int** und **long** sind immer genau.
- Typ **float**  
Fließkommazahlen im Wertebereich  $-3.4 \cdot 10^{38}$  ...  $+3.4 \cdot 10^{38}$
- Typ **double**  
Fließkommazahlen im Wertebereich  $-1.8 \cdot 10^{308}$  ...  $+1.8 \cdot 10^{308}$
- Fließkommazahlen-Typen decken *nicht* alle möglichen Werte ihres Wertebereichs ab.  
Bei Rechnungen treten daher Rundungsfehler auf. \*)
- Beim Verlassen eines Wertebereichs treten bei allen Typen Rechenfehler auf.

\*) Erklärung erfolgt in der Vorlesung Rechnerstrukturen.

## Einfache Typen

(Fortsetzung)

### Literale

#### Typ **int**

Folgen von Ziffern:

2        19        374        2147483647

*kein* zulässiges Literal:

59474836474 (nicht im Wertebereich von **int**)

#### Typ **long**

Folgen von Ziffern, endet mit L:

2L        19L        59474836474L

#### Typ **float**

Folgen von Ziffern, endet mit F, auch mit Dezimalpunkt und Exponent:

19F        37.4F        21E7F

*kein* zulässiges Literal:

6.9E44F (nicht im Wertebereich von **float**)

#### Typ **double**

Folgen von Ziffern mit Dezimalpunkt, auch mit Exponent:

2.        19.0        37.4        5.9474836474E45

## Einfache Typen

(Fortsetzung)

### Kompatibilität

- ❑ Werte von Typen mit einem kleineren Wertebereich sind kompatibel mit Typen mit einem größeren Wertebereich.
- ❑ 2, 17 und 2891 sind Literale für den Typ **int**.  
Die folgenden Initialisierungen sind also erlaubt:

```
long v = 2891;  
float w = 17;  
double x = v;
```

- ❑ 2.5, 16.0 und 4E22 sind Literale für den Typ **double**.  
Die folgenden Initialisierungen sind also erlaubt:

```
double y = 4E22;  
double z = 16.0;
```

- ❑ Nicht erlaubt sind hingegen:

```
v = z;  
v = 2.0;  
w = 2.5;
```

# Einfache Typen

(Fortsetzung)

## Typ **boolean**

- ❑ Literale sind **true** und **false**
- ❑ logische Operatoren
  - Konjunktion (*und*, mathematisches Zeichen:  $\wedge$ ) als Java-Operator &
  - Disjunktion (*oder*, mathematisches Zeichen:  $\vee$ ) als Java-Operator |
  - Negation (mathematisches Zeichen:  $\neg$ ) als Java-Operator !

## Einfache Typen

(Fortsetzung)

### Anmerkungen zu Operatoren

- ❑ Mit Ausnahme der Zuweisung werden alle Operatoren von links nach rechts ausgewertet.
- ❑ Operatoren mit höherer Priorität oder Klammern ändern die Auswertungsreihenfolge.
- ❑ Operatoren liefern immer einen Wert eines *bestimmten* Typs.
- ❑ Der Typ des Ergebnisses eines Operators ergibt sich *implizit* aus den Typen der Operanden. Dabei wird bei numerischen Berechnungen zwangsläufig der Typ des Operanden mit dem größeren Wertebereich zum Typ des Ergebnisses.

Beispiele:

**double** x = 2.51;

4 + 5	ergibt den <b>int</b> -Wert	9
5 / 2 * 2	ergibt den <b>int</b> -Wert	4
2 * 5 / 2	ergibt den <b>int</b> -Wert	5
4.4 + 5	ergibt den <b>double</b> -Wert	9.4
5 / 2.0	ergibt den <b>double</b> -Wert	2.5
10 * x	ergibt den <b>double</b> -Wert	25.1

# Einfache Typen

(Fortsetzung)

## Anmerkungen zu Operatoren

- ❑ Der Typ, den ein Operator liefert, ergibt sich *implizit* aus den Typen der Operanden.  
Das Ergebnis erhält zwangsläufig den Typ des Operanden mit dem größeren Wertebereich.

## Sonderfall Operator +:

- Addition
- Konkatenation von Texten (siehe Folie 74)

über	"hello" + " " + "world"	ergibt	den String	"hello world"
über	"hello" + 2 + 3	ergibt	den String	"hello23"
über	2 + 3 + "hello"	ergibt	den String	"5hello"

# Operatorprioritäten

Die Priorität eines Operators bestimmt den Zeitpunkt seiner Auswertung in einem Ausdruck.

Priorität	Operatoren
hoch	1 <code>++, --, !</code>
	2 <code>*, /</code>
	3 <code>+, -</code>
	4
	5 <code>&gt;, &gt;=, &lt;, &lt;=</code>
	6 <code>==, !=</code>
	7 <code>&amp;</code>
	8
	9 <code> </code>
	...
	13 <code>=</code>
niedrig	14 <code>+=, *=</code>

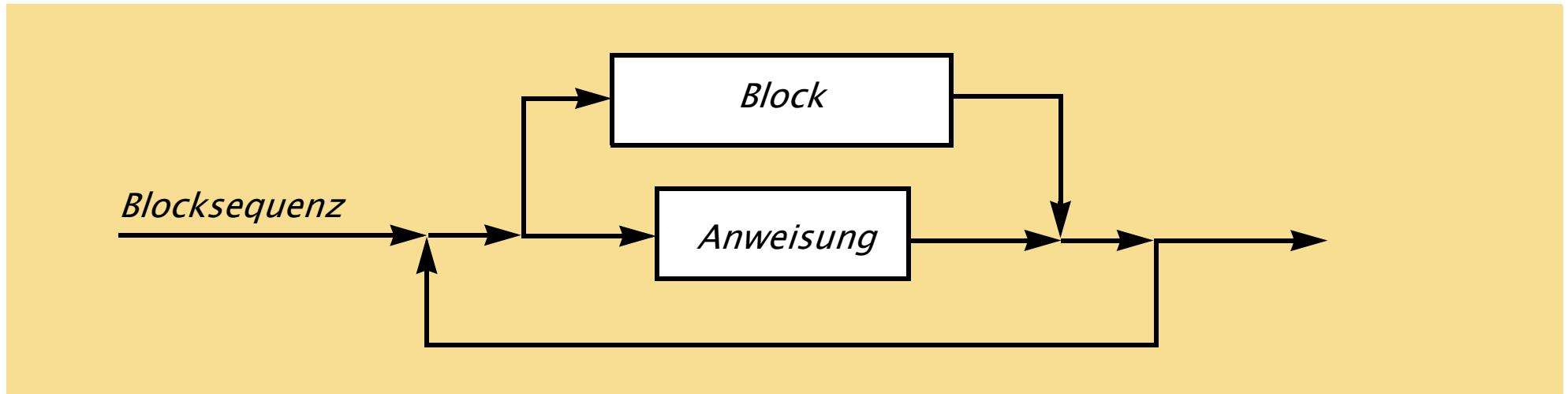
*alle einstelligen (unären) Operatoren*  
*Punktrechnung*  
*Strichrechnung*

*Vergleiche*  
*(Un-)Gleichheit*  
*Konjunktion*

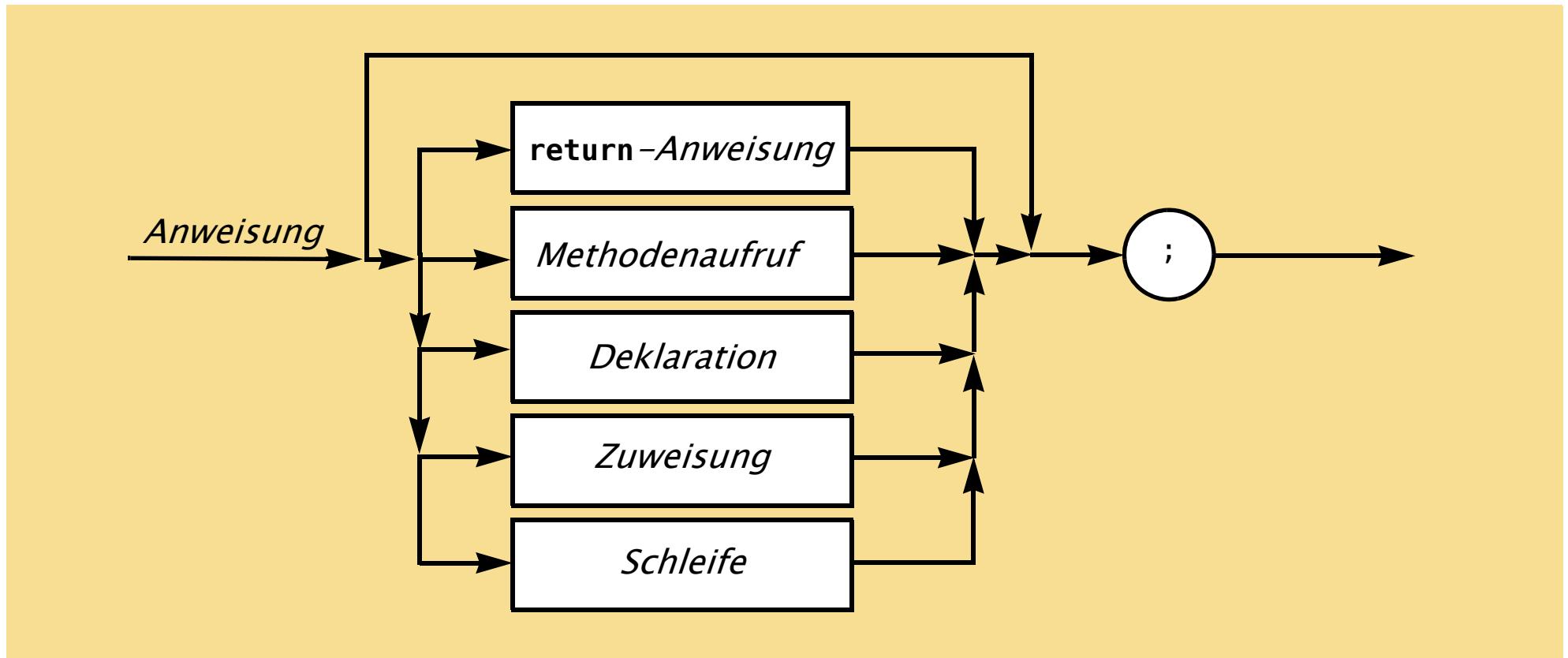
*Disjunktion*

*Zuweisung*  
*Verbundzuweisungen*

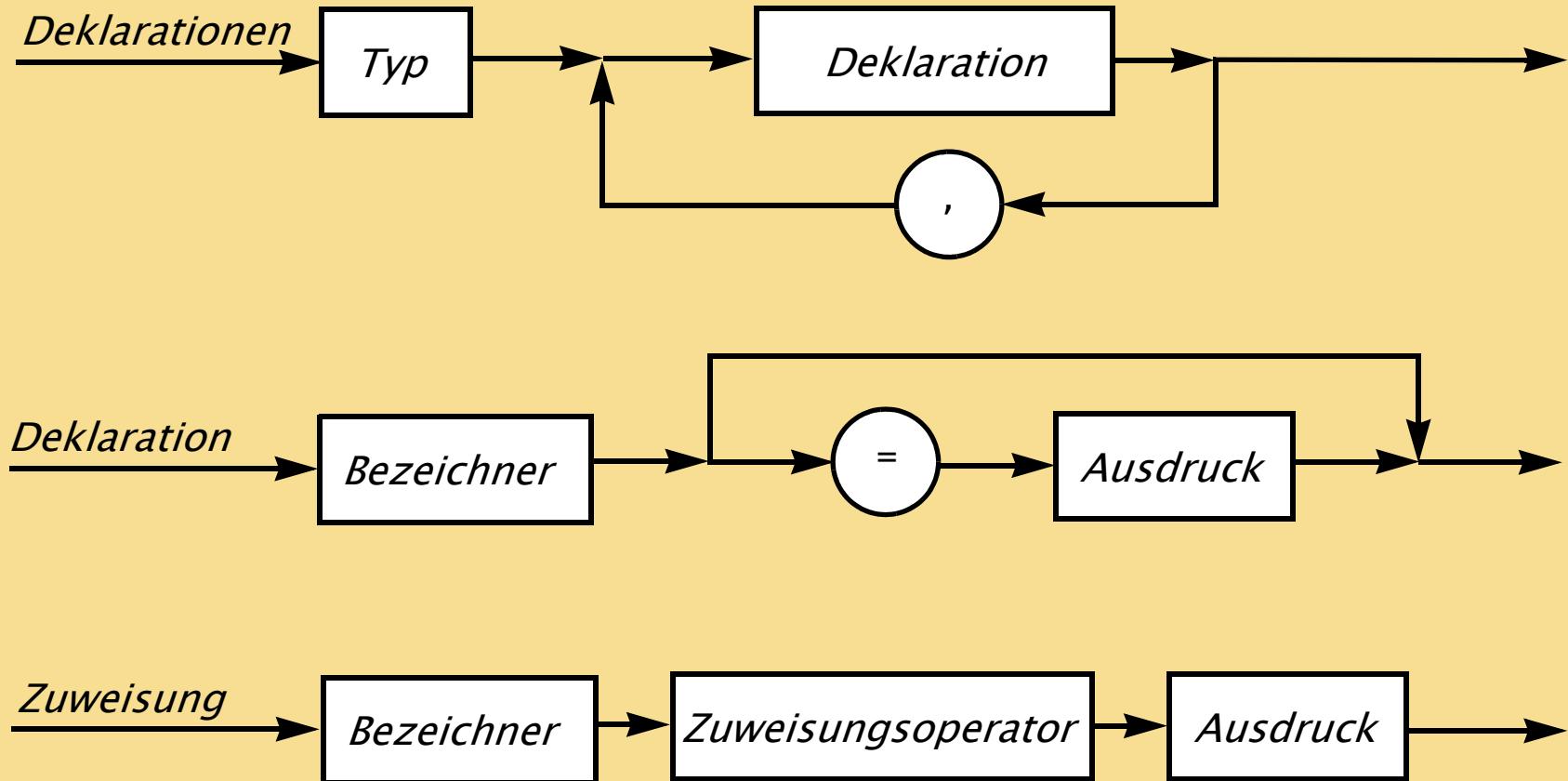
## Syntaxdiagramm zu *Blocksequenz* (siehe Folie 85)



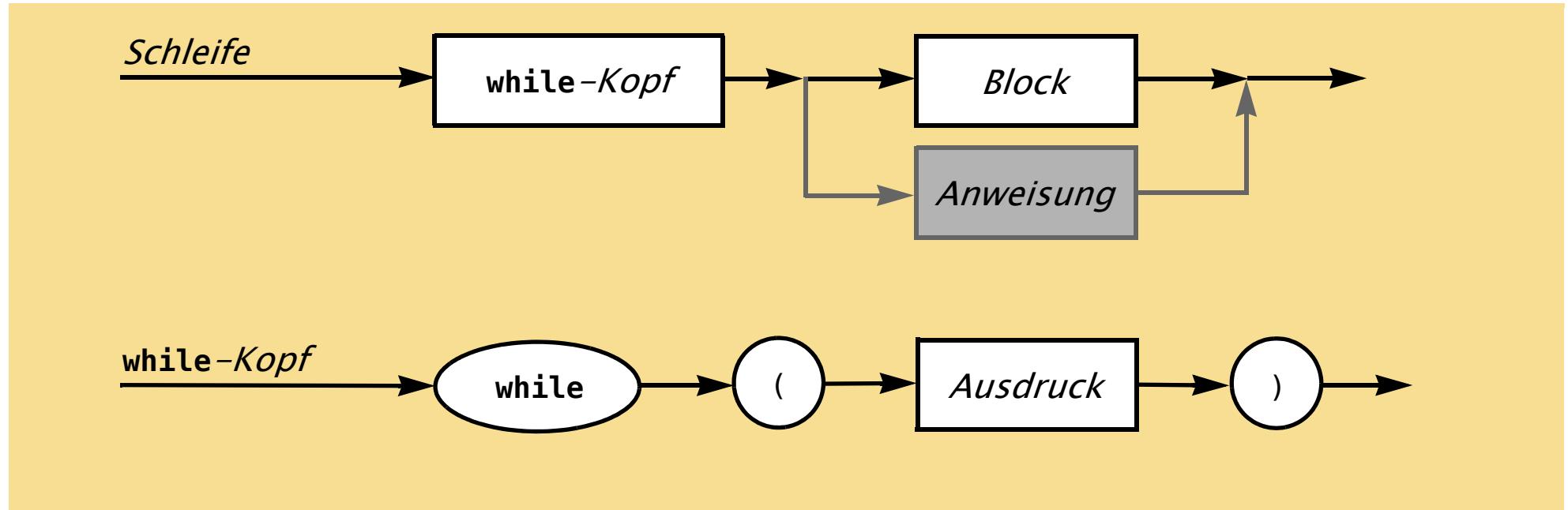
## Syntaxdiagramm zu *Anweisung*



## Syntaxdiagramm zu **Deklarationen** und **Zuweisung**



## Syntaxdiagramm zu *Schleife*



- ❑ Der Ausdruck eines Schleifenkopfs muss immer ein Ergebnis des Typs **boolean** liefern.
- ❑ Die **while**-Schleife sollte aus Gründen der Übersichtlichkeit und zur Fehlervermeidung *immer nur* mit einem nachfolgenden Block verwendet werden.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 2.2. Einfache Algorithmen - weitere Berechnungen

Dr. Stefan Dissmann  
Fakultät für Informatik



## Problemstellung «Bestimmung ganzzahlig teilbarer Summen von Quadraten»

Aufgabe:

Bestimme die Anzahl der Summen von Quadraten aller natürlichen Zahlen kleiner 1000, die ganzzahlig durch eine vorgegebene Zahl divisor teilbar sind.

oder mathematischer: Wie viele  $i$  mit  $i < 1000$  gibt es, so dass  
 $1^2 + 2^2 + 3^2 + 4^2 + \dots + i^2$  ganzzahlig durch divisor teilbar ist.

*Lösungsansatz:*

- Bestimme für jedes  $i$  die Summe der Quadratzahlen auf die bekannte Weise.
- Prüfe bei jeder Summe, ob sie ganzzahlig durch divisor teilbar ist.
- Falls das der Fall ist, merke, dass ein Treffer (mehr) erfolgt ist.
- Gib am Ende die Anzahl der Treffer zurück.

# Problemstellung «Bestimmung ganzzahlig teilbarer Summen von Quadraten»

## 1. Implementierung

```
public static int sumsDivisibleBy( int divisor )
{
    int value = 1;
    int sum = 0;
    int count = 0;
    while ( value < 1000 )
    {
        sum += value * value;
        if ( sum % divisor == 0 )
        {
            count++;
        }
        value++;
    }
    return count;
}
```

## Problemstellung «Bestimmung ganzzahlig teilbarer Summen von Quadraten»

### 1. Implementierung

```

public static int sumsDivisibleBy( int divisor )
{
    int value = 1;
    int sum = 0;
    int count = 0;
    while ( value < 1000 )
    {
        sum += value * value;
        if ( sum % divisor == 0 )
        {
            count++;
        }
        value++;
    }
    return count;
}
  
```

Bedingung

bedingte Anweisung

- Eine *bedingte Anweisung* beginnt mit dem Schlüsselwort **if**, auf das eine Bedingung und ein Block folgen.
- Eine *bedingte Anweisung* kontrolliert über die Bedingung die Ausführung des folgenden Blocks. Nur wenn die Bedingung den Wert **true** ergibt, wird der Block *einmal* ausgeführt.

## Problemstellung «Bestimmung ganzzahlig teilbarer Summen von Quadraten»

### 1. Implementierung

```

public static int sumsDivisibleBy( int divisor )
{
    int value = 1;
    int sum = 0;
    int count = 0;
    while ( value < 1000 )
    {
        sum += value * value;
        if ( sum % divisor == 0 )
        {
            count++;
        }
        value++;
    }
    return count;
}

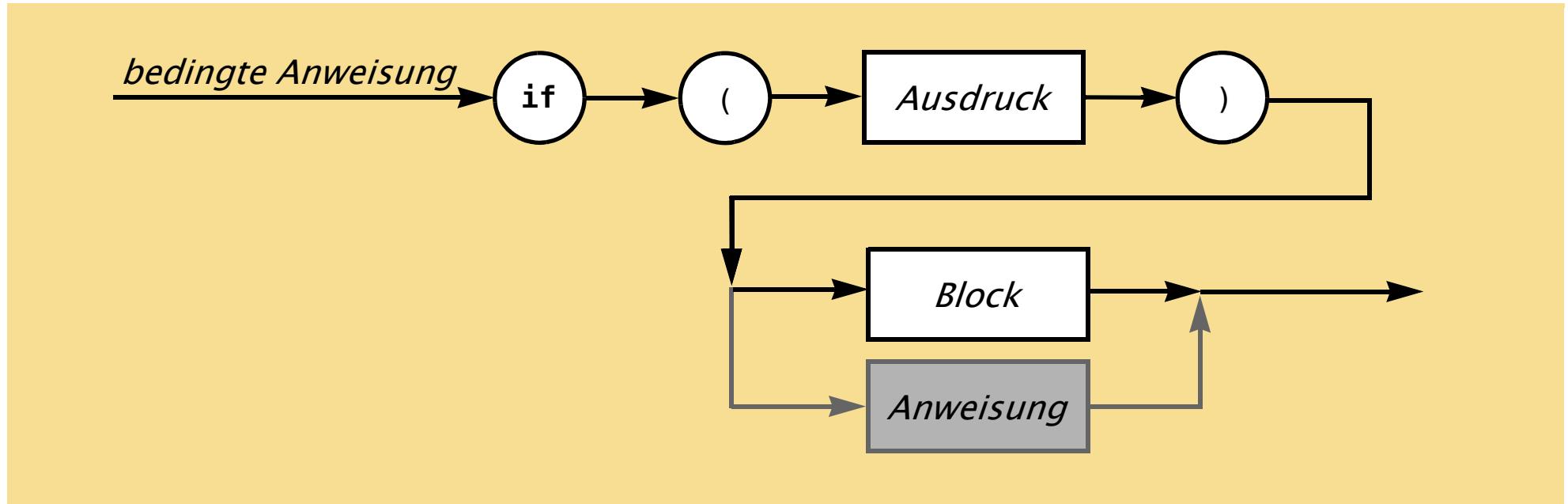
```



Modulo-Operator

- ❑ Der **Modulo**-Operator `%` liefert den Rest der *ganzzahligen* Division `sum / divisor`, also gilt: `sum % divisor == sum - (sum / divisor) * divisor`
- ❑ **Hinweis** Für negative Dividenden kann ein negativer Rest entstehen:  
 $-11 \% 2$  ergibt  $-1$ , da gilt:  $-11 - (-11 / 2) * 2 == -1$

## Syntaxdiagramm zu *bedingte Anweisung*



- ❑ Die bedingte Anweisung sollte aus Gründen der Übersichtlichkeit und Fehlervermeidung *immer nur* mit einem nachfolgenden Block verwendet werden.

## Problemstellung «Bestimmung ganzzahlig teilbarer Summen von Quadraten»

### 2. Implementierung

Übersichtlichere Darstellung mit einer **for-Schleife**,

die die wesentlichen Teile der Schleifenkonstruktion in ihrem Kopf zusammenfasst:

- Deklaration und Initialisierung von (Zähl-)Variablen
- Veränderung von (Zähl-)Variablen

```
public static int sumsDivisibleBy( int divisor )
{
    int sum = 0;
    int count = 0;
    for ( int value = 1; value < 1000; value++ )
    {
        sum += value * value;
        if ( sum % divisor == 0 )
        {
            count++;
        }
    }
    return count;
}
```

**for-Schleife**

**Deklaration und Initialisierung**

**Bedingung**

**Veränderung**

## Problemstellung «Bestimmung ganzzahlig teilbarer Summen von Quadraten»

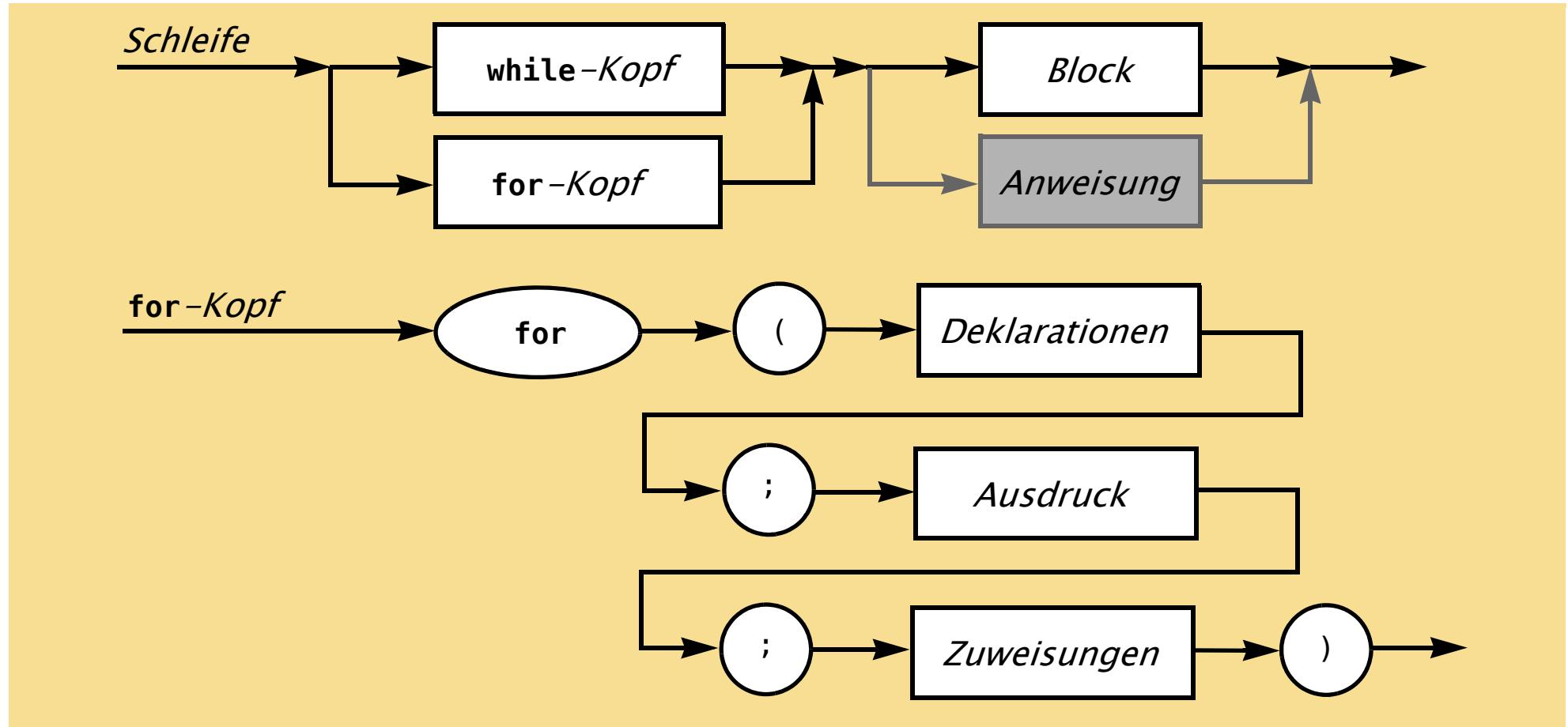
### 2. Implementierung

Hinweise zur **for**-Schleife:

- ❑ Der Abschnitt mit Deklaration und Initialisierung wird vor den anderen Bestandteilen der Schleife *genau einmal* ausgeführt.  
Anschließend wird die Bedingung ausgewertet.
  - ❑ Liefert die Bedingung das Ergebnis **true**, so werden die Anweisungen im Rumpf der Schleife ausgeführt.  
Liefert die Bedingung das Ergebnis **false**, so wird die Ausführung der Schleife beendet.
  - ❑ Der Veränderungsabschnitt wird *nach* den Anweisungen im Rumpf der Schleife ausgeführt.  
Anschließend wird die Bedingung ausgewertet.
- 
- ❑ Initialisierungs- und Veränderungsabschnitt können auch leer bleiben.  
Dann entspricht die **for**-Schleife einer **while**-Schleife:

```
for (; value < 1000;)
```

## Syntaxdiagramm zu *Schleife*



- ❑ Die **for**-Schleife sollte aus Gründen der Übersichtlichkeit *immer nur* mit einem nachfolgenden Block verwendet werden.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 2.3. Einfache Algorithmen - Primzahlbestimmung

Dr. Stefan Dissmann  
Fakultät für Informatik



## Problemstellung «Bestimmung von Primzahlen»

Aufgabe:

Bestimme alle Primzahlen bis zu einer vorgegebenen Zahl  $n$ .

Erinnerung:

Primzahlen sind natürliche Zahlen, die nur durch 1 und sich selbst ganzzahlig teilbar sind.

Beispiele:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...

*Lösungsansatz (Sieb des Eratosthenes, griechischer Mathematiker, geb. ca. 275 v. Chr.):*

- Schreibe alle natürlichen Zahlen ( $>1$  und  $\leq n$ ) als Liste auf.
- Beginne vorne in der Liste.
- Wähle die nächste Zahl aus und streiche alle ihre Vielfachen aus der Liste.
- Falls das Ende der Liste noch nicht erreicht ist, fahre fort.
- Falls das Ende der Liste erreicht ist, stehen in der Liste nur noch Primzahlen.



## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

*Lösungsansatz:*

- Schreibe alle natürlichen Zahlen ( $>1$  und  $\leq n$ ) als Liste auf.
- Beginne vorne in der Liste.
- Wähle die nächste Zahl aus und streiche alle ihre Vielfachen aus der Liste.
- Falls das Ende der Liste noch nicht erreicht ist, fahre fort.
- Falls das Ende der Liste erreicht ist, stehen in der Liste nur noch Primzahlen.



*Beispiel:*

- Schreibe die natürlichen Zahlen bis 30 auf:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Lösungsansatz:

- Schreibe alle natürlichen Zahlen ( $>1$  und  $\leq n$ ) als Liste auf.
- Beginne vorne in der Liste.
- Wähle die nächste Zahl aus und streiche alle ihre Vielfachen aus der Liste.
- Falls das Ende der Liste noch nicht erreicht ist, fahre fort.
- Falls das Ende der Liste erreicht ist, stehen in der Liste nur noch Primzahlen.



Beispiel:

- Schreibe die natürlichen Zahlen bis 30 auf:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

- Wähle 2 aus und streiche die Vielfachen von 2 aus der Liste:

2 3 5 7 9 11 13 15 17 19 21 23 25 27 29

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Lösungsansatz:

- Schreibe alle natürlichen Zahlen ( $>1$  und  $\leq n$ ) als Liste auf.
- Beginne vorne in der Liste.
- Wähle die nächste Zahl aus und streiche alle ihre Vielfachen aus der Liste.
- Falls das Ende der Liste noch nicht erreicht ist, fahre fort.
- Falls das Ende der Liste erreicht ist, stehen in der Liste nur noch Primzahlen.



Beispiel:

- Schreibe die natürlichen Zahlen bis 30 auf:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

- Wähle 2 aus und streiche die Vielfachen von 2 aus der Liste:

2 3 5 7 9 11 13 15 17 19 21 23 25 27 29

- Wähle 3 aus und streiche die Vielfachen von 3 aus der Liste:

2 **3** 5 7 11 13 17 19 23 25 27 29

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Lösungsansatz:

- Schreibe alle natürlichen Zahlen ( $>1$  und  $\leq n$ ) als Liste auf.
- Beginne vorne in der Liste.
- Wähle die nächste Zahl aus und streiche alle ihre Vielfachen aus der Liste.
- Falls das Ende der Liste noch nicht erreicht ist, fahre fort.
- Falls das Ende der Liste erreicht ist, stehen in der Liste nur noch Primzahlen.



Beispiel:

- Schreibe die natürlichen Zahlen bis 30 auf:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

- Wähle 2 aus und streiche die Vielfachen von 2 aus der Liste:

2 3 5 7 9 11 13 15 17 19 21 23 25 27 29

- Wähle 3 aus und streiche die Vielfachen von 3 aus der Liste:

2 3 5 7 11 13 17 19 23 25 29

- Wähle 5 aus und streiche die Vielfachen von 5 aus der Liste:

2 3 5 7 11 13 17 19 23 29

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Lösungsansatz:

- Schreibe alle natürlichen Zahlen ( $>1$  und  $\leq n$ ) als Liste auf.
- Beginne vorne in der Liste.
- Wähle die nächste Zahl aus und streiche alle ihre Vielfachen aus der Liste.
- Falls das Ende der Liste noch nicht erreicht ist, fahre fort.
- Falls das Ende der Liste erreicht ist, stehen in der Liste nur noch Primzahlen.



Bestandsaufnahme:

- natürliche Zahlen auflisten:  
Implementierung mit Schleife  
über Modulo-Operator möglich
- Vielfache ermitteln:  
Implementierung mit einer Schleife
- wiederholtes Ermitteln der Vielfachen:  
Implementierung mit einer Schleife,  
`System.out.println`
- verbleibende Zahlen ausgeben:  
  
*aber:*  
*Das Anlegen einer Liste von Zahlen ist mit den bekannten Hilfsmitteln unmöglich.  
Es können bisher nur einzelne Werte in Variablen abgelegt werden.*
- Benötigt wird also eine Möglichkeit, um viele Werte (des gleichen Typs) zu speichern.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 2.4. Einführung – Feld als einfache Datenstruktur

Dr. Stefan Dissmann  
Fakultät für Informatik



## Felder

Ein *Feld* (englisch Array) ist eine Folge einer festen Anzahl von Werten des gleichen Typs.

Der Zugriff auf ein Feld erfolgt über eine *Feldvariable*.

Der Umgang mit Feldvariablen und Felder wird auf den folgenden Folien eingeführt:

- Deklaration von Feldvariablen
- Erzeugen von Feldern
- Initialisieren von Feldern
- Zuweisen
  - von einzelnen Werten
  - von ganzen Feldern
- Zugriff auf einzelne Werte eines Feldes

# Felder

(Fortsetzung)

Erinnerung

`int value;`

deklariert eine Variable `value`,  
die genau einen `int`-Wert aufbewahren kann.

Deklaration einer *Feldvariablen*`int[]`**Sonderzeichen für Feld: []**

ist der *Typ aller Felder* mit `int`-Werten,  
`int` ist der *Grundtyp* dieser Felder.

`int[] values;`

deklariert eine Feldvariable `values`,  
die ein Feld mit dem Grundtyp `int` erreichbar macht.

Die Anzahl der Werte des Feldes ist *kein* Bestandteil  
der Deklaration der Feldvariablen.

## Felder

(Fortsetzung)

- Das *Erzeugen* eines Feldes erfordert das Festlegen der Anzahl der speicherbaren Werte.

```
int count = 25;  
int[] values;  
values = new int[5];  
  
values = new int[231];  
values = new int[count];  
values = new int[2 * count];
```

*Nach der Deklaration besitzt die Variable keinen Wert!  
Nach der Zuweisung ist der «Wert» der Variablen das neu erzeugte Feld.*

- Schlüsselwort **new** *erzeugt* ein neues Feld mit nachfolgend genannten Grundtyp.

- Die Angabe zwischen [...] legt die *Anzahl* der Elemente des Feldes fest.

- In jedem Element kann ein Wert des Grundtyps abgelegt werden.

- Auch möglich ist: `values = new int[0];`

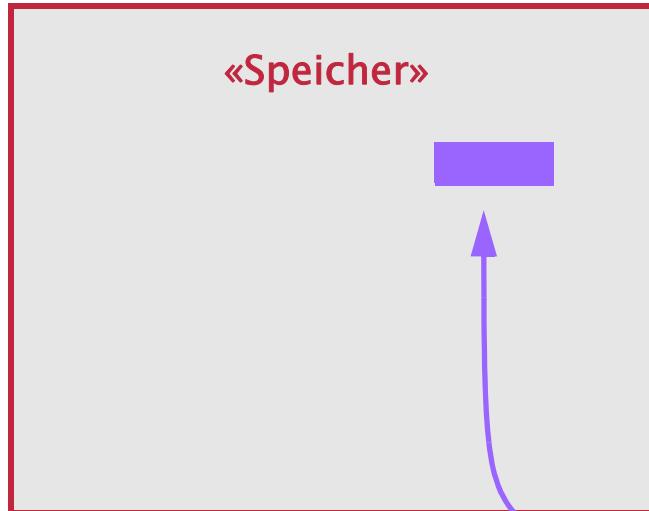
Es wird ein Feld *ohne* Elemente erzeugt.

- Die Anzahl der Elemente wird auch als *Länge* oder *Größe* des Feldes bezeichnet.

## einfaches Modell der Speicherung von Feldern

(vergleiche Folie 118)

Hardware:



Software:

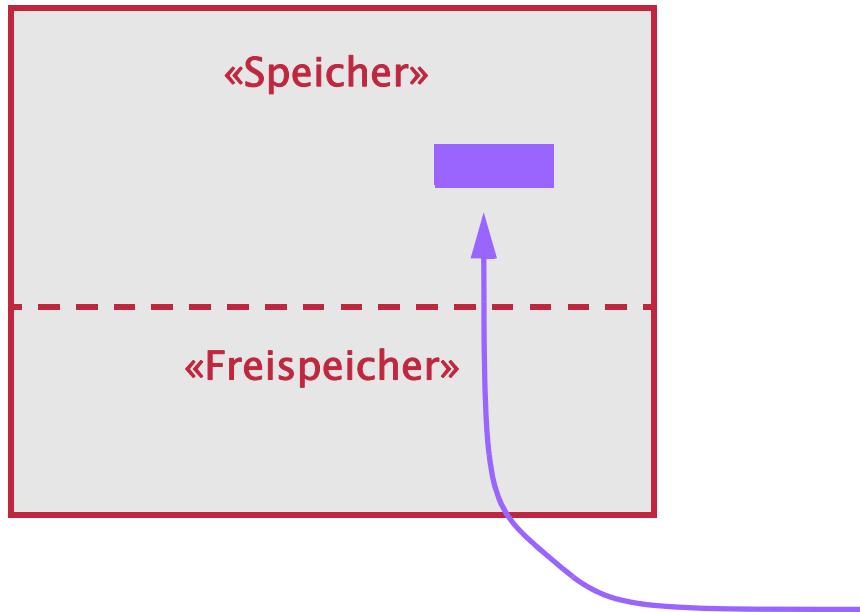
Die Anzahl der Elemente eines Feldes wird erst während der Ausführung festgelegt. Der Compiler kann daher *vor der Ausführung* nur einen Speicherbereich bestimmen, von dem aus auf das Feld mit seinen Elementen verwiesen wird.

`int[] values`

## einfaches Modell der Speicherung von Feldern

(Fortsetzung)

Hardware:



Software:

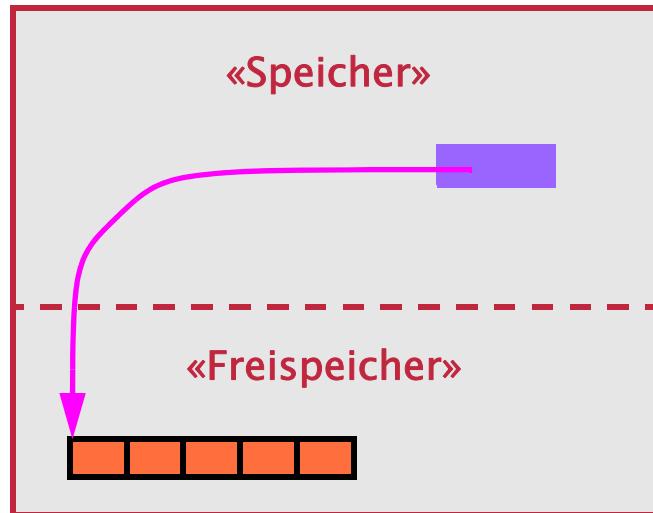
Für Felder und ihre Elemente wird ein Teil des Speichers (*Freispeicher*) reserviert, der während der Ausführung dynamisch den Feldvariablen zugeordnet wird.

`int[] values`

## einfaches Modell der Speicherung von Feldern

(Fortsetzung)

Hardware:



Software:

Für Felder und ihre Elemente wird ein Teil des Speichers (*Freispeicher*) reserviert, der während der Ausführung dynamisch den Feldvariablen zugeordnet wird.

```
int[] values = new int[5]
```

## Felder

(Fortsetzung)

- ☐ Feldvariablen können auch in der bekannten Weise initialisiert werden:

```
int[] values = new int[5];
```

Die Feldvariable `values` wird mit dem neu erzeugten Feld initialisiert.  
Alle Elemente des Feldes des Grundtyps `int` werden mit `0` initialisiert.

Visualisierung: `values` 

- ☐ *Nur* das Initialisieren kann auch über die direkte Angabe der Werte des Feldes erfolgen:

```
int[] values = { 17, 2, 23, -5, 0, 7 };
```

Die Feldvariable `values` wird mit einem neu erzeugten Feld der Länge 6 initialisiert,  
dessen Elemente die Werte `{ 17, 2, 23, -5, 0, 7 }` besitzen.

Visualisierung: `values` 

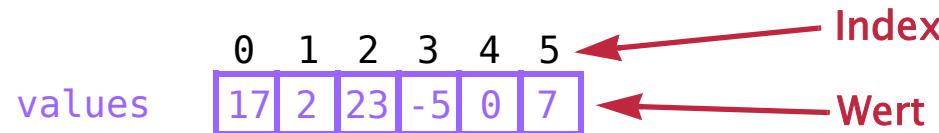
## Felder

(Fortsetzung)

- Die Elemente des Feldes sind durchnummeriert.  
Die Nummerierung beginnt mit 0.  
Die Nummer eines Elements wird als *Index* bezeichnet.

```
int[] values = { 17, 2, 23, -5, 0, 7 }
```

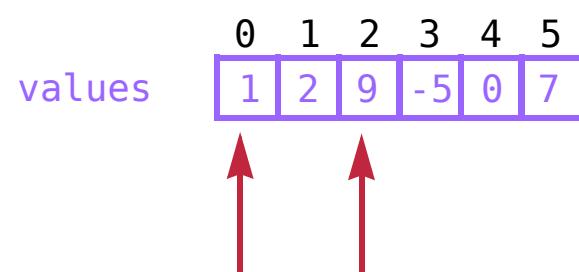
Visualisierung:



- Der Zugriff auf ein Element des Feldes erfolgt über die Angabe des Index in [...]:

values[2] = 9;	Dem 3. Element wird der Wert 9 zugewiesen.
values[0] = 1;	Dem 1. Element wird der Wert 1 zugewiesen.

Visualisierung:



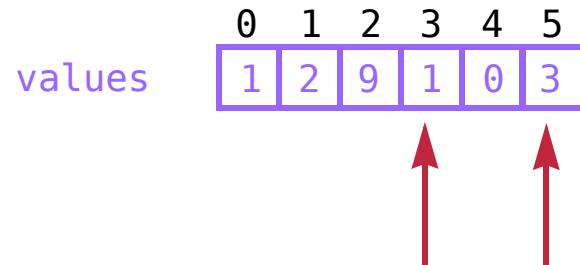
# Felder

(Fortsetzung)

- Der Index kann auch berechnet werden:

```
int position = 4;  
int[] values = { 1, 2, 9, -5, 0, 7 };  
values[position + 1] = 3;      Dem 6. Element wird der Wert 3 zugewiesen.  
values[position - 1] = 1;      Dem 4. Element wird der Wert 1 zugewiesen.
```

Visualisierung:



- Die Länge des Feldes kann ermittelt werden durch: `values.length`  
`values.length` liefert als Ergebnis: 6
- Zulässige Werte für den indexbasierten Zugriff zu den Elementen eines Feldes `x` liegen daher immer nur im Bereich `0, ..., x.length-1`

## Felder

(Fortsetzung)

- Der Zugriff auf die Elemente eines Feldes erfolgt ebenfalls indexbasiert:

```
int[] values = { 1, 2, 9, -5, 0, 7 };  
int v;  
v = values[3];
```

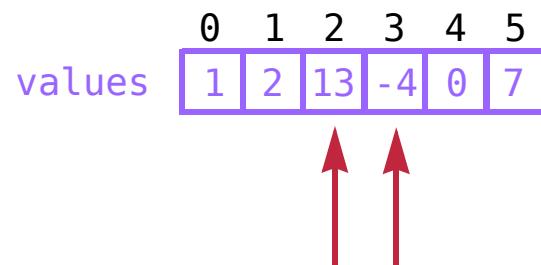
Der Variablen `v` wird der Wert `-5` zugewiesen.

- Auch die Anwendung von Zuweisungsoperatoren ist auf den Elementen eines Feldes möglich:

```
values[2] += 4;  
values[3]++;
```

Der Wert des 3. Elements wird um 4 erhöht.  
Der Wert des 4. Elements wird um 1 erhöht.

Visualisierung:



## Felder

(Fortsetzung)

- Zugriffe mit unzulässigem Indexwert:

```
int[] values = { 1, 2, 9, -5, 0, 7 };  
int i = values[8];  
  
values[-1] = 1;
```

Es wird versucht, den Wert eines nicht vorhandenen Elements abzurufen.  
Es wird versucht, einem nicht vorhandenen Element einen Wert zuzuweisen.

- Immer dann, wenn ein Wert für den indexbasierten Zugriff zu einem Element eines Feldes `x` angegeben wird, der nicht im Bereich `0, ..., x.length-1` liegt, entsteht eine *Ausnahme*(-situation) (siehe Folie 100).
- Fehlermeldungen bei der Programmausführung:

java.lang.ArrayIndexOutOfBoundsException: 8  
bzw. java.lang.ArrayIndexOutOfBoundsException: -1

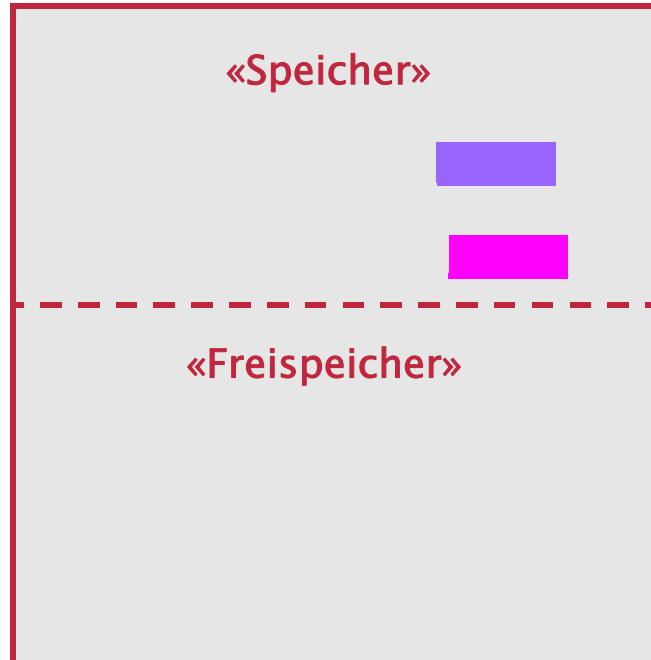
**fehlerhafte Indexangabe**

**Ursache des Fehlers**

## Zuweisungen an Feldvariablen (Modell)

(vergleiche Folie 168)

Hardware:



Software:

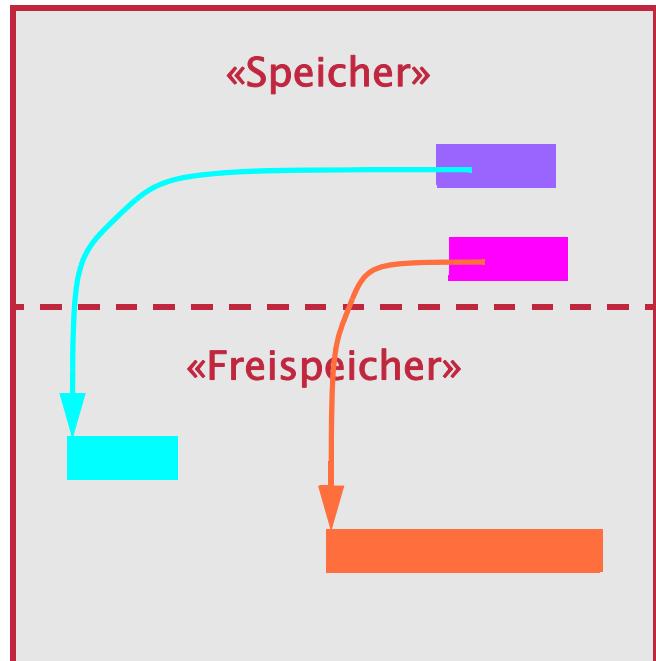
```
int[] values1;  
int[] values2;
```

Der Compiler legt fest, an welcher Stelle des Speichers die Feldvariablen eingerichtet werden.

## Zuweisungen an Feldvariablen (Modell)

(Fortsetzung)

Hardware:



Software:

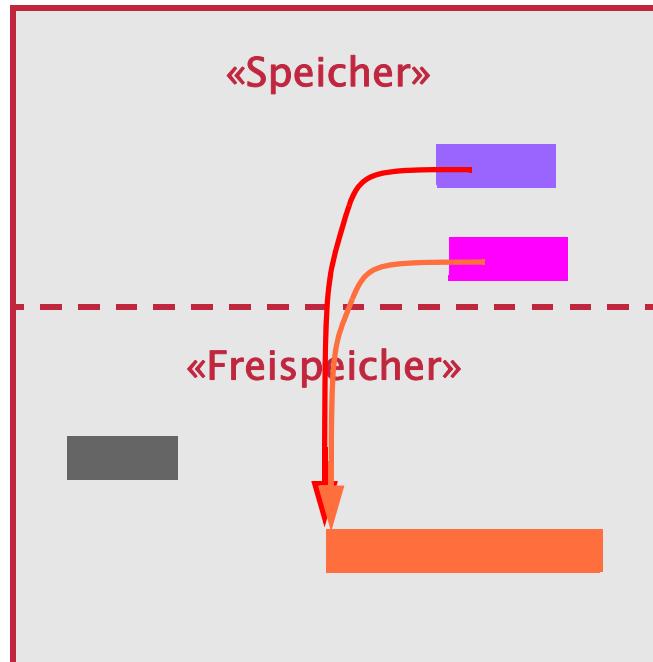
```
int[] values1;  
int[] values2;  
values1 = new int[3];  
values2 = new int[6];
```

Während der Ausführung werden Speicherbereiche in der benötigten Größe für die Felder reserviert. Die Positionen dieser Speicherbereiche werden in den Feldvariablen abgelegt.

## Zuweisungen an Feldvariablen (Modell)

(Fortsetzung)

Hardware:



Software:

```
int[] values1;  
int[] values2;  
values1 = new int[3];  
values2 = new int[6];  
values1 = values2;
```

Der Feldvariablen `values1` wird der Speicherbereich zugewiesen, den die Feldvariable `values2` kennt. Beide Feldvariablen verweisen jetzt auf den gleichen Speicherbereich (mit Platz für sechs `int`-Werte). Der Speicherbereich für drei `int`-Werte ist nicht mehr erreichbar.

## Zuweisungen an Feldvariablen

(Fortsetzung)

Die Zuweisung von Feldvariablen führt also zu folgendem Verhalten:

```
int[] values1 = { 4, 7, 13 };
```

values1 

```
int[] values2 = { 1, 2, 9, 1, 0, 7 };
```

values2 

```
values1 = values2;
```

Effekt: Die Feldvariable `values1` bezeichnet  
*dasselbe* Feld wie die Feldvariable `values2`.

values1   
values2 

Konsequenzen:

- ❑ Es entsteht *keine* Kopie des Feldes `values2`.
- ❑ Das Feld besitzt jetzt «*zwei Namen*».
- ❑ Das Feld  ist nicht mehr erreichbar.
- ❑ Alle weiteren Änderungen an den Elementen des Feldes  wirken in gleicher Weise für den Zugriff über `values1` oder `values2`.

## Kopieren von Feldinhalten

Das Kopieren von Feldinhalten muss also immer elementweise geschehen:

```
int[] values1;
int[] values2 = {1,2,3,4};  
  
values1 = new int[values2.length];  
for ( int i = 0; i < values2.length; i++ )  
{  
    values1[i] = values2[i];  
}
```

Erzeugen des neuen Feldes  
Bestimmen der Länge  
Zuweisen der einzelnen Werte

- ❑ Jedes Feld muss *explizit* mit **new** angelegt werden.
- ❑ Die Elemente eines neu angelegten Feldes sind immer mit einem (Null-)Wert initialisiert:
  - für **int[]** ist das **0**
  - für **double[]** ist das **0.0**
  - für **boolean[]** ist das **false**

## Kopieren von Feldinhalten

(Fortsetzung)

Das Kopieren lässt sich auch als Methode implementieren:

```
public static int[] copy( int[] in )
{
    int[] out = new int[in.length];
    for ( int i = 0; i < in.length; i++ )
    {
        out[i] = in[i];
    }
    return out;
}
```

Typ der Rückgabe:  
Feld mit Grundtyp int

## Kopieren von Feldinhalten

(Fortsetzung)

Auch das Übertragen der Inhalte eines Feldes in ein (passendes) zweites Feld muss elementweise erfolgen:

```
public static void moveTo( int[] source, int[] into )
{
    if ( source.length == into.length )
    {
        for ( int i = 0; i < source.length; i++ )
        {
            into[i] = source[i];
        }
    }
}
```

Für den Aufruf von `moveTo` muss ein passendes Feld bereitgestellt werden:

```
int[] values1 = new int[values2.length];
moveTo( values2, values1 );
```

Anschließend enthalten die Elemente des über `values1` erreichbaren Feldes die Werte der Elemente des Feldes, das durch `values2` erreicht wird.

## Kopieren von Feldinhalten

(Fortsetzung)

Ausführung der Methode `moveTo`:

```
public static void moveTo( int[] source, int[] into )
{
    if ( source.length == into.length )
    {
        for ( int i = 0; i < source.length; i++ )
        {
            into[i] = source[i];
        }
    }
}
```

- Beim Aufruf der Methode `moveTo` wird der Wert des zweiten Arguments dem Parameter `into` zugewiesen.
- Da es sich um eine Feldvariable handelt, erreicht `into` jetzt das Feld, das auch über die als zweites Argument übergebene Feldvariable erreichbar ist.
- Die Zuweisung in der Methode `moveTo` ändert also die Elemente des über die als zweites Argument übergebene Feldvariable auch außerhalb der Methode sichtbaren Feldes.
- Auf diese Weise ist das geänderte Feld nach der Ausführung der Methode erreichbar.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 2.5. Einführung – Primzahlbestimmung in Java

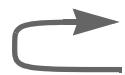
Dr. Stefan Dissmann  
Fakultät für Informatik



## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Lösungsansatz:



- Schreibe alle natürlichen Zahlen ( $>1$  und  $\leq n$ ) als Liste auf.
- Beginne vorne in der Liste.
- Wähle die nächste Zahl aus und streiche alle ihre Vielfachen aus der Liste.
- Falls das Ende der Liste noch nicht erreicht ist, fahre fort.
- Falls das Ende der Liste erreicht ist, stehen in der Liste nur noch Primzahlen.

Überlegungen zur Realisierung:

- Die sich im Verlauf der Ausführung ändernde Liste von natürlichen Zahlen wird durch ein Feld des Grundtyps **boolean** repräsentiert, bei dem der Wert **true** am Index *i* anzeigt, dass *i* eine Zahl in der Liste ist.
- Soll eine Zahl *i* aus der Liste gestrichen werden, so wird das Element am Index *i* auf den Wert **false** gesetzt.
- Nach Beenden des Algorithmus sind die Primzahlen dann durch alle Indizes gegeben, deren zugehöriger Wert **true** ist.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Implementierung:

```
public static void computePrimes( int n )
{
    boolean[] numbers = initializeNumbers( n ); ← erzeugt ein
                                                passendes Feld

    inspectNumbers( numbers ); ← markiert Primzahlen

    show( numbers ); ← zeigt die Primzahlen
                      auf dem Bildschirm an
}
```

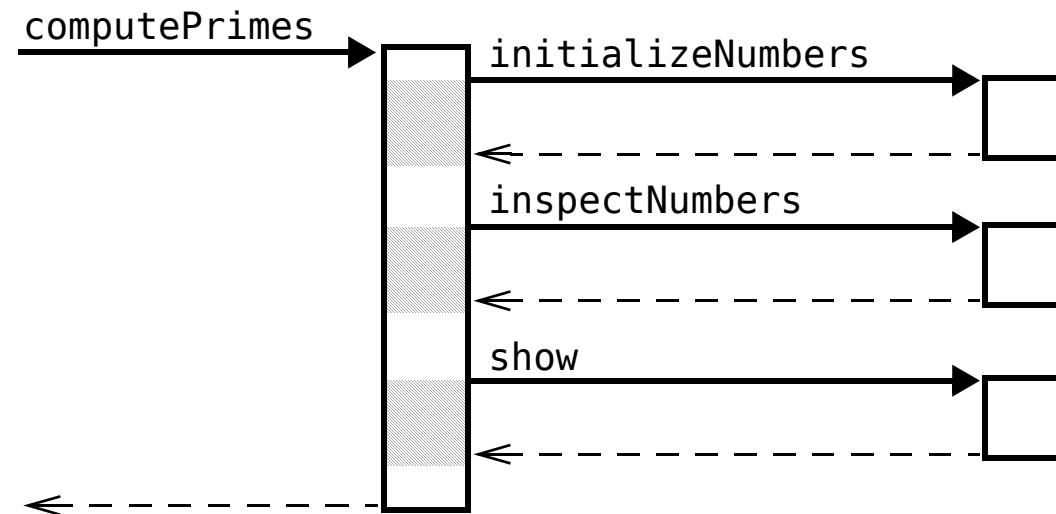
## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Implementierung:

```
public static void computePrimes( int n )
{
    boolean[] numbers = initializeNumbers( n );
    inspectNumbers( numbers );
    show( numbers );
}
```

Ausführung:



## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 1: Die Methode `initializeNumbers` erzeugt ein Feld des Typs **boolean** der Länge `n+1`, bei dem in allen Elementen mit Index `i>1` der Wert **true** steht.

```
public static boolean[] initializeNumbers( int n )
{
    boolean[] numbers = new boolean[n+1];
    for ( int i = 2; i < numbers.length; i++ )
    {
        numbers[i] = true;
    }
    return numbers;
}
```

- Visualisierung:

numbers	0	1	2	3	4	5	...
	false	false	true	true	true	true	

- Alle Werte des Feldes sind beim Erzeugen mit **false** initialisiert, da **false** der Nullwert des Typs **boolean** ist.
- Die Methode ändert die Werte für die Indizes 0 und 1 nicht.
- Der größte für das Feld `numbers` zulässige Index ist `n`.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 2: *1. Implementierung* der Methode `inspectNumbers`, die alle Zahlen betrachtet.

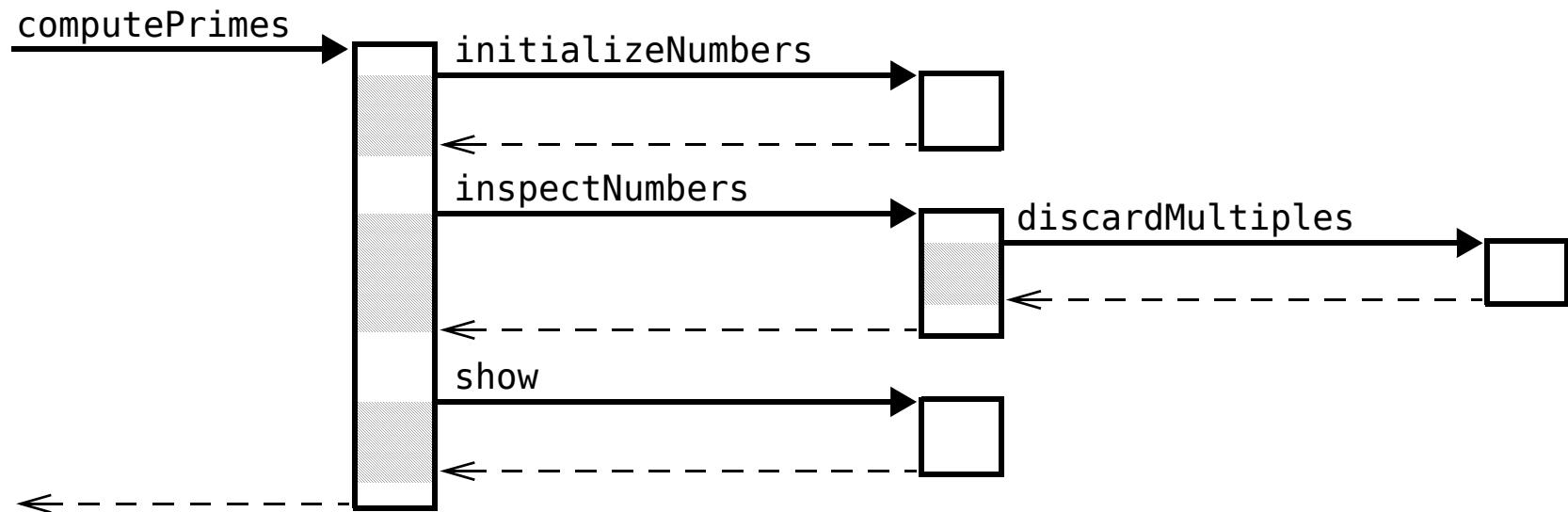
```
public static void inspectNumbers( boolean[] numbers )
{
    for ( int i = 2; i < numbers.length; i++ )
    {
        if ( numbers[i] ) ← nur für nicht gestrichenen Zahlen
        {
            discardMultiples( numbers, i ); ← streicht die Vielfachen
        }                                     von i aus numbers
    }
}
```

- Nach Ausführung von `inspectNumbers` besitzt `numbers` nur noch für solche Indizes den Wert `true`, die Primzahlen sind.
- `if ( numbers[i] )`     *entspricht*     `if ( numbers[i] == true )`

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

```
public static void inspectNumbers( boolean[] numbers )
{
    for ( int i = 2; i < numbers.length; i++ )
    {
        if ( numbers[i] )
        {
            discardMultiples( numbers, i );
        }
    }
}
```



## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 2: *1. Implementierung* der Methode `discardMultiples`, die das Aussieben durchführt.

```
public static void discardMultiples( boolean[] numbers, int i )
{
    for ( int j = i+1; j < numbers.length; j++ )
    {
        if ( j % i == 0 )
        {
            numbers[j] = false;
        }
    }
}
```

Streichen wird als  
«setze auf false» realisiert

- In jedem Element des Feldes `numbers`, dessen Index `j` ein ganzzahliges Vielfaches von `i` ist, wird der Wert `false` eingetragen.
- `j` ist dann ein ganzzahliges Vielfaches von `i`, wenn die Division `j/i` zu keinem Rest führt.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 2: *1. Implementierung* der Methode `discardMultiples`, die das Aussieben durchführt.

```
public static void discardMultiples( boolean[] numbers, int i )
{
    for ( int j = i+1; j < numbers.length; j++ )
    {
        if ( j % i == 0 )
        {
            numbers[j] = false;
        }
    }
}
```

### Analyse

- ❑ Es werden sehr viele überflüssige %-Berechnungen ausgeführt:  
beispielsweise wird `i+1` niemals ganzzahlig durch `i` teilbar sein für `i>1`.
- ❑ Falls `i` die Primzahl ist, deren Vielfache gestrichen werden sollen, so sind ja bereits alle Vielfachen `p*i` gestrichen worden, bei denen `p` eine Primzahl ist und `p<i` gilt.
- ❑ Das erste Vielfache `j`, das gestrichen werden muss, ist also `j=i*i`.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 2: *2. Implementierung* der Methode `discardMultiples`, die das Aussieben durchführt.

```
public static void discardMultiples( boolean[] numbers, int i )
{
    for ( int j = i*i; j < numbers.length; j++ )
    {
        if ( j % i == 0 )
        {
            numbers[j] = false;
        }
    }
}
```

### weitere Analyse

- ❑ Es werden trotz der Änderung noch sehr viele überflüssige %-Berechnungen ausgeführt, da ab  $i^2$  jeder Wert überprüft wird.
- ❑ Vielfache von  $i$  haben untereinander aber immer genau einen Abstand von  $i$ , es sind also die Werte  $i^2, i^2+i, i^2+i+i, \dots$   
Es reicht also, gezielt die Elemente an diesen Indizes unmittelbar auf **false** zu setzen.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 2: *3. Implementierung* der Methode `discardMultiples`, die das Aussieben durchführt.

```
public static void discardMultiples( boolean[] numbers, int i )
{
    for ( int j = i*i; j < numbers.length; j+=i )
    {
        numbers[j] = false;
    }
}
```

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 2: *Erinnerung: 1. Implementierung* der Methode `inspectNumbers`, die alle Zahlen betrachtet.

```
public static void inspectNumbers( boolean[] numbers )
{
    for ( int i = 2; i < numbers.length; i++ )
    {
        if ( numbers[i] )
        {
            discardMultiples( numbers, i );
        }
    }
}
```

- Bei der Analyse von `discardMultiples` wurde festgestellt:  
Das erste Vielfache, das in `discardMultiples` gestrichen werden muss, ist an der Position  $i \cdot i$ .
- Daraus folgt direkt, dass für alle  $i$ , die größer als die Wurzel aus `numbers.length - 1` sind,  
*kein Vielfaches mehr gefunden werden kann*, das noch gelöscht werden muss.
- Die Berechnung dieser Wurzel muss nur einmal erfolgen.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

### Berechnung der (Quadrat-)Wurzel einer Zahl

- Java stellt eine entsprechende Methode in der Klasse `Math` zur Verfügung:  
`double sqrt( double a )` (`sqrt` steht für *square root*)
- `double sqrt( double a )` berechnet die Wurzel des als Argument übergebenen Wertes und liefert ein Ergebnis des Typs `double`.
- Der Aufruf der Methode erfolgt in der Form: `Math.sqrt(...)` \*)

\*) Die genaue Bedeutung dieser Schreibweise wird später erläutert.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

- ❑ Die Klasse `Math` stellt mathematische Funktionen als Methoden bereit.
- ❑ Informationen finden sich in der Literatur oder in der offiziellen Java-Dokumentation unter:

<https://docs.oracle.com/en/java/javase/>

- ❑ Diese Methoden können in der Form `Math....` aufgerufen werden.
- ❑ Die Klasse `Math` bietet auch eine Methode `double pow( double a, double b )`, mit der (allgemeine) Potenzen  $a^b$  berechnet werden können.  
Da dabei jedoch Ungenauigkeiten auftreten können, erfolgt in der Methode `discardMultiples` die Berechnung des (einfach zu berechnenden) Quadrats als exakte `int`-Operation durch `i*i`.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 2: *2. Implementierung* der Methode `inspectNumbers`, die alle Zahlen betrachtet.

```
public static void inspectNumbers( boolean[] numbers )
{
    double limit = Math.sqrt( numbers.length );
    for ( int i = 2; i <= limit; i++ )
    {
        if ( numbers[i] )
        {
            discardMultiples( numbers, i );
        }
    }
}
```

A red arrow points from the word "limit" in the code to the text "Methode aus der Klasse Math" in red.

- Ergebnisse des Typs `double` können immer geringe Rundungsungenauigkeiten aufweisen. Vorbeugend wird hier die Wurzel aus dem etwas zu großen Wert `numbers.length` gezogen.
- Der Vergleich `<=` zwischen einem `double`- und einem `int`-Wert ist möglich, da `int` kompatibel zu `double` ist.
- Der `int`-Wert wird für den Vergleich implizit in einen `double`-Wert umgewandelt.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 2: Experimentelle Überprüfung der Laufzeiten  
der Versionen der Methoden `inspectNumbers` und `discardMultiples`

Laufzeiten für die Bestimmung der Primzahlen bis 100000

(Die absoluten Werte sind prozessorabhängig, interessant ist nur die relative Veränderung.)

- ❑ erste Implementierungen:  
in mehreren Versuchen immer knapp unter 4 Sekunden
  
- ❑ abschließende Implementierung:  
in mehreren Versuchen immer unter 10 Millisekunden

*Die kritische Analyse der ersten Implementierung hat zu Änderungen geführt,  
die die Laufzeit um den Faktor 400 verbessert haben.*

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

Schritt 3: Ausgabe der ermittelten Primzahlen.

```
public static void show( boolean[] numbers )
{
    for ( int i = 0; i < numbers.length; i++ )
    {
        if ( numbers[i] )
        {
            System.out.print( i + " " );
        }
    }
    System.out.println();
}
```

Ausgabe OHNE Zeilenumbruch

NUR Zeilenumbruch (ohne Ausgabe)

## Problemstellung «Bestimmung von Primzahlen» (von Folie 184)

(Fortsetzung)

```
public static void computePrimes( int n )
{
    boolean[] numbers = initializeNumbers( n );
    inspectNumbers( numbers );
    show( numbers );
}
```

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

zusätzliche Funktionalität: Ermitteln der Anzahl der bestimmten Primzahlen.

```
public static int countPrimeNumbers( boolean[] numbers )
{
    int count = 0;
    for ( boolean b : numbers )
    {
        if ( b )
        {
            count++;
        }
    }
    return count;
}
```

for-each-Schleife

- ❑ Die **for-each**-Schleife erlaubt den sequentiellen Zugriff auf alle Elemente *ohne* explizite Angabe der Indizes der Elemente.
- ❑ Bei jedem Durchlauf durch die **for-each**-Schleife wird der Variablen **b** der Wert des nächsten Elements von **numbers** zugewiesen.
- ❑ Die Position des Werts im Feld kann im Block der **for-each**-Schleife nicht festgestellt werden.
- ❑ Mit der **for-each**-Schleife ist daher nur ein Lesen möglich.

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

### for-each-Schleife

```
for ( boolean b : numbers )
{
    ...
}
```

for-each-Schleife

- Diese **for-each**-Schleife *entspricht*:

```
boolean b;
for ( int i = 0; i < numbers.length; i++)
{
    b = numbers[i];
    ...
}
```

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

zusätzliche Funktionalität: Erzeugen eines **int**-Feldes mit den ermittelten Primzahlen.

```
public static int[] primeNumbersToArray( boolean[] numbers )
{
    int[] primes = new int[countPrimeNumbers( numbers )];
    int nextPosition = 0;
    for (int i = 0; i < numbers.length; i++)
    {
        if ( numbers[i] )
        {
            primes[nextPosition] = i;
            nextPosition++;
        }
    }
    return primes;
}
```

bestimmt die Größe  
des benötigten Feldes

## Problemstellung «Bestimmung von Primzahlen»

(Fortsetzung)

zusätzliche Funktionalität: Ausgeben eines `int`-Feldes.

```
public static void show( int[] primes )
{
    for ( int p : primes )
    {
        System.out.print( p + "  " );
    }
    System.out.println();
}
```

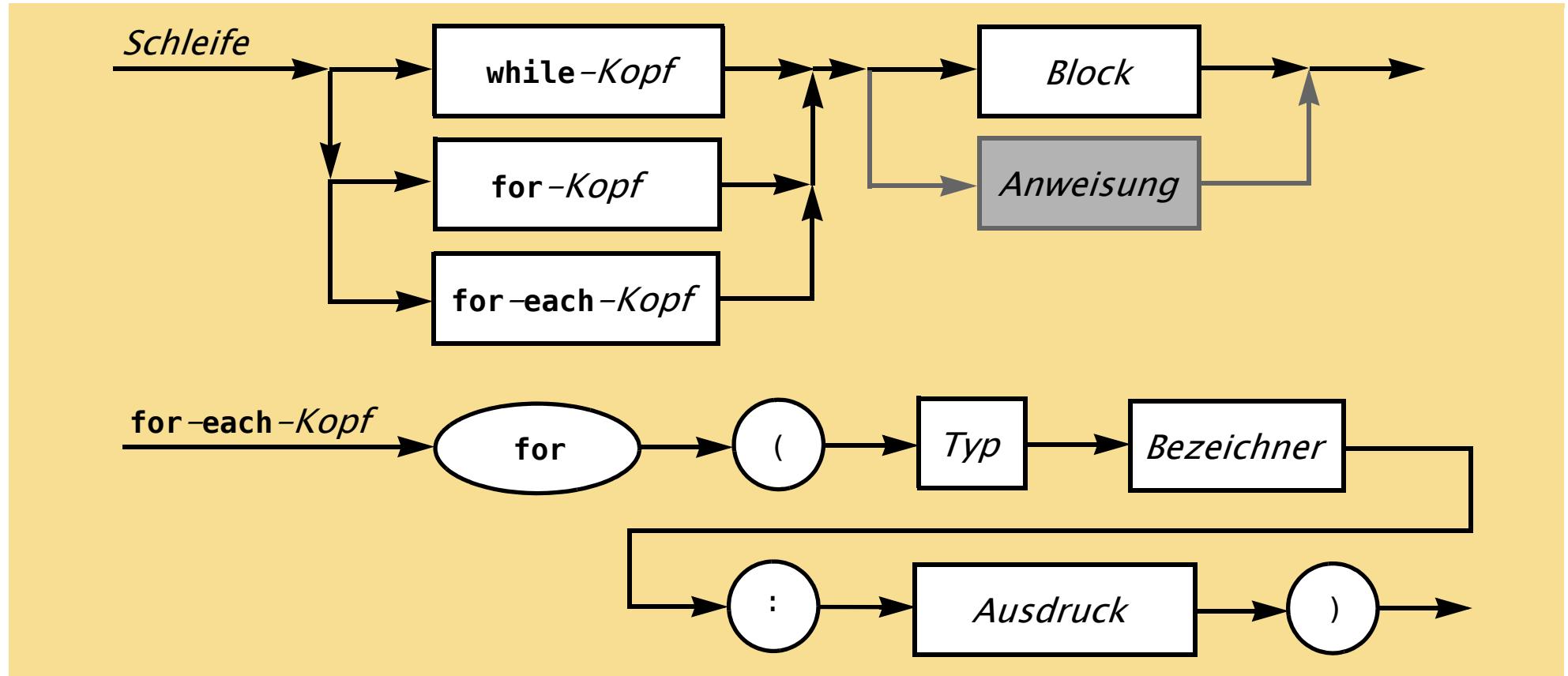
- ❑ Zwei Methoden mit dem Namen `show` sind in einer Klasse möglich, da sich die Signaturen im Typ des Parameters unterscheiden:

```
public static void show( boolean[] numbers )
```

```
public static void show( int[] primes )
```

- ❑ Beim Aufruf der Methode `show` kann daher anhand des Typs des Parameters entschieden werden, welche Implementierung gemeint ist.

## Syntaxdiagramm zu *Schleife* und *for-each-Kopf*



- ❑ Der Ausdruck im Kopf der **for-each**-Schleife muss ein Feld als Ergebnis liefern.
- ❑ Der Typ des Bezeichners muss mit dem Grundtyp des Feldes kompatibel sein.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## **3.1. Klasse und Objekt - Klasse für Rationale Zahlen - Teil 1**

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-205

## Lernziele des Kapitels 3. Klasse und Objekt

Nach Durcharbeiten des Kapitels 3. Klasse und Objekt sollen die teilnehmenden Studierenden

- eine *Klasse* deklarieren können
- in einer Klasse (Instanz-)*Attribute* anlegen können
- in einer Klasse *Konstruktoren* anlegen können
- in einer Klasse (Instanz-)*Methoden* anlegen können
- Objekte* einer Klasse erzeugen können
- die Bedeutung von *Referenzvariablen* verstanden haben
- Methoden eines Objekts über eine Referenzvariable aufrufen können
- die Bedeutung des Wertes `null` kennen
- mehrere Klassen durch Deklarationen und Aufrufe im Verbund arbeiten lassen können
- die dabei entstehenden Klassen- und Objektstrukturen visualisieren können
- die Klasse `String` nutzen können

## Motivation für das Konzept «Klasse»

Rückblick auf Konzept *Feld*:

- ❑ Ein Feld dient dazu, viele Werte des gleichen Grundtyps unter einem Namen abzulegen.
- ❑ Diese Werte sind nacheinander angeordnet und werden über aufeinander folgende Indexwerte adressiert.
- ❑ Ein Feld ist daher gut geeignet, wenn alle diese Werte auch eine ähnliche Bedeutung besitzen.
  
- ❑ Ein Feld ist *weniger* geeignet, um mehrere zusammengehörende Werte des gleichen Typs mit *unterschiedlicher* Semantik aufzubewahren.

Beispiel:

Ein Bruch (rationale Zahl) besteht aus zwei ganzzahligen Werten, die aber unterschiedliche Bedeutungen – Zähler und Nenner – haben.

- ❑ Ein Feld ist *ungeeignet*, um mehrere zusammengehörende Werte mit unterschiedlichen Typen aufzubewahren.

Beispiel:

Eine Studierende hat einen Namen, eine Matrikelnummer, ein Studienfach, eine Semesterzahl, ...

## Konzept «Klasse»

Klassen sind seit dem ersten Beispiel bekannt (Folie 55):

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

- Bisher kennengelernt:
  - Eine Klasse beinhaltet Methoden.
  - Methoden einer Klasse können aufgerufen und ausgeführt werden.
  - Methoden aus anderen Klassen können aufgerufen und ausgeführt werden (z.B. `Math.sqrt`).

## Konzept «Klasse»

(Fortsetzung)

Klassen sind seit dem ersten Beispiel bekannt (Folie 55):

```
public class FirstProgram
{
    public static void main( String[] args )
    {
        System.out.println( "hello world" );
    }
}
```

- ❑ Bisher kennengelernt:
  - Eine Klasse beinhaltet Methoden.
  - Methoden einer Klasse können aufgerufen und ausgeführt werden.
  - Methoden aus anderen Klassen können aufgerufen und ausgeführt werden (z.B. Math.sqrt).
- ❑ *Nun werden weitere Eigenschaften von Klassen vorgestellt:*
  - Eine Klasse kann *auch* beschreiben, wie Werte gespeichert werden.
  - Von einer Klasse können *mehrere* Einheiten erzeugt werden:  
mehrere Brüche, mehrere Studierende
  - Der Aufruf einer Methode der Klasse bezieht sich *nur auf eine* bestimmte Einheit.

## Konzept «Klasse»

(Fortsetzung)

Vorgehen:

- ❑ Beispiel 1: Klasse für *rationale Zahlen* (Brüche) vorstellen
- ❑ Beispiel 2: Klasse für *Vorlesungen* vorstellen
- ❑ Beispiel 3: Klasse für *Studierende* vorstellen
- ❑ wesentliche Aspekte von Klassen zusammenfassen
- ❑ Grundprinzipien objektorientierter Software herausarbeiten

## Problemstellung «Rationale Zahlen»

Erinnerung an die Schulmathematik:

$r$  ist eine *rationale Zahl*  
 $\Leftrightarrow r = z/n$  und  $z$  und  $n$  sind ganze Zahlen und  $n \neq 0$

Visualisierung:

$$\frac{101}{231} \quad \text{Bruch, } z \text{ heißt Zähler, } n \text{ heißt Nenner}$$

Ziel:

Definiere eine Klasse, die den Typ der rationalen Zahlen mit folgenden Eigenschaften realisiert:

- Die Grundrechenarten  $+, -, *, /$  sind verfügbar und arbeiten ohne Genauigkeitsverlust.
- Es treten nur gekürzte rationale Zahlen auf:  
Zähler und Nenner haben keinen gemeinsamen Teiler, der größer als 1 ist.
- Alle rationalen Zahlen besitzen immer einen positiven Nenner,  
negative rationale Zahlen besitzen einen negativen Zähler.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Die Klasse für den Umgang mit rationalen Zahlen wird schrittweise erweitert:

```
public class Fraction {  
    private int numerator;      // Zaehler  
    private int denominator;   // Nenner  
}
```

**Deklaration der Klasse (bereits bekannt)**

**Attribute**

- ❑ Ein *Instanzattribut* speichert einen Wert des bei seiner Deklaration vorgegeben Typs:  
`int numerator` deklariert das Attribut `numerator`, das `int`-Werte speichern kann.
- ❑ Ein Instanzattribut ist im Rumpf der Klasse bekannt und zugreifbar.
- ❑ Solange ausschließlich Instanzattribute betrachtet werden,  
werden diese im Folgenden einfach als *Attribute* bezeichnet.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Ziel:

*Die Klasse soll als Vorlage für viele gleichartige Objekte dienen.*

Visualisierung:

### Deklaration

```
public class Fraction
{
    private int numerator;
    private int denominator;
```

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

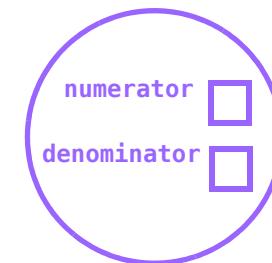
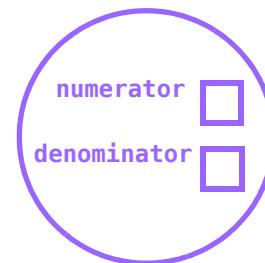
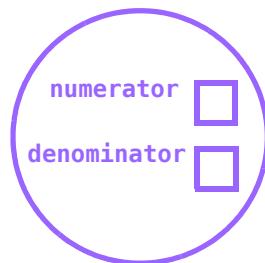
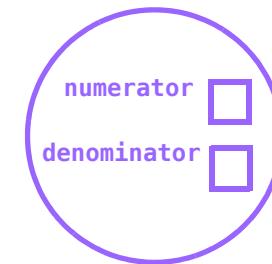
Ziel:

*Die Klasse soll als Vorlage für viele gleichartige Objekte dienen.*

Visualisierung:

### Deklaration

```
public class Fraction
{
    private int numerator;
    private int denominator;
```



Objekte der Klasse Fraction

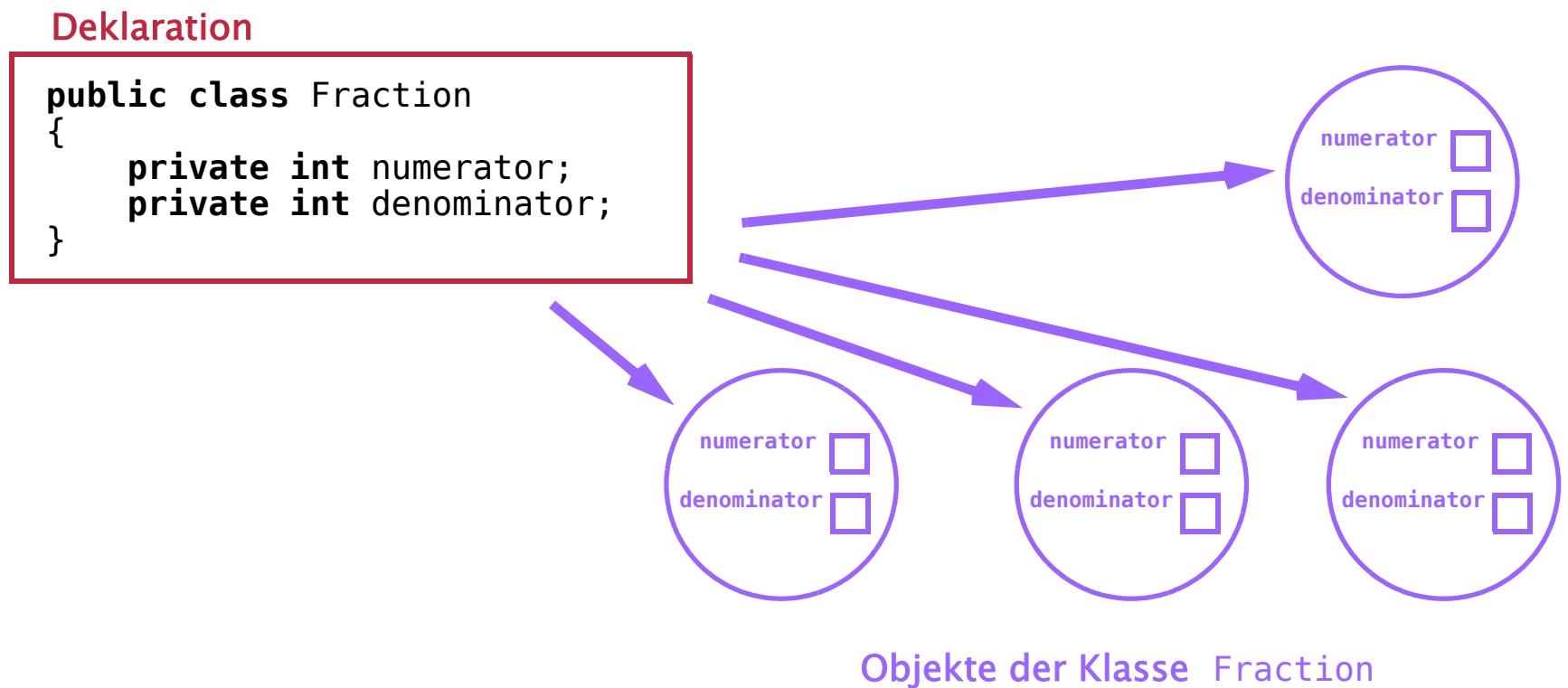
## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Ziel:

*Die Klasse soll als Vorlage für viele gleichartige Objekte dienen.*

Visualisierung:



## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Die Klasse für den Umgang mit rationalen Zahlen wird schrittweise erweitert:

```
public class Fraction
{
    private int numerator;      // Zaehler
    private int denominator;   // Nenner

Konstruktor public Fraction ( int num, int denom )
{
    // ensure:
    // denominator > 0
    // there exists no x>1 with numerator%x==0 and denominator%x==0
}
```

- Ein *Konstruktor erzeugt ein Objekt* der Klasse und ähnelt im Aufbau einer Methode.
- Ein Konstruktor besitzt keine Angabe für einen Rückgabewert.
- Ein Konstruktor besitzt immer den gleichen Namen wie die Klasse.
- Jedes Objekt der Klasse `Fraction` besitzt eigene *Attribute* `numerator` und `denominator`.
- Der Konstruktor dient dazu, die Attribute des erzeugten Objekts so mit Werten zu belegen, dass das Objekt sinnvoll benutzbar ist.

## Problemstellung «Rationale Zahlen» Visualisierung

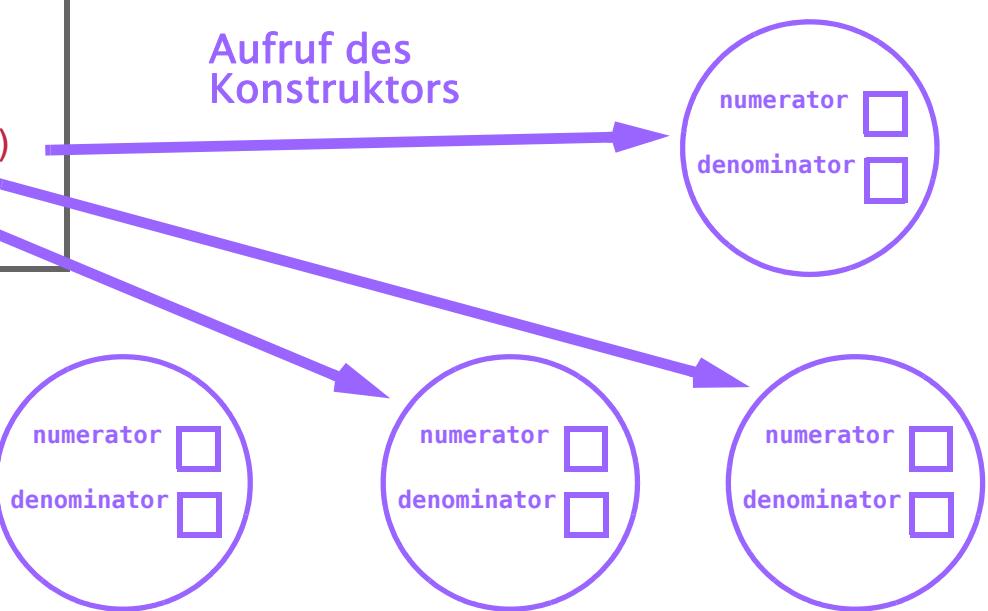
(Fortsetzung)

Aufgabe des Konstruktors:

Deklaration

```
public class Fraction
{
    private int numerator;
    private int denominator;
    public Fraction(int num, int denom)
    { ... }
}
```

Aufruf des  
Konstruktors



Objekte der Klasse Fraction

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Die Klasse für den Umgang mit rationalen Zahlen wird schrittweise erweitert:

```
public class Fraction
{
    private int numerator;           // Zahler
    private int denominator;         // Nenner

    public Fraction ( int num, int denom )
    {
        // ensure:
        // denominator > 0
        // there exists no x with numerator%x==0 and denominator%x==0
    }
}
```

Zugriffsrecht

- ❑ Für jedes Attribut und jeden Konstruktor wird ein *Zugriffsrecht* festgelegt.
- ❑ Das Zugriffsrecht **private** (*privat*) beschränkt die Sichtbarkeit auf genau die Klasse, in der die Deklaration erfolgt:  
Das Attribut `numerator` ist nur innerhalb der Klasse `Fraction` bekannt.
- ❑ Das Zugriffsrecht **public** (*öffentlich*) schränkt die Sichtbarkeit nicht ein:  
Der Konstruktor `Fraction( ... )` kann auch in anderen Klassen aufgerufen werden.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Zugriffsrechte:

- ❑ Die Klasse Fraction bietet anderen Klassen bisher nur genau den Konstruktor an, da die beiden Attribute als **private** deklariert sind.
- ❑ Attribute *sollten* immer als **private** deklariert werden.  
(Die Gründe werden hoffentlich im Verlauf der folgenden Vervollständigung der Klasse Fraction deutlich.)
- ❑ Konstruktoren werden nur in sehr speziellen Fällen *nicht* als **public** deklariert.

Anmerkungen:

- ❑ Bei dem Fehlen eines explizit angegebenen Zugriffsrechts besitzt ein Element der Klasse (Attribut, Methode, Konstruktor) implizit das Zugriffsrecht package, das hier noch nicht erläutert werden kann.
- ❑ Attribute können als **public** deklariert werden.  
Dieses sollte aber unbedingt vermieden werden.  
Beispiel (*nicht zur Nachahmung*): length ist ein öffentliches Attribut eines Feldes.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Planung des Konstruktors:

- ❑ Der Aufruf des Konstruktors erzeugt eine neue rationale Zahl (bzw. einen neuen Bruch).
- ❑ Zähler und Nenner (Attribute `numerator`, `denominator`) müssen eine zulässige Kombination von Werten enthalten.
- ❑ Dem Nenner darf nicht der Wert 0 zugewiesen werden.  
Die Division durch 0 wäre unzulässig.
- ❑ Der Nenner des Bruchs soll nicht negativ sein.  
Wird ein negativer Wert für den Nenner als Argument übergeben,  
so müssen bei Zähler und Nenner die Vorzeichen gewechselt werden.
- ❑ Der Bruch muss gekürzt werden,  
d.h. Zähler und Nenner müssen durch ihren größten gemeinsamen Teiler geteilt werden.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung des Konstruktors

```
public Fraction ( int num, int denom )
{
    if ( denom != 0 )
    {
        if ( denom < 0 )
        {
            numerator = -num;
            denominator = -denom;
        }
        else
        {
            numerator = num;
            denominator = denom;
        }
        reduce();
    }
    else
    {
        // error: division by zero
        throw new IllegalArgumentException();
    }
}
```

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung des Konstruktors

```

public Fraction ( int num, int denom )
{
    if ( denom != 0 )
    {
        if ( denom < 0 )
        {
            numerator = -num;
            denominator = -denom;
        }
        else
        {
            numerator = num;
            denominator = denom;
        }
        reduce();
    }
    else
    {
        // error: division by zero
        throw new IllegalArgumentException();
    }
}

```

The diagram illustrates the control flow of the code. Two red arrows originate from the text "if-else-Anweisung" and point to the two nested if statements. The first arrow points to the opening brace of the outer if-block. The second arrow points to the opening brace of the inner if-block.

## Problemstellung «Rationale Zahlen»

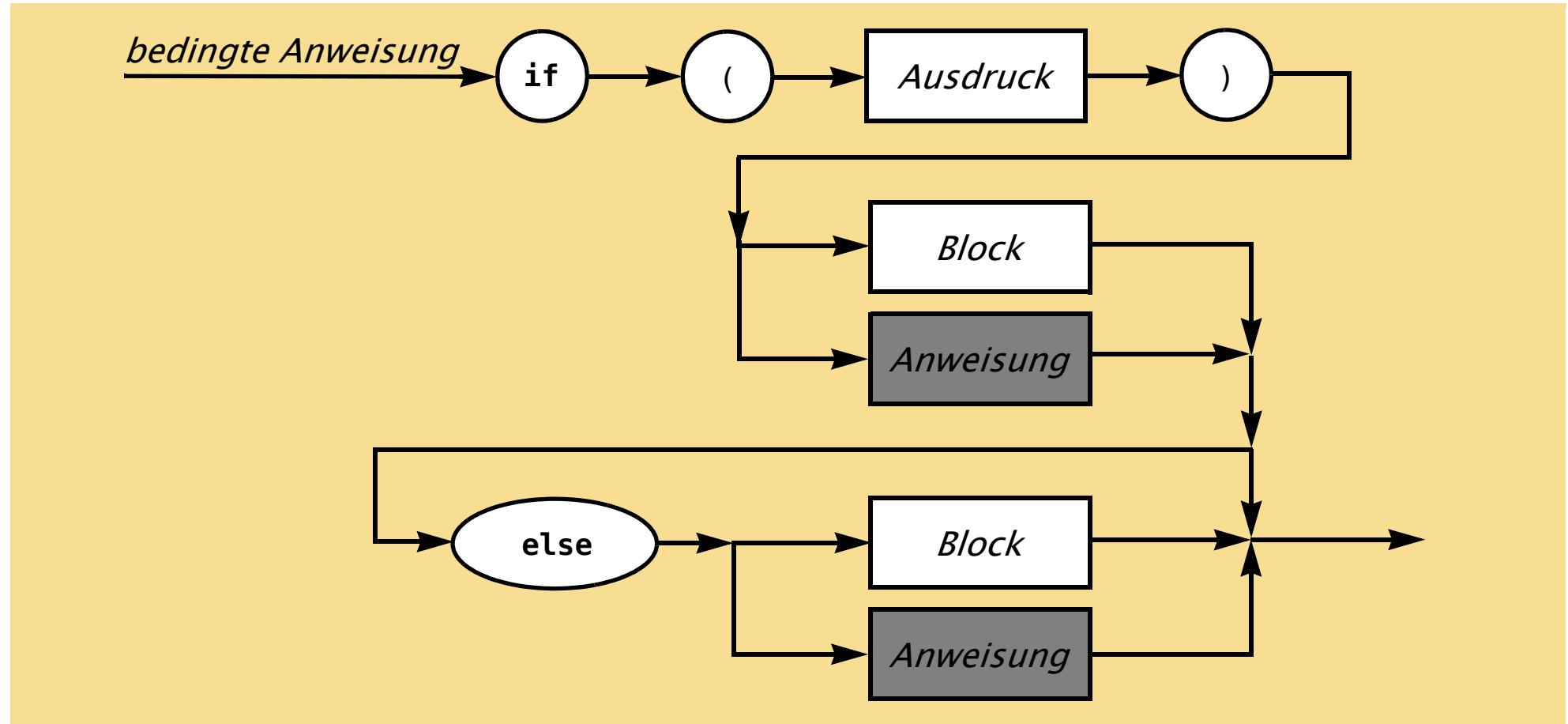
### Semantik der **if-else**-Anweisung

(Fortsetzung)

```
if ( denom < 0 )
{
    numerator = -num;
    denominator = -denom;
}
else
{
    numerator = num;
    denominator = denom;
}
```

- Ein **else**-Konstrukt ist nur mit vorangehendem **if**-Konstrukt möglich.
- Das **else**-Konstrukt bezieht sich immer auf das vorangehende **if**-Konstrukt auf der gleichen Ebene der Blockschachtelung.
- if ( Bedingung ) { Block1 } else { Block2 }**  
entspricht immer:  
**boolean b = Bedingung;**  
**if ( b ) { Block1 }**  
**if ( !b ) { Block2 }**
- Es wird also immer *entweder* nur das **if**-Konstrukt *oder* nur das **else**-Konstrukt ausgeführt.

## Syntaxdiagramm zu *bedingte Anweisung*



## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung des Konstruktors

```

public Fraction ( int num, int denom )
{
    if ( denom != 0 )           ← keine Division durch 0
    {
        if ( denom < 0 )         ← negatives Argument für Nenner
        {
            numerator = -num;
            denominator = -denom;   ← (einstelliger) - -Operator
        }
        else
        {
            numerator = num;
            denominator = denom;
        }
        reduce();                ← Wertzuweisungen an die Attribute
    }
    else
    {
        // error: division by zero
        throw new IllegalArgumentException();
    }
}

```

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung des Konstruktors

```
public Fraction ( int num, int denom )
{
    if ( denom != 0 )
    {
        if ( denom < 0 )
        {
            numerator = -num;
            denominator = -denom;
        }
        else
        {
            numerator = num;
            denominator = denom;
        }
        reduce();
    }
    else
    {
        // error: division by zero
        throw new IllegalArgumentException(); ← Fehlerfall behandeln *)
    }
}
```

\*) Erklärung folgt auf Folie 242

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

### Implementierung des Konstruktors

#### Anmerkungen:

- ❑ Ein Attribut übernimmt wie eine Variable das Speichern von Werten.
- ❑ Ein Attribut wird im Rumpf der Klasse deklariert und ist in *allen* ihren Methoden zugreifbar.
- ❑ Bei Ausführung eines Konstruktors erfolgen *nacheinander*:
  - Es wird im «Rechner» Speicherplatz für die Attribute besorgt und reserviert.
  - Die Initialisierung von Attributen wird ausgeführt.
  - Wird ein Attribut *nicht explizit* initialisiert, ist es implizit mit einem ausgezeichneten Wert seines Typs initialisiert. Die Attribute `numerator` und `denominator` werden nicht explizit initialisiert und erhalten daher implizit jeweils den Wert 0.
  - Der Rumpf des Konstruktors wird ausgeführt.
- ❑ Hinweis:  
Im Gegensatz zu den bisher vorgestellten Methoden *darf* ein Konstruktor *nicht* mit dem Modifizierer `static` versehen werden.
- ❑ Das Kürzen eines Bruchs stellt eine eigenständige Aufgabe dar, die in dieser Implementierung in eine private Methode `reduce` ausgelagert ist:
  - Der Programmcode des Konstruktors ist so übersichtlicher und leichter lesbar.
  - Die Methode `reduce` besitzt weder Parameter noch Rückgabe, da sie direkt die beiden Attribute verändert.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung der Methode `reduce`

```
private void reduce()
{
    if ( numerator != 0 )
    {
        int gcd = calculateGcd();
        numerator /= gcd;
        denominator /= gcd;
    }
    else
    {
        denominator = 1;
    }
}
```

Zugriff auf Attribut

normierte Speicherung der 0

- Die Methode `reduce` greift auf die beiden Attribute der Klasse zu und verändert ihre Werte: `reduce` benötigt daher weder Parameter noch eine Rückgabe. Da die Wirkung der Methode `reduce` nicht an der Deklaration ihres Kopfes sichtbar wird, spricht man davon, dass `reduce` *einen Seiteneffekt auf die Attribute hat*.
- Das Berechnen des für das Kürzen benötigten größten gemeinsamen Teilers ist ebenfalls in eine private Methode `calculateGcd()` ausgelagert, um die Lesbarkeit zu verbessern.

## Problemstellung «Rationale Zahlen»

Aufbau der Klasse

(Fortsetzung)

```
public class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction( int num, int denom )
    { ... }

    private void reduce()
    { ... }

    private int calculateGcd()
    { ... }

}
```

The diagram illustrates the internal structure of the Fraction class. It shows the class definition with its constructor, a reduce() method, and a calculateGcd() method. Two red arrows, each labeled 'ruft auf' (calls), point from one method call to another. One arrow points from the constructor call to the reduce() method, and another points from the reduce() method call to the calculateGcd() method.

## Problemstellung «Rationale Zahlen»

Aufrufreihenfolge

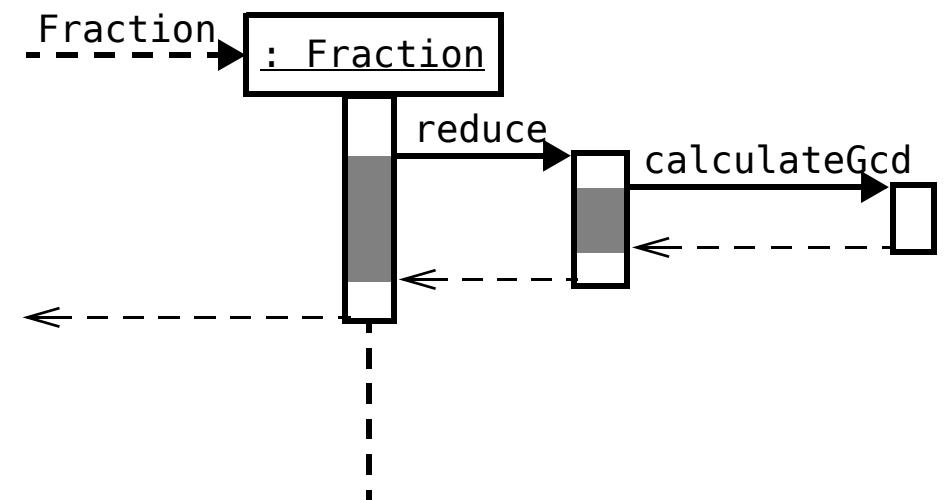
(Fortsetzung)

```
public class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction( int num, int denom )
    {
        ...
    }

    private void reduce()
    {
        ...
    }

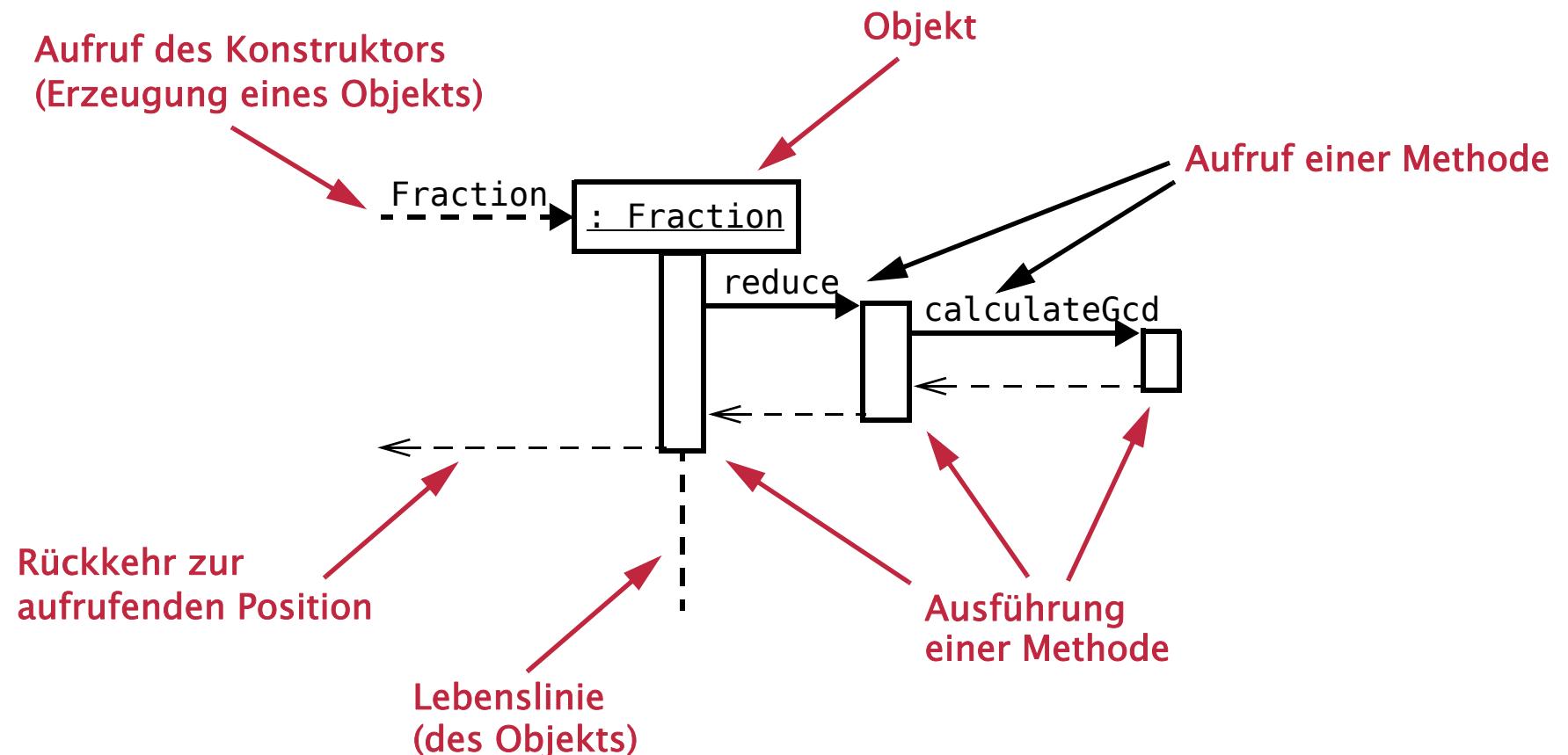
    private int calculateGcd()
    {
        ...
    }
}
```



## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Notation



(Notation: angelehnt an Sequenzdiagramme, siehe Vorlesung SWT)

## Problemstellung «Rationale Zahlen»

Implementierung der Methode `reduce`

(Fortsetzung)

```
private void reduce()
{
    ...
}
```

- Die Methode `reduce` ist ohne den Modifizierer **static** deklariert.  
Das macht sie zu einer *Instanzmethode*, die auf die (Instanz-)Attribute zugreifen kann.
- Die Instanzmethode `reduce` ist mit dem Zugriffsrecht **private** versehen und daher nur innerhalb der Klasse `Fraction` sichtbar.  
Da die durch `Fraction` realisierten Brüche immer gekürzt sein sollen,  
ist ein Aufruf außerhalb der Klasse `Fraction` auch nicht erforderlich.
- Als Instanzmethode bezieht sich die Ausführung von `reduce` immer auf genau ein Objekt.
- Terminologie:
  - Instanzmethoden werden in der Folge als *Methoden* bezeichnet.
  - Methoden, die mit dem Modifizierer **static** deklariert werden,  
werden im Folgenden explizit als *statische Methoden* bezeichnet.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung der Methode calculateGcd

```

public int calculateGcd()
{
    int value1 = Math.abs( numerator );
    int value2 = denominator;
    while (value1 != 0 & value2 != 0)
    {
        if ( value1 > value2 )
        {
            value1 = value1 % value2;
        } else {
            value2 = value2 % value1;
        }
    }
    if ( value1 == 0 )
    {
        return value2;
    } else {
        return value1;
    }
}

```

Betragbestimmung mit Methode aus Math

berechnet den größten gemeinsamen Teiler (greatest common divisor, gcd) nach dem Algorithmus von Euklid (geb. ca. 360 v. Chr.)

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Erzeugen eines Objekts

- ❑ Der Konstruktor dient dazu, ein Objekt der Klasse `Fraction` zu erzeugen.
- ❑ Der Aufruf des Konstruktors erfolgt immer hinter dem `new`-Operator.
- ❑ Beispiel:

```
public class FractionCalculator
{
    public static void main( String[] args )
    {
        Fraction f1 = new Fraction( 4, 5 );
        Fraction f2 = new Fraction( 6, -9 );
    }
}
```

*Aufruf des Konstruktors*

*new-Operator*

- ❑ Jeder Aufruf des Konstruktors erzeugt ein neues *Objekt* der Klasse `Fraction`.
- ❑ Die erzeugten Objekte werden hier den Variablen `f1` und `f2` zugewiesen.
- ❑ `f1` und `f2` bezeichnen nun zwei unterschiedliche rationale Zahlen.

## Problemstellung «Rationale Zahlen»

### Klassenstruktur

(Fortsetzung)

```
public class FractionCalculator
{
    public static void main( String[] args )
    {
        Fraction f1 = new Fraction ( 4, 5 );
        Fraction f2 = new Fraction ( 6, -9 );
    }
}
```

```
public class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction( int num, int denom )
    {
        ...
    }

    private void reduce()
    {
        ...
    }

    private int calculateGcd()
    {
        ...
    }
}
```

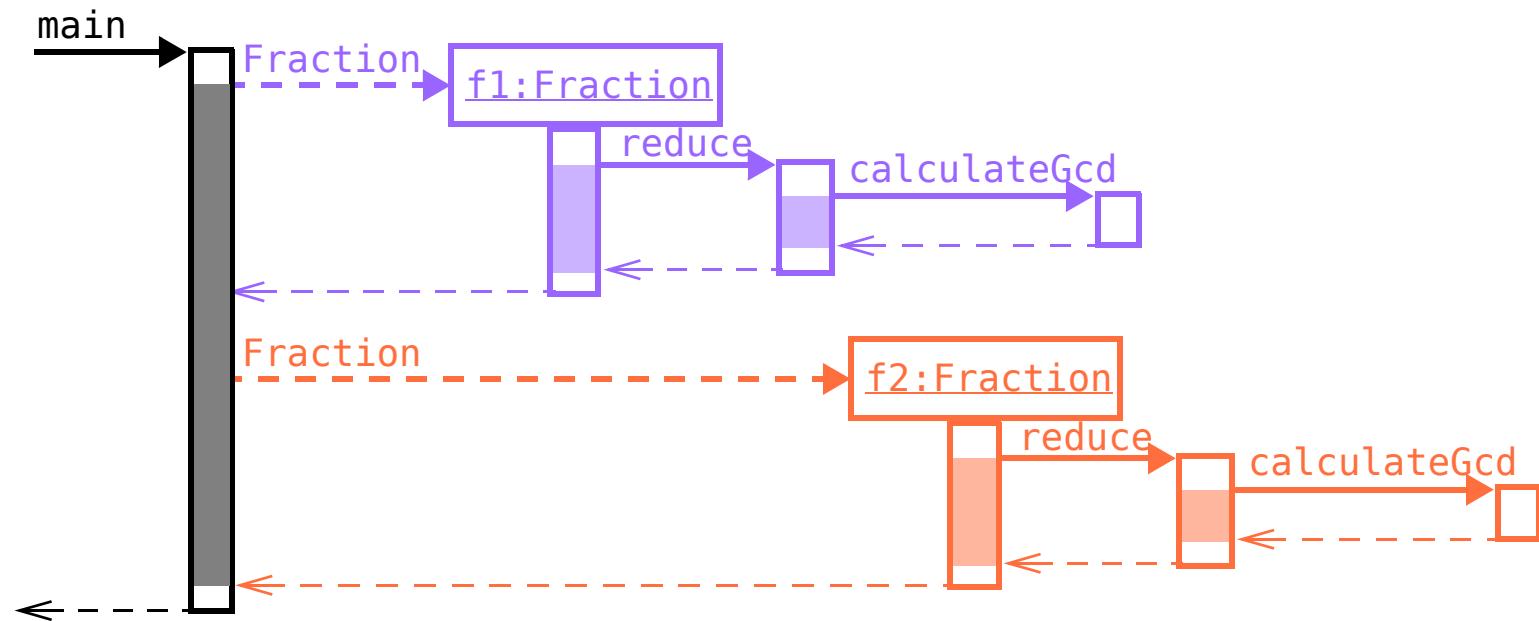
The diagram illustrates the class hierarchy and method calls between `FractionCalculator` and `Fraction`. It shows three method calls from `FractionCalculator` to `Fraction`: one to the constructor, one to `reduce()`, and one to `calculateGcd()`. Red arrows labeled "ruft auf" (calls) indicate these relationships.

## Problemstellung «Rationale Zahlen»

### Klassenstruktur

(Fortsetzung)

```
public class FractionCalculator
{
    public static void main( String[] args )
    {
        Fraction f1 = new Fraction ( 4, 5 );
        Fraction f2 = new Fraction ( 6, -9 );
    }
}
```



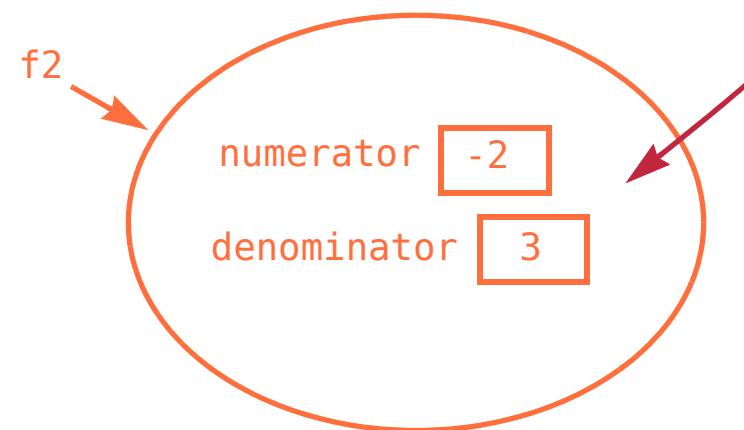
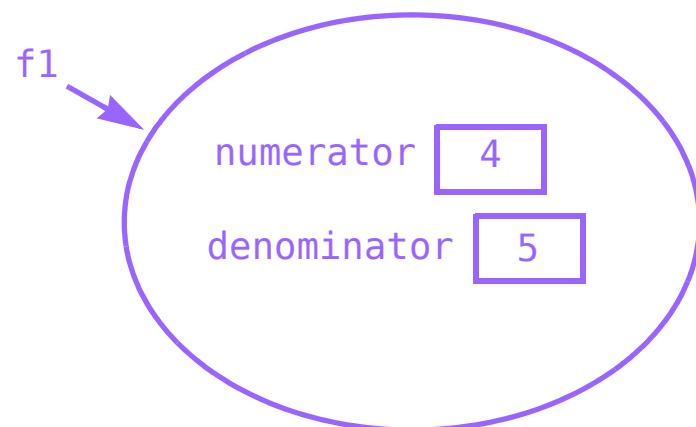
## Problemstellung «Rationale Zahlen»

### Visualisierung

(Fortsetzung)

```
public class FractionCalculator
{
    public static void main( String[] args )
    {
        Fraction f1 = new Fraction ( 4, 5 );
        Fraction f2 = new Fraction ( 6, -9 );
    }
}
```

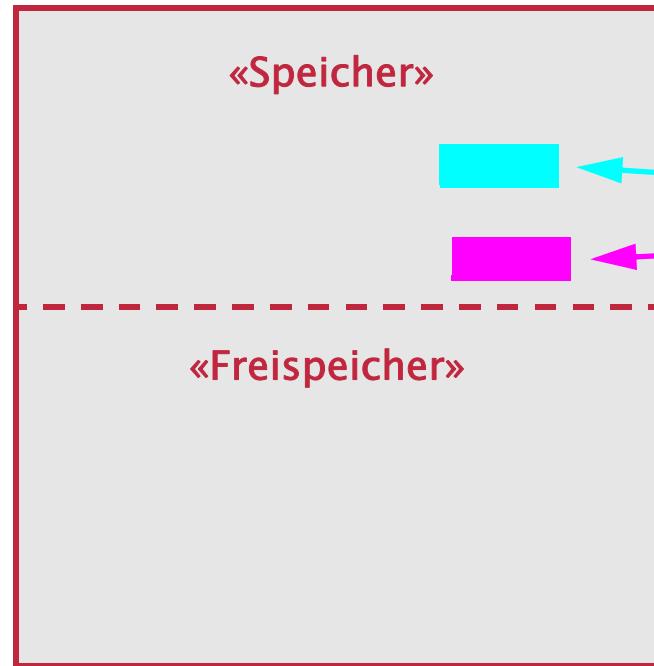
Konstruktor kürzt  
und normiert



## Exkurs: Modell der Speicherung von Objekten

(vergleiche auch Folie 166)

Hardware:



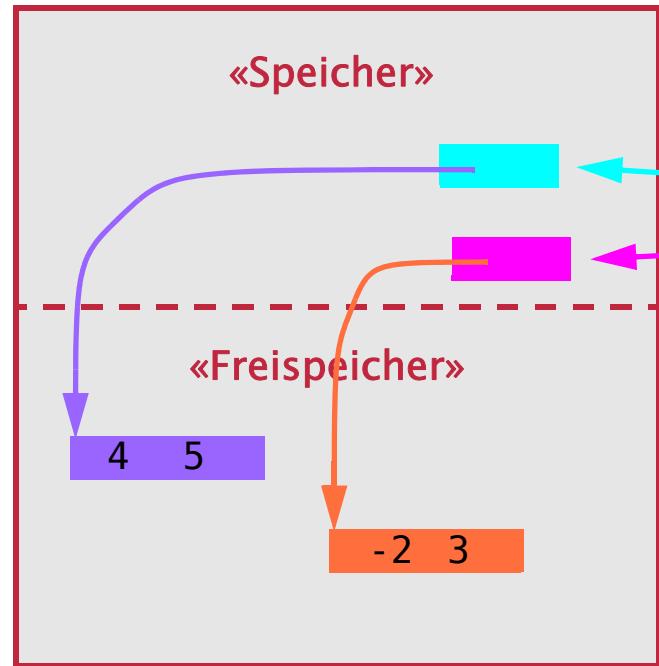
Software:

```
Fraction f1 = new Fraction ( 4, 5 )
Fraction f2 = new Fraction ( 6, -9 )
```

## Exkurs: Modell der Speicherung von Objekten

(Fortsetzung)

Hardware:



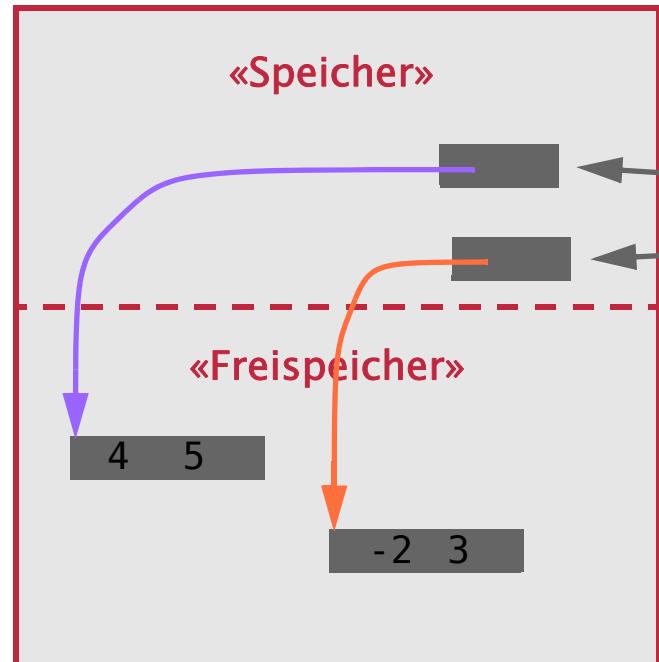
Software:

```
Fraction f1 = new Fraction ( 4, 5 )
Fraction f2 = new Fraction ( 6, -9 )
```

## Exkurs: Modell der Speicherung von Objekten

(Fortsetzung)

Hardware:



Software:

```
Fraction f1 = new Fraction ( 4, 5 )  
Fraction f2 = new Fraction ( 6, -9 )
```

Die Werte von `f1` und `f2` werden als *Referenzen* auf die entsprechenden Objekte bezeichnet.

## Referenzvariablen und Referenzattribute

- ❑ Die Werte von Variablen und Attributen, über die Objekte erreicht werden können, sind die *Referenzen* zu solchen Objekten.
- ❑ Diese Variablen bzw. Attribute werden als *Referenzvariablen* bzw. *-attribute* bezeichnet.
- ❑ Referenzvariable bzw. Referenzattribute kennen also den Speicherort eines Objekts, speichern aber das Objekt nicht selbst ab.
- ❑ In Java sind alle Variablen oder Attribute, deren Typ eine Klasse ist, Referenzvariablen bzw. Referenzattribute.
- ❑ Abgrenzung:

`int value;`      `value` bezeichnet einen Speicherbereich, in dem ein `int`-Wert abgelegt werden kann.

`Fraction f;`      `f` bezeichnet einen Speicherbereich, in dem die Referenz auf ein `Fraction`-Objekt abgelegt werden kann.

- ❑ Der *Dereferenzierungsoperator* `.` liefert für eine Referenzvariable das Objekt, dessen Speicherort sie kennt. Er wertet dazu die in der Referenzvariablen gespeicherte Referenz aus.
- ❑ In Java kann auch jede Feldvariable (vergleiche Folie 170) als eine Referenzvariable auf ein Feld(-Objekt) angesehen werden.

## Ausnahmesituation für «Rationale Zahlen»

Implementierung des Konstruktors (Rückblick)

```
public Fraction ( int num, int denom )
{
    if ( denom != 0 )
    {
        if ( denom < 0 )
        {
            numerator = -num;
            denominator = -denom;
        }
        else
        {
            numerator = num;
            denominator = denom;
        }
        reduce();
    }
    else
    {
        // error: division by zero
        throw new IllegalArgumentException(); ← Fehlerfall als Ausnahme
    }
}
```

behandeln

## Ausnahmesituation für «Rationale Zahlen»

(Fortsetzung)

```
throw new IllegalArgumentException();
```

- ❑ Wird als Argument für den Nenner der Wert 0 übergeben, so kann kein Bruch mit einer mathematisch erlaubten Belegung der Attribute erzeugt werden.
- ❑ In diesem Fall wird als Lösung die **throw**-Anweisung aufgerufen, die ein neu erzeugtes Objekt der Klasse `IllegalArgumentException` wirft. \*)
- ❑ `new IllegalArgumentException()` ist das Erzeugen eines solchen Objekts mit Hilfe des **new**-Operators und des (parameterlosen) Konstruktors der Klasse `IllegalArgumentException`.
- ❑ Java stellt zahlreiche spezielle Klassen bereit, mit denen auf diese Weise Ausnahmesituationen angezeigt werden können, u.a.:  
`ArithmetricException, IllegalArgumentException, IndexOutOfBoundsException,`  
`ArrayIndexOutOfBoundsException, NoSuchElementException, NullPointerException`
- ❑ Das Werfen eines Objekts einer dieser Klassen führt zum Abbruch der Programmausführung mit einer entsprechenden Meldung, die den Namen der Klasse des Objekts enthält.

\*) Die technischen Details der **throw**-Anweisung können hier noch nicht erklärt werden.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## **3.2. Klasse und Objekt - Klasse für Rationale Zahlen - Teil 2**

Dr. Stefan Dissmann  
Fakultät für Informatik



## Problemstellung «Rationale Zahlen»

(Fortsetzung)

### Weitere Planung der Klasse Fraction

Benötigt werden:

□ Weitere Konstruktoren:

    Fraction() ohne Parameter soll die rationale Zahl mit dem Wert 0 erzeugen.

    Fraction( **int** num ) soll die rationale Zahl num als Bruch num/1 erzeugen.

□ Eine Methode zur Darstellung einer rationalen Zahl als Bruch:

    String **toString()** gibt einen Text mit einem Bruch der Form 7/41 zurück.

□ Methoden für die Rechenoperationen:

    Fraction **changeSign()**

    Fraction **reverse()**

    Fraction **add( Fraction other )**

    Fraction **subtract( Fraction other )**

    Fraction **multiply( Fraction other )**

    Fraction **divideBy( Fraction other )**

**double** **toDouble()**

    Fraction **clone()**

**boolean** **equals( Fraction other )**

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

```
public class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction( int num, int denom ) { ... }
    public Fraction() { ... }
    public Fraction( int num ) { ... }
    public String toString() { ... }
    public Fraction changeSign() { ... }
    public Fraction reverse() { ... }
    public Fraction add( Fraction other ) { ... }
    public Fraction subtract( Fraction other ) { ... }
    public Fraction multiply( Fraction other ) { ... }
    public Fraction divideBy( Fraction other ) { ... }
    public double toDouble() { ... }
    public Fraction clone() { ... }
    public boolean equals( Fraction other ) { ... }

    private void reduce() { ... }
    private int calculateGcd() { ... }
}
```



## Problemstellung «Rationale Zahlen»

### Implementierung der weiteren Konstruktoren

(Fortsetzung)

```
public Fraction() {  
    numerator = 0;  
    denominator = 1;  
}  
  
public Fraction( int num ) {  
    numerator = num;  
    denominator = 1;  
}
```



Anlegen des Wertes 0.

Anlegen einer ganzen Zahl num

- Um den Komfort bei der Benutzung einer Klasse in anderen Klassen zu verbessern, ist es manchmal angebracht, Variationen mit der gleichen Operationalität bereitzustellen.
- Konstruktoren oder Methoden mit gleichen Namen müssen sich in ihren Signaturen unterscheiden, damit während der Ausführung eindeutig eine der Implementierungen ausgewählt werden kann.

## Problemstellung «Rationale Zahlen»

*alternative Implementierung* der weiteren Konstruktoren

(Fortsetzung)

```
public Fraction()  
{  
    this( 0, 1 );    ← this( ... ) ist der Aufruf eines Konstruktors  
}  
  
public Fraction( int num )  
{  
    this( num, 1 ); ← this( ... ) ist der Aufruf eines Konstruktors  
}
```

- ❑ Ein Konstruktor der eigenen Klasse kann unabhängig von dem Namen seiner Klasse immer nur durch **this( ... )** aufgerufen werden, wobei die Folge der Argumente der Parameterliste der Deklaration entsprechen muss.
- ❑ **this( ... )** darf genau einmal als erste Anweisung im Rumpf eines Konstruktors auftreten.
- ❑ Anmerkung:  
Der Aufruf des (komplexeren) Konstruktors mit zwei Parametern bringt im Beispiel oben keine Vorteile, sondern führt zur Ausführung unnötiger Abfragen in diesem Konstruktor.

## Problemstellung «Rationale Zahlen»

Implementierung der Methode `toString`

(Fortsetzung)

```
public String toString()
{
    return numerator + " / " + denominator;
}
```

- Ganze einfache Methode, die den Wert des Zählers, das `/`-Zeichen und den Wert des Nenners zu einem Text zusammensetzt.
- Die Klasse `String` ist in Java bereits definiert und realisiert Zeichenketten (Texte).
- Die `toString`-Methode kann benutzt werden, um den Wert eines Objekts der Klasse `Fraction` als Bruch auszugeben.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Aufruf der Methode `toString`

```
public String toString()
{
    return numerator + " / " + denominator;
}
```

Beispiel für einen Aufruf:

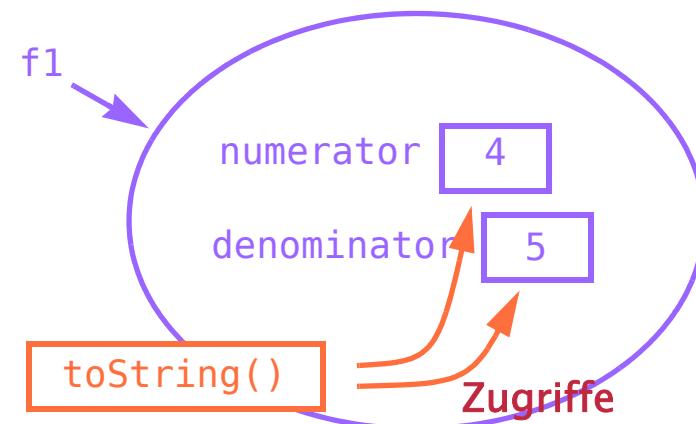
```
Fraction f1 = new Fraction ( 4, 5 );
System.out.print( f1.toString() );
```

erzeugt:

4 / 5

Aufruf der Methode `toString`  
für das Objekt `f1`

"4 / 5" ←  
Rückgabe



## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Aufruf der Methode `toString`

Beispiel für einen Aufruf:

```
Fraction f1 = new Fraction ( 4, 5 );  
System.out.print( f1.toString() );
```

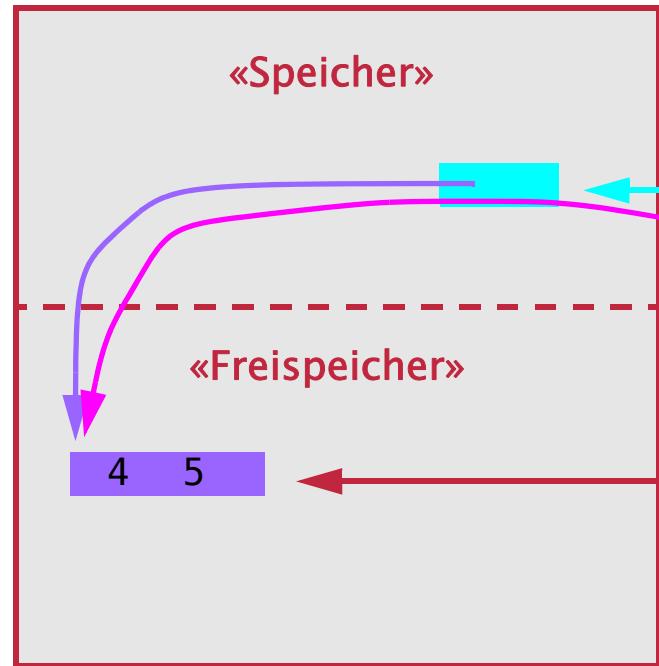


- Notation: `f1.toString()`  
Die Methode `toString` wird für das über `f1` erreichbare Objekt aufgerufen und ausgeführt.  
Dieses Objekt wird daher als das `toString` *ausführende Objekt* bezeichnet.
- Der *Dereferenzierungsoperator* `.` ermöglicht den Zugriff zu dem über `f1` erreichbaren Objekt und damit die Ausführung einer Methode auf diesem Objekt.  
Voraussetzung hierfür ist, dass das Zugriffsrecht der Methode diesen Aufruf gestattet:  
Da `toString` eine öffentliche Methode ist, ist dieses immer der Fall.

## Exkurs: Modell der Speicherung von Objekten

(siehe Folie 238)

Hardware:



Software:

```
Fraction f1 = new Fraction ( 4, 5 )  
System.out.print( f1.toString() )
```

ausführendes Objekt

## Problemstellung «Rationale Zahlen»

Implementierung der Methode `changeSign`

(Fortsetzung)

```
public Fraction changeSign()
{
    return new Fraction( -numerator, denominator );
}
```

- Die Methode `changeSign` erzeugt ein Objekt, das gegenüber dem ausführenden Objekt ein geändertes Vorzeichen besitzt.
- Die Methode `changeSign` benötigt daher keinen Parameter.

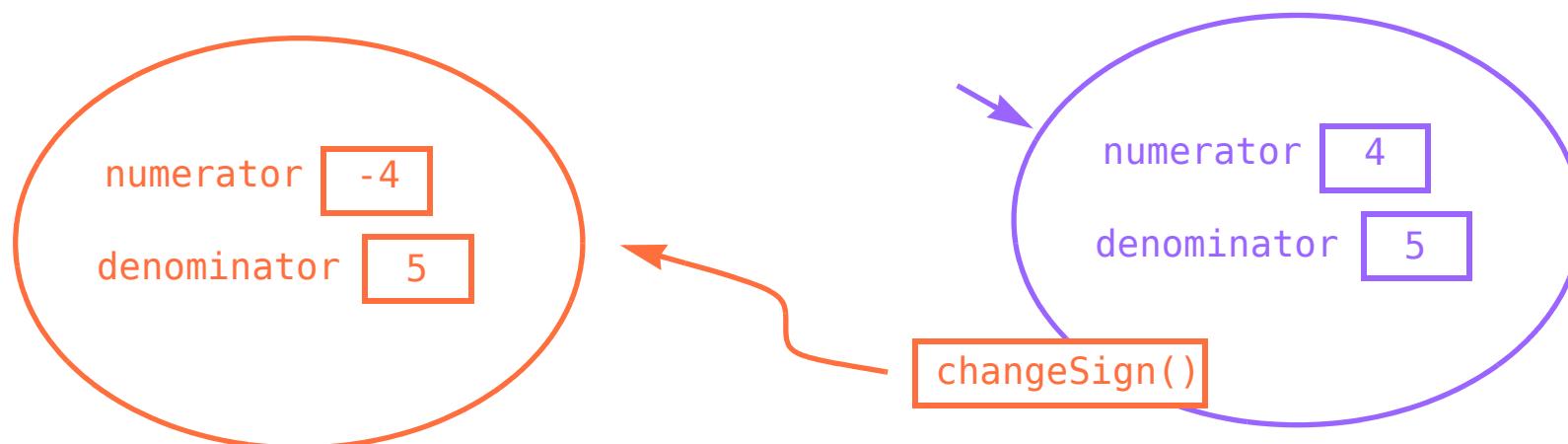
## Problemstellung «Rationale Zahlen»

Implementierung der Methode `changeSign`

(Fortsetzung)

```
public Fraction changeSign()
{
    return new Fraction( -numerator, denominator );
}
```

- ❑ Die Methode `changeSign` erzeugt ein Objekt, das gegenüber dem ausführenden Objekt ein geändertes Vorzeichen besitzt.
- ❑ Die Methode `changeSign` benötigt daher keinen Parameter.



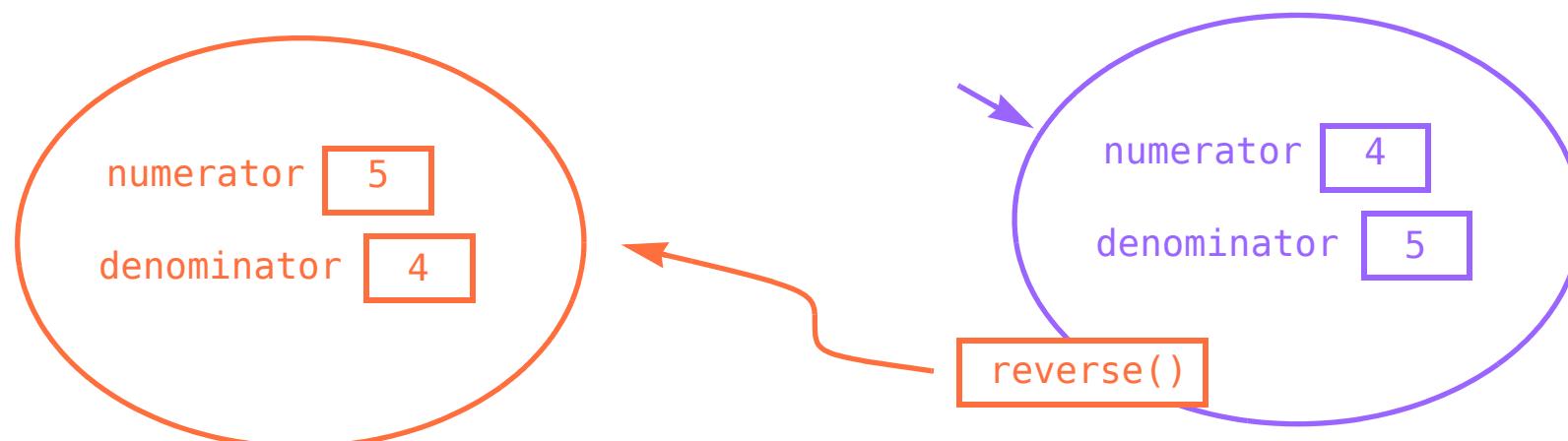
## Problemstellung «Rationale Zahlen»

Implementierung der Methode `changeSign`

(Fortsetzung)

```
public Fraction reverse()
{
    return new Fraction( denominator, numerator );
}
```

- Die Methode `reverse` erzeugt ein Objekt, das den Kehrwert des ausführenden Bruchs enthält, bei dem also Zähler (`numerator`) und Nenner (`denominator`) getauscht worden sind.



## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung der Methode add

```
public Fraction add( Fraction other )
{
    int num = numerator * other.denominator + other.numerator * denominator;
    int denom = denominator * other.denominator;
    return new Fraction ( num, denom );
}
```

Dereferenzierung

- Die Methode add bestimmt nach den Regeln der Bruchrechnung die Werte von Zähler (num) und Nenner (denom) der Addition.
- Die Methode add erzeugt durch den Aufruf des Konstruktors ein neues Objekt der Klasse Fraction und über gibt dabei die *berechneten* Werte für Zähler und Nenner an diesen Konstruktorauf ruf als Argumente.
- Der Konstruktor sorgt für ein eventuell notwendiges Kürzen des neuen Bruchs.
- Der *Dereferenzierungsoperator* . ermöglicht den Zugriff auf das über other erreichbare Objekt der Klasse Fraction.
- other.numerator liefert also den Wert des Attributs numerator des über other erreichbaren Objekts.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Aufruf von Methoden für Objekte

```
public class FractionCalculator
{
    public static void main( String[] args )
    {
        Fraction f1 = new Fraction ( 4, 5 );
        Fraction f2 = new Fraction ( 6, -9 );
        Fraction f3 = f1.add( f2 );
        System.out.println( "add: " + f1.toString() + " + " + f2.toString() +
                            " = " + f3.toString() );
    }
}
```

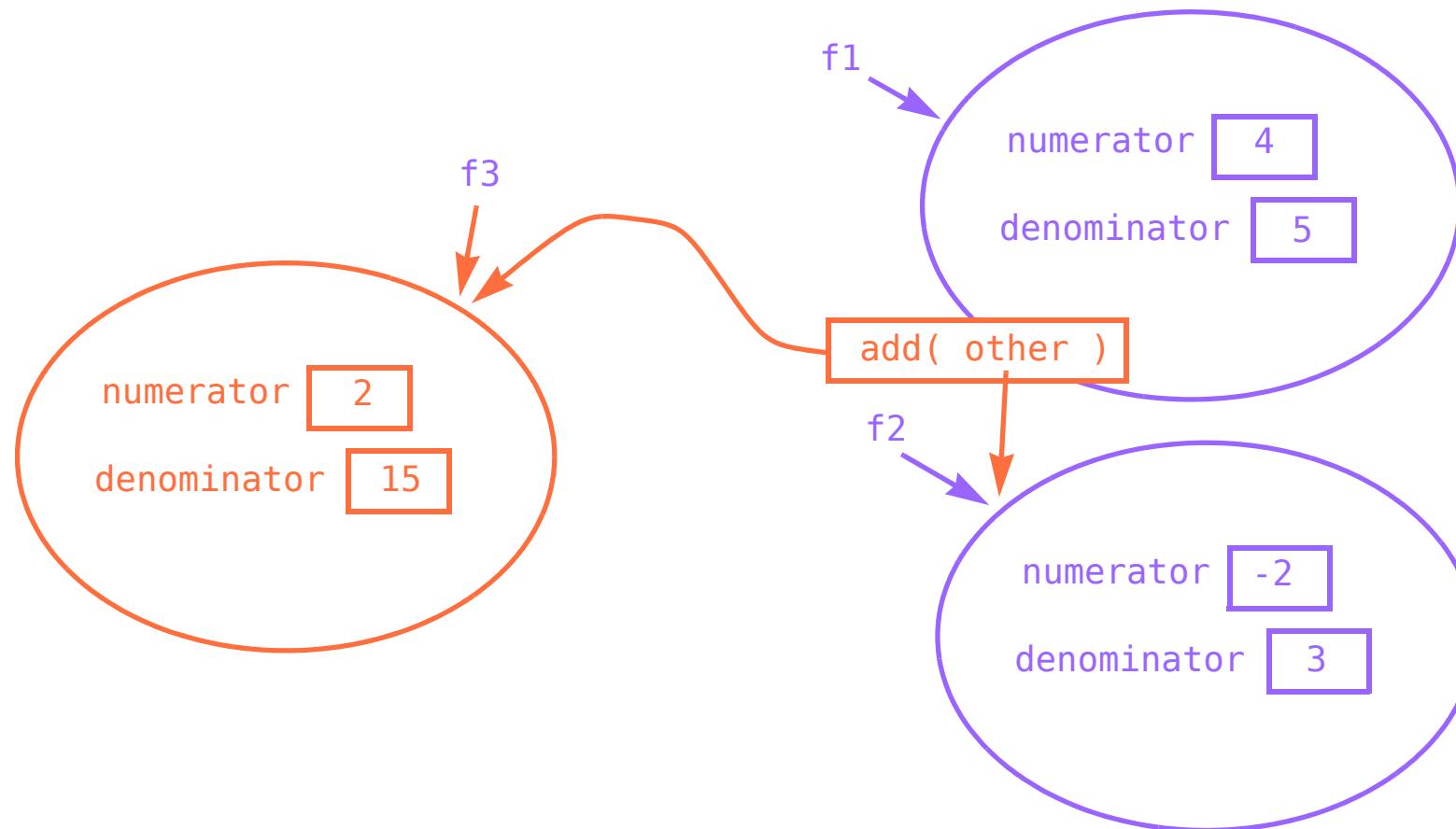
- Notation: `f1.add( f2 )`  
Die Methode `add` wird für das über `f1` erreichbare Objekt mit dem Parameter `f2` aufgerufen, `f1` verweist auf das *ausführende Objekt*.
- In der Klasse `FractionCalculator` können nur die öffentlich sichtbaren (`public`) Methoden der Klasse `Fraction` aufgerufen werden.

## Problemstellung «Rationale Zahlen»

### Visualisierung

(Fortsetzung)

Fraction f3 = f1.add( f2 );

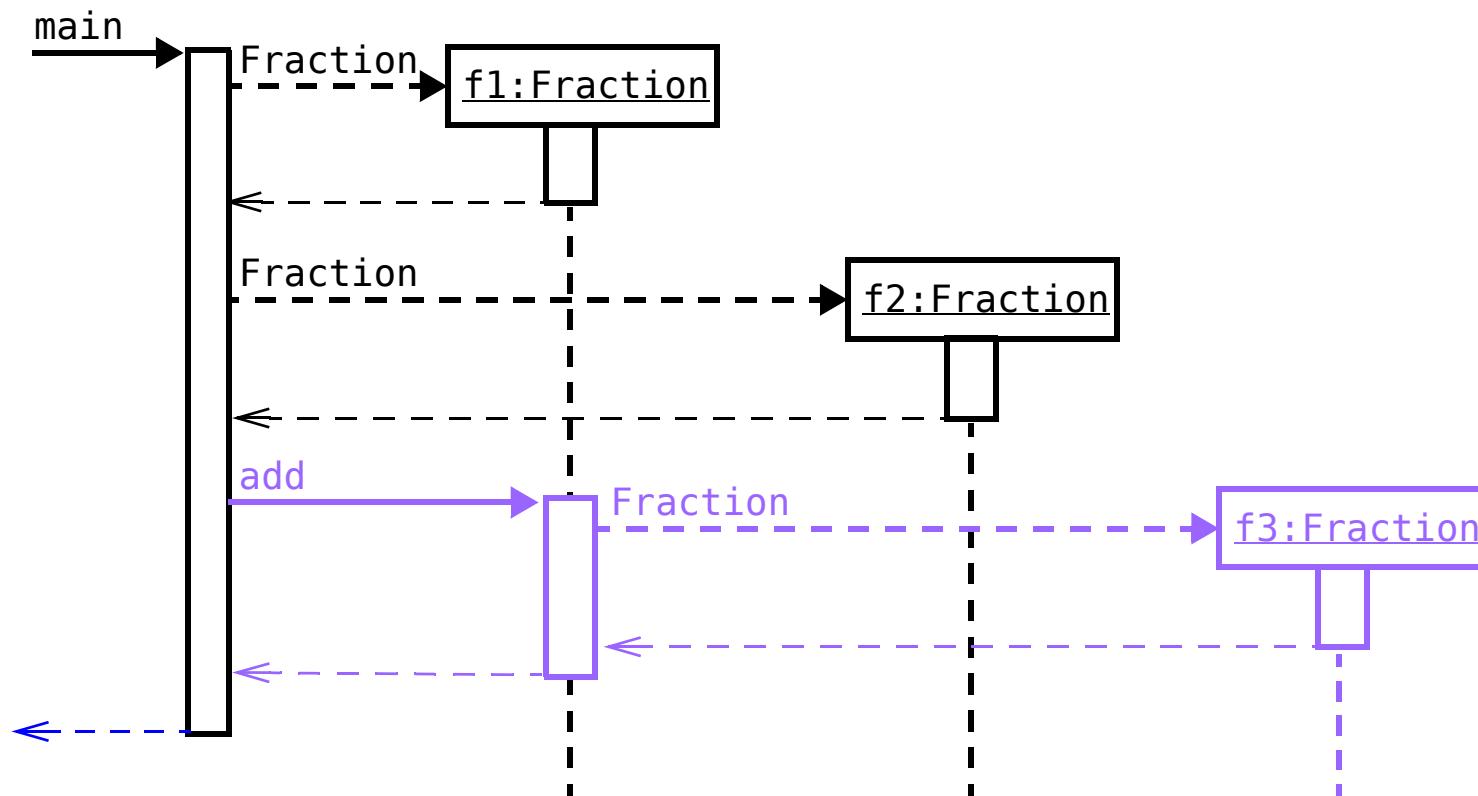


## Problemstellung «Rationale Zahlen»

### Visualisierung

(Fortsetzung)

Fraction f3 = f1.add( f2 );



## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung der Methode `subtract`

```
public Fraction subtract( Fraction other )
{
    int num = numerator * other.denominator - other.numerator * denominator;
    int denom = denominator * other.denominator;
    return new Fraction ( num, denom );
}
```

- Der Aufbau des Algorithmus und der Implementierung sind analog zu dem der Methode `add`.
- Eine *alternative* Implementierung greift auf die schon vorhandenen Methoden zurück:

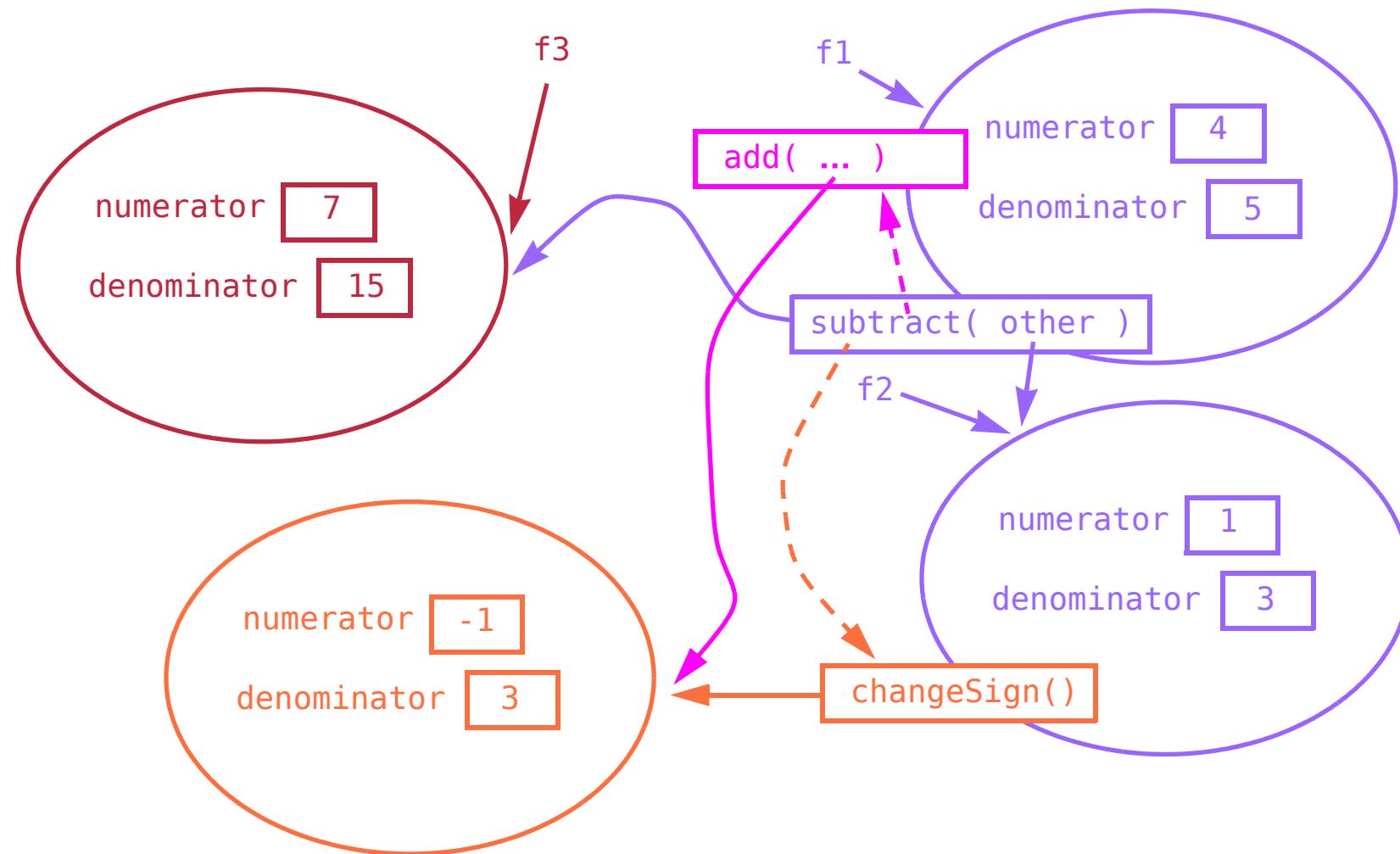
```
public Fraction subtract( Fraction other )
{
    return add ( other.changeSign() );
}
```

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Visualisierung: alternative Implementierung

Fraction f3 = f1.subtract( f2 );



## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Implementierung der Methoden `multiply` und `divideBy`

```
public Fraction multiply( Fraction other )
{
    int num = numerator * other.numerator;
    int denom = denominator * other.denominator;
    return new Fraction ( num, denom );
}

public Fraction divideBy( Fraction other )
{
    return multiply ( other.reverse() );
}
```

- ❑ Die Division erfolgt durch Multiplikation mit dem Kehrbruch.
- ❑ Der Aufbau der Algorithmen und der Implementierungen sind analog zu dem der Methoden `add` bzw. `subtract`.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

Anwendung der Klasse `Fraction`

```
Fraction f1 = new Fraction ( 4, 5 );
Fraction f2 = new Fraction ( 6, -9 );
Fraction f3 = f1.add( f2 );
Fraction f4 = new Fraction(7);
Fraction f5 = f4.add( f2.multiply( f3.subtract( f1 ) ) );
System.out.println( f5.toString() );
```

Ausgabe:

67 / 9

Mit der Klasse `Fraction` lassen sich Berechnungen mit Brüchen im Rahmen der Genauigkeit des Typs `int` programmieren.

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

weitere Methode

```
public double toDouble()
{
    return (double)numerator / (double)denominator;
}
```

explizite Typumwandlung

- Die Methode `toDouble` liefert das Ergebnis der Division von Zähler und Nenner als reelle Zahl.
- Um die ganzzahlige Division mit einem ganzzahligen Ergebnis zu vermeiden, muss mindestens einer der Operatoren der Division explizit in einen `double`-Wert konvertiert werden (Type-Cast).
- Der *Type-Cast* (`double`) erzeugt den dem `int`-Wert des nachfolgenden Attributs entsprechenden `double`-Wert, der dann in die Division eingeht.
- Ein Type-Cast der Form (*primitiver-Typ*) kann auch benutzt werden, um Werte eines Typs mit einem größeren Wertebereich in Werte eines Typs mit einem kleineren Wertebereich umzuwandeln. Da es dabei zu Rundungsfehlern kommen kann, muss eine solche Umwandlung in Java explizit angestoßen werden.

```
double x = 13.567;
int n = (int)x * 4;
int m = (int)(x * 4);
```

Ergebnis ist: 52  
Ergebnis ist: 54

## Problemstellung «Rationale Zahlen»

(Fortsetzung)

weitere Methoden

```
public Fraction clone() {  
    return new Fraction( numerator, denominator );  
}
```

- Die Methode `clone` erzeugt eine Kopie des Objekts, für das sie aufgerufen wird.
- Die Methode `clone` benötigt daher keine Parameter, liefert aber ein mit den Attributwerten des ausführenden Objekts neu erzeugtes Objekt zurück.
- Die Werte der Attribute von beiden Objekten sind gleich, die Objekte müssen aber sehr wohl unterschieden werden.

## Problemstellung «Rationale Zahlen»

weitere Methoden

(Fortsetzung)

```
public boolean equals( Fraction other )
{
    return numerator == other.numerator & denominator == other.denominator;
}
```

- ❑ Die Methode `equals` vergleicht die Werte von Zähler und Nenner von zwei Brüchen, um so die Gleichheit festzustellen.
- ❑ Stimmen die Werte überein, wird `true` zurückgegeben, sonst wird `false` zurückgegeben.
- ❑ Da Brüche in der Klasse `Fraction` immer in gekürzer Form vorliegen, ist sichergestellt, dass nur bei gleichen Attributwerten auch Brüche mit identischen Zahlenwert vorliegen.

## Problemstellung «Rationale Zahlen»

### Zusammenfassung

(Fortsetzung)

- Eine Klasse dient als Vorlage für Objekte, die anhand der Deklaration der Klasse erzeugt werden.
- Eine Klasse beschreibt ein semantisch zusammen gehörendes Konzept, das an anderer Stelle bei der Entwicklung von Software wiederverwendet werden soll.
- Eine Klasse enthält verschiedene Elemente: Attribute, Konstruktoren und Methoden.
- Diese Elemente müssen mit Zugriffsrechten versehen werden:
  - *private* Elemente können nur im Rumpf der Klasse genutzt werden, in der sie deklariert sind,
  - *öffentliche* Elemente können in beliebigen anderen Klassen genutzt werden.
- Zugriffsrechte dienen dazu, den korrekten Umgang bei der Benutzung der Elemente durch Einschränkungen sicherzustellen.
- Attribute dienen der (dauerhaften) Aufbewahrung von Werten, die über die Ausführung einer Methode hinausgeht.
- Konstruktoren bestimmen den Ablauf beim Erzeugen von Objekten und sorgen insbesondere für die korrekte erste Belegung von Attributen mit Werten.
- Methoden greifen auf die Werte der Attribute zu und führen Algorithmen damit aus. Methoden liefern dabei Rückgabewerte oder manipulieren die Werte von Attributen.

## Problemstellung «Rationale Zahlen»

### Zusammenfassung (Fortsetzung)

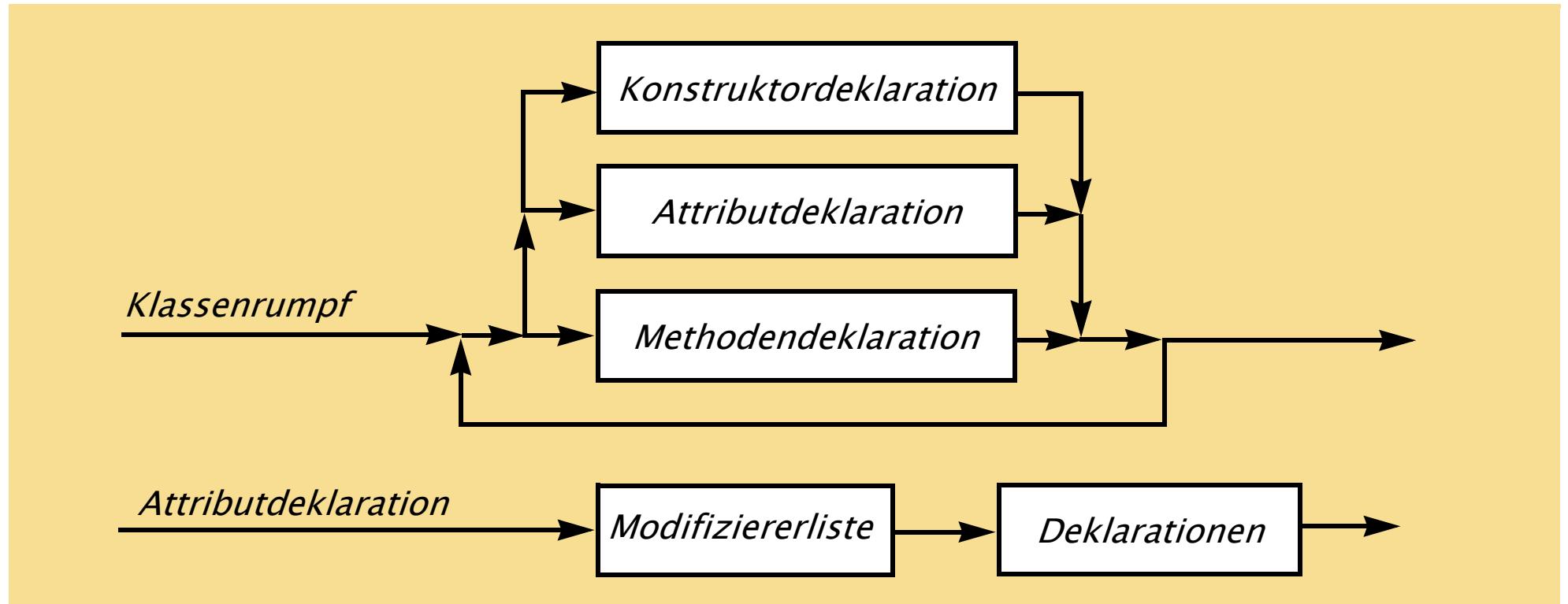
(Fortsetzung)

- ❑ Methoden sollen immer die innere Konsistenz der Objekte erhalten.
- ❑ Es kann mehrere Konstruktoren oder Methoden gleichen Namens geben, die sich dann durch ihre Signaturen unterscheiden müssen.
- ❑ Methoden können mit dem Dereferenzierungsoperator über den Namen eines Objekts aufgerufen werden.
- ❑ In Methoden einer Klasse kann mit dem Dereferenzierungsoperator über den Namen eines Objekts der *gleichen* Klasse auch auf dessen private Elemente zugegriffen werden. Das Zugriffsrecht **private** schränkt den Zugriff auf den Bereich *innerhalb der Deklaration* einer Klasse ein.

#### Anmerkungen:

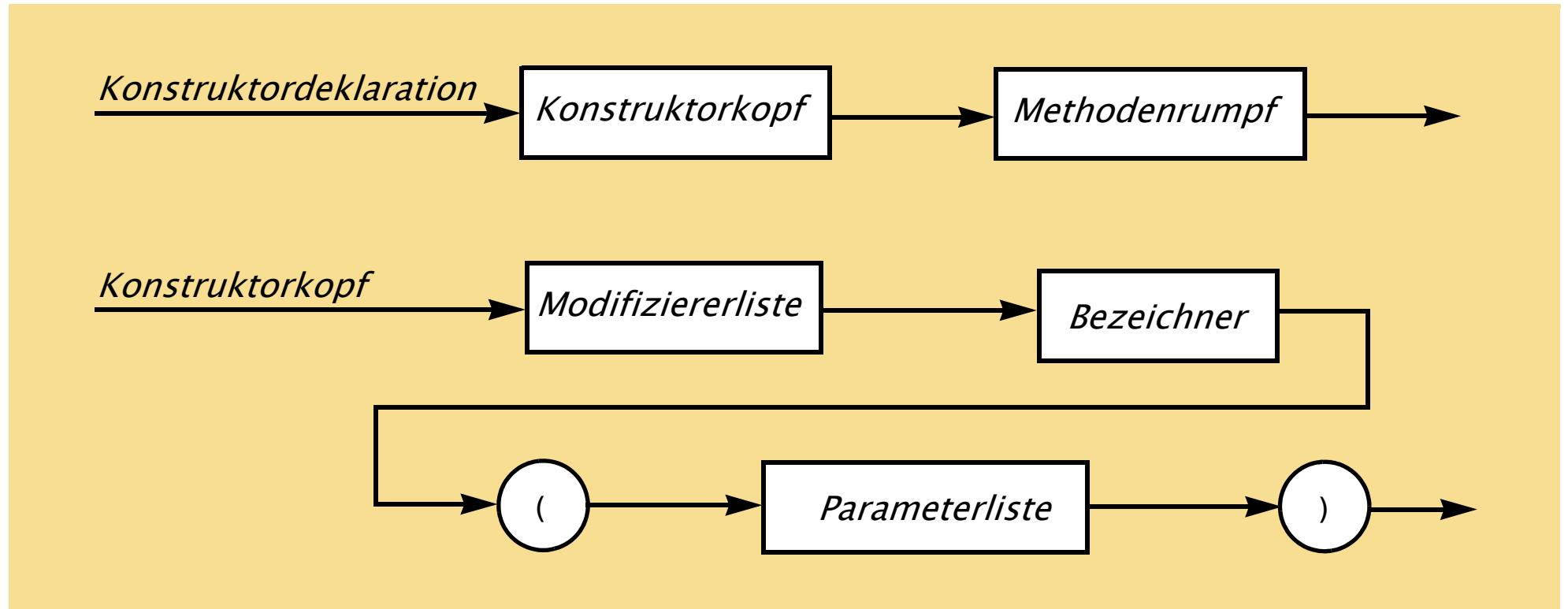
- ❑ Statische Methoden können nicht auf (nicht-statische) *Attribute* eines Objekts zugreifen.
- ❑ Statische Methoden können keine (nicht-statischen) *Methoden* eines Objekts aufrufen.

## Syntaxdiagramm zu *Klassenrumpf*



- ❑ Da Attribute grundsätzlich wie Variablen deklariert werden, können auch Attribute bei der Deklaration initialisiert werden.
- ❑ Eine solche Initialisierung erfolgt vor der Ausführung der Anweisungen des Konstruktors.
- ❑ Aus Gründen der Übersichtlichkeit sollte so eine Initialisierung aber nur dann erfolgen, wenn kein Konstruktor deklariert wird.

## Syntaxdiagramm zu *Konstruktor*



- ❑ Die Nebenbedingung, dass der Bezeichner eines Konstruktors dem Namen der Klasse entsprechen muss, in der die Deklaration vorgenommen wird, lässt sich im Syntaxdiagramm *nicht* ausdrücken.

## Standardkonstruktor

Anmerkungen zu Konstruktoren:

- ❑ In der Beispielklasse `Fraction` werden benutzt:
  - Konstruktoren mit Parametern,  
die den Attributen die als Argumente übergebenen Werte zuweisen
  - der parameterlose Konstruktor,  
der den Attributen feste Werte zuweist
- ❑ Der parameterlose Konstruktor heißt *Standardkonstruktor*.
- ❑ *Default-Konstruktor*:  
Falls in einer Klasse überhaupt kein Konstruktor deklariert wird,  
stellt eine solche Klasse *implizit* einen Standardkonstruktor als Default-Konstruktor bereit:  
Der Default-Konstruktor legt ein Objekt der Klasse an und  
initialisiert alle Attribute mit ihren typspezifischen *Nullwerten*.

```
public class SimpleClass
{
    int att;
}
```

Der Aufruf `new SimpleClass()` ist erlaubt und  
erzeugt ein Objekt, bei dem `att` mit dem Wert `0` initialisiert wird.

## Exkurs – `main`-Methode

Nun kann der Aufbau des Kopfes der `main`-Methode erklärt werden:

```
public static void main( String[] args )
```

- ❑ Der Parameter `args` ist ein Feld, dessen Elemente Texte beinhalten.
- ❑ Ein Java-Programm wird von einem Betriebssystem immer in folgender Form gestartet:  
`java className option1 option2 option3`
- ❑ `java` bezeichnet die Virtuelle Maschine (Folie 32).
- ❑ `className` ist der Name der `.class`-Datei, die die `main`-Methode enthält.
- ❑ `option1, option2, option3` sind beliebige Texte,  
die in dem Feld `args` anschließend in der `main`-Methode verfügbar sind.
- ❑ Die Methode `main` ist als `static` deklariert,  
damit sie ohne das Erzeugen eines Objekts aufgerufen werden kann.
- ❑ Beim Aufruf über eine Entwicklungsumgebung hat `args` in der Regel die Länge 0.
- ❑ Die meisten Entwicklungsumgebungen erwarten trotzdem eine Methode `main` mit der vorgestellten Signatur als Einstiegspunkt in die Ausführung.

*Hinweis:* `BlueJ` erlaubt jedoch das einfache Ausführen beliebiger statischer Methoden.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 3.3. Klasse und Objekt - Beispiel «Vorlesung»

Dr. Stefan Dissmann  
Fakultät für Informatik



## Problemstellung «Vorlesung»

Entwicklung zweier Klassen, um das Konzept von Klasse und Objekt weiter zu verdeutlichen.

Ziel:

Definiere eine Klasse, die die Verwaltung einer Vorlesung mit diesen Eigenschaften realisiert:

- Eine Vorlesung hat eine Bezeichnung und kennt die teilnehmenden Studierenden. \*)
- Eine Vorlesung hat eine Obergrenze für die Zahl der teilnehmenden Studierenden.
- Eine Studierende hat einen Namen, ein Studienfach und eine Matrikelnummer. \*)

\*) Auf weitere Attribute wird verzichtet.

## Problemstellung «Vorlesung»

(Fortsetzung)

```
public class Lecture
{
    private String title;           ← Attribut für den Veranstaltungsnamen
    private Student[] students;     ← Attribut für die teilnehmenden Studierenden

    public Lecture( String t, int cap ) ← Konstruktor
    {
        title = t;
        students = new Student[cap];
    }
}
```

- ❑ Das Attribut `students` ist ein Feld, dessen Grundtyp die Referenz auf die Klasse `Student` ist.
- ❑ Die Elemente des Feldes `students` sind also Referenzen auf Objekte der Klasse `Student`.
- ❑ Der Wert des Parameters `cap` (`capacity`) wird direkt zur Initialisierung der Größe des Feldes `students` verwendet.
- ❑ Der Wert von `cap` muss daher nicht in einem eigenen Attribut abgelegt werden, da er immer durch `students.length` ermittelt werden kann.

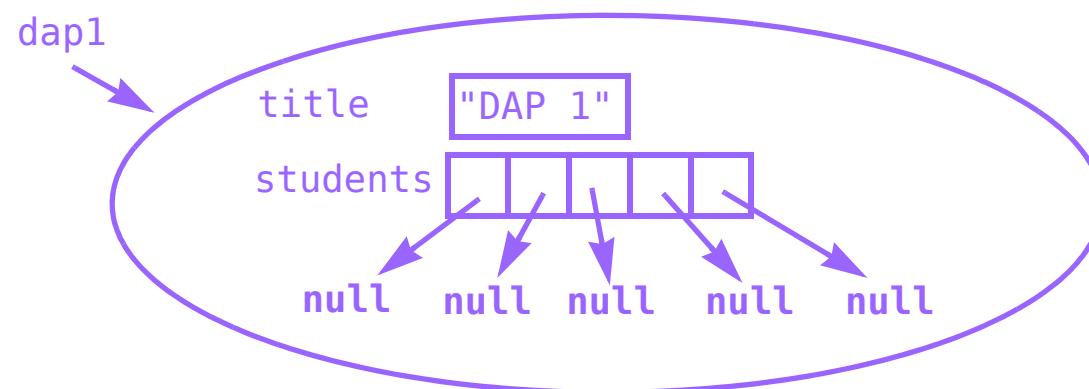
## Problemstellung «Vorlesung»

(Fortsetzung)

Anmerkungen:

- ❑ Beim Erzeugen des Feldes `students` werden die Elemente mit dem Wert `null` initialisiert, der eine Referenz auf kein Objekt anzeigt.
- ❑ `null` ist ein *Schlüsselwort*.
- ❑ Nach dem Erzeugen eines Objekts der Klasse `Lecture` liegt also folgende Situation vor:

```
Lecture dap1 = new Lecture( "DAP 1", 5 );
```



## Problemstellung «Vorlesung»

(Fortsetzung)

### «Anmelden» von Studierenden zur Vorlesung

Ziel:

- Ein Objekt der Klasse `Student` soll in das Element des Feldes `students` mit dem kleinsten Index eintragen werden, in dem der Wert `null` steht.
- Falls die Vorlesung «voll» ist, das Feld `students` also kein Element mit dem Wert `null` aufweist, soll nichts geschehen.
- Die Methode soll anzeigen, ob das Eintragen erfolgreich war (oder nicht).

Implementierung der folgenden Methode:

```
public boolean add( Student s )  
{  
    ...  
}
```

## Problemstellung «Vorlesung»

### «Anmelden» von Studierenden zur Vorlesung

(Fortsetzung)

```
public boolean add( Student s )
{
    for ( int i = 0; i < students.length; i++ )
    {
        if ( students[i] == null ) ← Überprüfen auf null
        {
            students[i] = s;
            return true;
        }
    }
    return false;
}
```

- Die Konzeption und auch die Implementierung der Klasse Lecture können ohne detaillierte Kenntnisse der Implementierung der Klasse Student erstellt werden, da in den bisher betrachteten Methoden der Klasse Lecture keine Methoden der Klasse Student benötigt werden.
- Für das Compilieren der Klasse Lecture wird aber eine Implementierung der Klasse Student benötigt.
- Das Vorliegen von `null` kann in Bedingungen überprüft werden.

## Problemstellung «Vorlesung» «Anmelden» von Studierenden zur Vorlesung

(Fortsetzung)

Analyse:

- ❑ Solange *keine* Methode zum Löschen in der Klasse `Lecture` vorgesehen ist, ist die vorhandene Implementierung der Methode `add` umständlich und ineffizient:  
Obwohl nach jedem Einfügen klar ist, an welcher Position im Feld `students` das nächste Einfügen stattfinden wird, erfolgt immer wieder eine Suche vom Beginn des Feldes aus.
- ❑ Das Problem ist, das «Wissen» von einem Aufruf der Methode `add` zum nächsten Aufruf dieser Methode zu übertragen.
- ❑ Diese Aufgabe kann ein Attribut übernehmen, da der dort gespeicherte Wert über den Aufruf einer Methode hinaus erhalten bleibt:  
Das Attribut `firstUnused` soll die Position des ersten unbelegten Elements speichern, also die Position, an der das nächste Student-Objekt abgelegt wird.
- ❑ Attribute können also nicht nur Informationen speichern, die das Objekt beschreiben, sondern auch Informationen, die die Arbeit der Methoden des Objekts unterstützen.

Anmerkung:

Nach außen wird das Austauschen der Implementierung der Methode nicht sichtbar:  
Vor und nach der Änderung stellt die Klasse `Lecture` eine Methode `add` bereit.

## Problemstellung «Vorlesung»

(Fortsetzung)

«Anmelden»« von Studierenden zur Vorlesung (geänderte Implementierung)

```
public class Lecture
{
    private String title;
    private Student[] students;
    private int firstUnused; ← weiteres Attribut

    public Lecture( String t, int cap )
    {
        title = t;
        students = new Student[cap];
        firstUnused = 0; ← Startwert setzen
    }

    public boolean add( Student s )
    {
        if ( firstUnused < students.length )
        {
            students[firstUnused] = s;
            firstUnused++;
            return true;
        }
        return false;
    }
}
```

... ← weiteres Attribut

Startwert setzen

add ohne Schleife

## Problemstellung «Vorlesung»

Klasse für Studierende

(Fortsetzung)

```
public class Student
{
    private String name;
    private String subject;
    private int registrationNo;

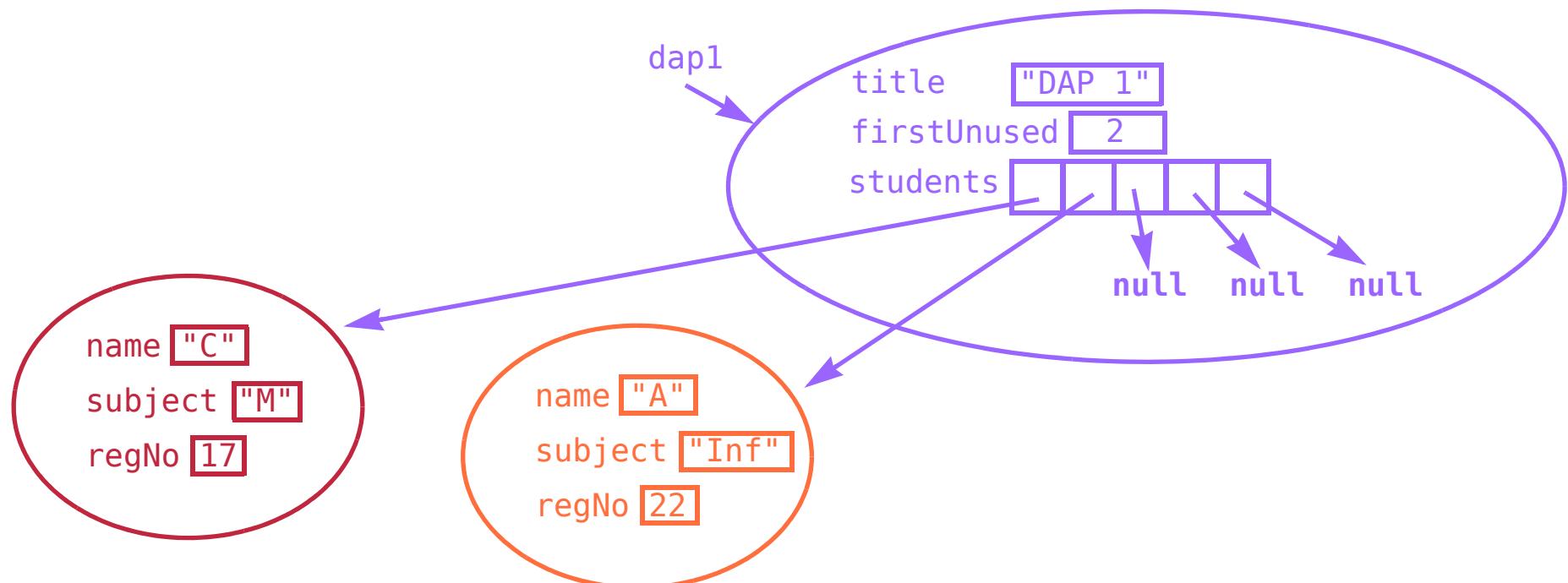
    public Student( String n, String sub, int no ) {
        name = n;
        subject = sub;
        registrationNo = no;
    }

    public String toString() {
        return "student: " + name +
               ", registration number: " + registrationNo +
               "(" + subject + ")";
    }
}
```

## Problemstellung «Vorlesung»

(Fortsetzung)

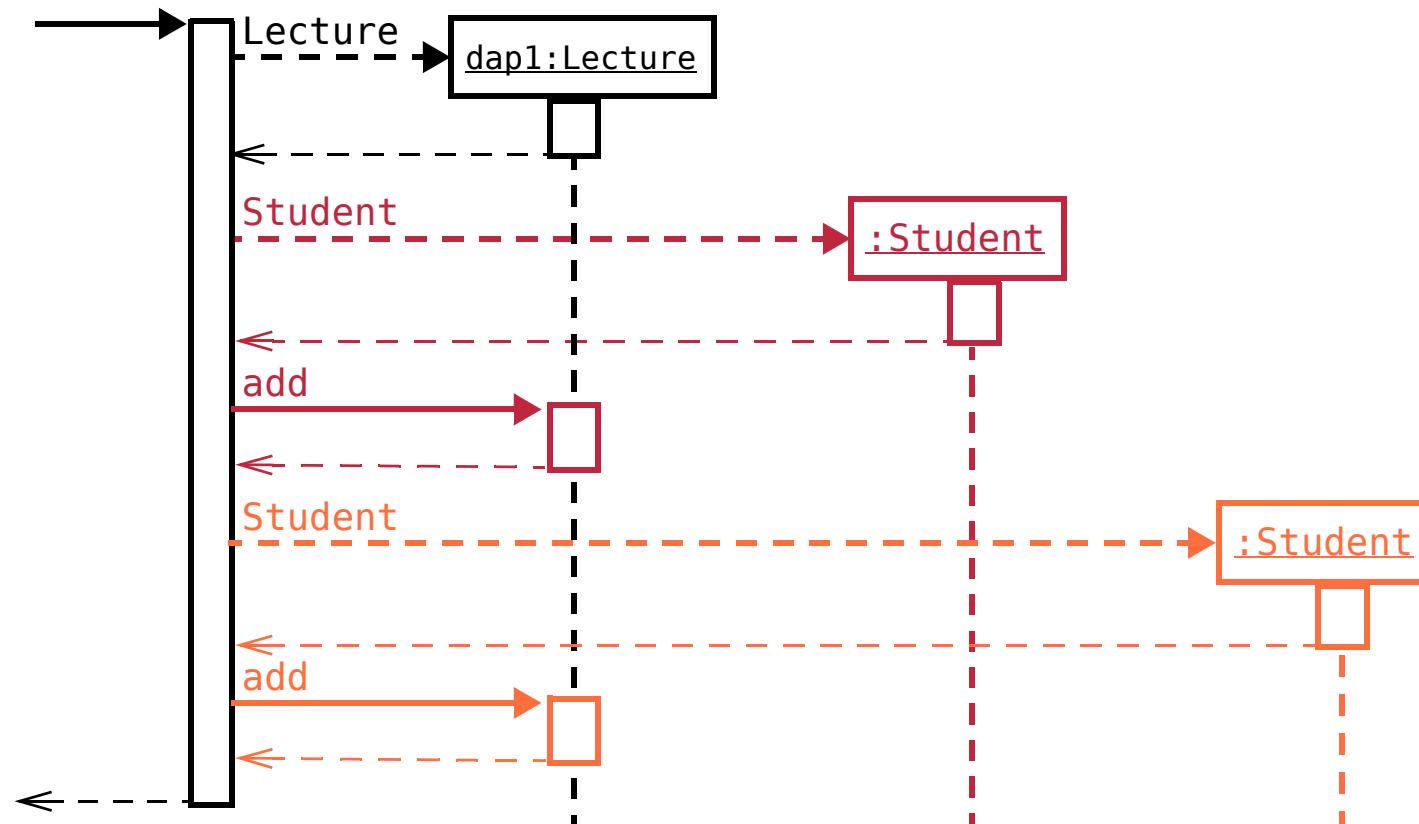
- Erzeugen eines Objekts der Klasse Lecture:  
`Lecture dap1 = new Lecture( "DAP 1", 5 );`
- 2 Studierende werden erzeugt und der Vorlesung hinzugefügt:  
`dap1.add( new Student( "C", "M", 17 ) );`  
`dap1.add( new Student( "A", "Inf", 22 ) );`
- Visualisierung der Situation:



## Problemstellung «Vorlesung»

(Fortsetzung)

```
Lecture dap1 = new Lecture( "DAP 1", 5 );
dap1.add( new Student( "C", "M", 17 ) );
dap1.add( new Student( "A", "Inf", 22 ) );
```



## Problemstellung «Vorlesung»

(Fortsetzung)

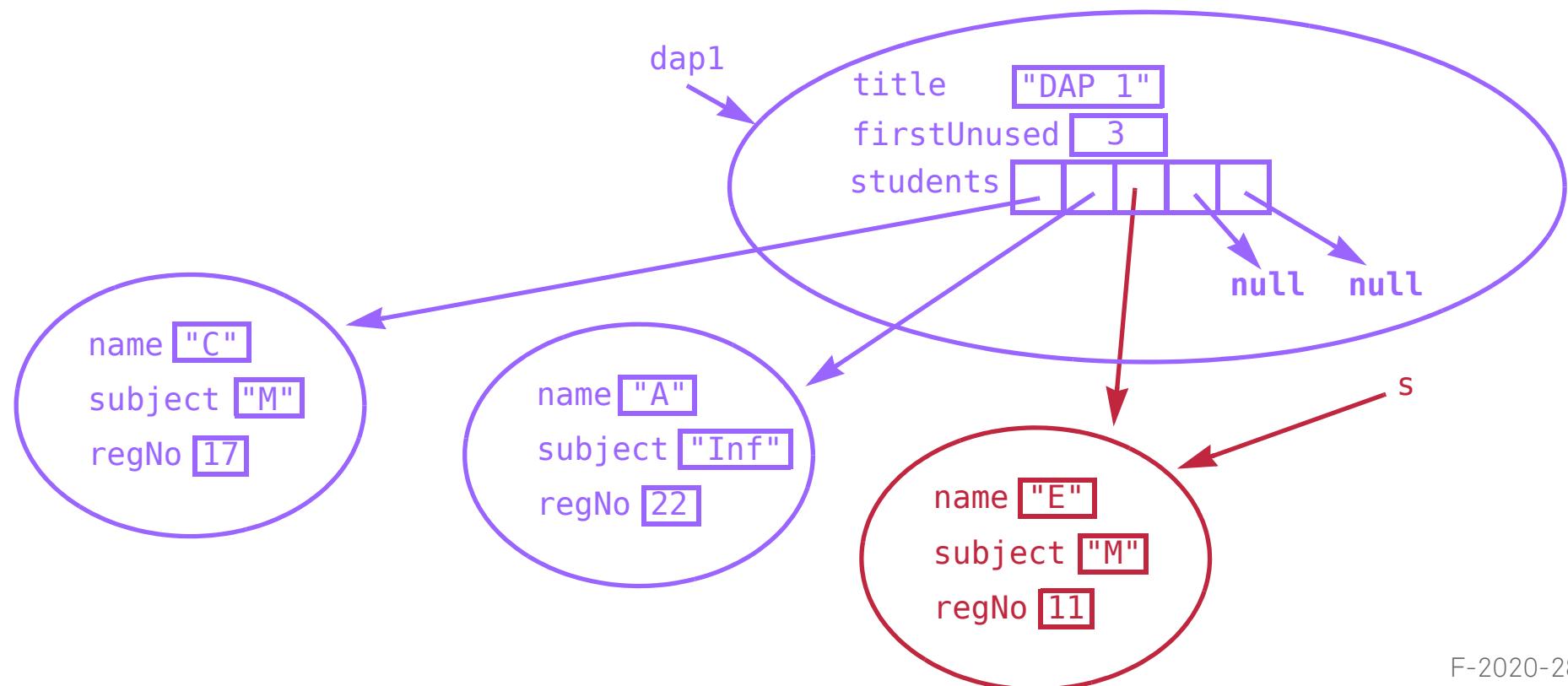
- Erzeugen eines weiteren Studierenden:

```
Student s = new Student( "E", "M", 11 );
```

- Hinzufügen dieses Studierenden:

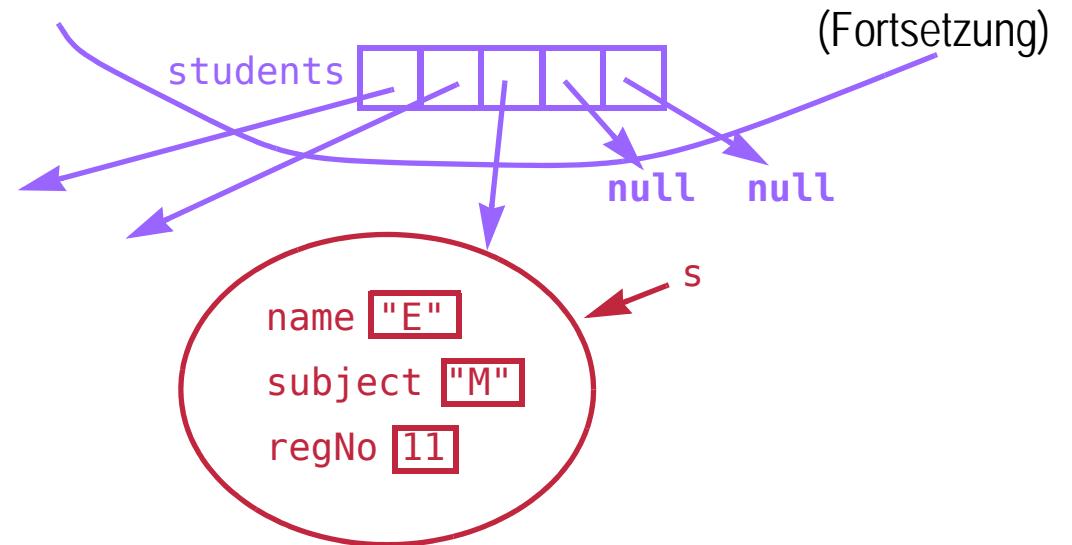
```
dap1.add( s );
```

- Visualisierung der Situation:



## Problemstellung «Vorlesung»

### Analyse



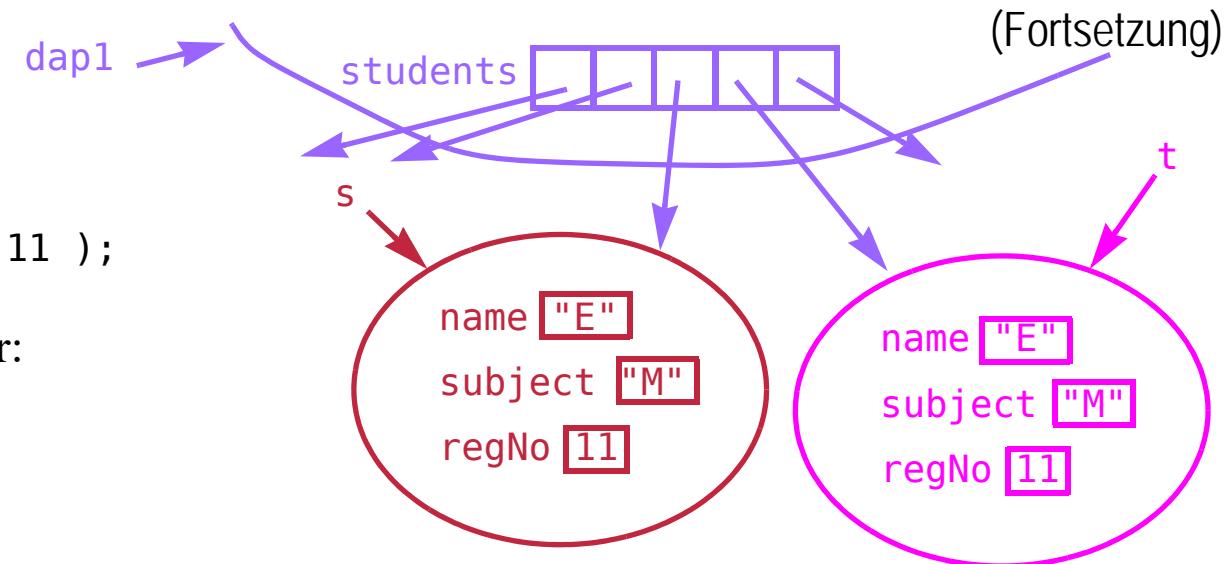
(Fortsetzung)

- ❑ Dasselbe Objekt ist sowohl über das Feld als auch über die Variable `s` erreichbar.
- ❑ Der Aufruf `dap1.add( s )` weist dem dritten Element des Feldes `students` die in der Referenzvariablen `s` abgelegte Referenz zu.
- ❑ Da die Referenz immer das gleiche Objekt erreicht, führen nun `students[2]` und `s` zum gleichen Ziel.
- ❑ Objekte werden also – wie auch Felder – bei einer Zuweisung (an den Parameter der Methode `add`) *nicht* kopiert.
- ❑ Nach einer Zuweisung unter Referenzvariablen ist das gleiche Objekt über mehrere Variablen (`students[2]` und `s`) erreichbar.

## Referenzvariablen

- ❑ nach  

```
Student t =
    new Student( "E", "M", 11 );
dap1.add( t );
liegt die folgende Situation vor:
```



- ❑ `s` ist eine Referenzvariable,  
die das **rote Objekt** kennt.
- ❑ `students[2]` ist eine Referenzvariable,  
die das **rote Objekt** kennt.
- ❑ `s == students[2]` vergleicht daher die in beiden Referenzvariablen gespeicherten Referenzen,  
die das gleiche Objekt bezeichnen. Als Ergebnis wird daher **true** zurückgegeben.
- ❑ Das **magenta Objekt** besitzt als Inhalte die **gleichen** Attributwerte wie das **rote Objekt**,  
die aber, da es sich um eine eigenständiges Objekt handelt, an einem **anderen** Ort liegen.
- ❑ `s == t` vergleicht wieder die von den beiden Referenzvariablen gespeicherten Referenzen  
die unterschiedliche Objekte bezeichnen. Als Ergebnis wird daher **false** zurückgegeben.
- ❑ Sollen die **Inhalte zweier Objekt** miteinander verglichen werden, wird immer eine Methode  
benötigt, die auf die Werte der Attribute zugreift und diese miteinander vergleicht.

## Problemstellung «Vorlesung»

(Fortsetzung)

Anmerkungen zum Wert **null**:

- ❑ **null** ist keine Referenz auf ein Objekt, sondern ermöglicht es auszudrücken, dass einer Referenzvariable *kein* Objekt zugeordnet ist.
- ❑ Jede Referenzvariable kann den Wert **null** annehmen.  
Sie verweist dann *nicht* auf ein Objekt.
- ❑ Auf den Wert **null** kann der Dereferenzierungsoperator *nicht* angewandt werden, da ja kein Objekt existiert, zu dem dereferenziert werden könnte.
- ❑ Wird versucht, den Dereferenzierungsoperator auf den Wert **null** anzuwenden, so wird eine Ausnahme der Klasse `NullPointerException` geworfen und damit die Programmausführung abgebrochen.

## Problemstellung «Vorlesung»

get-Methoden für die Klasse Student

(Fortsetzung)

```
public String getName()
{
    return name;
}

public String getSubject()
{
    return subject;
}

public int getRegistrationNo()
{
    return registrationNo;
}
```

- ❑ Die get-Methoden für die Klasse Student erlauben einen Zugriff auf die Attributwerte, ohne eine Möglichkeit zur Änderung zu geben.
- ❑ Das schließt zufällige und unerwünschte Zuweisungen an die Attribute eines schon erzeugten Objekts aus.

## Problemstellung «Vorlesung»

(Fortsetzung)

Vergleichsmethoden für die Klasse Student

```
public boolean hasGreaterNumber( Student s )
{
    return getRegistrationNo() > s.getRegistrationNo();
}
```

- Diese Methode erlaubt einen Größenvergleich von zwei Objekten der Klasse Student anhand der Werte des Attributs registrationNo.
- Nutzung:

```
Student s1 = new Student( "C", "M", 17 );
Student s2 = new Student( "A", "Inf", 22 );
if ( s1.hasGreaterNumber( s2 ) )
{ ... }
```

- Der eine Operand für den Vergleich mit `>` wird durch das Attribut des ausführenden Objekts gegeben, der zweite Operand durch das Attribut des als Argument an den Parameter übergebenen Objekts.
- Das Ergebnis entspricht der ausgesprochenen Notation:  
`true` wird zurückgegeben, falls gilt: *s1 has greater number (than) s2*

## Problemstellung «Vorlesung»

Vergleichsmethoden für die Klasse Student

(Fortsetzung)

```
public boolean hasEqualNumber( Student s )
{
    return getRegistrationNo() == s.getRegistrationNo();
}
```

- ❑ Die Implementierung ist analog zu der der Methode hasGreaterNumber.

## Problemstellung «Vorlesung»

show-Methode für die Klasse Lecture

(Fortsetzung)

```
public void show()
{
    System.out.println( "lecture: " + title );
    System.out.println( "participants:" );
    for ( Student s : students )
    {
        if ( s != null )
        {
            System.out.println( s.toString() );
        }
    }
}
```

- ❑ Die Methode `show` dient zur Unterstützung von Testläufen, da sich mit ihr der Inhalt aller mit einer Vorlesung verbundenen Objekte einfach ausgeben lässt.
- ❑ Die Methode `show` beinhaltet eine geeignete zeilenorientierte Formatierung durch `println`.

## Klasse String, primitiver Typ char

- ❑ `String` ist eine vordefinierte Klasse, die in Java immer zur Verfügung steht.
- ❑ Java erlaubt deshalb eine besondere Form für die Erzeugung von `String`-Objekten (Textliterale), die ohne den Aufruf eines Konstruktors auskommt:

```
String s = "hello";  
statt      String s = new String ( ... );
```

- ❑ Die Beschreibung der Klasse `String` findet sich unter:  
<https://docs.oracle.com/en/java/javase/>
- ❑ Ein `String`-Objekt beinhaltet eine nicht änderbare Folge von Zeichen, ein `String`-Objekt kann sich daher während der Ausführung *nicht* ändern.

Anmerkung:

Daher können `String`-Objekte mit gleichen Inhalten durch ein einziges Objekt mit diesem Inhalt ersetzt werden. Das Laufzeitsystem nimmt solche Ersetzungen vor, um u.a. Speicherplatz zu sparen.

Als Folge kann für Referenzvariablen der Klasse `String` ein Vergleich mit dem Operator `==` manchmal zu unerwarteten Ergebnissen führen. Zum Vergleich zweier `String`-Objekte sollte immer eine Methode benutzt werden.

## Klasse String, primitiver Typ **char**

(Fortsetzung)

- ❑ Ein einzelnes Zeichen wird in Java durch den Typ **char** repräsentiert.
- ❑ Literale des Typs **char** werden in ' ' angegeben:

```
char c = 'a';
```

- ❑ Der Typ **char** ist ein Teilbereich des Typs **int**:  
Für jedes Zeichen aus **char** ist eine Abbildung auf einen Wert aus **int** definiert:

'0'	→	48
'9'	→	57
'A'	→	65
'Z'	→	90
'a'	→	97
'z'	→	122

**int-Addition**

```
char c1 = '1';
char c2 = 'A';
char c3 = (char)(c1 + c2);
System.out.println( "c1: " + c1 + ", c2: " + c2 + ", c3: " + c3 );
```

```
c1: 1, c2: A, c3: r
```

## Klasse String, primitiver Typ `char`

(Fortsetzung)

Konstruktoren (u.a.):

- `String()`  
Initializes a newly created `String` object so that it represents an empty character sequence. \*)
- `String( char[] value )`  
Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument.
- `String( char[] value, int offset, int count )`  
Allocates a new `String` that contains characters from a subarray of the character array argument.
- `String( String original )`  
Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

\*) Beschreibungen aus <https://docs.oracle.com/en/java/javase/>

## Klasse String, primitiver Typ `char`

(Fortsetzung)

Methoden (u.a.):

- ❑ `char charAt( int index )`  
Returns the `char` value at the specified index.
- ❑ `int compareTo( String anotherString )`  
Compares two strings lexicographically.
- ❑ `int compareToIgnoreCase( String str )`  
Compares two strings lexicographically, ignoring case differences.
- ❑ `String concat( String str )`  
Concatenates the specified string to the end of this string.
- ❑ `int indexOf( int ch )`  
Returns the index within this string of the first occurrence of the specified character.
- ❑ `boolean isEmpty()`  
Returns `true` if, and only if, `length()` is 0.

Ergebnis ist `int`, um  
`>`, `<` und `==`  
anzeigen zu können

## Klasse String, primitiver Typ `char`

(Fortsetzung)

Methoden (u.a.):

- ❑ `int lastIndexOf( int ch )`  
Returns the index within this string of the last occurrence of the specified character.
- ❑ `int length()`  
Returns the length of this string.
- ❑ `String replace( char oldChar, char newChar )`  
Returns a **new** string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
- ❑ `String substring( int beginIndex, int endIndex )`  
Returns a **new** string that is a substring of this string.
- ❑ `char[] toCharArray()`  
Converts this string to a **new** character array.
- ❑ `String trim()`  
Returns a copy of the string, with leading and trailing whitespace omitted.

## Problemstellung «Vorlesung»

weitere Methode für die Klasse Student

(Fortsetzung)

```
public boolean hasGreaterName( Student s )
{
    return getName().compareTo( s.getName() ) > 0;
}

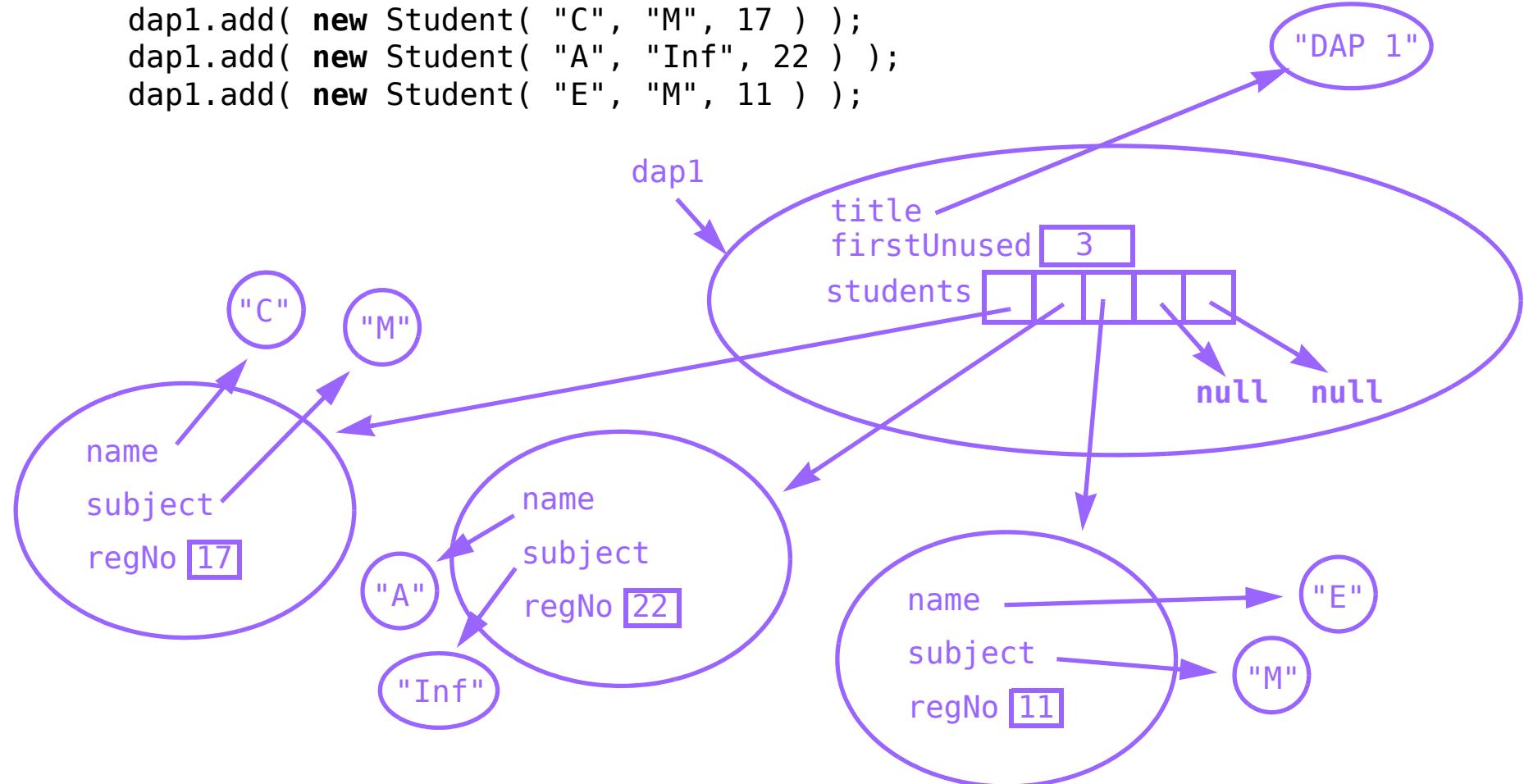
public boolean hasEqualName( Student s )
{
    return getName().compareTo( s.getName() ) == 0;
}
```

- Diese Methoden erlauben einen Vergleich von zwei Objekten der Klasse Student anhand der Werte des Attributs name.

## Problemstellung «Vorlesung» Visualisierung der Objektstruktur

(Fortsetzung)

```
Lecture dap1 = new Lecture( "DAP 1", 5 );
dap1.add( new Student( "C", "M", 17 ) );
dap1.add( new Student( "A", "Inf", 22 ) );
dap1.add( new Student( "E", "M", 11 ) );
```



## Problemstellung «Vorlesung»

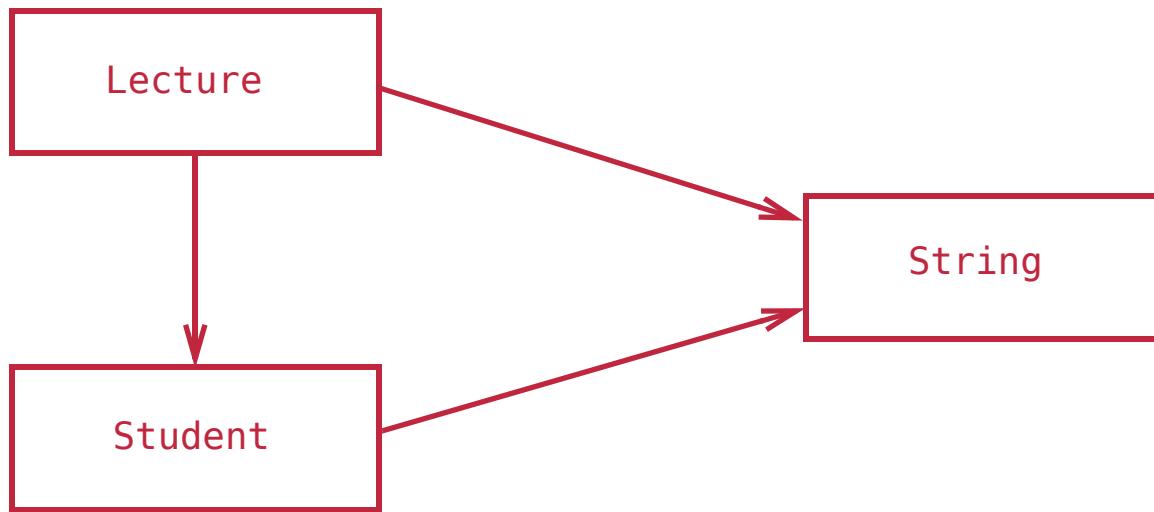
Anmerkung:

Die beiden Klassen `Lecture` und `Student` unterscheiden sich von der Klasse `Fraction`:

- Die Attribute besitzen unterschiedliche Typen.
- Objekte der Klasse `Student` können Attributwerte der Klasse `Lecture` sein.
- Die Methoden beider Klassen berechnen keine neuen Attributwerte, sondern speichern diese nur.
- Alle Methoden der Klasse `Student` sind recht trivial.

## Problemstellung «Vorlesung» Visualisierung der Klassenstruktur \*)

(Fortsetzung)



Bedeutung von → : kennt Objekt(e) «länger als nur für einen Methodenaufruf»

\*) Notation: *UML (Unified Modeling Language)*, Details in der Vorlesung Softwaretechnik)

## Gestaltung von Klassen (und Objekten)

- ❑ Klassen sollen helfen, Programme übersichtlich und verständlich zu gestalten.
- ❑ Eine Klasse sollte eine abgeschlossene Aufgabe erfüllen.
- ❑ Die Namen von Klassen, Methoden, Attributen und Variablen sollten selbsterklärend sein:
  - Klassennamen sind immer Substantive.
  - Methodennamen sind immer Verben.
- ❑ Die Sichtbarkeit von Attributen sollte als **private** deklariert werden.
- ❑ Die Sichtbarkeit von Methoden sollten aufgrund ihrer Verwendung deklariert werden:
  - Methoden, die außerhalb der Klasse nicht benötigt werden, sollten privat bleiben.
  - Methoden, die außerhalb der Klasse benötigt werden,  
müssen öffentlich gemacht werden.
- ❑ Öffentliche Methoden sollen den Zugang zu den privaten Attributen kontrollieren.
- ❑ Methoden können
  - Attributwerte zurückgeben,
  - Attributwerte setzen,
  - Attributwerte manipulieren,
  - Attributwerte zur Berechnung von Ergebnissen verwenden.
- ❑ Konstruktoren sollten dafür sorgen, dass nur Objekte entstehen, die konzeptionell sinnvolle Werte enthalten.

## Gestaltung von Klassen (und Objekten)

(Fortsetzung)

- ❑ Attribute können internes Wissen der Klasse speichern.
- ❑ Das zeigen die Beispiele:
  - Mit der Klasse `Fraction` wird ein bekannter mathematischer *Datentyp* realisiert, der so in Java nicht verfügbar ist.
  - Die Klassen `Student` und `Lecture` schaffen spezielle Datentypen, die genau auf eine bestimmte Aufgabe zugeschnitten sind, die wir selbst vorher definiert haben.
- ❑ Ziel der objektorientierten Softwareentwicklung mit Klassen und Objekten ist es, situationsangemessene (*abstrakte*) *Datentypen* zu bestimmen, die Daten und die darauf zulässigen Operationen zusammenfassen.
- ❑ Die Deklaration einer Java-Klasse ist dann die Konkretisierung eines solchen Datentyps.
- ❑ Daher sollen die Attribute einer Klasse als `private` deklariert werden, so dass bei der Benutzung der Klasse nur die – durch Methoden realisierten – zulässigen Operationen durchgeführt werden können.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 4.1. Sortieralgorithmen - «Sortieren durch Auswählen»

Dr. Stefan Dissmann  
Fakultät für Informatik



## Kapitel 4. Sortieralgorithmen

# So geht es weiter:

Sortieren der Studierenden einer Vorlesung nach  
– Matrikelnummern oder  
– Namen

- ❑ Betrachtung verschiedener Sortieralgorithmen
- ❑ Betrachtung des zum Sortieren notwendigen Aufwandes
- ❑ Betrachtung von Sonderfällen wie einer bereits vorliegenden Sortierung

## Lernziele des Kapitels 4. Sortieralgorithmen

Nach Durcharbeiten des Kapitels Sortieralgorithmen sollen die teilnehmenden Studierenden

- den Sortieralgorithmus *SelectionSort* kennen
- den Sortieralgorithmus *InsertionSort* kennen
- den Sortieralgorithmus *QuickSort* kennen
- grobe Analysen zur Laufzeit dieser Algorithmen erklären können
- die *O-Notation* erklären und anwenden können
- die Implementierungen der Algorithmen erklären können
- den rekursiven Ansatz von *QuickSort* verstanden haben

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Ziel:

Die im Feld `students` eines Objekts der Klasse `Lecture` abgelegten Studierenden sollen *aufsteigend* nach ihren Matrikelnummern sortiert werden.

### Lösungsansatz über Analogie:

Wie würde man einen ungeordneten Stapel von Karteikarten von Studierenden sortieren?

Algorithmus *Sortieren durch Auswählen (SelectionSort)*:

- 
- Suche im Ausgangsstapel die Karte mit der kleinsten Matrikelnummer heraus.
  - Lege mit dieser Karte einen neuen Stapel an: die bereits sortierten Karten.
  - Suche im Ausgangsstapel erneut die Karte mit der kleinsten Matrikelnummer heraus.
  - Lege diese Karte unter den Stapel der sortierten Karten.
  - Fahre mit diesen letzten beiden Schritten fort, bis der Ausgangsstapel abgearbeitet ist.

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.

Ausgangssituation:



## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:

Ausgangssituation:



## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

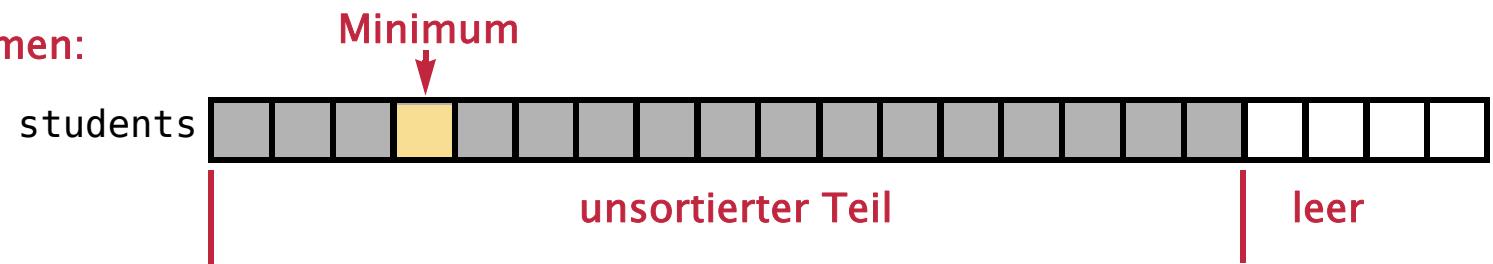
Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:

Ausgangssituation:



Minimum bestimmen:



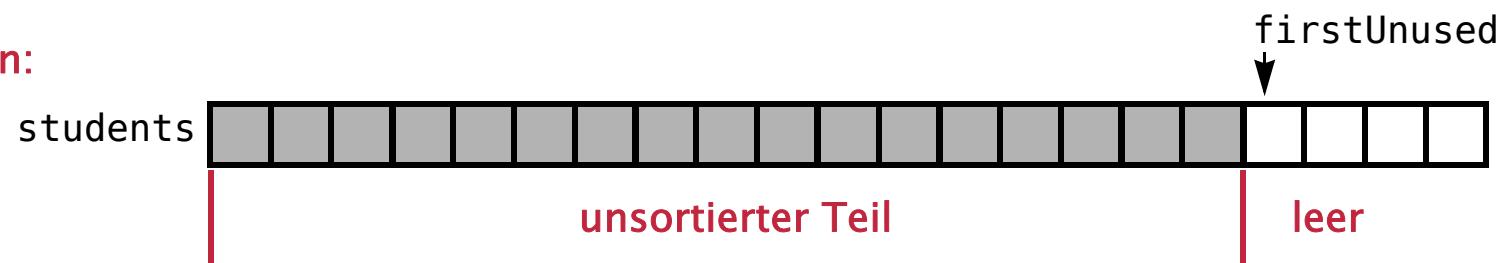
## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

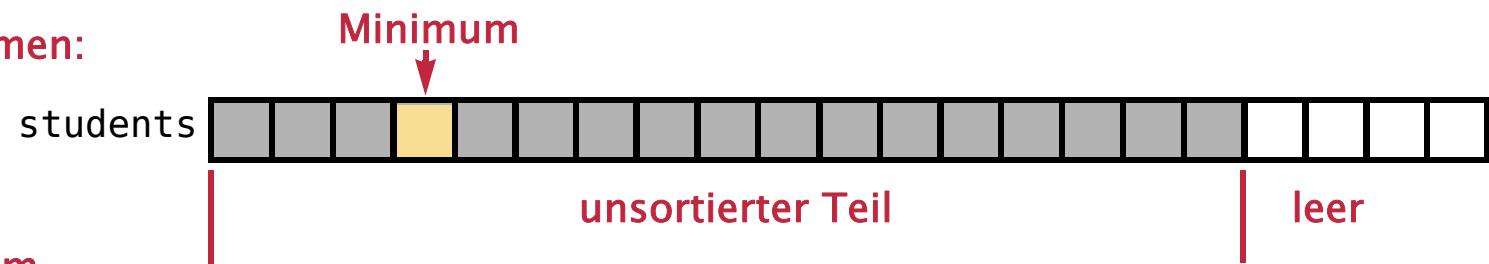
Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:

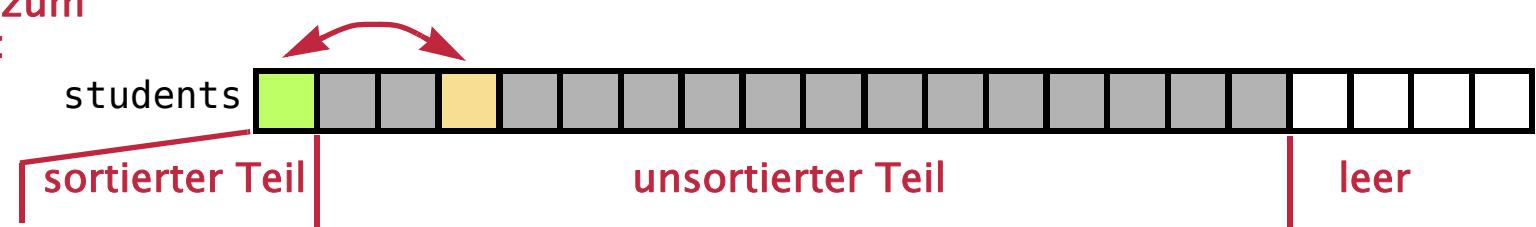
Ausgangssituation:



Minimum bestimmen:



Minimum wird zum sortiertem Teil:

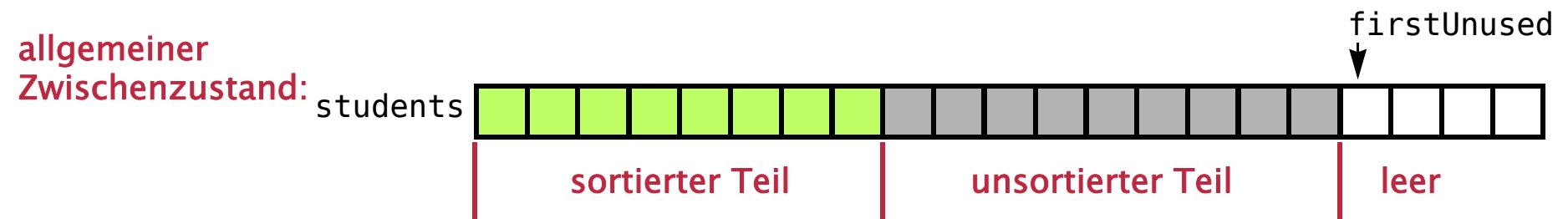


## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:

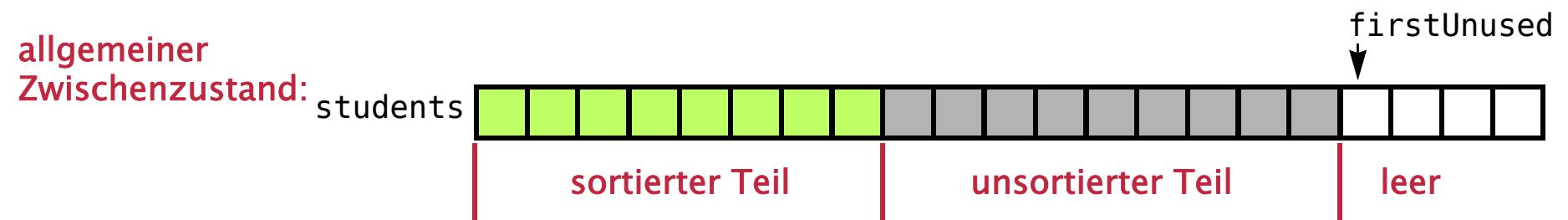


## Problemstellung «Sortieren durch Auswählen»

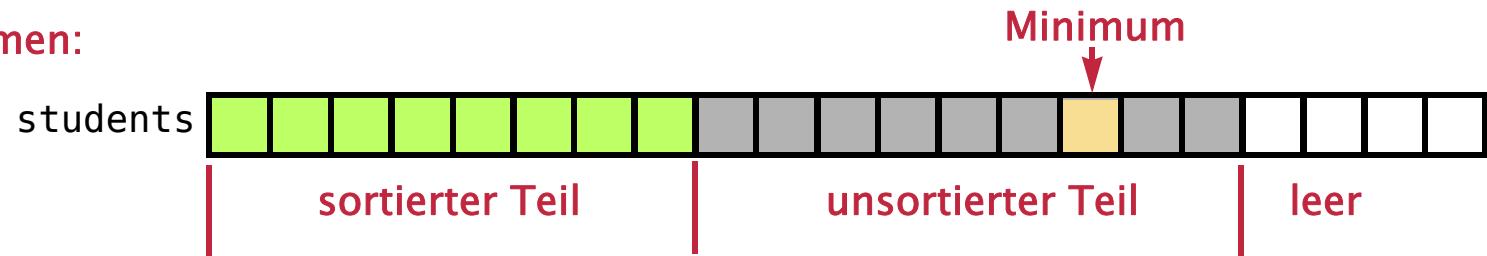
(Fortsetzung)

Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:



**Minimum bestimmen:**

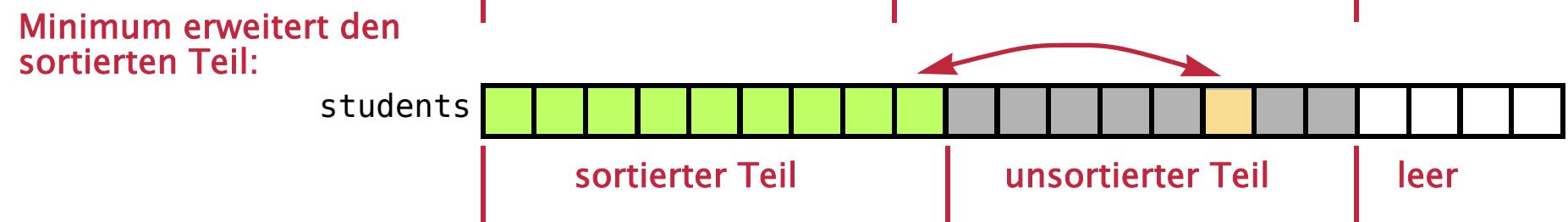
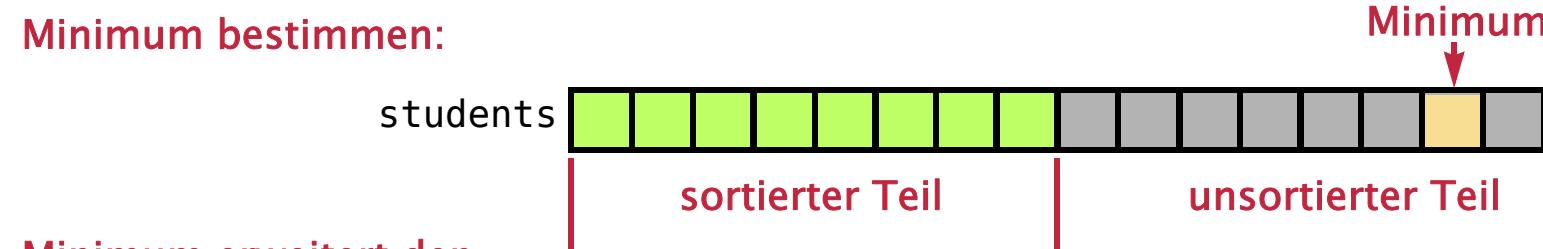
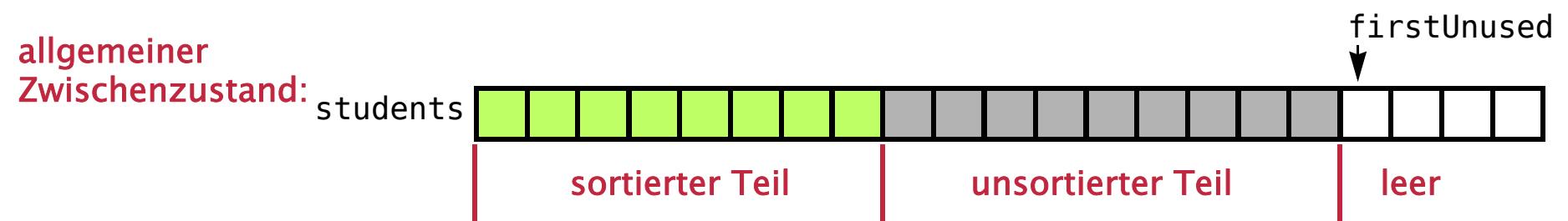


## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:



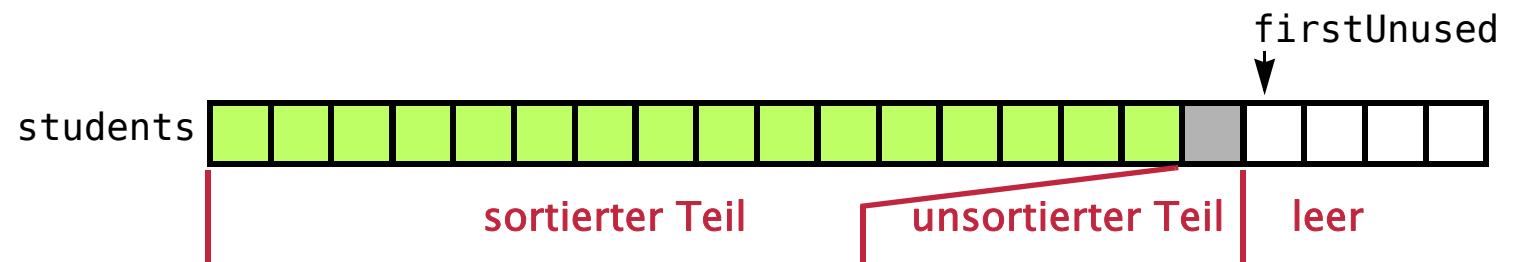
## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:

**Endzustand:**



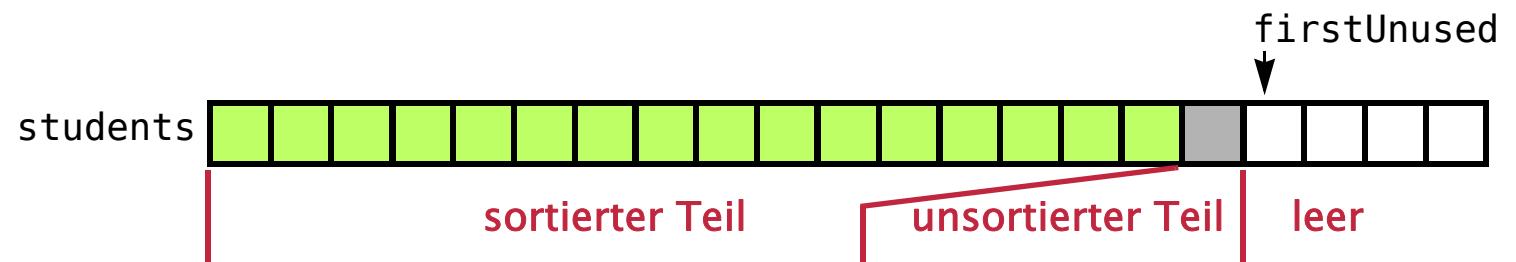
## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

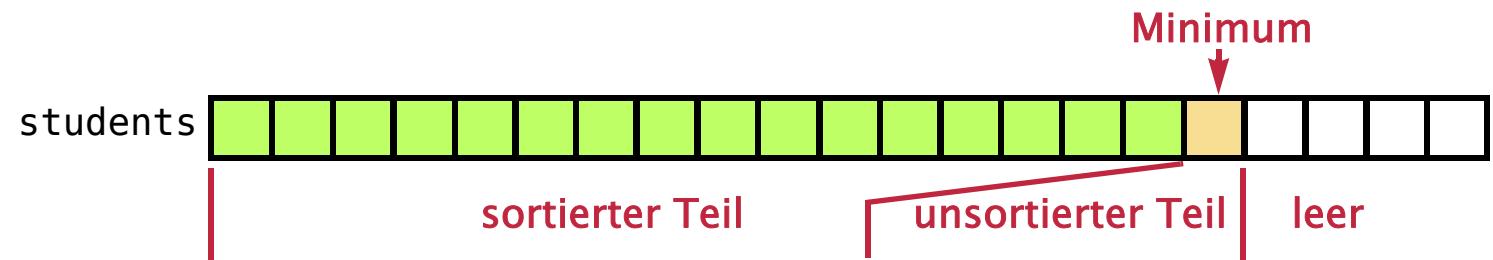
Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:

Endzustand:



letzter Wert  
ist Minimum:



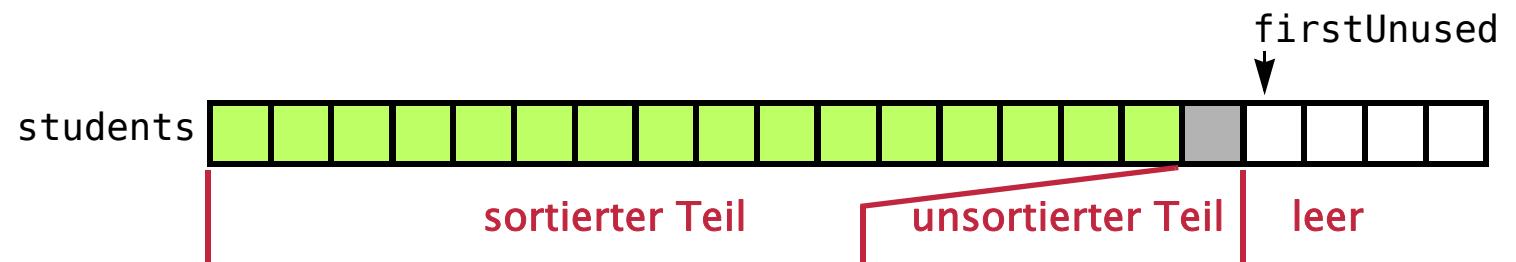
## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

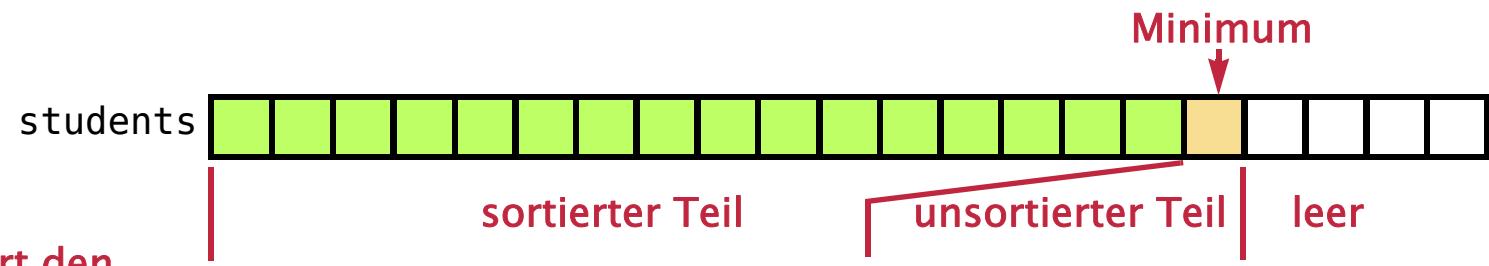
Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:

Endzustand:



letzter Wert  
ist Minimum:



Minimum erweitert den  
sortierten Teil:

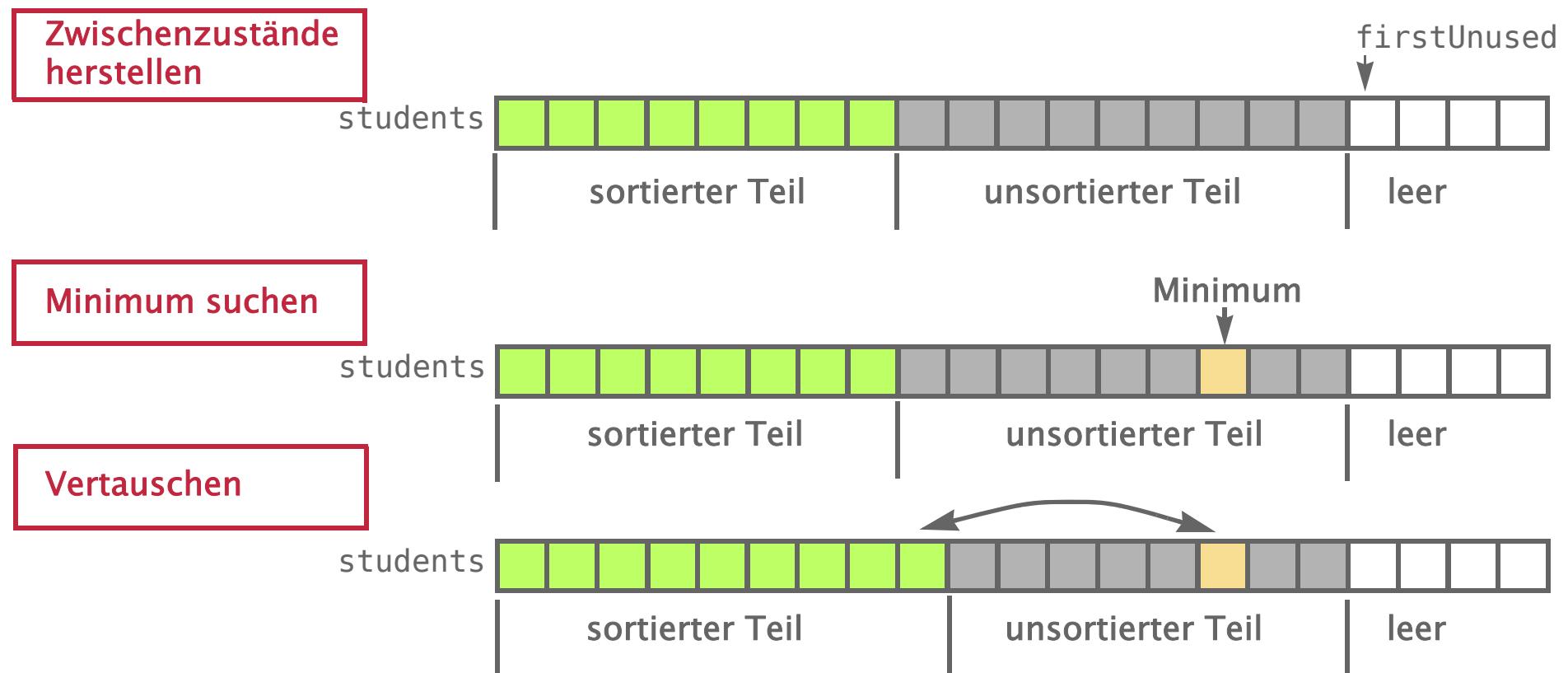


## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:



## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Implementierung *SelectionSort*:

- ❑ Es liegt bereits ein Feld `students` vor, das Platz für alle zu sortierenden Objekte besitzt.
- ❑ Es wird daher kein neues Feld für die sortierten Objekte angelegt, sondern im bestehenden Feld `students` werden der sortierte und der unsortierte Teil unterschieden:

Zwischenzustände  
herstellen

Methode `void selectionSortByNumber()`

Minimum suchen

Methode `searchForMinimalNumber( int start )`

Vertauschen

Methode `void swapStudents( int i, int j )`

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

```

public void selectionSortByNumber()
{
    for ( int i = 0; i < firstUnused - 1; i++ )
    {
        int position = searchForMinimalNumber( i );
        swapStudents( i, position );
    }
}

```

- ❑ Der letzte zu sortierende Eintrag muss nicht betrachtet werden, da er immer sein eigenes Minimum bildet.
- ❑ Die Methode `searchForMinimalNumber` gibt den Index des `Student`-Objekts mit der kleinsten Matrikelnummer zurück.
- ❑ Die Suche mit `searchForMinimalNumber` erfolgt im unsortierten Teil des Feldes `students`:
  - Der unsortierte Teil beginnt bei Index `i`.
  - Der unsortierte Teil endet bei Index `firstUnused-1`.

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

```
private int searchForMinimalNumber( int start )
{
    int selected = start;
    for ( int i = start + 1; i < firstUnused; i++ )
    {
        if ( students[selected].hasGreaterNumber( students[i] ) )
        {
            selected = i;
        }
    }
    return selected;
}
```

- ❑ Die Methode `searchForMinimalNumber` gibt den Index zurück, an dem das `Student`-Objekt mit der kleinsten Matrikelnummer innerhalb des unsortierten Teils des Feldes `students` steht.
- ❑ Die Variable `selected` enthält nach jedem Durchlauf durch die Schleife den Index, an dem das bis dahin gefundene `Student`-Objekt mit der kleinsten Matrikelnummer steht.

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

```
private void swapStudents( int i, int j )
{
    if ( i != j )
    {
        Student temp = students[i];
        students[i] = students[j];
        students[j] = temp;
    }
}
```

- ❑ Das Vertauschen der Inhalte von Elementen des Feldes `students` wird in verschiedenen Methoden innerhalb der Klasse `Lecture` benötigt.  
Daher wird dieses Vertauschen als eigenständige, private Methode realisiert.
- ❑ Das Vertauschen von Inhalten von zwei Variablen oder Attributen erfordert immer den Einsatz einer Hilfsvariablen, um durch Zwischenspeichern ein Überschreiben zu vermeiden.
- ❑ Die Methode vermeidet das (unnütze) Vertauschen von Objekten an der gleichen Position.

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

```
public void selectionSortByNumber()
{
    for ( int i = 0; i < firstUnused - 1; i++ )
    {
        int position = searchForMinimalNumber( i );
        swapStudents( i, position );
    }
}
```

Analyse:

Wie häufig wird vertauscht (Aufruf der Methode `swapStudents`), wenn  $n$  Einträge vorliegen, also `firstUnused` den Wert  $n$  hat?

$n-1$  -mal

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

```

public void selectionSortByNumber()
{
    for ( int i = 0; i < firstUnused - 1; i++ )
    {
        int position = searchForMinimalNumber( i );
        swapStudents( i, position );
    }
}

private int searchForMinimalNumber( int start )
{
    int selected = start;
    for ( int i = start + 1; i < firstUnused; i++ )
    {
        if ( students[selected] hasGreaterNumber( students[i] ) )
        {
            selected = i;
        }
    }
    return selected;
}

```

Analyse:

Wie häufig wird die Methode `hasGreaterNumber` in der Methode `searchForMinimalNumber` aufgerufen, wenn  $n$  Student-Objekte vorliegen, also `firstUnused` den Wert  $n$  hat?

$$n-1 + n-2 + \dots + 1 = (n-1) \cdot n/2 = n^2/2 - n/2 \text{-mal}$$

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

```

public void selectionSortByNumber()
{
    for ( int i = 0; i < firstUnused - 1; i++ )
    {
        int position = searchForMinimalNumber( i );
        swapStudents( i, position );
    }
}

private int searchForMinimalNumber( int start )
{
    int selected = start;
    for ( int i = start + 1; i < firstUnused; i++ )
    {
        if ( students[selected] hasGreaterNumber( students[i] ) )
        {
            selected = i;
        }
    }
    return selected;
}

```

Analyse:

Hängt die Zahl der Vergleiche von der Verteilung der Werte in der Ausgangsfolge ab?

*Nein, da die Schleifen beider Methoden immer vollständig durchlaufen werden.  
Auch eine schon sortierte Folge erfordert  $n^2/2-n/2$  Vergleiche.*

## Problemstellung «Sortieren durch Auswählen»

Variation: Sortieren anhand der Namen der Studierenden

(Fortsetzung)

```
public void selectionSortByName()
{
    for ( int i = 0; i < firstUnused - 1; i++ )
    {
        int position = searchForMinimalName( i );
        swapStudents( i, position );
    }
}
```



Algorithmus bleibt gleich,  
nur die Suche erfolgt mit  
einer anderen Methode

## Problemstellung «Sortieren durch Auswählen»

(Fortsetzung)

Variation: Sortieren anhand der Namen der Studierenden

```
private int searchForMinimalName( int start )
{
    int selected = start;
    for ( int i = start + 1; i < firstUnused; i++ )
    {
        if ( students[selected].hasGreaterName( students[i] ) )
        {
            selected = i;
        }
    }
    return selected;
}
```



Algorithmus bleibt gleich,  
nur Vergleich erfolgt mit  
einer anderen Methode

Anmerkung: Es ist ungeschickt, den ganzen Code zu duplizieren, nur um das Sortierkriterium zu ändern. Eine elegante Lösung zur Lösung dieser Problematik wird in später folgenden Abschnitten dieser Vorlesung vorgestellt.

## Abschätzung der Laufzeit

Analyseergebnis für *selectionSort*:

- Zahl der Vertauschungen für  $n$  Elemente:  $n-1$       also:  $f(n) = n-1$
- Zahl der Vergleiche für  $n$  Elemente:  $n^2/2-n/2$       also:  $g(n) = n^2/2-n/2$

Was ist die Aussage dieser beiden Funktionen?

## Abschätzung der Laufzeit

(Fortsetzung)

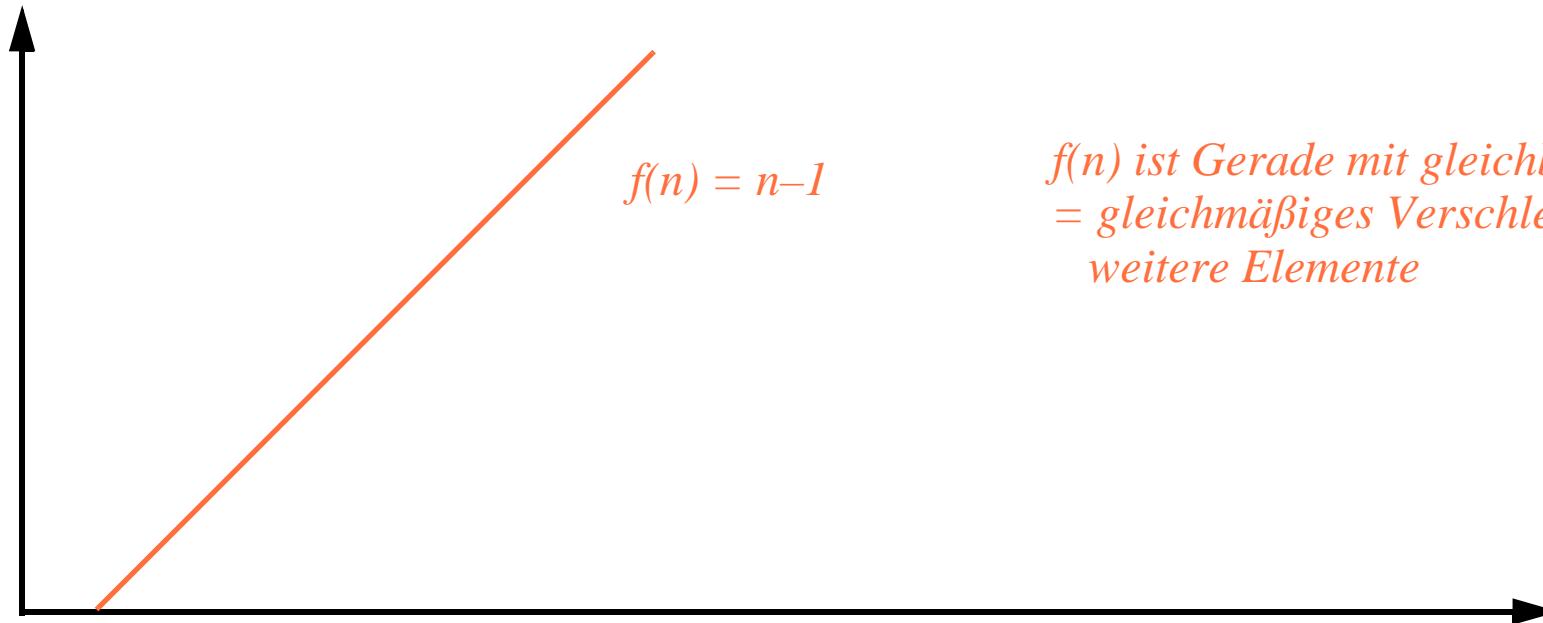
Analyseergebnis für *selectionSort*:

- Zahl der Vertauschungen für *n* Elemente:  $n-1$
- Zahl der Vergleiche für *n* Elemente:  $n^2/2-n/2$

also:  $f(n) = n-1$

also:  $g(n) = n^2/2-n/2$

Was ist die Aussage dieser beiden Funktionen?



## Abschätzung der Laufzeit

(Fortsetzung)

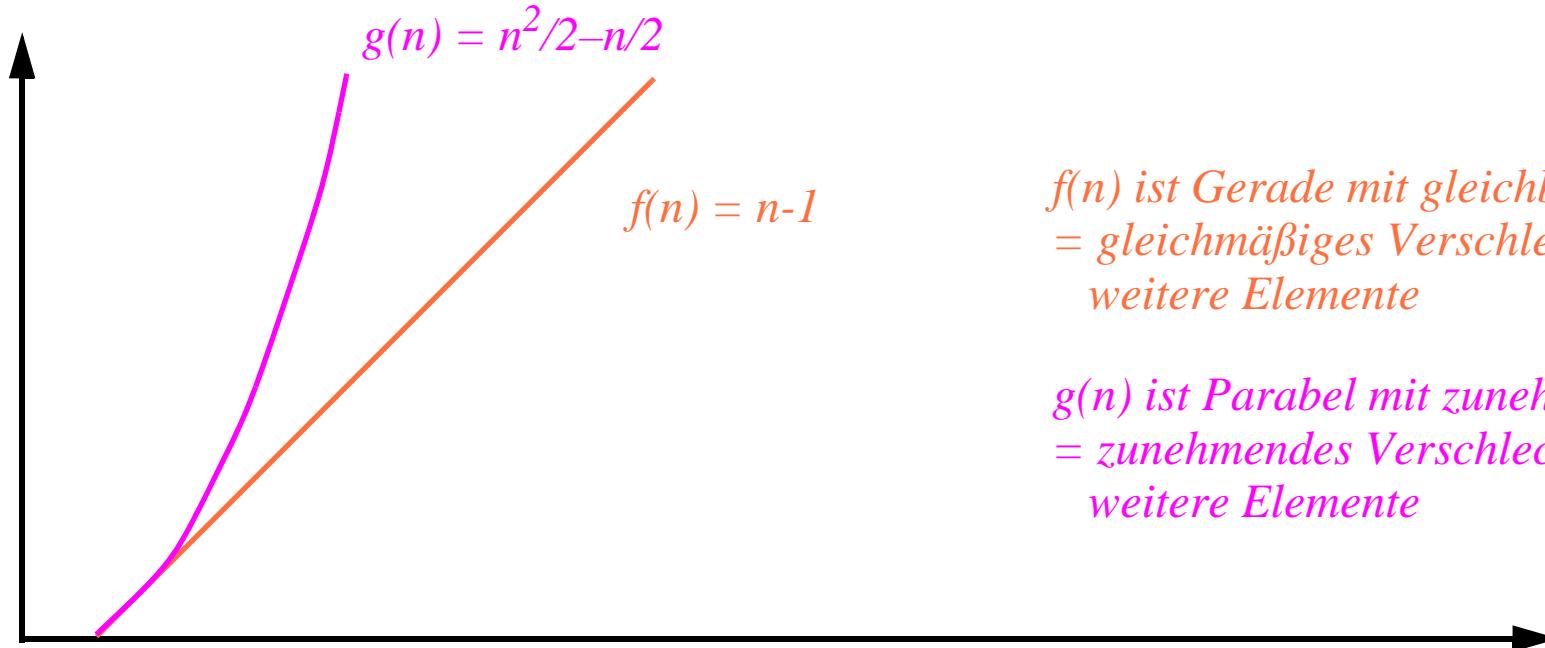
Analyseergebnis für *selectionSort*:

- Zahl der Vertauschungen für  $n$  Elemente:  $n-1$
- Zahl der Vergleiche für  $n$  Elemente:  $n^2/2-n/2$

also:  $f(n) = n-1$

also:  $g(n) = n^2/2-n/2$

Was ist die Aussage dieser beiden Funktionen?



## Abschätzung der Laufzeit

(Fortsetzung)

Ideen für die Analyse der *Laufzeit*:

- ❑ Es soll die Laufzeit von Algorithmen eingeordnet werden:  
Das ist eine (abstrakte) qualitative Aussage, nicht eine Angabe für die Ausführungszeit.
- ❑ Die Laufzeit dient insbesondere zur Unterscheidung und zum Vergleich von verschiedenen Algorithmen zur Lösung des gleichen Problems.
- ❑ Die konkrete Ausführungzeit hängt von der Ausführungsumgebung und insbesondere von der verwendeten Hardware ab. Die konkrete Ausführungzeit wird nicht betrachtet.
- ❑ Konstante Faktoren werden ignoriert.
- ❑ Es wird die Laufzeit von  $f(n)$  für  $n \rightarrow \infty$  betrachtet:  
Dann ist das Wachstum von  $f(n)$  wesentlich. \*)
- ❑ Um Laufzeiten miteinander vergleichen zu können, werden Algorithmen durch die Angabe sich ähnlich verhaltender *einfacher Funktionen* klassifiziert.

\*) Falls nicht  $n \rightarrow \infty$  gilt, muss die Aussage einer Analyse kritisch geprüft werden.

## Abschätzung der Laufzeit

(Fortsetzung)

- *O-Notation:*

$$O(f(n)) = \{ g(n) \mid \exists c > 0, n_0 > 0 \quad \forall n \geq n_0 : g(n) \leq c \cdot f(n) \} \quad \text{mit } f(n) > 0, g(n) > 0$$

- Hinweis:

$O(f(n))$  ist eine Menge von Funktionen.

- Interpretation:

$g(n) \in O(f(n))$  bedeutet, dass ab einem  $n_0$   $g(n)$  höchstens so stark wächst wie  $f(n)$ .

## Abschätzung der Laufzeit

(Fortsetzung)

### □ *O-Notation:*

$$O(f(n)) = \{ g(n) \mid \exists c > 0, n_0 > 0 \quad \forall n \geq n_0 : g(n) \leq c \cdot f(n) \} \quad \text{mit } f(n) > 0, g(n) > 0$$

### □ Hinweis:

$O(f(n))$  ist eine Menge von Funktionen.

### □ Interpretation:

$g(n) \in O(f(n))$  bedeutet, dass ab einem  $n_0$   $g(n)$  höchstens so stark wächst wie  $f(n)$ .

### □ Beispiele:

$$10n \in O(n) \quad \text{als Kurzfassung für} \quad n \rightarrow 10n \in O(n \rightarrow n)$$

$$O(10n) = O(n)$$

$$n^2 + 10 \in O(n^2)$$

$$10n \in O(n^2)$$

$$n^2 \notin O(n)$$

$$O(n) \subseteq O(n^2)$$

## Abschätzung der Laufzeit

(Fortsetzung)

Einordnung der Laufzeit von *SelectionSort*:

$$O(n^2)$$

da für  $f(n) = n^2/2 - n/2$  gilt:  $f(n) \in O(n^2)$

## Algorithmus «Sortieren durch Auswählen»

Idee des Sortierverfahrens «Sortieren durch Auswählen» (selection sort)

- Während des Sortierens werden unterschieden:
  - die Menge  $M$  der noch nicht sortierten Einheiten
  - die Folge  $F$  der bereits sortierten Einheiten
- In jedem Sortierschritt
  - wird eine Einheit  $e$  mit vorgegebenen Eigenschaften in  $M$  gesucht,
  - deren Position in  $F$  aufgrund der Eigenschaften feststeht.
- Aufwändig ist also immer das Bestimmen von  $e$ , während das Hinzufügen zu  $F$  einfach ist.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 4.2. Sortieralgorithmen - «Sortieren durch Einfügen»

Dr. Stefan Dissmann  
Fakultät für Informatik



## Problemstellung «Sortieren durch Einfügen»

### neuer Lösungsansatz über Analogie:

Wie würde man einen ungeordneten Stapel von Karteikarten von Studierenden sortieren?

### Algorithmus *Sortieren durch Einsetzen (InsertionSort)*:

- 
- Entferne aus dem Ausgangsstapel die erste Karte.
  - Lege mit dieser Karte einen neuen Stapel der sortierten Karten an.
  - Entferne aus dem Ausgangsstapel die erste Karte.
  - Sortiere diese Karte im Stapel der sortierten Karten an der richtigen Stelle ein.
  - Fahre mit diesen beiden Schritten fort, bis der Ausgangsstapel abgearbeitet ist.

### Anmerkungen:

- Dieser Algorithmus ist anders als *SelectionSort*.
- Das Suchen des Minimums entfällt, dafür muss eingesortiert werden.
- Macht sich dieser Unterschied  
in der Zahl der notwendigen Vergleiche und Vertauschungen bemerkbar?

## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

Ausgangssituation:



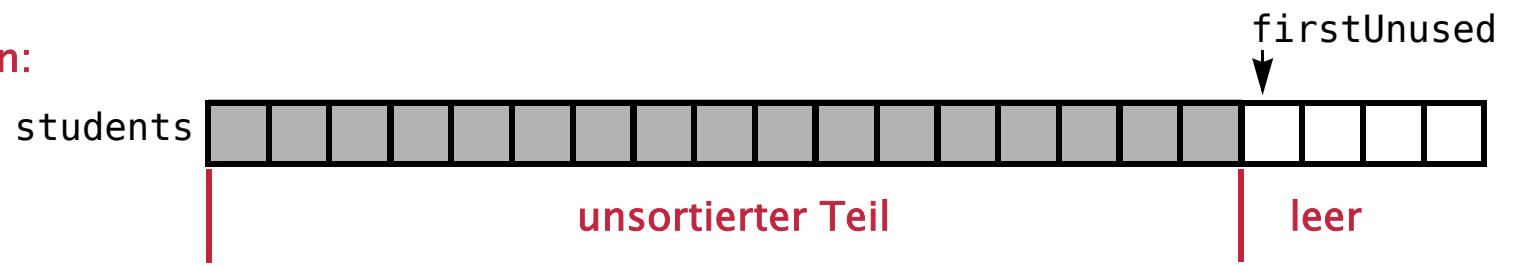
## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

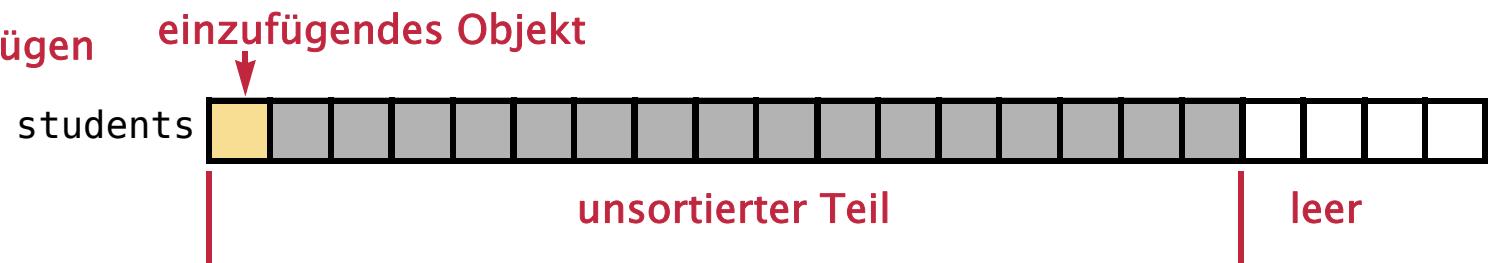
- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

Ausgangssituation:



Position zum Einfügen  
bestimmen:

einzufügendes Objekt

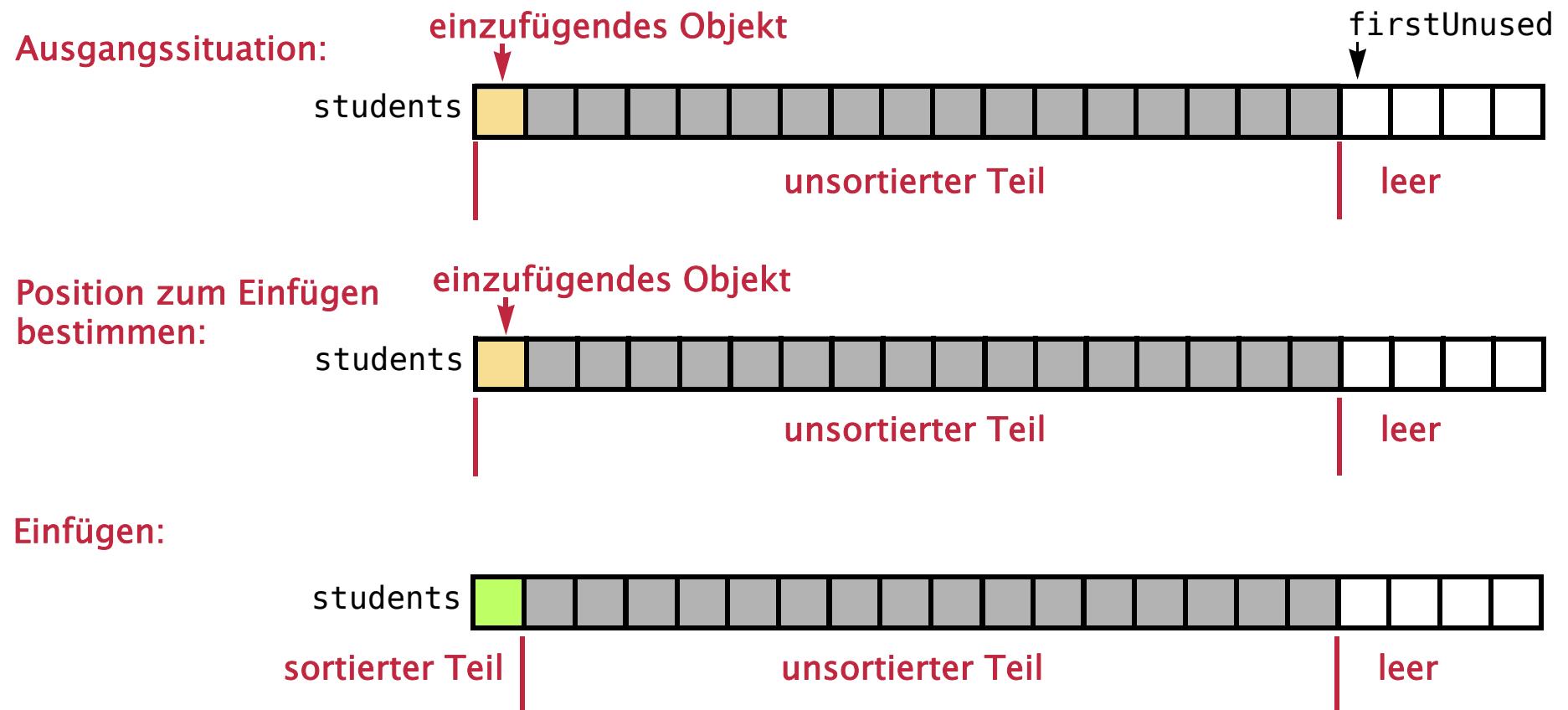


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

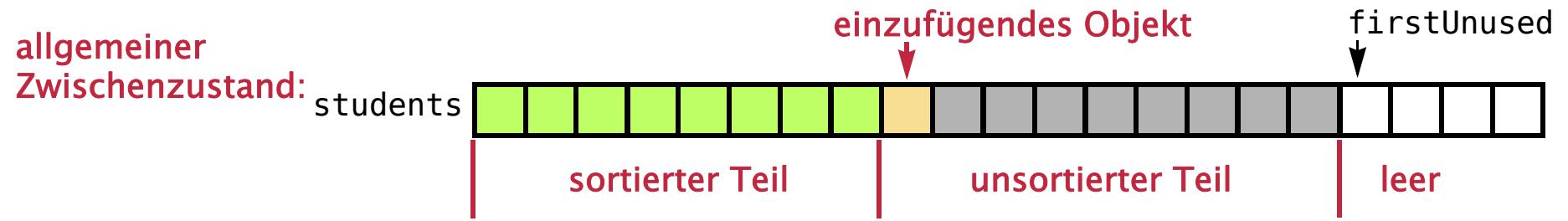


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

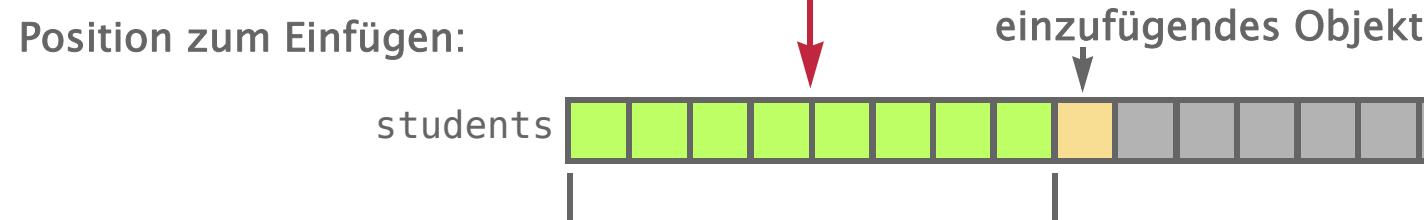
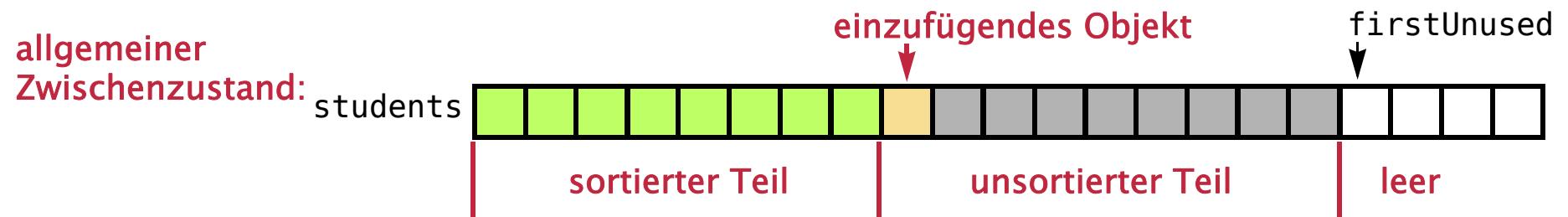


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

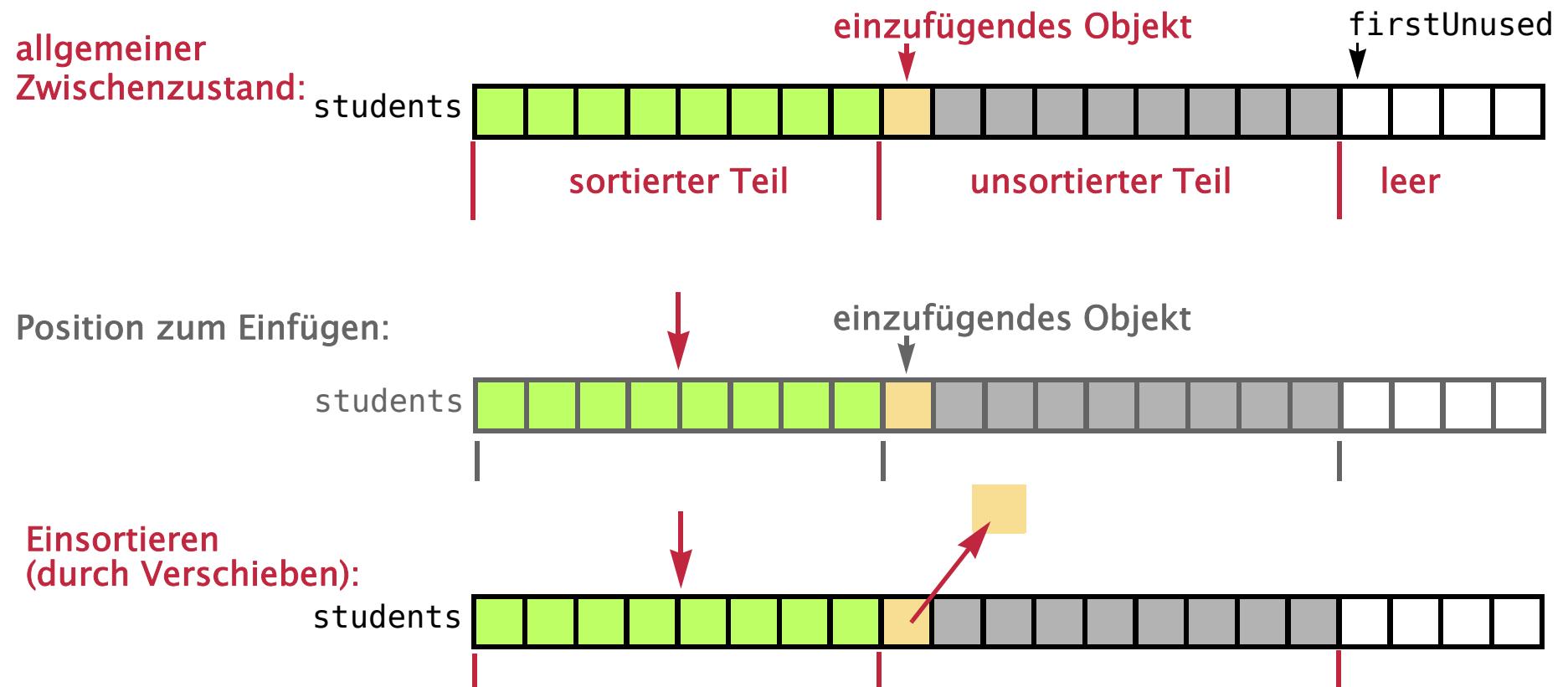


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

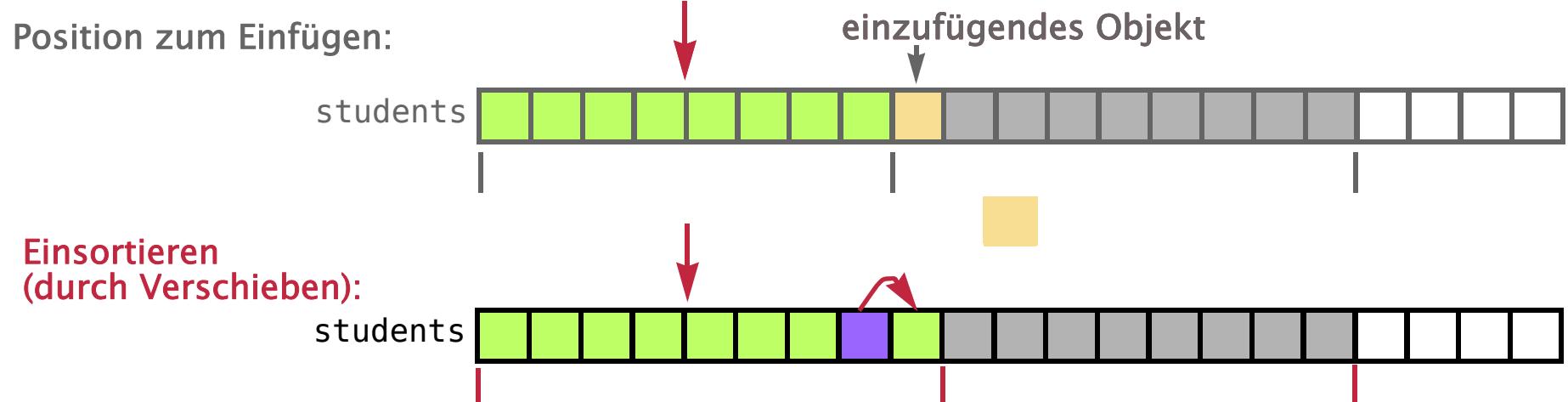
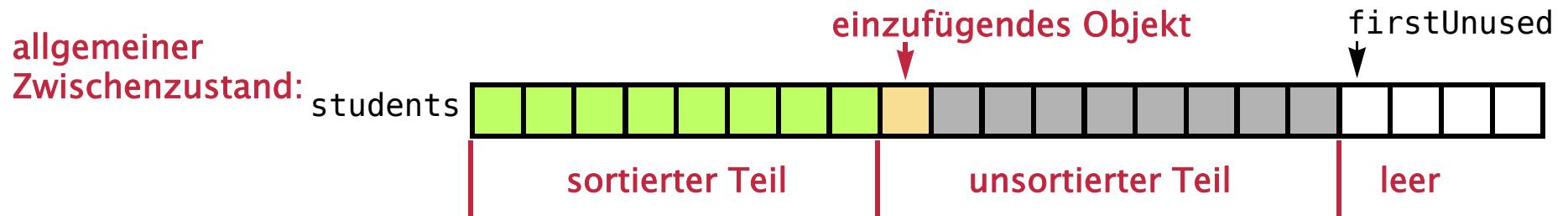


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

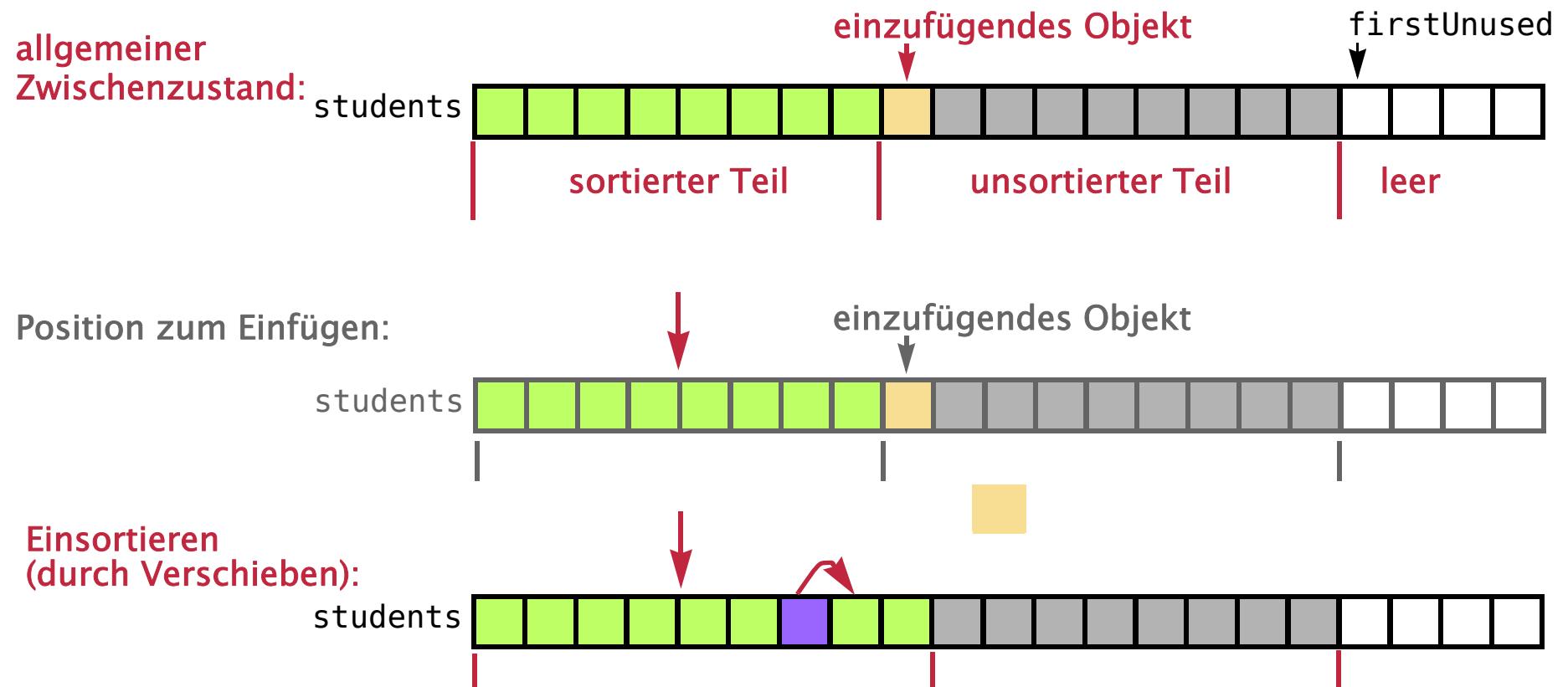


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

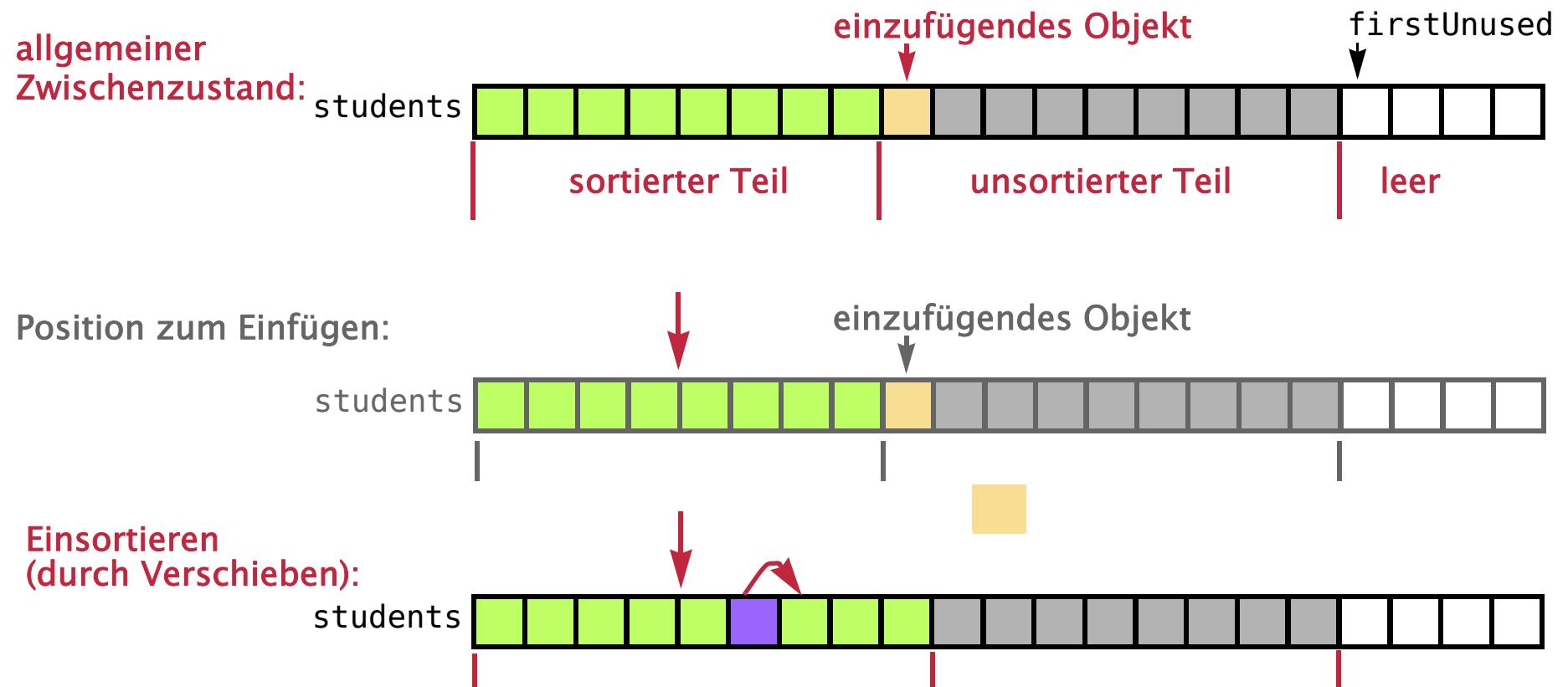


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

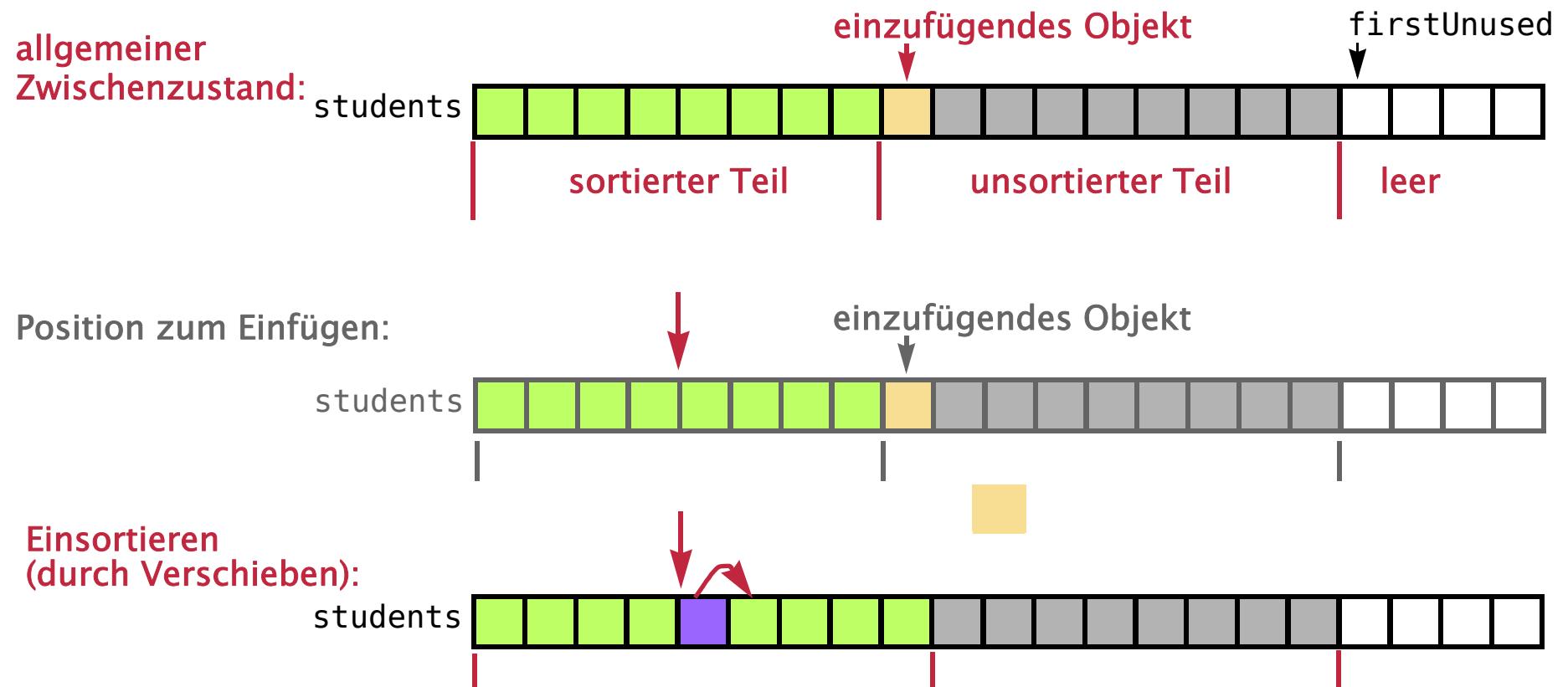


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

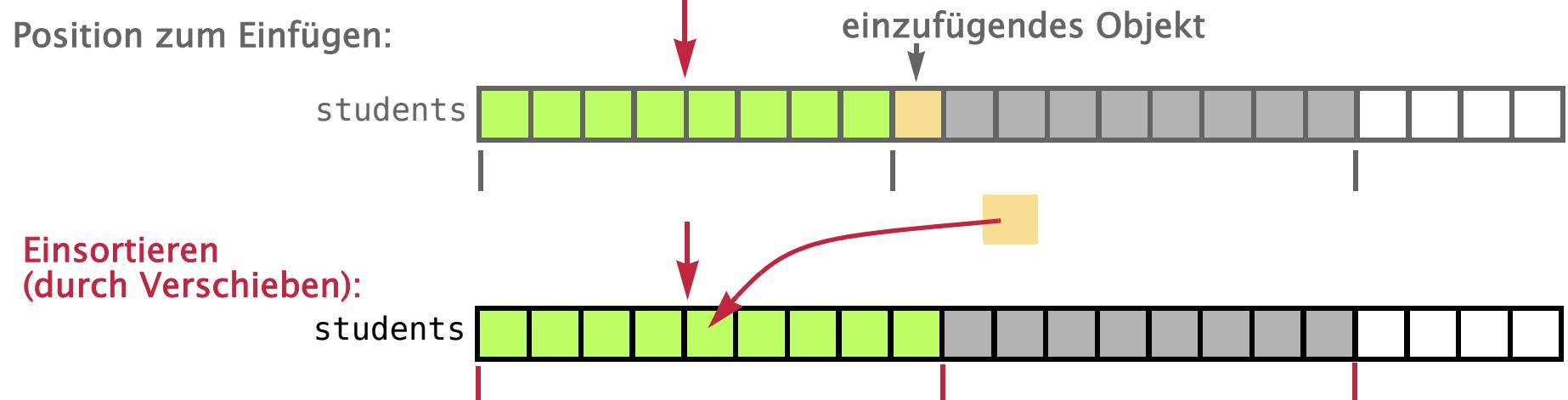
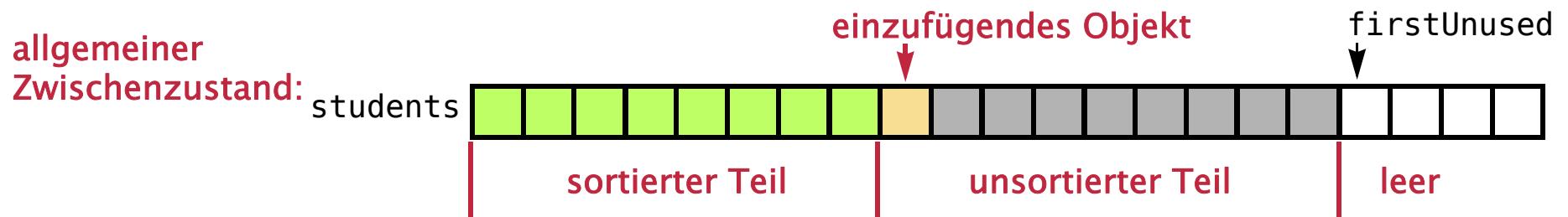


## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.



## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Implementierung *InsertionSort*:

- Es liegt die gleiche Ausgangssituation wie bei *SelectionSort* vor.

allgemeiner  
Zwischenzustand

Methode `void insertionSortByNumber()`

Position zum Einfügen

Einfügen  
(durch Verschieben)

Methode `shiftStudentsByNumber( int start )`

## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

```
public void insertionSortByNumber()
{
    for ( int i = 1; i < firstUnused; i++ )
    {
        shiftStudentsByNumber( i );
    }
}
```

- ❑ Die Implementierung setzt unmittelbar den geplanten Algorithmus um.

## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

```
private void shiftStudentsByNumber( int start )
{
    Student toInsert = students[start];
    int i = start;
    while ( i > 0 && students[i - 1].hasGreaterNumber( toInsert ) )
    {
        students[i] = students[i - 1];
        i--;
    }
    students[i] = toInsert;
}
```

- Die Implementierung setzt unmittelbar den geplanten Algorithmus um.

## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

```

private void shiftStudentsByNumber( int start )
{
    Student toInsert = students[start];
    int i = start;
    while ( i > 0 && students[i - 1].hasGreaterNumber( toInsert ) )
    {
        students[i] = students[i - 1];
        i--;
    }
    students[i] = toInsert;
}

```

UND-Operator  
(Short-Cut-Evaluation)

*Short-Cut-Evaluation* der logischen Operatoren `&&` und `||`:

- Die Auswertung des linken Operanden erfolgt *immer vor* der Auswertung des rechten Operanden.
- Die Auswertung *bricht ab*, sobald das Ergebnis der Auswertung feststeht:
  - Konjunktion mit `&&`-Operator: linker Operand ist `false`
  - Disjunktion mit `||`-Operator: linker Operand ist `true`
- daher im Beispiel:  
für `i==0` wird der dann fehlerhafte Zugriff `students[-1]` nicht mehr ausgeführt

## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

```

public void insertionSortByNumber()
{
    for ( int i = 1; i < firstUnused; i++ )
    {
        shiftStudentsByNumber( i );
    }
}

private void shiftStudentsByNumber( int start )
{
    Student toInsert = students[start];
    int i = start;
    while ( i > 0 && students[i - 1].hasGreaterNumber( toInsert ) )
    {
        students[i] = students[i - 1];
        i--;
    }
    students[i] = toInsert;
}

```

Analyse:

Wie häufig wird die Methode `hasGreaterNumber` aufgerufen, wenn  $n$  Einträge vorliegen?

maximal:

$$n-1 + n-2 + \dots + 1 = (n-1) \cdot n/2 = n^2/2 - n/2 \text{ -mal}$$

minimal:

$n-1$  -mal (falls die Einträge schon sortiert vorliegen)

durchschnittlich:

*dazwischen*

## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

```

public void insertionSortByNumber()
{
    for ( int i = 1; i < firstUnused; i++ )
    {
        shiftStudentsByNumber( i );
    }
}

private void shiftStudentsByNumber( int start )
{
    Student toInsert = students[start];
    int i = start;
    while ( i > 0 && students[i - 1].hasGreaterNumber( toInsert ) )
    {
        students[i] = students[i - 1];
        i--;
    }
    students[i] = toInsert;
}

```

Analyse:

Wie häufig wird umkopiert, wenn  $n$  Einträge vorliegen?

maximal:	$n-1 + n-2 + \dots + 1 = (n-1) \cdot n/2 = n^2/2 - n/2$ -mal
minimal:	0 -mal (falls Einträge bereits sortiert vorliegen)
durchschnittlich:	dazwischen

## Problemstellung «Sortieren durch Einfügen»

(Fortsetzung)

Variation: Sortieren anhand der Namen der Studierenden

```

private void shiftStudentsByNumber( int start )
{
  Student toInsert = students[start];
  int i = start;
  while ( i > 0 && students[i - 1].hasGreaterNumber( toInsert ) )
  {
    students[i] = students[i - 1];
    i--;
  }
  students[i] = toInsert;
}

```


  
 stoppt bei  
 Gleichheit

Anmerkung:

- Bei *InsertionSort* werden gleiche Objekte in der Reihenfolge eingesortiert, in der sie in der Ausgangsfolge auftreten:  
 Das Sortierverfahren verhält sich *stabil*.
- Studierende mit dem gleichen Namen werden also in ihrer Reihenfolge nicht geändert.

## Problemstellung «Sortieren der Studierenden einer Vorlesung»

Vergleich der Algorithmen *SelectionSort* und *InsertionSort*:

	<i>SelectionSort</i>	<i>InsertionSort</i>
bei $n$ zu sortierenden Einträgen:		
maximale Zahl an Vergleichen	$O(n^2)$ *)	$O(n^2)$
durchschnittliche Zahl an Vergleichen	$O(n^2)$	?
minimale Zahl an Vergleichen	$O(n^2)$	$=n-1$ also $O(n)$
maximale Zahl an Zuweisungen	$=n-1$ also $O(n)$	$O(n^2)$
durchschnittliche Zahl an Zuweisungen	$=n-1$ also $O(n)$	?
minimale Zahl an Zuweisungen	$=n-1$ also $O(n)$	0
Reihenfolge gleicher Elemente	<i>nicht stabil</i> **)	<i>stabil</i>
Speicherbedarf	<i>in situ</i> ***)	<i>in situ</i>

\*) Beispiel zur Verdeutlichung:  $n = 100\,000 \Rightarrow n^2 = 10\,000\,000\,000$

\*\*) Die Position der in den unsortierten Teil getauschten Objekte ist zufällig.

\*\*\*) *in situ* (lat.) = am (Ursprungs-)Ort, d.h. es wird in dem vorhandenen Feld sortiert

## Algorithmus «Sortieren durch Einfügen»

Idee des Sortierverfahrens «Sortieren durch Einfügen» (insertion sort)

- Während des Sortierens werden unterschieden:
  - die Menge  $M$  der noch nicht sortierten Einheiten
  - die Folge  $F$  der bereits sortierten Einheiten
- In jedem Sortierschritt
  - wird eine beliebige Einheit  $e$  aus  $M$  genommen,
  - für die die Position in  $F$  aufgrund ihrer Eigenschaften bestimmt wird.
- Einfach ist also immer das Bestimmen von  $e$ , während das Hinzufügen zu  $F$  aufwändig ist.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 4.3. Sortieralgorithmen - «Quicksort»

Dr. Stefan Dissmann  
Fakultät für Informatik



## Beschleunigung des Sortierens

Feststellung:

Wenn die Zahl der Vergleiche/Zuweisungen sich quadratisch bezüglich der Anzahl der zu sortierenden Elemente verhält, dann wird das Sortieren großer Datenmengen sehr aufwändig.

Überlegung:

Man könnte den Aufwand reduzieren, wenn man die zu sortierenden Datenmengen geschickt aufteilt, die Teile sortiert und die sortierten Teile zusammenfügt.

Eine grobe Abschätzung, die die Idee verdeutlichen soll:

Das Sortieren von  $n$  Einträgen erfordert etwa  $n^2/2$  Operationen.

Das Sortieren von 2 Teilen mit je  $n/2$  Einträgen erfordert  $2 \cdot (n/2)^2/2 = n^2/4$  Operationen.

Das Sortieren von 4 Teilen mit je  $n/4$  Einträgen erfordert  $4 \cdot (n/4)^2/2 = n^2/8$  Operationen.

...

Voraussetzung:

Das Aufteilen und das Zusammenfügen darf selbst nicht aufwändig sein.

## Sortieren durch «QuickSort»

Ziel:

Das Sortieren soll folgendermaßen realisiert werden:

- die zu sortierenden Elemente werden in eine Menge mit den kleineren und eine Menge mit den größeren Elementen aufgeteilt,
- für beide Mengen wird dieses Verfahren solange erneut angewendet, bis die zu sortierenden Teilmengen nur noch aus einem Element bestehen (und damit sortiert sind) und
- alle Teilmengen werden so zusammengesetzt, dass die sortierten kleineren vor den sortierten größeren Elementen stehen.

Diese Art des Sortierens heißt *QuickSort*, da so tatsächlich relativ schnell sortiert werden kann. \*)

\*) Grundidee zu QuickSort stammt von C. Tony R. Hoare, 1962

## Sortieren durch «QuickSort»

(Fortsetzung)

### Beispiel

Sei diese Folge von Matrikelnummern von Studierenden gegeben:

28 19 4 12 17 3 9 11 1 7

Daraus entstehen durch Aufteilen in kleine und große Werte:

4 3 9 1 7 und 28 19 12 17 11

und durch weiteres Aufteilen:

4 3 1 und 9 7 und 12 17 11 und 28 19

und:

1 | 4 3 | 7 | 9 | 12 11 | 17 | 19 | 28

und:

1 | 3 | 4 | 7 | 9 | 11 | 12 | 17 | 19 | 28

## Sortieren durch «QuickSort»

(Fortsetzung)

Anmerkungen:

- Das Beispiel zeigt:  
Sortieren ist möglich durch ein wiederholtes Aufteilen  
in Folgen mit größeren und kleineren Elementen.
- *Problem*: Voraussetzung für das Bestimmen gleichmächtiger Folgen ist eine  
vorliegende Sortierung (– die erst erzeugt werden soll).
- *Lösung*: Es wird ein geeignetes *Pivot*-Element  $p$  ausgewählt  
anhand dessen eine Folge zerlegt wird in  
eine *Folge* mit Elementen, die kleiner als  $p$  sind, und  
eine *Folge* von Elementen, die größer oder gleich  $p$  sind.
- Das Zusammenfügen der Teile ist einfach,  
wenn der Algorithmus innerhalb eines Feldes arbeitet:  
Die Teile sind nur Abschnitte des Feldes,  
in denen verschoben wird,  
in ihrer Gesamtheit aber immer zusammen bleiben.

Methode  
`groupByNumber`

## Sortieren durch «QuickSort»

(Fortsetzung)

Anmerkungen:

- Es wird ein geeignetes *Pivot*-Element  $p$  ausgewählt anhand dessen eine Folge zerlegt wird in eine *Folge* mit Elementen, die kleiner als  $p$  sind, und eine *Folge* von Elementen, die größer oder gleich  $p$  sind.
- Möglichkeiten zur Auswahl des Pivot-Elements sind zum Beispiel:
  - ein zufällig gewähltes Element
  - ein Element an immer der gleichen festen Position
  - mehrere Elemente, von denen das mit dem mittleren Wert gewählt wird.

## Sortieren durch «QuickSort»

(Fortsetzung)

Entscheidungen für die hier folgende Implementierung:

- Auswahl des Pivot-Elements:  
Das Element an der letzten Position des betrachteten Abschnitts des Feldes wird gewählt.
- Zuordnung des Pivot-Elements:  
Das Pivot-Element wird in den Teil mit den größeren Elementen eingeordnet.
- Die Implementierung folgt damit der *Partitionierung* nach *Nico Lomuto*.

## Sortieren durch «QuickSort»

(Fortsetzung)

```
private void groupByNumber( int leftBound, int rightBound ) {
```

- anhand des Pivot-Elements in Teilfolgen zerlegen
- Algorithmus auf Teilfolgen anwenden

Grenzen der Ausgangsfolge

```
}
```

- Die Parameter geben den kleinsten (`leftBound`) und den größten (`rightBound`) Index eines Ausschnitts des Feldes `students` an, der mit der Methode bearbeitet werden soll.
- Die Methode ist privat, da außerhalb der Klasse `Lecture` das Feld unbekannt ist und damit eine Angabe von Indizes sinnvoll nicht erfolgen kann.

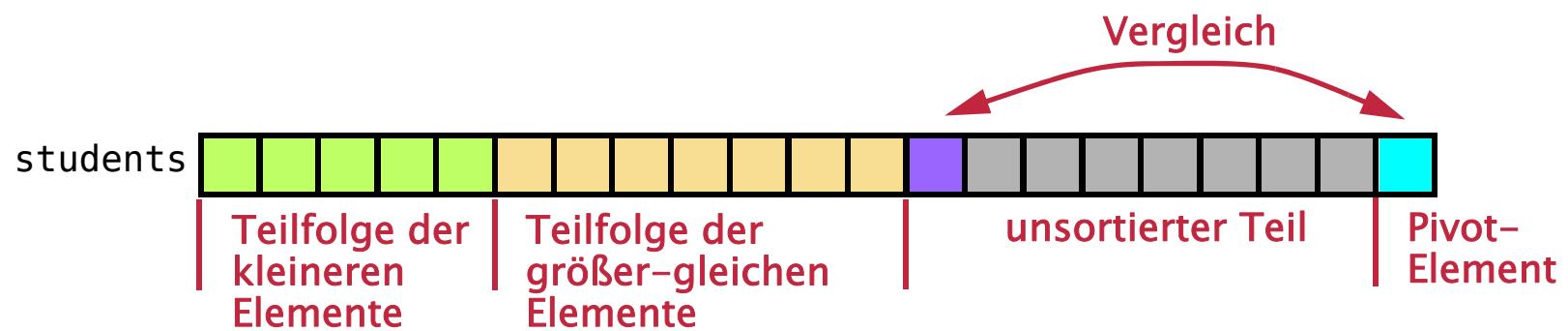
## Sortieren durch «QuickSort»

(Fortsetzung)

```
private void groupByNumber( int leftBound, int rightBound )
{
    if ( leftBound < rightBound ) ←
    {
        - anhand des Pivot-Elements in Teilstufen zerlegen
        - Algorithmus auf Teilstufen anwenden
    }
}
```

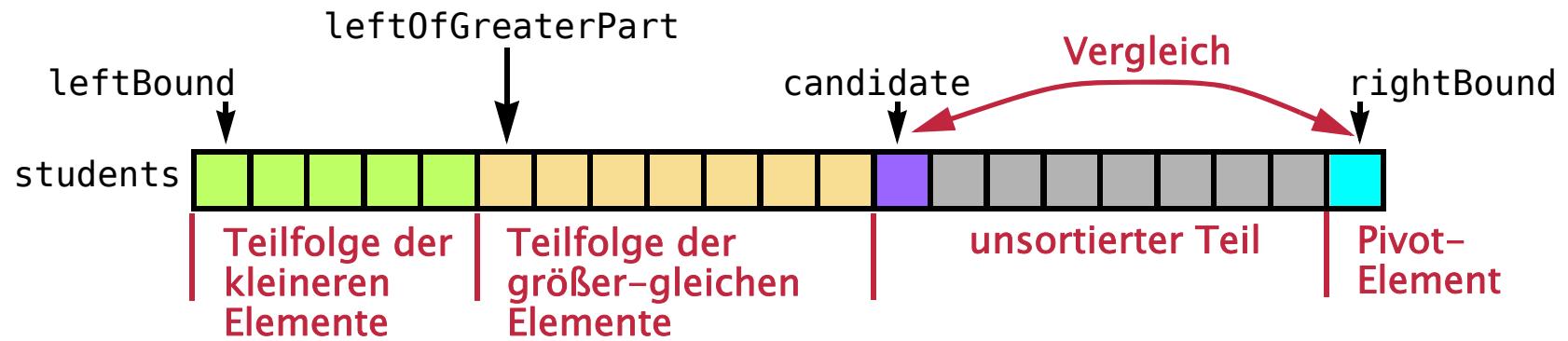
mindestens zwei  
Elemente müssen  
sortiert werden

- Die Idee für das Zerlegen in Teilstufen ist:

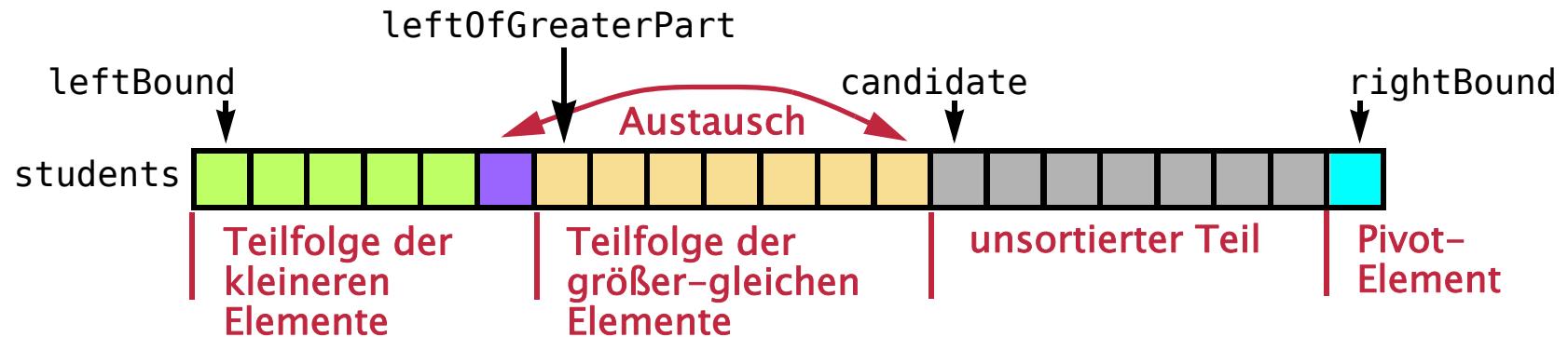


## Sortieren durch «QuickSort»

(Fortsetzung)

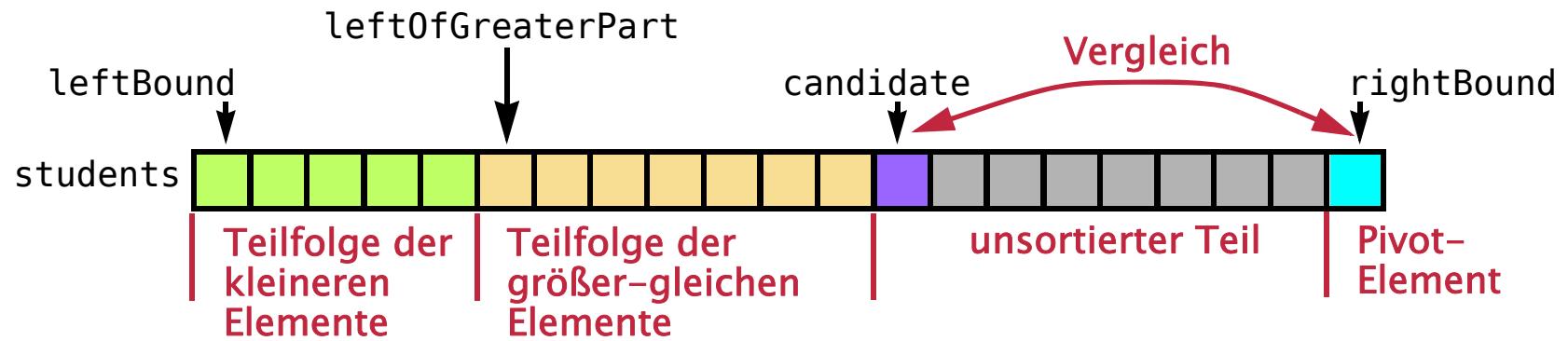


- ❑ falls `students[candidate]` *kleiner als* `students[rightBound]` ist, muss die Teilfolge der kleineren Elemente verlängert werden:

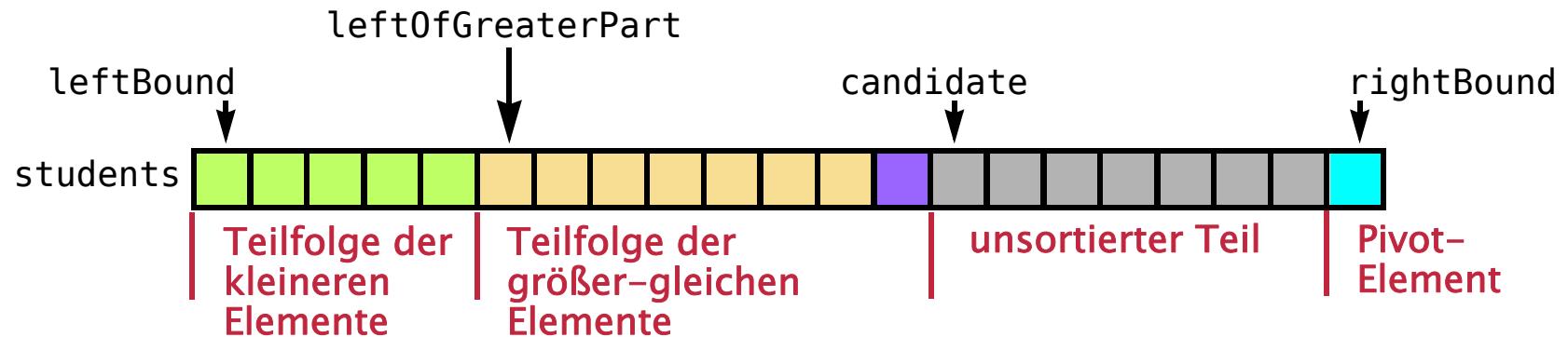


## Sortieren durch «QuickSort»

(Fortsetzung)



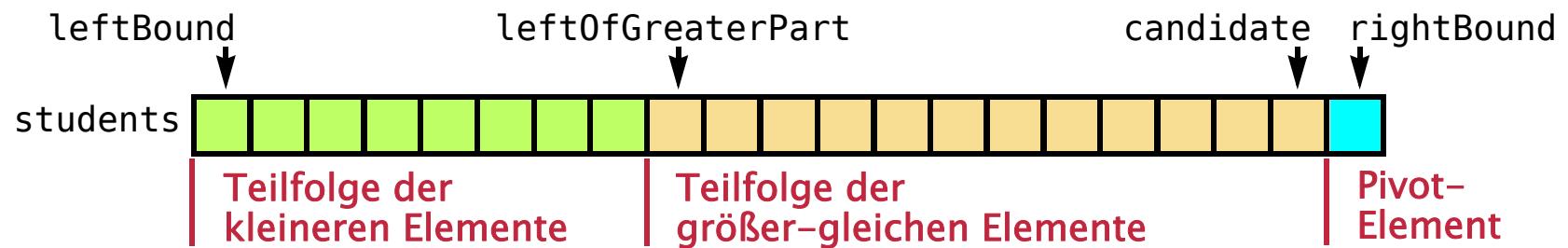
- ❑ falls `students[candidate]` *größer als oder gleich* `students[rightBound]` ist, muss die Teilfolge der größer-gleichen Elemente verlängert werden:



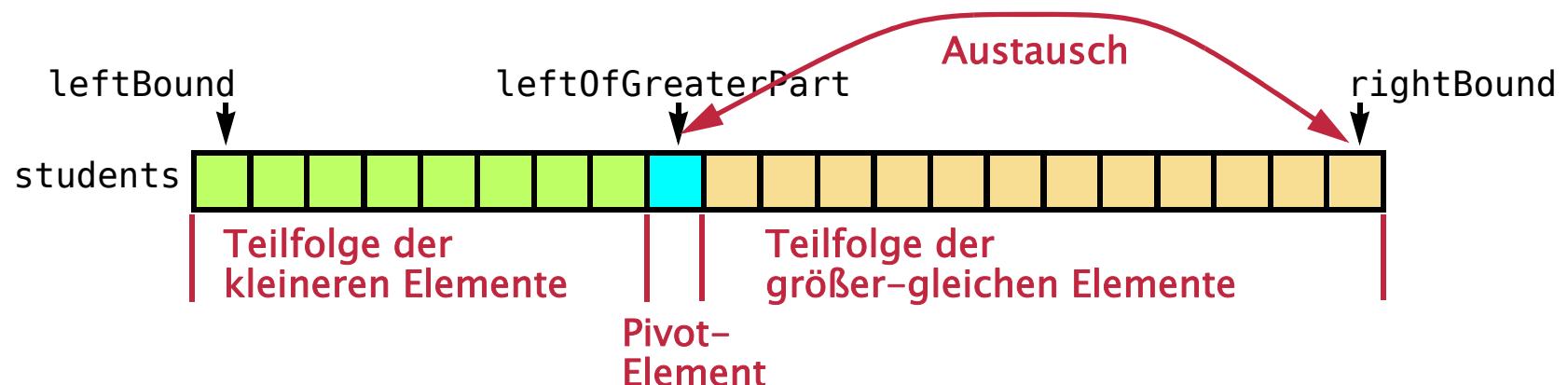
## Sortieren durch «QuickSort»

(Fortsetzung)

- Endsituation der Aufteilung:



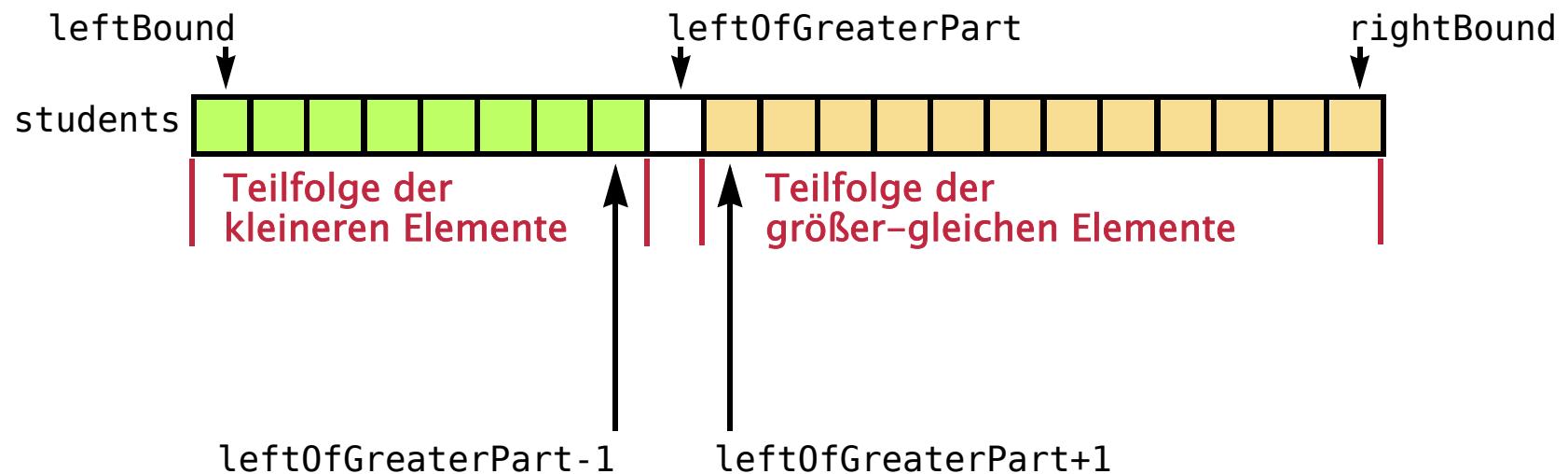
- Das Pivot-Element kann durch Austausch eingesortiert werden, da das Pivot-Element immer der kleinste Wert in der Teilfolge der größer-gleichen Elemente ist:



## Sortieren durch «QuickSort»

(Fortsetzung)

- Fortsetzung:



- Algorithmus anwenden auf den Bereich `leftBound ... leftOfGreaterPart-1`
- Algorithmus anwenden auf den Bereich `leftOfGreaterPart+1 ... rightBound`
- Das Pivot-Element liegt an der Position `leftOfGreaterPart` genau an der Position, an der es in der sortierten Folge stehen muss. Es muss daher nicht weiter betrachtet werden.

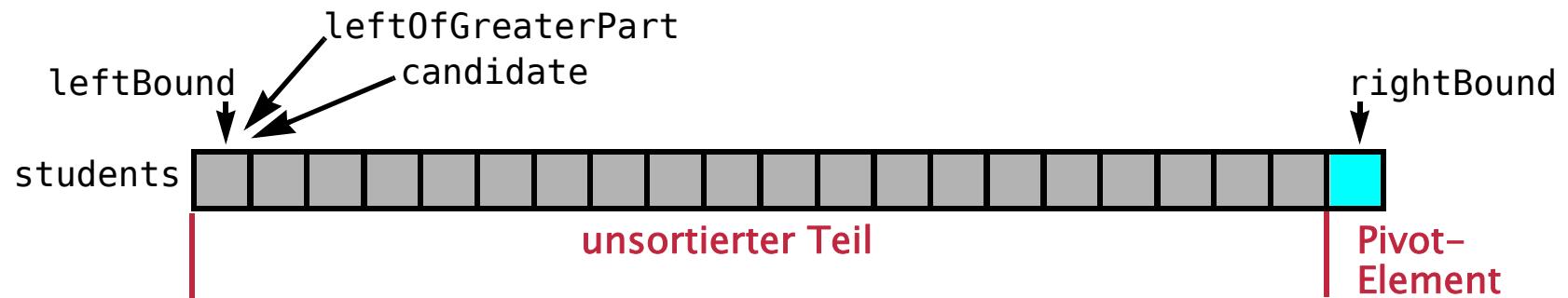
## Sortieren durch «QuickSort»

(Fortsetzung)

```
private void groupByNumber( int leftBound, int rightBound )
{
    if ( leftBound < rightBound ) ←
    {
        int candidate = leftBound;
        int leftOfGreaterPart = leftBound;

        - anhand des Pivot-Elements in Teilstücke zerlegen
        - Algorithmus auf Teilstücke anwenden
    }
}
```

mindestens zwei  
Elemente müssen  
sortiert werden



## Sortieren durch «QuickSort»

(Fortsetzung)

Beispiel für die Zerlegung der bekannten Folge von Matrikelnummern von Studierenden gegeben:

28 19 4 12 17 3 9 11 1 7

Dann wird das Student-Objekt mit der Matrikelnummer 7 als Pivot-Element genutzt.

Nun wird gesucht:



## Sortieren durch «QuickSort»

(Fortsetzung)

```
private void groupByNumber( int leftBound, int rightBound )
{
    if (leftBound < rightBound )
    {
        int leftOfGreaterPart = leftBound;
        for ( int candidate = leftBound; candidate < rightBound; candidate++ )
        {
            if ( students[rightBound].hasGreaterNumber( students[candidate] ) )
            {
                swapStudents( candidate, leftOfGreaterPart );
                leftOfGreaterPart++;
            }
        }

        swapStudents( leftOfGreaterPart, rightBound );

        groupByNumber( leftBound, leftOfGreaterPart - 1 );
        groupByNumber( leftOfGreaterPart + 1 , rightBound );
    }
}
```

es müssen noch Elemente verteilt werden

## Sortieren durch «QuickSort»

(Fortsetzung)

```

private void groupByNumber( int leftBound, int rightBound )
{
    if (leftBound < rightBound )
    {
        int leftOfGreaterPart = leftBound;

        for ( int candidate = leftBound; candidate < rightBound; candidate++ )
        {
            if ( students[rightBound].hasGreaterNumber( students[candidate] ) )
            {
                swapStudents( candidate, leftOfGreaterPart );
                leftOfGreaterPart++;
            }
        }

        swapStudents( leftOfGreaterPart, rightBound );

        groupByNumber( leftBound, leftOfGreaterPart - 1 );
        groupByNumber( leftOfGreaterPart + 1 , rightBound );
    }
}

```

**betrachtetes Element ist kleiner als das Pivot-Element**

**Vertauschen in den Teil der kleineren Elemente**

## Sortieren durch «QuickSort»

(Fortsetzung)

```

private void groupByNumber( int leftBound, int rightBound )
{
  if (leftBound < rightBound )
  {
    int leftOfGreaterPart = leftBound;

    for ( int candidate = leftBound; candidate < rightBound; candidate++ )
    {
      if ( students[rightBound].hasGreaterNumber( students[candidate] ) )
      {
        swapStudents( candidate, leftOfGreaterPart );
        leftOfGreaterPart++;
      }
    }

    swapStudents( leftOfGreaterPart, rightBound );

    groupByNumber( leftBound, leftOfGreaterPart - 1 );
    groupByNumber( leftOfGreaterPart + 1 , rightBound );
  }
}

```

Pivot-Element wird zwischen die Teile verschoben



## Sortieren durch «QuickSort»

(Fortsetzung)

```

private void groupByNumber( int leftBound, int rightBound )
{
    if (leftBound < rightBound )
    {
        int leftOfGreaterPart = leftBound;

        for ( int candidate = leftBound; candidate < rightBound; candidate++ )
        {
            if ( students[rightBound].hasGreaterNumber( students[candidate] ) )
            {
                swapStudents( candidate, leftOfGreaterPart );
                leftOfGreaterPart++;
            }
        }
        swapStudents( leftOfGreaterPart, rightBound );
Pivot-Element wird zwischen die Teile verschoben
eventuell gilt
leftOfGreaterPart==rightBound:
Dann wird nicht getauscht.
        groupByNumber( leftBound, leftOfGreaterPart - 1 );
        groupByNumber( leftOfGreaterPart + 1 , rightBound );
    }
}

```

## Sortieren durch «QuickSort»

(Fortsetzung)

```

private void groupByNumber( int leftBound, int rightBound )
{
    if (leftBound < rightBound )
    {
        int leftOfGreaterPart = leftBound;

        for ( int candidate = leftBound; candidate < rightBound; candidate++ )
        {
            if ( students[rightBound].hasGreaterNumber( students[candidate] ) )
            {
                swapStudents( candidate, leftOfGreaterPart );
                leftOfGreaterPart++;
            }
        }

        swapStudents( leftOfGreaterPart, rightBound );

        groupByNumber( leftBound, leftOfGreaterPart - 1 );
        groupByNumber( leftOfGreaterPart + 1 , rightBound );
    }
}

```

Algoritmus auf  
Teilfolgen anwenden

## Sortieren durch «QuickSort»

(Fortsetzung)

```
private void groupByNumber( int leftBound, int rightBound )
{
    if (leftBound < rightBound )
    {
        int leftOfGreaterPart = leftBound;

        for ( int candidate = leftBound; candidate < rightBound; candidate++ )
        {
            if ( students[rightBound].hasGreaterNumber( students[candidate] ) )
            {
                swapStudents( candidate, leftOfGreaterPart );
                leftOfGreaterPart++;
            }
        }

        swapStudents( leftOfGreaterPart, rightBound );

        groupByNumber( leftBound, leftOfGreaterPart - 1 );    «kleine Matrikelnummern»
        groupByNumber( leftOfGreaterPart + 1 , rightBound ); «große Matrikelnummern»
    }
}
```

## Sortieren durch «QuickSort»

(Fortsetzung)

Aufruf der Methode                                   groupByNumber  
im Rumpf der (eigenen) Deklaration von   groupByNumber

### Geht das überhaupt?

**Ja, weil**

- für jeden Aufruf eigene Speicherbereiche für die Parameter `leftBound` und `rightBound` angelegt werden und
- für jeden Aufruf eigene Speicherbereiche für die lokalen Variablen `leftOfGreaterPart` und `candidate` angelegt werden.

Das Konzept des Selbstaufrufs einer Methode heißt **Rekursion**.

## Sortieren durch «QuickSort»

(Fortsetzung)

### *Beispiel – Überblick:*

Studierende einer Vorlesung (unsortiert) im Feld `students`:

```
student: A, registration number: 11(Inf)
student: C, registration number: 3(M)
student: B, registration number: 14(Inf)
student: B, registration number: 8(M)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
```

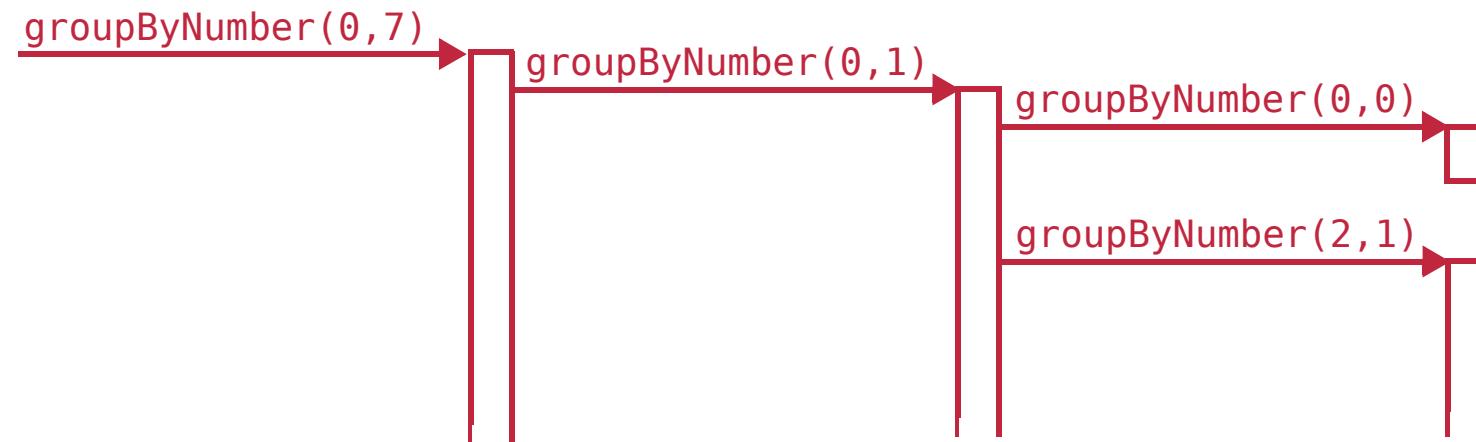
## Sortieren durch «QuickSort»

(Fortsetzung)

### Beispiel – Überblick:

Studierende einer Vorlesung (unsortiert) im Feld `students`:

```
student: A, registration number: 11(Inf)
student: C, registration number: 3(M)
student: B, registration number: 14(Inf)
student: B, registration number: 8(M)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
```



## Sortieren durch «QuickSort»

(Fortsetzung)

### *Beispiel – Überblick:*

Studierende einer Vorlesung (unsortiert) im Feld `students`:

```
student: A, registration number: 11(Inf)
student: C, registration number: 3(M)
student: B, registration number: 14(Inf)
student: B, registration number: 8(M)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
```

Aufrufe:

```
groupByNumber( 0, 7 )
    groupByNumber( 0, 1 )
        groupByNumber( 0, 0 )
        groupByNumber( 2, 1 )
    groupByNumber( 3, 7 )
        groupByNumber( 3, 4 )
            groupByNumber( 3, 3 )
            groupByNumber( 5, 4 )
        groupByNumber( 6, 7 )
            groupByNumber( 6, 5 )
            groupByNumber( 7, 7 )
```

## Sortieren durch «QuickSort»

(Fortsetzung)

*Beispiel – 1. Aufruf:* groupByNumber( 0, 7 )

Studierende einer Vorlesung im Feld `students`:

```
student: A, registration number: 11(Inf)
student: C, registration number: 3(M)
student: B, registration number: 14(Inf)
student: B, registration number: 8(M)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
```

← Pivot-Element

Aufruf von `groupByNumber( 0, 7 )` führt zu:

```
student: C, registration number: 3(M)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
student: B, registration number: 8(M)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
student: A, registration number: 11(Inf)
student: B, registration number: 14(Inf)
```

## Sortieren durch «QuickSort»

(Fortsetzung)

*Beispiel – 2. Aufruf:* groupByNumber( 0, 1 )

Studierende einer Vorlesung im Feld `students`:

```
student: C, registration number: 3(M)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
student: B, registration number: 8(M)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
student: A, registration number: 11(Inf)
student: B, registration number: 14(Inf)
```

← Pivot-Element  
← richtig positioniert

Aufruf von `groupByNumber( 0, 1 )` führt zu:

```
student: C, registration number: 3(M)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
student: B, registration number: 8(M)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
student: A, registration number: 11(Inf)
student: B, registration number: 14(Inf)
```

## Sortieren durch «QuickSort»

(Fortsetzung)

*Beispiel – 5. Aufruf: groupByNumber( 3, 7 ) \**

Studierende einer Vorlesung im Feld `students`:

```
student: C, registration number: 3(M)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
student: B, registration number: 8(M)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
student: A, registration number: 11(Inf)
student: B, registration number: 14(Inf)
```

← richtig positioniert  
← richtig positioniert  
← richtig positioniert  
← Pivot-Element

Aufruf von `groupByNumber( 3, 7 )` führt zu:

```
student: C, registration number: 3(M)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
student: B, registration number: 8(M)
student: A, registration number: 11(Inf)
student: B, registration number: 14(Inf)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
```

\*) Die Aufrufe `groupByNumber( 0, 0 )` und `groupByNumber( 2, 1 )` führen zu keinen Änderungen.

## Sortieren durch «QuickSort»

(Fortsetzung)

*Beispiel – 6. Aufruf:* groupByNumber( 3, 4 )

Studierende einer Vorlesung im Feld `students`:

```
student: C, registration number: 3(M)          ← richtig positioniert
student: F, registration number: 4(Inf)         ← richtig positioniert
student: E, registration number: 7(Inf)         ← richtig positioniert
student: B, registration number: 8(M)           ← Pivot-Element
student: A, registration number: 11(Inf)          ← richtig positioniert
student: B, registration number: 14(Inf)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
```

Aufruf von `groupByNumber( 3, 4 )` führt zu:

```
student: C, registration number: 3(M)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
student: B, registration number: 8(M)           ← Pivot-Element
student: A, registration number: 11(Inf)
student: B, registration number: 14(Inf)
student: B, registration number: 22(Inf)
student: D, registration number: 19(Ph)
```

## Sortieren durch «QuickSort»

(Fortsetzung)

*Beispiel – 9. Aufruf: groupByNumber( 6, 7 ) \*)*

Studierende einer Vorlesung im Feld `students`:

```
student: C, registration number: 3(M) ← richtig positioniert
student: F, registration number: 4(Inf) ← richtig positioniert
student: E, registration number: 7(Inf) ← richtig positioniert
student: B, registration number: 8(M) ← richtig positioniert
student: A, registration number: 11(Inf) ← richtig positioniert
student: B, registration number: 14(Inf) ← richtig positioniert
student: B, registration number: 22(Inf) ← richtig positioniert
student: D, registration number: 19(Ph) ← Pivot-Element
```

Aufruf von `groupByNumber( 6, 7 )` führt zu:

```
student: C, registration number: 3(M)
student: F, registration number: 4(Inf)
student: E, registration number: 7(Inf)
student: B, registration number: 8(M)
student: A, registration number: 11(Inf)
student: B, registration number: 14(Inf)
student: D, registration number: 19(Inf)
student: B, registration number: 22(Ph)
```

\*) Der 7. und 8. Aufruf führen nicht zu Änderungen.

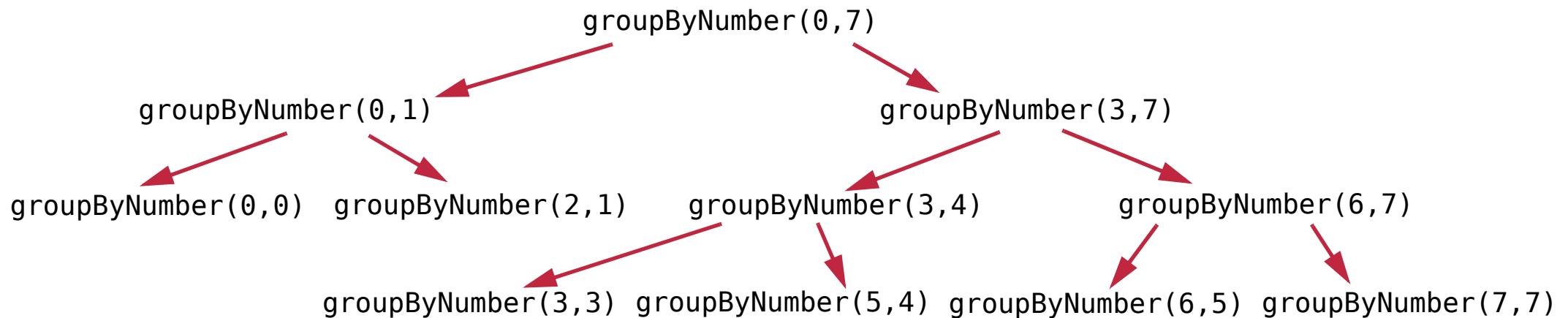
## Sortieren durch «QuickSort»

(Fortsetzung)

*Beispiel:*

Analyse des Beispieldurchlaufs:

- ❑ 11 Aufrufe, davon 5 mit Bestimmung eines Pivot-Elements
- ❑ 6 Vertauschungen
- ❑ Aufrufbaum:



## Sortieren durch «QuickSort»

(Fortsetzung)

Wie kann eine Analyse eines konkreten Durchlaufes erfolgen?

Durch geeignete Instrumentierung des Programmcodes:

- Einbau von Testausgaben
- Einbau von Zählern

```
private int countCalls;  
  
...  
  
private void groupByNumber( int leftBound, int rightBound )  
{  
    countCalls++;  
    System.out.println( "groupByNumber( " + leftBound + ", " + rightBound + " )" );  
    ...  
}
```

## Sortieren durch «QuickSort»

(Fortsetzung)

Analyse des Algorithmus:

- Laufzeit – Sonderfall aufsteigend sortierte Eingabe:

Es wird *in jedem rekursiven Aufruf* das größte Element als Pivot-Element gewählt und daher *in jedem rekursiven Aufruf* nur ein Element abgetrennt.

Dabei müssen aber alle restlichen Elemente überprüft – also mit dem Pivot-Element verglichen – werden.

Bei  $n$  zu sortierenden Einträgen ergibt sich für Vergleiche die Größenordnung:  $\approx n^2$

Es müssen aber keine Elemente vertauscht werden, da die Sortierung schon vorliegt.

Bei  $n$  zu sortierenden Einträgen ergibt sich für Vertauschungen: 0

## Sortieren durch «QuickSort»

(Fortsetzung)

Analyse des Algorithmus:

- Laufzeit – Sonderfall aufsteigend sortierte Eingabe:

Es wird *in jedem rekursiven Aufruf* das größte Element als Pivot-Element gewählt und daher *in jedem rekursiven Aufruf* nur ein Element abgetrennt. Dabei müssen alle restlichen Elemente überprüft – also mit dem Pivot-Element verglichen – werden.

Bei  $n$  zu sortierenden Elementen ergibt sich für Aufrufe die Größenordnung:  $\approx 2 \cdot n$

Bei  $n$  zu sortierenden Elementen ergibt sich für Vergleiche die Größenordnung:  $\approx n^2$

Es müssen aber keine Elemente vertauscht werden, da die Sortierung schon vorliegt.

Bei  $n$  zu sortierenden Elementen ergibt sich für Vertauschungen: 0

- Laufzeit – Sonderfall absteigend sortierte Eingabe:

Es wird *in den rekursiven Aufrufen* abwechseln das kleinste und das größte Element als Pivot-Element gewählt und daher *in jedem rekursiven Aufruf* nur ein Element abgetrennt. Dabei müssen alle restlichen Elemente überprüft – also mit dem Pivot-Element verglichen – werden.

Bei  $n$  zu sortierenden Elementen ergibt sich für Aufrufe die Größenordnung:  $\approx 2 \cdot n$

Bei  $n$  zu sortierenden Elementen ergibt sich für Vergleiche die Größenordnung:  $\approx n^2$

Die Elemente werden in jedem zweiten Aufruf getauscht und drehen die Sortierung um.

Bei  $n$  zu sortierenden Elementen ergibt sich für Vertauschungen:  $n/2$

## Sortieren durch «QuickSort»

(Fortsetzung)

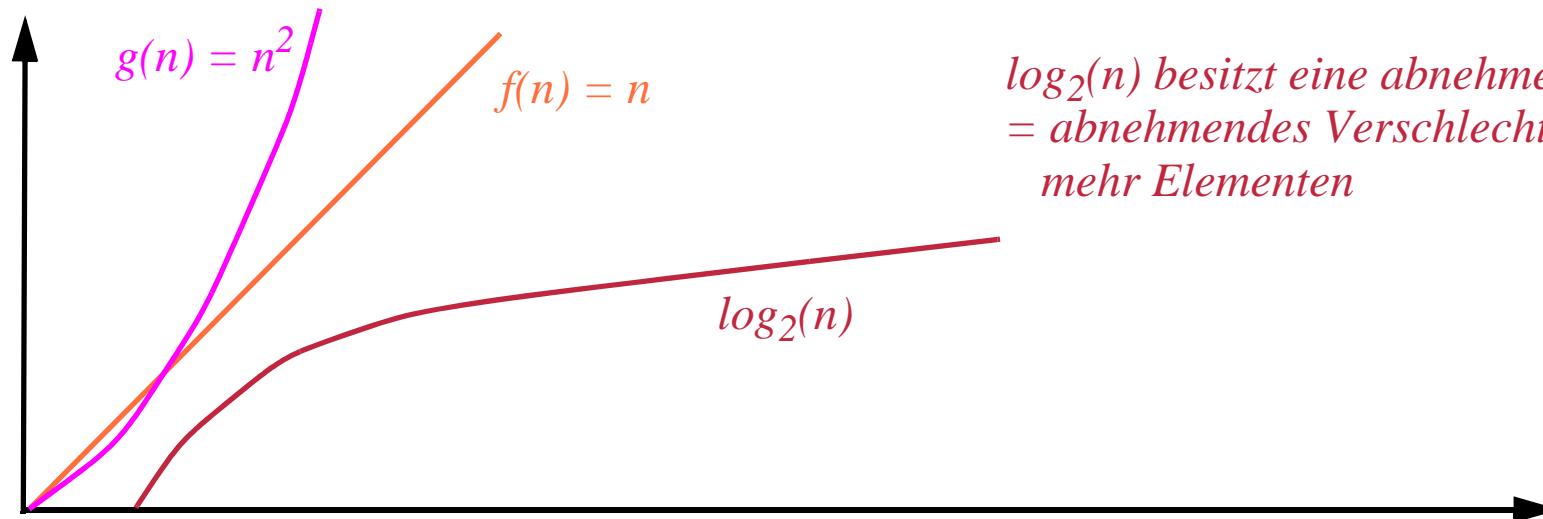
Analyse des Algorithmus:

- Laufzeit – bestmöglicher Fall:

Die zu betrachtende Folge wird durch das Pivot-Element *immer* genau halbiert.  
Dann hängt die Zahl der Aufrufe der Methode `groupByNumber` davon ab,  
wie häufig sich  $n$  durch 2 teilen lässt:

$$2^x \approx n$$

oder  $x = \log_2(n)$  (*Logarithmus zur Basis 2*)



## Sortieren durch «QuickSort»

(Fortsetzung)

Analyse des Algorithmus:

- Laufzeit – bestmöglicher Fall:

Die zu betrachtende Folge wird durch das Pivot-Element *immer* genau halbiert.

Bei  $n$  zu sortierenden Elementen ergibt sich für Aufrufe die Größenordnung:  $\approx \log_2(n)$

In jedem der  $\log_2(n)$  Aufrufe der Methode `groupByNumber` wird ein Ausschnitt der  $n$  zu sortierenden Elemente mit dem Pivot-Element verglichen.

Es ergibt sich für Vergleiche die Größenordnung:  $\approx n \cdot \log_2(n)$

Da Vertauschungen eine Konsequenz der Vergleiche sind, treten nie mehr Vertauschungen auf.

Es ergibt sich für Vertauschungen ebenfalls die Größenordnung:  $\approx n \cdot \log_2(n)$

## Sortieren durch «QuickSort»

(Fortsetzung)

Analyse des Algorithmus:

- Laufzeit – bestmöglicher Fall:

Die zu betrachtende Folge wird durch das Pivot-Element *immer* genau halbiert.

Bei  $n$  zu sortierenden Elementen ergibt sich für Aufrufe die Größenordnung:  $\approx \log_2(n)$

In jedem der  $\log_2(n)$  Aufrufe der Methode `groupByNumber` wird ein Ausschnitt der  $n$  zu sortierenden Elemente mit dem Pivot-Element verglichen.

Es ergibt sich für Vergleiche die Größenordnung:  $\approx n \cdot \log_2(n)$

Da Vertauschungen eine Konsequenz der Vergleiche sind, treten nie mehr Vertauschungen auf.

Es ergibt sich für Vertauschungen ebenfalls die Größenordnung:  $\approx n \cdot \log_2(n)$

- Laufzeit – durchschnittlicher Wert:

$O(n \cdot \log_2(n))$  (kann hier aber noch nicht bewiesen werden)

- *QuickSort* kann also deutlich schneller als *InsertionSort* und *SelectionSort* arbeiten. \*)

\*) Für  $n=100\,000$  ist  $n^2=10\,000\,000\,000$ , aber  $n \cdot \log_2(n) \leq 1\,700\,000$  (ganzzahlig).

## Sortieren durch «QuickSort»

(Fortsetzung)

Analyse des Algorithmus:

- Laufzeit – bestmöglicher Fall:

Die zu betrachtende Folge wird durch das Pivot-Element *immer* genau halbiert.

Bei  $n$  zu sortierenden Elementen ergibt sich für Aufrufe die Größenordnung:  $\approx \log_2(n)$

In jedem der  $\log_2(n)$  Aufrufe der Methode `groupByNumber` wird ein Ausschnitt der  $n$  zu sortierenden Elemente mit dem Pivot-Element verglichen.

Es ergibt sich für Vergleiche die Größenordnung:  $\approx n \cdot \log_2(n)$

Da Vertauschungen eine Konsequenz der Vergleiche sind, treten nie mehr Vertauschungen auf.

Es ergibt sich für Vertauschungen ebenfalls die Größenordnung:  $\approx n \cdot \log_2(n)$

- Laufzeit – durchschnittlicher Wert:

$O( n \cdot \log_2(n) )$  (kann hier aber noch nicht bewiesen werden)

- *QuickSort* kann also deutlich schneller als *InsertionSort* und *SelectionSort* arbeiten. \*)
- *QuickSort* ist *nicht stabil*, da die Elemente abhängig von ihrem Vorkommen getauscht werden.
- *QuickSort* arbeitet *in situ*, benötigt aber neben dem Feld den Speicherplatz für die lokalen Variablen und Parameter der gleichzeitig ausgeführten Instanzen der rekursiven Methode.

\*) Für  $n=100\,000$  ist  $n^2=10\,000\,000\,000$ , aber  $n \cdot \log_2(n) \leq 1\,700\,000$  (ganzzahlig).

## Algorithmus «QuickSort»

### Idee des Sortierverfahrens «QuickSort»

- Die zu sortierenden Elemente werden in zwei Folgen aufgeteilt
  - bezüglich ihrer Eigenschaften und
  - bezüglich einer Referenzeinheit  $p$  («Pivot-Element»), so dass die bezüglich  $p$  kleineren und die bezüglich  $p$  größeren Elemente jeweils eine Folge bilden.
- Auf die beiden Folgen wird das Verfahren immer wieder angewandt, bis Folgen der Länge 1 entstehen.
- Der gleiche Vorgang wird also immer wieder auf immer kleiner werdenden Folgen ausgeführt.

## Sortieren durch «QuickSort»

(Fortsetzung)

Aufruf für Objekte der Klasse `Lecture`

- ❑ Methode ist privat:  
`private void groupByNumber( int leftBound, int rightBound )`
- ❑ Sinnvoll, da die Parameterwerte ja auch abhängig sind von der Größe und der Belegung des Feldes `students` und dieses ebenfalls privat deklariert ist.
- ❑ Daher wird für eine Lösung, die ein einfaches Sortieren der Studierenden einer Vorlesung erlaubt, eine zusätzliche öffentliche Methode benötigt, die den ersten Aufruf von `groupByNumber` übernimmt.

```
public void quickSortByNumber()
{
    groupByNumber( 0, firstUnused - 1 );
}
```



berechnet das letzte benutzte Element in `students`

## Sortieren durch «QuickSort»

(Fortsetzung)

analoge Methode: quickSortByName

```
sorted by number:  
lecture: DAP 1  
participants:  
student: C, registration number: 3(M)  
student: F, registration number: 4(Inf)  
student: E, registration number: 7(Inf)  
student: B, registration number: 8(M)  
student: A, registration number: 11(Inf)  
student: B, registration number: 14(Inf)  
student: D, registration number: 19(Ph)  
student: B, registration number: 22(Inf)
```

```
sorted by name:  
lecture: DAP 1  
participants:  
student: A, registration number: 11(Inf)  
student: B, registration number: 22(Inf)  
student: B, registration number: 14(Inf)  
student: B, registration number: 8(M)  
student: C, registration number: 3(M)  
student: D, registration number: 19(Ph)  
student: E, registration number: 7(Inf)  
student: F, registration number: 4(Inf)
```

arbeitet auch mit  
mehrfachen Einträgen  
korrekt  
(aber nicht stabil)

## Sortieren durch «QuickSort»

Sortierung nach Namen

(Fortsetzung)

```
public void quickSortByName()
{
    groupByName( 0, firstUnused - 1 );
}

private void groupByName( int leftBound, int rightBound )
{
    if (leftBound < rightBound )
    {
        int leftOfGreaterPart = leftBound;
        for ( int candidate = leftBound; candidate < rightBound; candidate++ )
        {
            if ( students[rightBound].hasGreaterName( students[candidate] ) )
            {
                swapStudents( candidate, leftOfGreaterPart );
                leftOfGreaterPart++;
            }
        }
        swapStudents( leftOfGreaterPart, rightBound );
        groupByName( leftBound, leftOfGreaterPart - 1 );
        groupByName( leftOfGreaterPart + 1 , rightBound );
    }
}
```

- ❑ Die Algorithmen der Methoden sind identisch, nur die Namen unterscheiden sich, da eine andere Vergleichsmethode der Klasse `Student` zum Einsatz kommen muss.
- ❑ Im weiteren Verlauf der Vorlesung werden Konzepte eingeführt, mit denen sich das Duplizieren des Programmtextes vermeiden lässt.

## Vergleich der Sortierverfahren in der Praxis

Experiment:

- zufälliges Erzeugen einer großen Zahl von Student-Objekten
- Sortieren dieser Student-Objekte jeweils mit jedem der drei Sortierverfahren  
*SelectionSort, InsertionSort, Quicksort*
- Messen der benötigten Ausführungszeit
- mehrfaches Wiederholen des Erzeugens und Sortierens, um Einflüsse von für einzelne Verfahren besonders günstigen/ungünstigen Ausgangsfolgen auszuschließen
- Bilden von Mittelwerten

## Vergleich der Sortierverfahren in der Praxis

(Fortsetzung)

**zufällig erzeugte, unsortierte Folge**  
(durchschnittliche Ausführungszeit in Millisekunden)

Anzahl der Objekte	Sortierverfahren		
	<i>SelectionSort</i>	<i>InsertionSort</i>	<i>QuickSort</i>
10 000	95		
20 000	380		
30 000	859		
40 000	1517		

# Vergleich der Sortierverfahren in der Praxis

(Fortsetzung)

**zufällig erzeugte, unsortierte Folge**  
(durchschnittliche Ausführungszeit in Millisekunden)

Anzahl der Objekte	Sortierverfahren		
	<i>SelectionSort</i>	<i>InsertionSort</i>	<i>QuickSort</i>
10 000	95	63	
20 000	380	257	
30 000	859	588	
40 000	1517	1052	

## Vergleich der Sortierverfahren in der Praxis

(Fortsetzung)

**zufällig erzeugte, unsortierte Folge**  
(durchschnittliche Ausführungszeit in Millisekunden)

Anzahl der Objekte	Sortierverfahren		
	<i>SelectionSort</i>	<i>InsertionSort</i>	<i>QuickSort</i>
10 000	95	63	1
20 000	380	257	2
30 000	859	588	3
40 000	1517	1052	5

## Vergleich der Sortierverfahren in der Praxis

(Fortsetzung)

### bereits aufsteigend sortierte Folge

(Ausführungszeit in Millisekunden)

Anzahl der Objekte	Sortierverfahren		
	<i>SelectionSort</i>	<i>InsertionSort</i>	<i>QuickSort</i>
1000	1	0	1
2000	3	0	4
3000	7	0	8
10 000	71	0	<i>overflow</i> *)
20 000	290	0	<i>overflow</i>
30 000	656	0	<i>overflow</i>
40 000	1154	0	<i>overflow</i>

\*) Der Speicherplatzbedarf der rekursiven Aufrufe kann durch das Java Runtime System nicht erfüllt werden.

## Vergleich der Sortierverfahren in der Praxis

(Fortsetzung)

### bereits absteigend sortierte Folge

(Ausführungszeit in Millisekunden)

Anzahl der Objekte	Sortierverfahren		
	<i>SelectionSort</i>	<i>InsertionSort</i>	<i>QuickSort</i>
1000	1	1	2
2000	4	3	4
3000	7	10	9
10 000	73	112	<i>overflow</i>
20 000	292	449	<i>overflow</i>
30 000	671	1014	<i>overflow</i>
40 000	1057	1975	<i>overflow</i>

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## **5.1. Rekursive Algorithmen – «Die Töpfe der Java Star»**

Dr. Stefan Dissmann  
Fakultät für Informatik



## Lernziele des Kapitels 5. Rekursive Algorithmen

Nach Durcharbeiten des Kapitels Rekursive Algorithmen sollen die teilnehmenden Studierenden

- den Grundaufbau rekursiver Methoden kennen
- rekursive Algorithmen konzipieren und formulieren können
- rekursive Methoden implementieren können
- Backtracking* als Prinzip der Gestaltung von Algorithmen kennen
- die vorgestellten Beispiele für rekursive Methoden nachvollziehen können

## Erinnerung: «QuickSort» – wesentliche Eigenschaften

```

private void groupByNumber( int leftBound, int rightBound )
{
    if (leftBound < rightBound ) ← Berechnung ohne Rekursion
    {
        int leftOfGreaterPart = leftBound;

        for ( int candidate = leftBound; candidate < rightBound; candidate++ )
        {
            if ( students[rightBound].hasGreaterNumber( students[candidate] ) )
            {
                swapStudents( candidate, leftOfGreaterPart );
                leftOfGreaterPart++;
            }
        }

        swapStudents( leftOfGreaterPart, rightBound );

        groupByNumber( leftBound, leftOfGreaterPart - 1 ); ← Selbstaufruf
        groupByNumber( leftOfGreaterPart + 1 , rightBound );
    }
}

```

kleiner werdende Aufgabe

## Rekursion – technische Voraussetzungen

**Java stellt folgendes Konzept bereit:**

- Jedes Ausführen einer Methode ist unabhängig von anderen Ausführungen von Methoden.
- Inbesondere:*  
Jede Ausführung einer Methode ist unabhängig von anderen Ausführungen *derselben* Methodendeklaration.
- Jeder Aufruf einer Methode erhält für seine Ausführung neuen Speicherplatz für alle Parameter und alle Variablen zugeteilt.
- Gleichzeitig bleibt der Speicherplatz von Parametern und Variablen erhalten, der für bereits laufende Ausführungen der *derselben* Methodendeklaration benutzt wird.
- Beispiel:*  
Jede Ausführung der Methode `groupByNumber` benötigt und erhält also Speicherplatz für vier Werte:  
`leftBound, rightBound, leftOfGreaterPart, candidate`

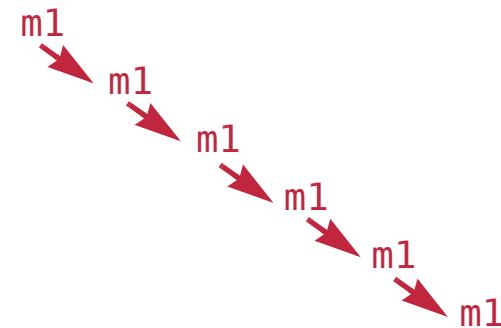
## Rekursion – konzeptionelle Voraussetzungen

Damit ein rekursiv formulierter Algorithmus korrekt arbeitet, müssen immer drei Eigenschaften vorhanden sein:

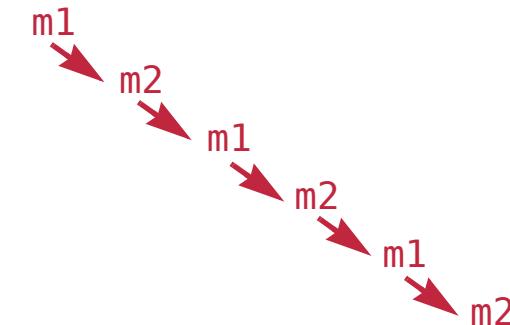
*Selbstaufruf*

Die Ausführung einer Methode führt zu einer weiteren Ausführung derselben Methode.

- Die Methode kann sich im Rumpf ihrer Deklaration selbst aufrufen (*direkte Rekursion*).
- Es gibt aber auch *indirekte Rekursion*, bei der eine Methode  $m_1$  die Ausführung einer anderen Methode  $m_2$  veranlasst, von der aus wiederum die Methode  $m_1$  aufgerufen wird.



direkte Rekursion



indirekte Rekursion

## Rekursion – konzeptionelle Voraussetzungen

(Fortsetzung)

Damit ein rekursiver Algorithmus korrekt arbeitet,  
müssen immer drei Eigenschaften vorhanden sein:

□ *Selbstaufruf*

Die Ausführung einer Methode führt zu einer weiteren Ausführung derselben Methode.

- Die Methode kann sich im Rumpf ihrer Deklaration selbst aufrufen (*direkte Rekursion*).
- Es gibt aber auch *indirekte Rekursion*,  
bei der eine Methode  $m_1$  die Ausführung einer anderen Methode  $m_2$  veranlasst,  
von der aus wiederum die Methode  $m_1$  aufgerufen wird.

□ *Abbruchkriterium für den Selbstaufruf*

Der Algorithmus einer rekursiven Methode muss in ausgewählten Situationen  
ein Ergebnis ohne erneuten rekursiven Aufruf berechnen können.

(Nur dann kann die Folge der rekursiven Ausführungen irgendwann abbrechen.)

## Rekursion – konzeptionelle Voraussetzungen

(Fortsetzung)

Damit ein rekursiver Algorithmus korrekt arbeitet,  
müssen immer drei Eigenschaften vorhanden sein:

□ *Selbstaufruf*

Die Ausführung einer Methode führt zu einer weiteren Ausführung der gleichen Methode.

- Die Methode kann sich im Rumpf ihrer Deklaration selbst aufrufen (*direkte Rekursion*).
- Es gibt aber auch *indirekte Rekursion*,  
bei der eine Methode  $m_1$  die Ausführung einer anderen Methode  $m_2$  veranlasst,  
von der aus wiederum die Methode  $m_1$  aufgerufen wird.

□ *Abbruchkriterium für den Selbstaufruf*

Der Algorithmus einer rekursiven Methode muss in ausgewählten Situationen  
ein Ergebnis ohne erneuten rekursiven Aufruf berechnen können.  
(Nur dann kann die Folge der rekursiven Ausführungen irgendwann abbrechen.)

□ *Reduktion der Problemstellung*

Der Selbstaufruf einer rekursiven Methode muss immer für ein verkleinertes Problem erfolgen,  
das «näher» an dem Abbruchkriterium für die Rekursion liegt.  
(Nur dann wird das Abbruchkriterium irgendwann erreicht.)

## Rekursion – Beispiel «Die Töpfe der Java Star»

Szenario:

Der Schiffskoch des Kreuzfahrtschiffes *Java Star* hat 28 neue Töpfe bekommen, die einfach auf dem Herd seiner Bordküche abgestellt worden sind. Diese muss er nun in den Schrank schaffen, hat dabei aber folgende Probleme:

- Die Töpfe sind so schwer, dass er immer nur genau einen Topf anheben und bewegen kann.
- Um Platz zu sparen besitzen die Töpfe unterschiedliche Durchmesser, so dass sie sich ineinander stellen lassen.
- In der engen Bordküche gibt es nur noch die Spüle, auf der nur genau ein Topf Platz hat.
- Wegen des starken Seegangs können die Töpfe nicht gestapelt werden ohne sofort umzufallen. Die Töpfe müssen also immer ineinander gestellt werden.

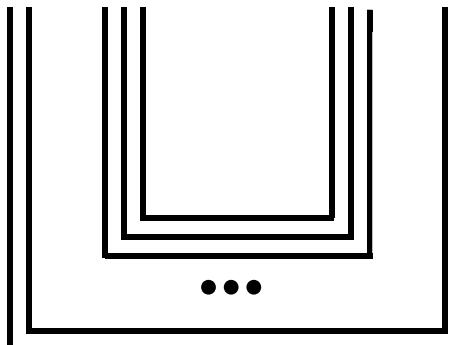
## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Szenario:

Der Schiffskoch des Kreuzfahrtschiffes *Java Star* hat 28 neue Töpfe bekommen, die einfach auf dem Herd seiner Bordküche abgestellt worden sind. Diese muss er nun in den Schrank schaffen, hat dabei aber folgende Probleme:

- Die Töpfe sind so schwer, dass er immer nur genau einen Topf anheben und bewegen kann.
  - Um Platz zu sparen besitzen die Töpfe unterschiedliche Durchmesser, so dass sie sich ineinander stellen lassen.
  - In der engen Bordküche gibt es nur noch die Spüle, auf der nur genau ein Topf Platz hat.
  - Wegen des starken Seegangs können die Töpfe nicht gestapelt werden ohne sofort umzufallen. Die Töpfe müssen also immer ineinander gestellt werden.
- Ausgangssituation:



Herd

Spüle

Schrank

## Rekursion – Beispiel «Die Töpfe der Java Star»

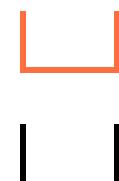
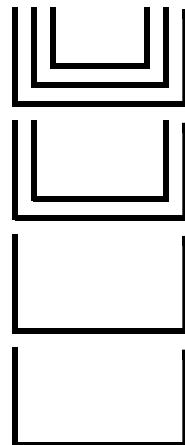
(Fortsetzung)

- Das Problem lässt sich wohl nur durch geschicktes Umlagern lösen.

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

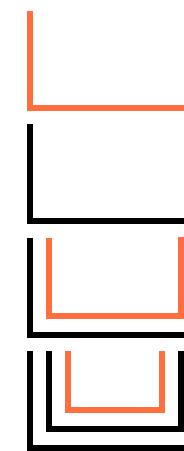
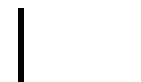
- ❑ Das Problem lässt sich wohl nur durch geschicktes Umlagern lösen.
- ❑ Beispiel für drei Töpfe:



Herd



Spüle



Schrank

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

- Das Problem lässt sich durch geschicktes Umlagern lösen!
- *Aber:* Das Beispiel mit drei Töpfen zeigt, dass die Situation mit mehr Töpfen leicht unübersichtlich werden kann.
- *Und:* Bei Fehlern im Umlagern stehen alle Töpfe am Ende möglicherweise nicht im Schrank – sondern auf der Spüle oder (wieder) auf dem Herd.
- Der Schiffskoch entwickelt also einen Algorithmus, der angibt, in welcher Folge die Töpfe versetzt werden müssen.

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Um zu einem Algorithmus zu kommen,  
betrachten wir die folgende Zwischenstufe des Beispiels:



Herd



Spüle



Schrank

- Damit der größte Topf vom Herd in den Schrank gelangen kann, müssen alle anderen Töpfe auf der Spüle stehen.
- Das Umlagern von  $n$  Töpfen vom Herd in den Schrank kann daher zerlegt werden in:

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Um zu einem Algorithmus zu kommen,  
betrachten wir die folgende Zwischenstufe des Beispiels:



Herd



Spüle



Schrank

- Damit der größte Topf vom Herd in den Schrank gelangen kann, müssen alle anderen Töpfe auf der Spüle stehen.
- Das Umlagern von  $n$  Töpfen vom Herd in den Schrank kann daher zerlegt werden in:
  - das Umlagern der inneren  $n-1$  Töpfe vom Herd auf die Spüle, also die einzige Lagermöglichkeit, die weder Start- noch Zielpunkt ist.

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Um zu einem Algorithmus zu kommen,  
betrachten wir die folgende Zwischenstufe des Beispiels:



Herd



Spüle



Schrank

- Damit der größte Topf vom Herd in den Schrank gelangen kann, müssen alle anderen Töpfe auf der Spüle stehen.
- Das Umlagern von  $n$  Töpfen vom Herd in den Schrank kann daher zerlegt werden in:
  - das Umlagern der inneren  $n-1$  Töpfe vom Herd auf die Spüle, also die einzige Lagermöglichkeit, die weder Start- noch Zielpunkt ist.
  - das Versetzen des größten, äußersten Topfes in den Schrank,

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Um zu einem Algorithmus zu kommen,  
betrachten wir die folgende Zwischenstufe des Beispiels:



Herd



Spüle

Schrank

- Damit der größte Topf vom Herd in den Schrank gelangen kann, müssen alle anderen Töpfe auf der Spüle stehen.
- Das Umlagern von  $n$  Töpfen vom Herd in den Schrank kann daher zerlegt werden in:
  - das Umlagern der inneren  $n-1$  Töpfe vom Herd auf die Spüle, also die einzige Lagermöglichkeit, die weder Start- noch Zielpunkt ist.
  - das Versetzen des größten, äußersten Topfes vom Herd in den Schrank,
  - das Umlagern der  $n-1$  Töpfe von der Spüle in den Topf im Schrank.

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

- Algorithmus für den ersten Teilschritt:

Das Umlagern der inneren  $n-1$  Töpfe vom Herd auf die Spüle kann zerlegt werden in:

- das Umlagern der inneren  $n-2$  Töpfe vom Herd in den Schrank,  
also die Lagermöglichkeit, die weder Start- noch Zielpunkt ist.
  - das Versetzen des größten, äußersten Topfes vom Herd auf die Spüle,
  - das Umlagern der  $n-2$  Töpfe vom Schrank in den Topf auf der Spüle.
- 
- Und so lässt sich das Umlagern fortsetzen,  
bis keine Töpfe mehr versetzt werden müssen.

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Es ergibt sich also folgender, rekursiver Algorithmus:

*Umlagern* von  $n$  Töpfen von einer Startposition zu einer Zielposition  
wird ausgeführt durch

falls  $n$  größer als 0 ist

*Umlagern* von  $n-1$  Töpfen von der Startposition zur *Hilfsposition*

(die weder Start- noch Zielposition ist)

Versetzen von einem Topf von der Startposition zur Zielposition

*Umlagern* von  $n-1$  Töpfen von der Hilfsposition zur Zielposition

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Es ergibt sich also folgender, rekursiver Algorithmus:

*Umlagern* von  $n$  Töpfen von einer Startposition zu einer Zielposition  
wird ausgeführt durch

falls  $n$  größer als 0 ist

*Umlagern* von  $n-1$  Töpfen von der Startposition zur *Hilfsposition*

(die weder Start- noch Zielposition ist)

Versetzen von einem Topf von der Startposition zur Zielposition

*Umlagern* von  $n-1$  Töpfen von der Hilfsposition zur Zielposition

Wie wird die Hilfsposition bestimmt?

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Es ergibt sich also folgender, rekursiver Algorithmus:

*Umlagern* von  $n$  Töpfen von einer Startposition zu einer Zielposition  
wird ausgeführt durch

falls  $n$  größer als 0 ist

*Umlagern* von  $n-1$  Töpfen von der Startposition zur Hilfsposition

(= weder Start- noch Zielposition)

Versetzen von einem Topf von der Startposition zur Zielposition

*Umlagern* von  $n-1$  Töpfen von der Hilfsposition zur Zielposition

Wie wird die Hilfsposition bestimmt?

- Die Positionen werden mit Nummern versehen:  
Herd = 0, Spüle = 1, Schrank = 2
- Die Summe aller Nummern ist dann:  
 $0 + 1 + 2 = 3$
- Die Hilfsposition ergibt sich daher immer als:  
 $3 - \text{Startposition} - \text{Zielposition}$

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

```

private static void rearrangePots( int quantity, int start, int target )
{
    String[] locations = { "stove", "sink", "cupboard" };
    if (quantity > 0)           ← Abbruchkriterium
    {
        rearrangePots( quantity - 1, start, computeInterimPosition( start, target ) );
        System.out.println(
            "pot " + quantity + ": " + locations[start] + " -> " + locations[target] );
        rearrangePots( quantity - 1, computeInterimPosition(start, target), target );
    }
}

```

**rekursive Aufrufe** →

→ **Reduktion**

```

private static int computeInterimPosition( int start, int target )
{
    return 3 - start - target;
}

```

- Die Implementierung setzt den zuvor entwickelten Algorithmus direkt um.

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

```
public static void rearrangeAllPots( int quantity )
{
    rearrangePots( quantity, 0, 2 );
}
```

- Die öffentliche Methode `rearrangeAllPots` dient wiederum dazu,  
für den ersten Aufruf der rekursiven Methode die Argumente geeignet zu übergeben.

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Ausgabe für das Beispiel mit drei Töpfen:

```
pot 1: stove -> cupboard
pot 2: stove -> sink
pot 1: cupboard -> sink
pot 3: stove -> cupboard
pot 1: sink -> stove
pot 2: sink -> cupboard
pot 1: stove -> cupboard
```

## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

rekursive Aufrufstruktur für das Beispiel mit drei Töpfen:

```
rearrangePots( 3, 0, 2 )
    rearrangePots( 2, 0, 1 )
        rearrangePots( 1, 0, 2 )
            rearrangePots( 0, 0, 1 )
            rearrangePots( 0, 1, 2 )
        rearrangePots( 1, 2, 1 )
            rearrangePots( 0, 2, 0 )
            rearrangePots( 0, 0, 1 )
    rearrangePots( 2, 1, 2 )
        rearrangePots( 1, 1, 0 )
            rearrangePots( 0, 1, 2 )
            rearrangePots( 0, 2, 0 )
        rearrangePots( 1, 0, 2 )
            rearrangePots( 0, 0, 1 )
            rearrangePots( 0, 1, 2 )
```

Schachtelung der rekursiven Aufrufe



## Rekursion – Beispiel «Die Töpfe der Java Star»

(Fortsetzung)

Erkenntnisse:

- ❑ Der rekursive Algorithmus hat sich nach kurzer Überlegung recht unmittelbar ergeben. Entscheidend für die Entwicklung war nur die Reduktion des Problems auf das Versetzen des größten Topfes.
- ❑ Da rekursive Algorithmen immer eine Reduktion des Problems für den rekursiven Aufruf enthalten müssen, ist es naheliegend, über einen passenden Denkansatz zu einem Algorithmus zu kommen.

Anmerkungen:

- ❑ Für das regelgerechte Umsetzen aller 28 Töpfe müssen 26 843 455 Bewegungen ausgeführt werden. Selbst wenn der Schiffskoch nur eine Sekunde für das Versetzen eines Topfes braucht, erfordert das Umsetzen aller 28 Töpfe etwa ein Jahr Arbeit *ohne* Pause.
- ❑ Das hier in die Schifffahrt verlagerte Problem ist in der Informatik unter dem Namen «Türme von Hanoi» bekannt.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## **5.2. Rekursive Algorithmen – «Beladen der Java Star»**

Dr. Stefan Dissmann  
Fakultät für Informatik



## Rekursion – Beispiel «Beladen der Java Star»

Szenario:

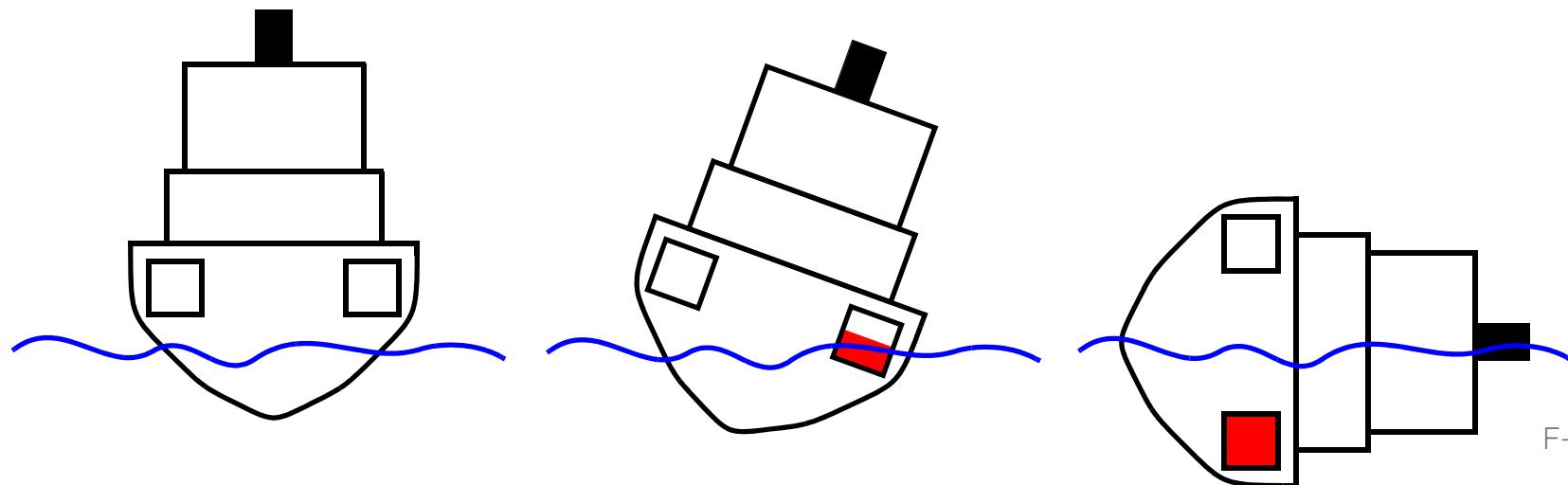
- ❑ Für seine Passagiere muss das Schiff Vorräte in Containern an Bord nehmen.
- ❑ Jeder Container hat ein bekanntes Gewicht.
- ❑ Die Container werden in einer nicht zu beeinflussenden Folge angeliefert.
- ❑ Container können in Laderäumen auf der Backbord- oder Steuerbordseite verstaut werden. Allerdings muss darauf geachtet werden, dass die Balance des Schiffes erhalten bleibt, damit das Schiff nicht bereits während des Beladevorgangs eine zu große Schlagseite bekommt.
- ❑ Der Lademeister möchte also bereits im Vorfeld der Beladung prüfen, ob eine geeignete Verteilung der ankommenden Container auf die beiden Seiten des Schiffes überhaupt möglich ist.

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

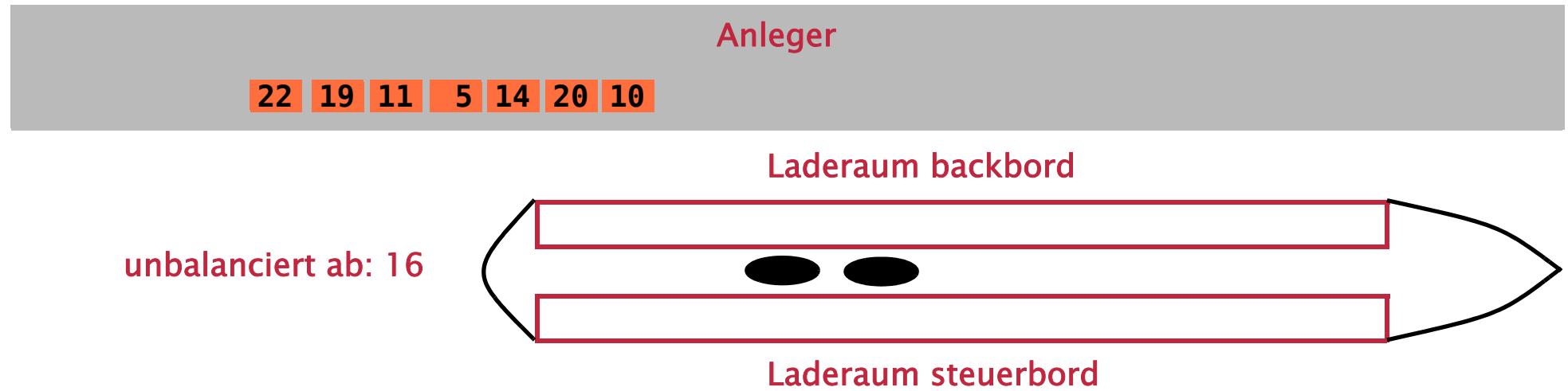
Szenario:

- ❑ Für seine Passagiere muss das Schiff Vorräte in Containern an Bord nehmen.
- ❑ Jeder Container hat ein bekanntes Gewicht.
- ❑ Die Container werden in einer nicht zu beeinflussenden Folge angeliefert.
- ❑ Container können in Laderäumen auf der Backbord- oder Steuerbordseite verstaut werden. Allerdings muss darauf geachtet werden, dass die Balance des Schiffes erhalten bleibt, damit das Schiff nicht bereits während des Beladevorgangs eine zu große Schlagseite bekommt.
- ❑ Der Lademeister möchte also bereits im Vorfeld der Beladung prüfen, ob eine geeignete Verteilung der ankommenden Container auf die beiden Seiten des Schiffes überhaupt möglich ist.



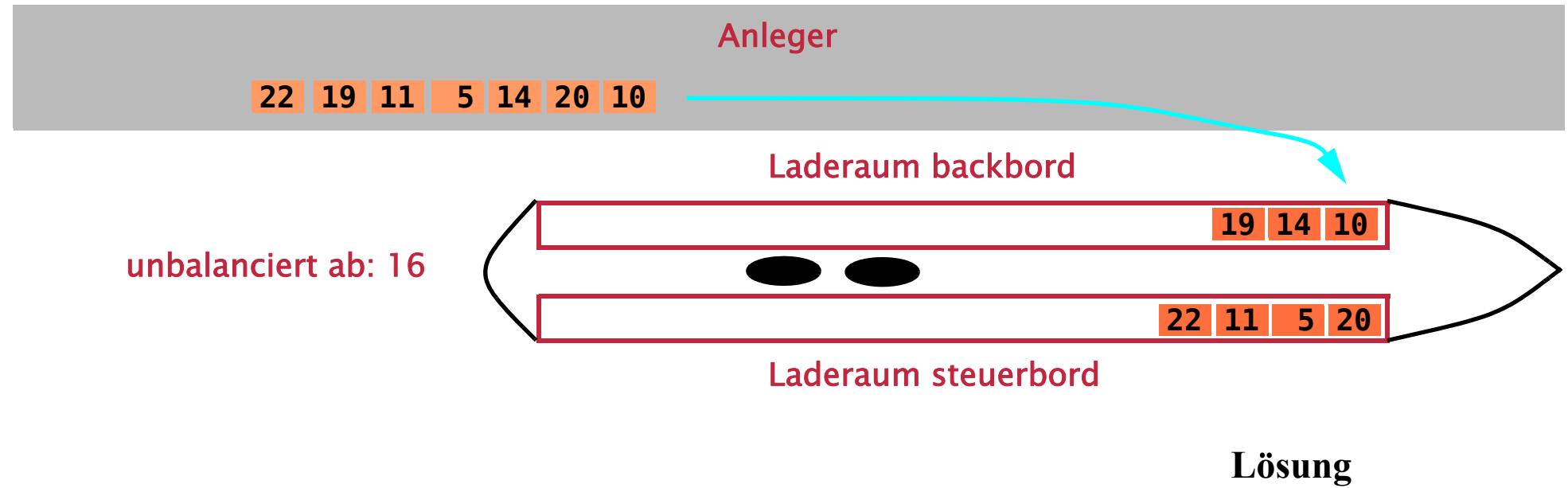
## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)



## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)



## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

Szenario:

- ❑ Für seine Passagiere muss das Schiff Vorräte in Containern an Bord nehmen.
  - ❑ Jeder Container hat ein bekanntes Gewicht.
  - ❑ Die Container werden in einer nicht zu beeinflussenden Folge angeliefert.
  - ❑ Container können in Laderäumen auf der Backbord- oder Steuerbordseite verstaut werden. Allerdings muss darauf geachtet werden, dass die Balance des Schiffes erhalten bleibt, damit das Schiff nicht bereits während des Beladevorgangs eine zu große Schlagseite bekommt.
  - ❑ Der Lademeister möchte also bereits im Vorfeld der Beladung prüfen, ob eine geeignete Verteilung der ankommenden Container auf die beiden Seiten des Schiffes überhaupt möglich ist.
- 
- ❑ Das Problem wird hier etwas vereinfacht modelliert:
    - Es gibt ein Feld `containers` mit positiven ganzen Zahlen, die die Gewichte der Container in der Reihenfolge ihres Ankommens darstellen.
    - Es gibt einen positiven Wert `limit`, der den Grenzwert für das Ungleichgewicht angibt, bei dessen Überschreiten das Schiff umkippen würde.

## Rekursion – Beispiel «Beladen der *Java Star*»

(Fortsetzung)

Voraussetzung:

- ❑ Da die Folge der zu ladenden Gewichte festliegt, muss von Beginn des Ladevorgangs an eine geeignete Verteilung der Gewichte vorgenommen werden.

## Rekursion – Beispiel «Beladen der *Java Star*»

(Fortsetzung)

Voraussetzung:

- Da die Folge der zu ladenden Gewichte festliegt, muss von Beginn des Ladevorgangs an eine geeignete Verteilung der Gewichte vorgenommen werden.

Lösungsidee:

- Lade den Container  $n$  auf eine Seite des Schiffes.
- Prüfe, ob das Schiff noch im Gleichgewicht liegt:
  - falls das der Fall ist, setze fort mit dem Laden des Containers  $n+1$
  - falls das nicht der Fall ist, lade den Container  $n$  auf die andere Seite um

## Rekursion – Beispiel «Beladen der *Java Star*»

(Fortsetzung)

Voraussetzung:

- Da die Folge der zu ladenden Gewichte festliegt, muss von Beginn des Ladevorgangs an eine geeignete Verteilung der Gewichte vorgenommen werden.

Lösungsidee:

- Lade den Container  $n$  auf eine Seite des Schiffes.
- Prüfe, ob das Schiff noch im Gleichgewicht liegt:
  - falls das der Fall ist, setze fort mit dem Laden des Containers  $n+1$
  - falls das nicht der Fall ist, lade den Container  $n$  auf die andere Seite um
    - Prüfe, ob das Schiff noch im Gleichgewicht liegt:
      - falls das der Fall ist, setze fort mit dem Laden des Containers  $n+1$
      - falls das nicht der Fall ist, passt die Ausgangssituation nicht:  
versuche, den Container  $n-1$  auf die andere Seite umzuladen  
und dann forzusetzen

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

Voraussetzung:

- Da die Folge der zu ladenden Gewichte festliegt, muss von Beginn des Ladevorgangs an eine geeignete Verteilung der Gewichte vorgenommen werden.

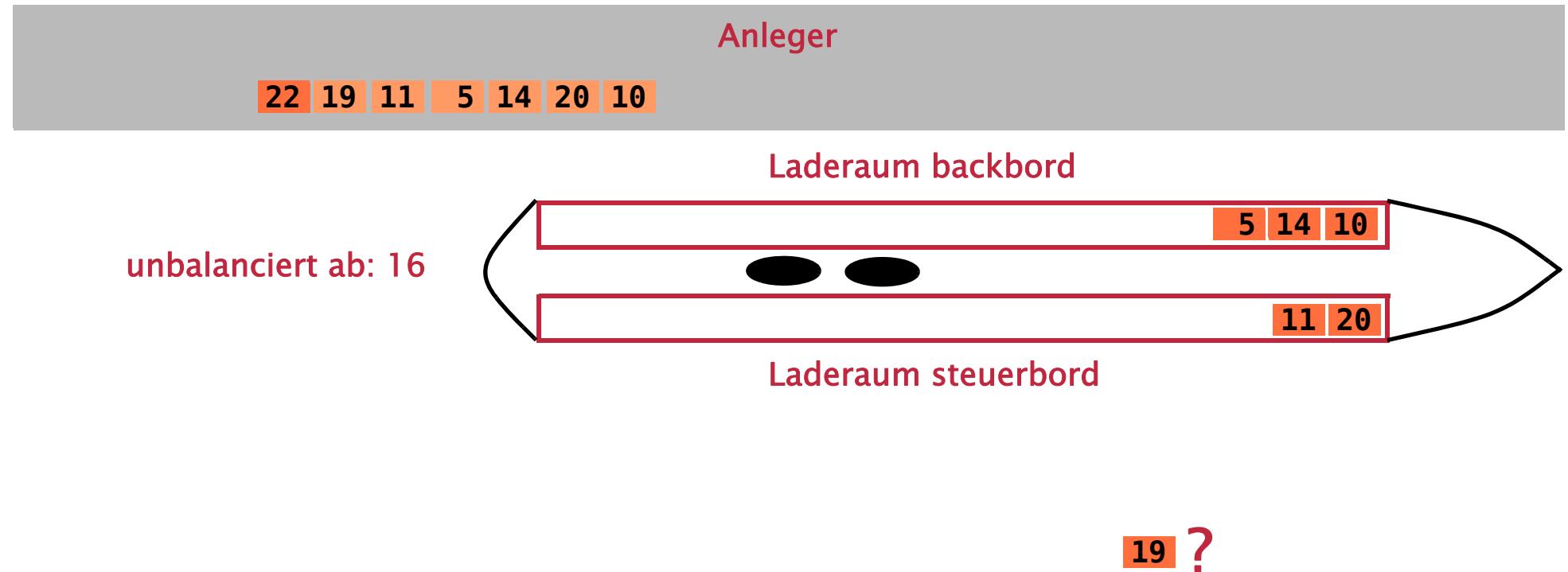
Lösungsidee:

- Lade den Container  $n$  auf eine Seite des Schiffes.
- Prüfe, ob das Schiff noch im Gleichgewicht liegt:
  - falls das der Fall ist, setze fort mit dem Laden des Containers  $n+1$
  - falls das nicht der Fall ist, lade den Container  $n$  auf die andere Seite um
    - Prüfe, ob das Schiff noch im Gleichgewicht liegt:
      - falls das der Fall ist, setze fort mit dem Laden des Containers  $n+1$
      - falls das nicht der Fall ist, passt die Ausgangssituation nicht:  
versuche, den Container  $n-1$  auf die andere Seite umzuladen  
und dann forzusetzen

**Gehe gegebenenfalls noch weiter zurück und versuche,  
eine andere Ausgangssituationen zu schaffen.**

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)



## Rekursion – Beispiel «Beladen der *Java Star*»

(Fortsetzung)

Lösungsidee (Kurzfassung):

Erstelle schrittweise eine Beladung.

Sobald das Gleichgewicht nicht gehalten werden kann, gehe schrittweise zurück und versuche von einer der vorherigen Beladungen aus anders fortzusetzen.

Das Verfahren endet, wenn

- entweder alle Container erfolgreich verladen werden konnten
- oder alle möglichen Beladungsalternativen erfolglos abgebrochen worden sind.

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

Lösungsidee (Kurzfassung):

Erstelle schrittweise eine Beladung.

Sobald das Gleichgewicht nicht gehalten werden kann, gehe schrittweise zurück und versuche von einer der vorherigen Beladungen aus anders fortzusetzen.

Das Verfahren endet, wenn

- entweder alle Container erfolgreich verladen werden konnten
- oder alle möglichen Beladungsalternativen erfolglos abgebrochen worden sind.

Rekursive Methode zur Lösung:

- ❑ Jede Instanz einer rekursiven Methode enthält einen eigenen Satz von Variablen mit Werten.
- ❑ Wird dieser Satz von Variablen dazu genutzt, um den Zwischenstand der Beladung festzuhalten, so kann einfach durch das Beenden von jüngeren Instanzen der Methode zu den älteren Instanzen und damit zu einem älteren Zwischenstand der Beladung zurückgekehrt werden.
- ❑ Die aufeinander folgenden rekursiven Aufrufe bilden also ein Gedächtnis der bisher vorgenommenen Entscheidungen zur Beladung des Schiffes.

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

Signatur der rekursiven Methode:

```
private static boolean existsBalanceUpTo(  
    int[] containers, int limit, int divergence, int unitsLoaded )  
{  
    ...  
}
```

Containergewichte

Gleichgewichtsgrenze

Ungleichgewicht  
bei aktueller Beladung

Anzahl der verladenen Container

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

```
private static boolean existsBalanceUpTo(  
    int[] containers, int limit, int divergence, int unitsLoaded )  
{
```

Abbruchkriterium: alle Container sind verladen

rekursiver Aufruf: Container auf Backbord-Seite laden

rekursiver Aufruf: Container auf Steuerbord-Seite laden

```
}
```

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

```
private static boolean existsBalanceUpTo(
    int[] containers, int limit, int divergence, int unitsLoaded )
{
    if ( unitsLoaded >= containers.length )
    {
        return true;
    }
}
```

Abbruchkriterium:  
alle Container verladen

rekursiver Aufruf: Container auf Backbord-Seite laden

rekursiver Aufruf: Container auf Steuerbord-Seite laden

}

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

```

private static boolean existsBalanceUpTo(
    int[] containers, int limit, int divergence, int unitsLoaded )
{
    if ( unitsLoaded >= containers.length )
    {
        return true;
    }
    int nextDivergence = divergence + containers[unitsLoaded];
    if ( Math.abs( nextDivergence ) <= limit )
    {
        boolean loadedOnPortside =
            existsBalanceUpTo( containers, limit, nextDivergence, unitsLoaded+1 );
        if ( loadedOnPortside )
        {
            return true;
        }
    }
}

```

Abbruchkriterium:  
alle Container verladen

rekursiver Aufruf:  
Container auf Backbord-Seite laden

rekursiver Aufruf: Container auf Steuerbord-Seite laden

}

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

```

private static boolean existsBalanceUpTo(
    int[] containers, int limit, int divergence, int unitsLoaded )
{
  if ( unitsLoaded >= containers.length )
  {
    return true;
  }
  int nextDivergence = divergence + containers[unitsLoaded];
  if ( Math.abs( nextDivergence ) <= limit )
  {
    boolean loadedOnPortside =
      existsBalanceUpTo( containers, limit, nextDivergence, unitsLoaded+1);
    if ( loadedOnPortside )
    {
      return true;
    }
  }
  nextDivergence = divergence - containers[unitsLoaded];
  if ( Math.abs( nextDivergence ) <= limit )
  {
    return existsBalanceUpTo(containers, limit, nextDivergence, unitsLoaded+1);
  }
  else
  {
    return false;
  }
}

```

rekursiver Aufruf:  
Container auf Steuerbord-Seite laden

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

```
public static boolean existsBalance( int[] containers, int limit )
{
    return existsBalanceUpTo( containers, limit, 0, 0 );
}
```

- initialer Aufruf der rekursiven Methode
  - das Schiff ist im Gleichgewicht: divergence = 0
  - es wurde noch kein Container verladen: unitsLoaded = 0
- Beide Angaben werden nur zur Steuerung der rekursiven Aufrufe benötigt.  
Die Methode `existsBalance` verbirgt die Parameter und die initial übergebenen Argumente.

## Rekursion – Beispiel «Beladen der Java Star»

(Fortsetzung)

- Die Algorithmen *QuickSort* und zum Umlagern der Töpfe streben unmittelbar auf das gewünschte Ergebnis zu. Die Rekursion dient dazu, die Ausführung von Teilen der Algorithmen zu koordinieren:
  - *QuickSort*:  
Zerlegen in zwei Abschnitte, Sortieren des einen Abschnitts, dann des anderen Abschnitts
  - Umstapeln der Töpfe:  
Stapeln auf dem Hilfsplatz, von dort Umstapeln zum Ziel
- Das Konzept der Rekursion wird beim Beladen anders eingesetzt:  
Hier wird zunächst versucht, eine Lösung zu finden.  
Scheitert der Versuch, so wird die Rekursion eingesetzt, um zurückgehen zu können.  
  
Diese Art der Gestaltung von Algorithmen wird als *Backtracking* bezeichnet.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## **5.3. Rekursive Algorithmen – «Die Treppe der Informatik»**

Dr. Stefan Dissmann  
Fakultät für Informatik



## Rekursion – Beispiel «Die Treppe der Informatik»

Szenario:

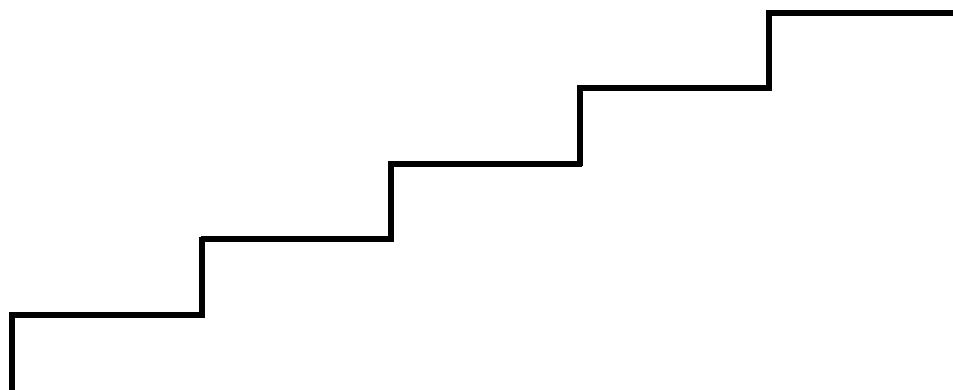
- ❑ Ein körperlich leistungsfähiger Mensch kann auf einer Treppe zwei Stufen mit einem Schritt überwinden.
- ❑ Die Treppe des Informatik-Gebäudes OH14 hat 80 Stufen.
- ❑ Wie viele Schrittfolgen gibt es, diese Treppe hinaufzugehen, wenn beliebig Schritte über eine oder zwei Stufen miteinander kombiniert werden können.

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Szenario:

- ❑ Ein körperlich leistungsfähiger Mensch kann auf einer Treppe zwei Stufen mit einem Schritt überwinden.
  - ❑ Die Treppe des Informatik-Gebäudes OH14 hat 80 Stufen.
  - ❑ Wie viele Schrittfolgen gibt es, diese Treppe hinaufzugehen, wenn beliebig Schritte über eine oder zwei Stufen miteinander kombiniert werden können.
- ❑ *Beispiel:* Treppe mit fünf Stufen



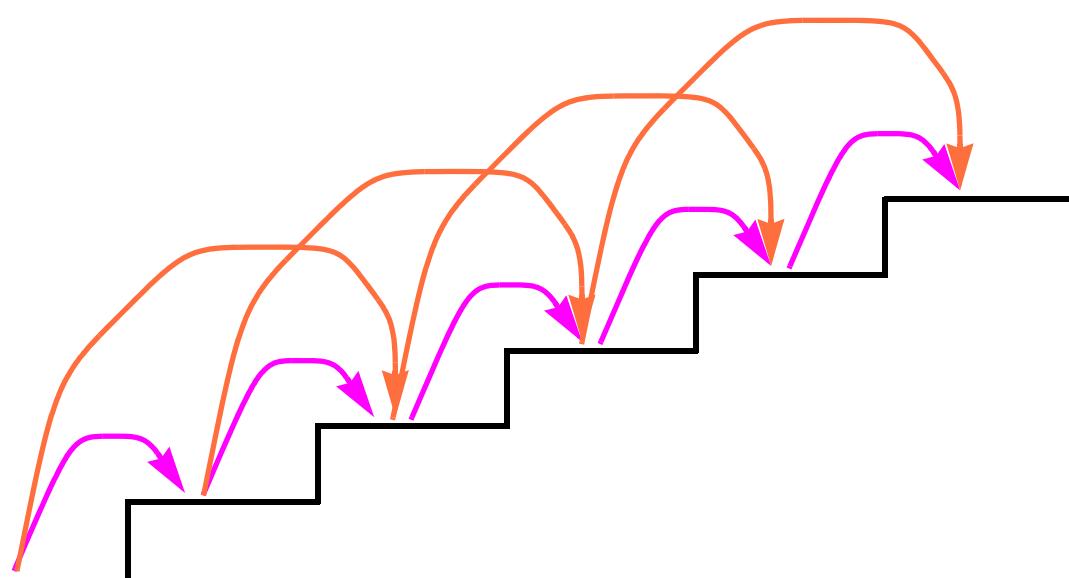
mögliche Schrittfolgen:

1-1-1-1-1  
1-1-1-2  
1-1-2-1  
1-2-1-1  
2-1-1-1  
1-2-2  
2-1-2  
2-2-1

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Visualisierung:



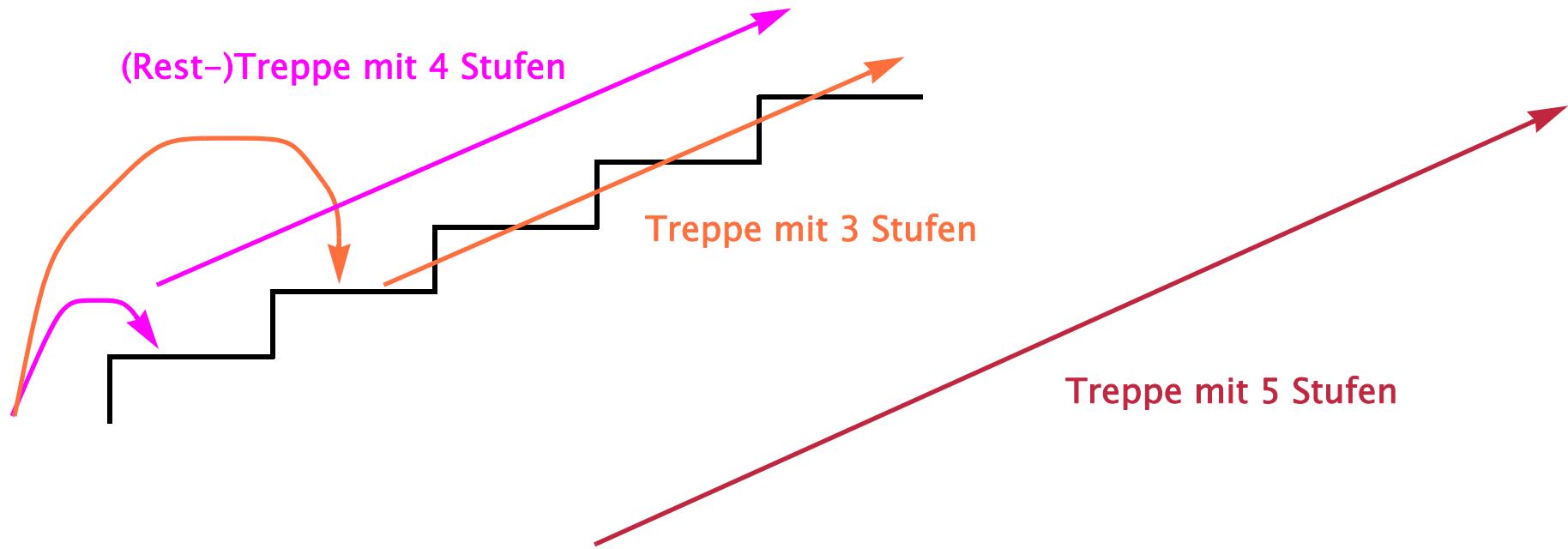
mögliche Schrittfolgen:

- 1-1-1-1-1
- 1-1-1-2
- 1-1-2-1
- 1-2-1-1
- 2-1-1-1
- 1-2-2
- 2-1-2
- 2-2-1

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Visualisierung:



## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Berechnung für das Beispiel:

Schrittfolgen für eine Treppe mit 5 Stufen =

Schrittfolgen für eine Treppe mit 4 Stufen

(wenn mit einem kleinen Schritt begonnen wurde)

- + Schrittfolgen für eine Treppe mit 3 Stufen  
(wenn mit einem großen Schritt begonnen wurde)

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Berechnung für das Beispiel:

Schrittfolgen für eine Treppe mit 5 Stufen =

Schrittfolgen für eine Treppe mit 4 Stufen

(wenn mit einem kleinen Schritt begonnen wurde)

- + Schrittfolgen für eine Treppe mit 3 Stufen  
(wenn mit einem großen Schritt begonnen wurde)

Berechnung (allgemein):

Schrittfolgen für eine Treppe mit  $n$  Stufen =

falls  $n \leq 2$ :

$n$

sonst:

Schrittfolgen für eine Treppe mit  $n-1$  Stufen

(wenn mit einem kleinen Schritt begonnen wurde)

- + Schrittfolgen für eine Treppe mit  $n-2$  Stufen  
(wenn mit einem großen Schritt begonnen wurde)

Abbruchkriterium

offensichtlich eine rekursive  
Berechnungsvorschrift

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

```
public static long routesToTop( long stairs )
{
    if ( stairs <= 2 )
    {
        return stairs;
    }
    else
    {
        return routesToTop( stairs - 1 ) + routesToTop( stairs - 2 );
    }
}
```

- Die Implementierung setzt den zuvor entwickelten Algorithmus direkt um.

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

```
public static long routesToTop( long stairs )
{
    if ( stairs <= 2 )           ← Abbruchkriterium
    {
        return stairs;
    }
    else
    {
        return routesToTop( stairs - 1 ) + routesToTop( stairs - 2 );
    }
}
```

- Die Implementierung setzt den zuvor entwickelten Algorithmus direkt um.

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

```
public static long routesToTop( long stairs )
{
    if ( stairs <= 2 )           ← Abbruchkriterium
    {
        return stairs;
    }
    else
    {
        return routesToTop( stairs - 1 ) + routesToTop( stairs - 2 );
    }
}
```

rekursive Aufrufe

Reduktion

- Die Implementierung setzt den zuvor entwickelten Algorithmus direkt um.
- Die Ergebnisse der rekursiven Aufrufe liefern jeweils ein Ergebnis.  
Die Ergebnisse werden addiert, um zum Ergebnis der aufrufenden Methode zu gelangen: +

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

```

public static long routesToTop( long stairs )
{
  if ( stairs <= 2 )           ← Abbruchkriterium
  {
    return stairs;
  }
  else
  {
    return routesToTop( stairs - 1 ) + routesToTop( stairs - 2 );
  }
}
  
```

rekursive Aufrufe

Reduktion

- Die Implementierung setzt den zuvor entwickelten Algorithmus direkt um.
- Die Ergebnisse der rekursiven Aufrufe liefern jeweils ein Ergebnis.  
Die Ergebnisse werden addiert, um zum Ergebnis der aufrufenden Methode zu gelangen: +
- Der Aufruf `routesToTop( 80 )` für das Informatik-Gebäude liefert: **37889062373143906**
- Der rechte Turm des Kölner Doms hat 533 Stufen:  
Die Berechnung übersteigt die Grenzen des Wertebereichs `long` deutlich.

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

rekursive Aufrufstruktur für das Beispiel der Treppe mit fünf Stufen:

```
routesToTop( 5 )
  routesToTop( 4 )
    routesToTop( 3 )
      routesToTop( 2 )
        routesToTop( 1 )
      routesToTop( 2 )
    routesToTop( 3 )
      routesToTop( 2 )
    routesToTop( 1 )
```

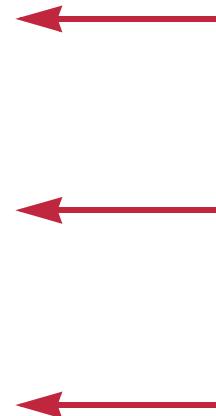
- Berechnungen für den gleichen Parameterwert werden immer wieder aufgerufen und daher auch neu berechnet: `routesToTop( 3 )` hier schon zweimal.

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

rekursive Aufrufstruktur für eine Treppe mit sechs Stufen:

```
routesToTop( 6 )
  routesToTop( 5 )
    routesToTop( 4 )
      routesToTop( 3 )
        routesToTop( 2 )
        routesToTop( 1 )
      routesToTop( 2 )
      routesToTop( 3 )
        routesToTop( 2 )
        routesToTop( 1 )
    routesToTop( 4 )
      routesToTop( 3 )
      routesToTop( 2 )
      routesToTop( 1 )
    routesToTop( 2 )
```



- Gleiche Berechnungen werden für große Treppen *sehr häufig* ausgeführt:  
Für die Berechnung von `routesToTop(20)` werden beispielsweise 2639 Aufrufe von `routesToTop( 3 )` durchgeführt.

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Wie lassen sich wiederholte Berechnungen vermeiden?

Durch Speichern der berechneten Werte,  
z.B. in einem Feld, das als Attribut für jeden Aufruf der Methode `routesToTop` zugreifbar ist.

```
routesToTop( 6 )
  routesToTop( 5 )
    routesToTop( 4 )
      routesToTop( 3 )
        routesToTop( 2 )
          routesToTop( 1 )
            routesToTop( 2 )
            routesToTop( 3 )
              routesToTop( 2 )
              routesToTop( 1 )
            routesToTop( 4 )
            routesToTop( 3 )
              routesToTop( 2 )
              routesToTop( 1 )
            routesToTop( 2 )
```

alle benötigten Werte werden berechnet



## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Wie lassen sich wiederholte Berechnungen vermeiden?

Durch Speichern der berechneten Werte,  
z.B. in einem Feld, das als Attribut für jeden Aufruf der Methode `routesToTop` zugreifbar ist.

```
routesToTop( 6 )
  routesToTop( 5 )
    routesToTop( 4 )
      routesToTop( 3 )
        routesToTop( 2 )
          routesToTop( 1 )
            routesToTop( 2 )
            routesToTop( 3 )
              routesToTop( 2 )
              routesToTop( 1 )
            routesToTop( 4 )
            routesToTop( 3 )
              routesToTop( 2 )
              routesToTop( 1 )
            routesToTop( 2 )
```

alle benötigten Werte werden berechnet

keine Berechnungen mehr notwendig

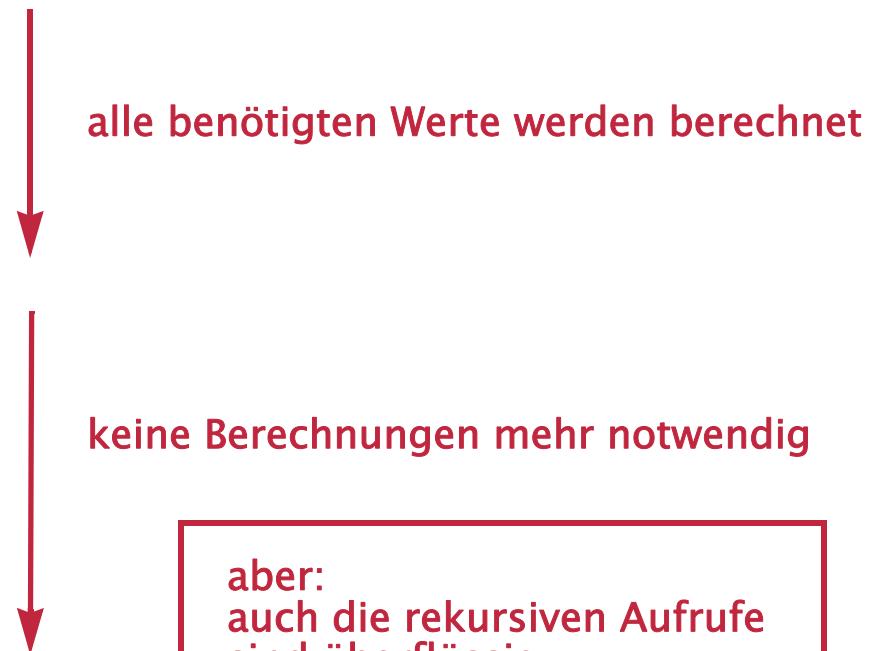
## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Wie lassen sich wiederholte Berechnungen vermeiden?

Durch Speichern der berechneten Werte,  
z.B. in einem Feld, das als Attribut für jeden Aufruf der Methode `routesToTop` zugreifbar ist.

```
routesToTop( 6 )
  routesToTop( 5 )
    routesToTop( 4 )
      routesToTop( 3 )
        routesToTop( 2 )
          routesToTop( 1 )
            routesToTop( 2 )
            routesToTop( 3 )
              routesToTop( 2 )
              routesToTop( 1 )
            routesToTop( 4 )
            routesToTop( 3 )
              routesToTop( 2 )
              routesToTop( 1 )
            routesToTop( 2 )
```



## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Wie lassen sich wiederholte Berechnungen vermeiden?

Durch Ändern der Reihenfolge der Berechnungen:

```
routesToTop( 6 )
  routesToTop( 5 )
    routesToTop( 4 )
      routesToTop( 3 )
        routesToTop( 2 )
          routesToTop( 1 )
```

mit Schleife und paarweiser Addition  
der berechneten Werte

□ also:

routesToTop( 3 )	←	routesToTop( 2 ) + routesToTop( 1 )
routesToTop( 4 )	←	routesToTop( 3 ) + routesToTop( 2 )
routesToTop( 5 )	←	routesToTop( 4 ) + routesToTop( 3 )
routesToTop( 6 )	←	routesToTop( 5 ) + routesToTop( 4 )

□ Es werden zur Berechnung des nächsten Wertes immer nur die Ergebnisse der letzten beiden Berechnungen benötigt:

zwei Variablen reichen aus

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Lösung *ohne* Rekursion mit einer Schleife (*Iteration*)

```
public static long iterativRoutesToTop( long stairs )
{
    if ( stairs <= 2 )           ← direkt berechenbar
    {
        return stairs;
    }
    else
    {
        long next = 0, old1 = 1, old2 = 2;   ← drei Variable
                                                zur Zwischenablage
        for (int i = 3; i <= stairs; i++)
        {
            next = old1 + old2;             ← Addition
            old1 = old2;                  ← "Verschieben" der Inhalte
            old2 = next;                 der Variablen
        }
        return next;
    }
}
```

## Rekursion – Beispiel «Die Treppe der Informatik»

Vergleich der rekursiven und der iterativen Lösung

(Fortsetzung)

Laufzeiten der rekursiven Lösung:

- routesToTop( 35 ): 45 Millisekunden
- routesToTop( 40 ): 421 Millisekunden
- routesToTop( 45 ): 4641 Millisekunden
- routesToTop( 50 ): 50560 Millisekunden

Laufzeiten der iterativen Lösung: < 1 Millisekunde

## Rekursion – Beispiel «Die Treppe der Informatik»

(Fortsetzung)

Schlussfolgerungen für die Berechnung der Wege auf einer Treppe:

- ❑ Der rekursive Lösungsansatz kann intuitiv gefunden werden.
- ❑ Die Formulierung der rekursiven Lösung ist einfach.
- ❑ Der Berechnungsaufwand der rekursiven Lösung ist hoch und nimmt schnell zu.
- ❑ Die Analyse des rekursiven Lösungsansatzes kann als Ausgangspunkt für die Entwicklung einer iterativen Lösung dienen.

Anmerkung:

- ❑ Bei der durch `routesToTop` berechneten Zahlenfolge handelt es sich um die *Fibonacci*-Zahlen.  
(von Leonardo Fibonacci ca. 1200 in Pisa entdeckt bei dem Versuch, das Wachstum einer Kanninchen-Population zu beschreiben.)

## Rekursion – Zusammenfassung

- ❑ Rekursion ist ein brauchbares Konzept zur Entwicklung von Algorithmen.
- ❑ Rekursion ist ein brauchbares Konzept zur Implementierung von Algorithmen:  
*QuickSort*, Umstapeln der Töpfe, *Backtracking*  
(Bei diesen Beispielen finden keine überflüssigen rekursiven Aufrufe statt.)
- ❑ Rekursive Algorithmen können dafür sorgen,  
dass kaskadenartig Werte abgearbeitet werden:
  - *QuickSort* bearbeitet so die ständig anwachsende Menge der kleiner werdenden Teile.
  - Das Treppensteigen bearbeitet so die ständig anwachsende Menge der kleiner  
werdenden Treppen.
- ❑ Rekursive Algorithmen können ausnutzen,  
dass nach Ausführung der rekursiven Aufrufe die Werte von Parametern und lokalen Variablen  
noch verfügbar sind. So kann in jedem der Beispiele ein zweiter rekursiver Aufruf mit  
vernünftigen Parameterwerten erfolgen:
  - `groupByNumber( leftBound, rightBound );`
  - `rearrangePots( quantity - 1, 3 - start - target, target );`
  - `existsBalanceUpTo( containers, limit, nextDivergence, unitsLoaded + 1 );`
  - `return routesToTop( stairs - 1 ) + routesToTop( stairs - 2 );`
- ❑ Rekursive Algorithmen können wie andere Methoden Werte zurückgeben (`routesToTop`),  
Seiteneffekte erzeugen (`groupByNumber`) oder Ausgaben vornehmen (`rearrangePots`).

## Rekursion – Folgerung

Wenn es möglich ist,  
zu einem rekursiven Algorithmus `routesToTop` einen äquivalenten iterativen Algorithmus anzugeben,  
kann dann auch zu iterativen Algorithmen ein äquivalenter rekursiver Algorithmus gefunden werden?

Dazu werden zwei (bekannte) Beispiele untersucht:

- ❑ Berechnung der Summe der Quadrate aller natürlichen Zahlen bis zu einer Grenze  $n$   
(siehe Folie 108)
- ❑ Berechnung des größten gemeinsamen Teilers  
(siehe Folie 233)

## Rekursion – Folgerung

(Fortsetzung)

Summe der Quadrate natürlicher Zahlen – rekursive Lösung

```
public static int recursiveSumOfSquares( int n )
{
    if ( n > 1 ) {
        return n * n + recursiveSumOfSquares( n - 1 );
    }
    else
    {
        return n;
    }
}
```

## Rekursion – Folgerung

(Fortsetzung)

Summe der Quadrate natürlicher Zahlen – rekursive Lösung

```
public static int recursiveSumOfSquares( int n )
{
    if ( n > 1 ) {
        return n * n + recursiveSumOfSquares( n - 1 );
    }
    else
    {
        return n;
    }
}
```

The diagram shows a Java code snippet for calculating the sum of squares of natural numbers. Red annotations explain the recursive process: 'rekursiver Aufruf' points to the recursive call line 'recursiveSumOfSquares( n - 1 )'; 'Abbruchkriterium' points to the base case condition 'if ( n > 1 )'; and 'Reduktion' points to the reduction step where the problem size is decreased by 1 in each recursive call.

## Rekursion – Folgerung

größter gemeinsamer Teiler – rekursive Lösung

(Fortsetzung)

```
public static int recursiveCalculateGcd( int v1, int v2 ) {  
    int i1 = Math.abs(v1);  
    int i2 = Math.abs(v2);  
    if (i1 != 0 & i2 != 0)  
    {  
        if ( i1 > i2 )  
        {  
            return recursiveCalculateGcd( i1 % i2, i2 );  
        }  
        else  
        {  
            return recursiveCalculateGcd( i1, i2 % i1 );  
        }  
    }  
    return i1 + i2;  
}
```

## Rekursion – Folgerung

(Fortsetzung)

größter gemeinsamer Teiler – rekursive Lösung

```

public static int recursiveCalculateGcd( int v1, int v2 ) {
    int i1 = Math.abs(v1);
    int i2 = Math.abs(v2);
    if (i1 != 0 & i2 != 0)
    {
        if ( i1 > i2 )
        {
            return recursiveCalculateGcd( i1 % i2, i2 );
        }
        else
        {
            return recursiveCalculateGcd( i1, i2 % i1 );
        }
    }
    return i1 + i2;
}

```

Abbruch-  
kriterium

rekursiver Aufruf

Reduktion

## Rekursive Algorithmen

Ein rekursiver Algorithmus

ist dadurch gekennzeichnet, dass eine Lösung für eine Problemsituation durch erneutes Anwenden des gleichen Algorithmus auf eine etwas veränderte Situation ermittelt wird (*rekursiver Aufruf*).

Ein Ende der Ausführung wird dadurch sichergestellt, dass für ausgezeichnete Situationen die Lösungen direkt ohne rekursiven Aufruf bestimmt werden können und die rekursiven Aufrufe letztlich immer zu einer solchen Situation hinführen.

## Backtracking-Algorithmus

Ein Backtracking-Algorithmus

ist ein rekursiver Algorithmus,  
bei dem die Schachtelung der rekursiven Aufrufe genutzt wird,  
um einen Zwischenzustand der Lösungsentwicklung abzuspeichern,  
um gegebenenfalls zu diesem Zwischenzustand zurückkehren zu können und  
eine andere Fortsetzung der Lösungsentwicklung vorzunehmen.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 6.1. Datenkompression – Algorithmus zur Huffman-Codierung

Dr. Stefan Dissmann  
Fakultät für Informatik



## Lernziele des Kapitels 6. Datenkompression

Nach Durcharbeiten des Kapitels Datenkompression sollen die teilnehmenden Studierenden

- Datenstrukturen aus Objekten konzipieren und implementieren können,
- rekursive Datenstrukturen* kennen,
- rekursive Datenstrukturen entwerfen und implementieren können,
- die Datenstruktur *binärer Baum* kennen, einsetzen und erweitern können,
- den Algorithmus der *Huffman-Codierung* kennen.

# Datenkompression – Motivation

## aktuelles, alltagsrelevantes Thema

**Bereiche**

- Filme: *mp4*-Format (H.264)
- Musik: *mp3*-Format, *vorbis*-Format, *flac*-Format
- Bilder: *jpeg*-Format, *png*-Format
- Texte: *zip*-Format, *gzip*-Format, *jar*-Format

**Beispiele****Unterschiede:**

- verlustfreie* Kompression
  - = Dekompression führt wieder zu den Originaldaten
    - (in der Regel bei Texten, Programmtexten, ausführbarem Code notwendig)

# Datenkompression – Motivation

## aktuelles, alltagsrelevantes Thema

**Bereiche**

- Filme: *mp4*-Format (H.264)
- Musik: *mp3*-Format, *vorbis*-Format, *flac*-Format
- Bilder: *jpeg*-Format, *png*-Format
- Texte: *zip*-Format, *gzip*-Format, *jar*-Format

**Beispiele**

## Unterschiede:

- verlustfreie* Kompression
  - = Dekompression führt wieder zu den Originaldaten  
(in der Regel bei Texten, Programmtexten, ausführbarem Code notwendig)
- verlustbehaftete* Kompression
  - = Dekompression kann nicht die Originaldaten herstellen  
(bei Filmen, Musik, Bildern möglich, da auf Details eventuell verzichtet werden kann:  
die Kompression erfolgt spezifisch für den Anwendungsbereich)

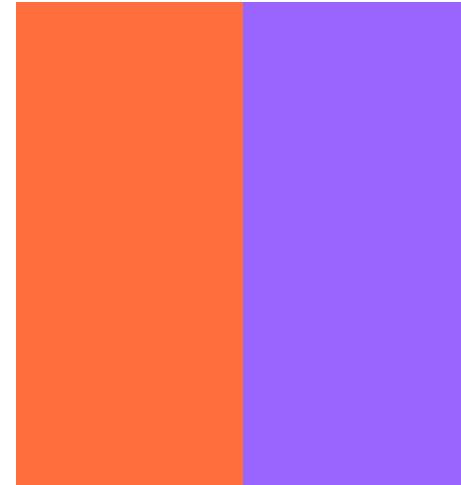
## Datenkompression – Motivation

(Fortsetzung)

Beispiel:

*True Color* ermöglicht die Unterscheidung von etwa 16,78 Millionen Farben auf der Basis von je 256 Abstufungen für die Farben Rot, Grün und Blau.

- Im folgenden Bild tauchen aber nur 2 Farben auf.
- Soll genau dieses Bild gespeichert werden, muss offensichtlich nicht für jedes Pixel eine 24-Bit-Farbinformation abgelegt werden.



## Datenkompression – Motivation

(Fortsetzung)

Beispiel:

**ASCII** ermöglicht die Unterscheidung von 256 Zeichen.

- Im folgenden Text tauchen aber nur 2 Zeichen auf.
- Soll genau dieser Text gespeichert werden, muss offensichtlich nicht für jedes Zeichen eine 8-Bit-Information abgelegt werden.

qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqxxxxxxxxxxxxxxxxxxxxxx

## Datenkompression – Motivation

(Fortsetzung)

Hier wird als Beispiel betrachtet:

### **verlustfreie Kompression von Textdaten**

(der vorgestellte Algorithmus wird aber als Ergänzung  
auch in verlustbehafteten Verfahren eingesetzt)

- ❑ Wann kann verlustfreie Kompression vorgenommen werden?

Wenn die Originaldaten überflüssige Informationen enthalten oder  
das Speichern auf Einheiten basiert, die unabhängig von den vorliegenden Informationen sind.

- ❑ Wie kann verlustfreie Kompression vorgenommen werden?

Indem genau die redundanten Informationen entfernt werden oder  
die Speicherungsform an den Bedarf der vorliegenden Information geknüpft wird.

## Datenkompression – Motivation

(Fortsetzung)

Beispiel:

- ❑ Dateien mit dem Beispiel-Java-Code zu dieser Vorlesung bestehen aus
  - 52 (normalen) Buchstaben
  - 10 Ziffern
  - ca. 30 Sonderzeichenalso ca. 100 verschiedenen Zeichen
- ❑ Dateien werden in den meisten Betriebssystemen so abgespeichert, dass 1 Zeichen in 1 Byte abgelegt wird (z.B. nach ISO Latin 1 oder UTF-8 \*).
  - 1 Byte entspricht 8 **Bit**, also einer 8-stelligen Binärzahl.
  - 1 Byte kann daher 256 Werte annehmen.
- ❑ einfache Analyse:  
Es ist offensichtlich, dass die Beispiel-Java-Programme nur etwa 40% der mit einem Byte darstellbaren Informationen benötigen.  
*aber*: Verkürzen der Speichergröße um 1 Bit (=12,5%) verkürzt die darstellbaren Werte um 50%.

\*) vergleiche Vorlesung Rechnerstrukturen

## Datenkompression – Motivation

(Fortsetzung)

Ziel der hier vorgestellten Kompression:

Reduktion des benötigten Speicherplatzes für einen *festen Inhalt* auf das notwendige *Minimum*.

Beispiel:

Kommen in einem Text nur die Buchstaben a, b, c, d, e vor,  
dann müssen auch nur fünf Zeichen unterschieden werden.

Denkbar wäre also eine *Kodierung* derart,  
dass maximal 2 Bit je Buchstabe vorgesehen werden:

$$a = (0)_2 \quad b = (1)_2 \quad c = (00)_2 \quad d = (01)_2 \quad e = (10)_2$$

## Datenkompression – Motivation

(Fortsetzung)

Ziel der hier vorgestellten Kompression:

Reduktion des benötigten Speicherplatzes für einen *festen Inhalt* auf das notwendige *Minimum*.

Beispiel:

Kommen in einem Text nur die Buchstaben a, b, c, d, e vor,  
dann müssen auch nur fünf Zeichen unterschieden werden.

Denkbar wäre also eine *Kodierung* derart,  
dass maximal 2 Bit je Buchstabe vorgesehen werden:

$$a = (0)_2 \quad b = (1)_2 \quad c = (00)_2 \quad d = (01)_2 \quad e = (10)_2$$

Diese Kodierung hat jedoch zwei Nachteile:

- ❑ Bei der Dekompression ist unklar, ob  $(01)_2$  zu ab oder zu d expandiert werden soll.  
Bei der Kompression muss also auch die Möglichkeit einer *eindeutigen Dekompression* berücksichtigt werden.
- ❑ Wenn beispielsweise e wesentlich häufiger als a im Text vorkommt,  
dann ist die Kompression mit der oben vorgestellten Kodierung *nicht* minimal.  
Die *Häufigkeit* des Auftretens eines Buchstabens muss also in die Kodierung eingehen.

## Datenkompression – Motivation

(Fortsetzung)

- Sei  $\Sigma_t$  die Menge der Zeichen, die in einem zu komprimierenden Text  $t$  vorkommen.  
Für das Beispiel gilt:  $\Sigma_t = \{ a, b, c, d, e \}$  für einen Text  $t$ .
- Dann ist  $\Sigma_t^*$  die Menge aller beliebig langen Folgen aus diesen Zeichen.  
Also gilt:  $aabcdab \in \Sigma_t^*$  oder  $cccea \in \Sigma_t^*$  oder  $t \in \Sigma_t^*$
- Gesucht wird eine Abbildung  $c: \Sigma_t \rightarrow \{0, 1\}^*$  (Kodierung der einzelnen Buchstaben)  
so dass die Abbildung  $c^*: \Sigma_t^* \rightarrow \{0, 1\}^*$  (Kodierung des ganzen Textes)  
ein *möglichst kurzes Bild*  $c^*(t)$  besitzt. (Kodierung ist spezifisch für Text  $t$ )

## Datenkompression – Motivation

(Fortsetzung)

- Sei  $\Sigma_t$  die Menge der Zeichen, die in einem zu komprimierenden Text  $t$  vorkommen.  
Für das Beispiel gilt:  $\Sigma_t = \{ a, b, c, d, e \}$  für einen Text  $t$ .
- Dann ist  $\Sigma_t^*$  die Menge aller beliebig langen Folgen aus diesen Zeichen.  
Also gilt:  $aabcdab \in \Sigma_t^*$  oder  $cccea \in \Sigma_t^*$  oder  $t \in \Sigma_t^*$
- Gesucht wird eine Abbildung  $c: \Sigma_t \rightarrow \{0, 1\}^*$  (Kodierung der einzelnen Buchstaben)  
so dass die Abbildung  $c^*: \Sigma_t^* \rightarrow \{0, 1\}^*$  (Kodierung des ganzen Textes)  
ein *möglichst kurzes Bild*  $c^*(t)$  besitzt. (Kodierung ist spezifisch für Text  $t$ )
- Um die eindeutige Dekompression zu gewährleisten,  
muss das Bild von  $c$  *präfixfrei* sein, d.h.  
für beliebige  $x_1 \in \Sigma_t, x_2 \in \Sigma_t$  darf es kein  $b \in \{0, 1\}^*$  geben mit  $c(x_1) = c(x_2)b$   
umgangssprachlich: Keine Kodierung eines Zeichens darf  
den Anfang der Kodierung eines anderen Zeichens bilden.  
(Beispielsweise müssen auch Telefonnummern präfixfrei sein.)

## Datenkompression – Motivation

(Fortsetzung)

Beispiel mit Text  $t$  mit  $\Sigma_t = \{ a, b, c, d, e \}$

Dann sind aufgrund der Forderung nach Präfixfreiheit folgende Bildmengen für Kodierungen denkbar:

- $c_1$  mit den folgenden Elementen im Bildbereich:
  - $(0)_2$  Präfixfreiheit verbietet weitere Nutzung von 0 als Anfang
  - $(10)_2$  Präfixfreiheit verbietet weitere Nutzung von 10 als Anfang
  - $(110)_2$  Präfixfreiheit verbietet weitere Nutzung von 110 als Anfang
  - $(1110)_2$  Präfixfreiheit verbietet weitere Nutzung von 1110 als Anfang
  - $(1111)_2$
- $c_2$  mit den folgenden Elementen im Bildbereich:
  - $(00)_2$  Präfixfreiheit verbietet weitere Nutzung von 00 als Anfang
  - $(01)_2$  Präfixfreiheit verbietet weitere Nutzung von 01 als Anfang
  - $(10)_2$  Präfixfreiheit verbietet weitere Nutzung von 10 als Anfang
  - $(110)_2$  Präfixfreiheit verbietet weitere Nutzung von 110 als Anfang
  - $(111)_2$
- Weitere Kodierungsabbildungen sind entweder bis auf die 0-1-Kodierung analog oder besitzen für jedes  $x \in \Sigma$ , längere Einzelkodierungen, so dass  $c^*$  zu einer längeren Gesamtkodierung führen muss.
- *Feststellung:* Die Auswahl und die Zuordnung der Bilder zu den  $x \in \Sigma_t$  muss sich aus der Häufigkeit ergeben, mit denen die einzelnen  $x$  auftreten.

## Datenkompression – Motivation

(Fortsetzung)

Beispiel mit Text  $t$  mit  $\Sigma_t = \{ a, b, c, d, e \}$

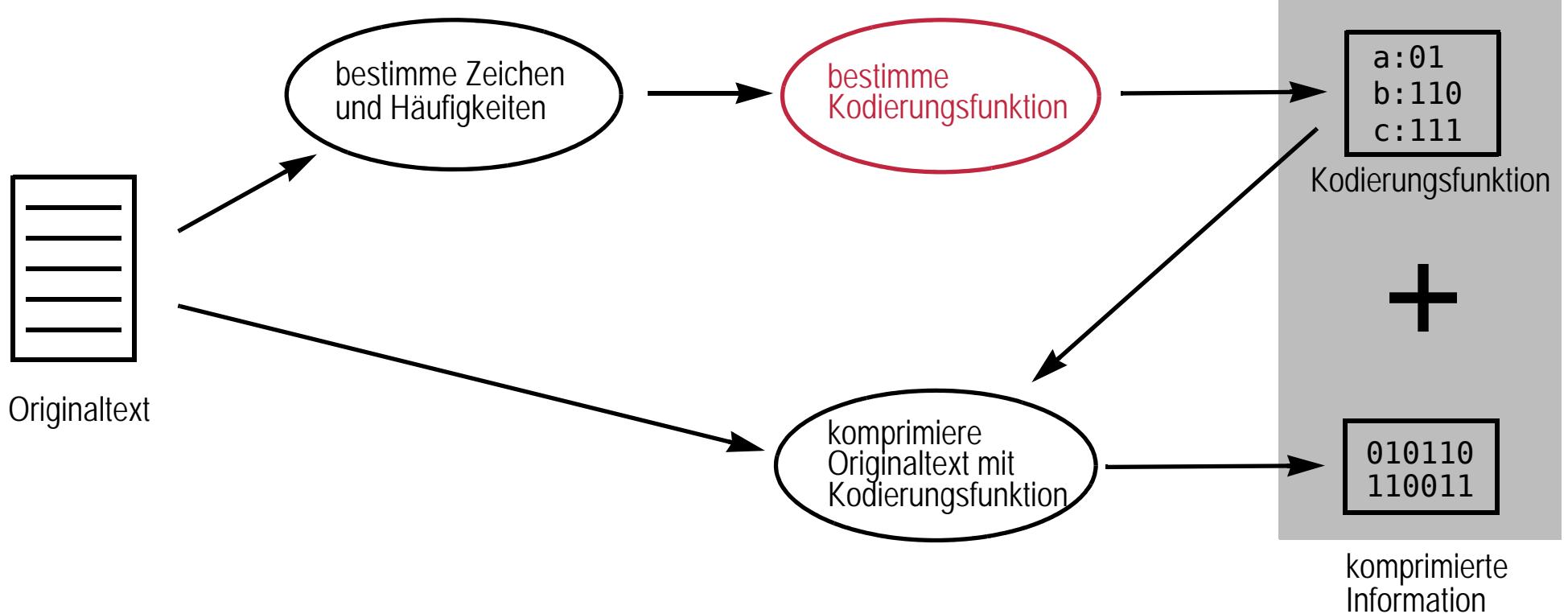
<i>Zeichen</i>	a	b	c	d	e	
<i>Häufigkeit</i>	30	20	10	15	80	
						<i>benötigte Bits</i>
$c_1$	$(10)_2$	$(110)_2$	$(1110)_2$	$(1111)_2$	$(0)_2$	300
$c_2$	$(00)_2$	$(01)_2$	$(110)_2$	$(111)_2$	$(10)_2$	335

- Dieses überschaubare Beispiel hat nur zwei sinnvolle Lösungsmöglichkeiten.  
Hier kann die Entscheidung leicht durch Ausprobieren getroffen werden.  
In der Realität existieren jedoch bei mehr Zeichen sehr viel mehr Lösungsmöglichkeiten.
- Begründung für die Auswahl der Codierung  $c_1$ :  
Die Kodierung mit der Länge 1 für e spart mehr Bits ein  
als die dafür notwendigen Verlängerungen der Kodierungen für b, c und d kosten.

## Datenkompression – Motivation

(Fortsetzung)

Ablauf einer Textkompression:



## Datenkompression – Algorithmus von Huffman

Der Algorithmus von David A. Huffman (aus dem Jahre 1952) bestimmt *konstruktiv* die Kodierungsfunktion  $c$ .

Konstruktionsidee:

Aufbau eines *binären Baums* derart, dass  
die in einem Text vorkommenden Zeichen die Blätter bilden,  
die selten vorkommenden Zeichen an langen Ästen platziert werden und  
die häufig vorkommenden Zeichen an kurzen Ästen platziert werden.

Dann lässt sich die Kodierung an den Pfaden im Baum ablesen.

## Datenkompression – Algorithmus von Huffman

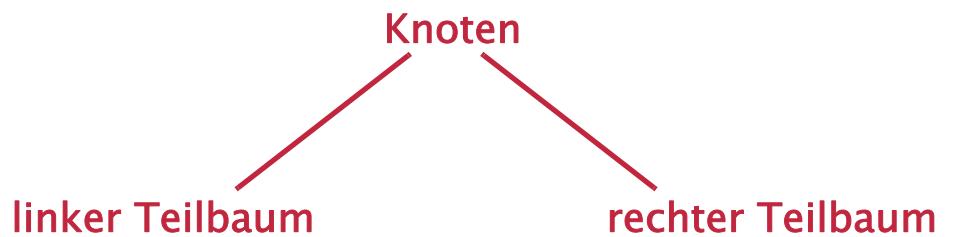
(Fortsetzung)

(rekursive) Definition:

Ein *binärer Baum* ist

- entweder ein leerer Baum
- oder besteht aus einem Knoten, der einen linken und einen rechten Teilbaum besitzt, die wieder binäre Bäume sind.

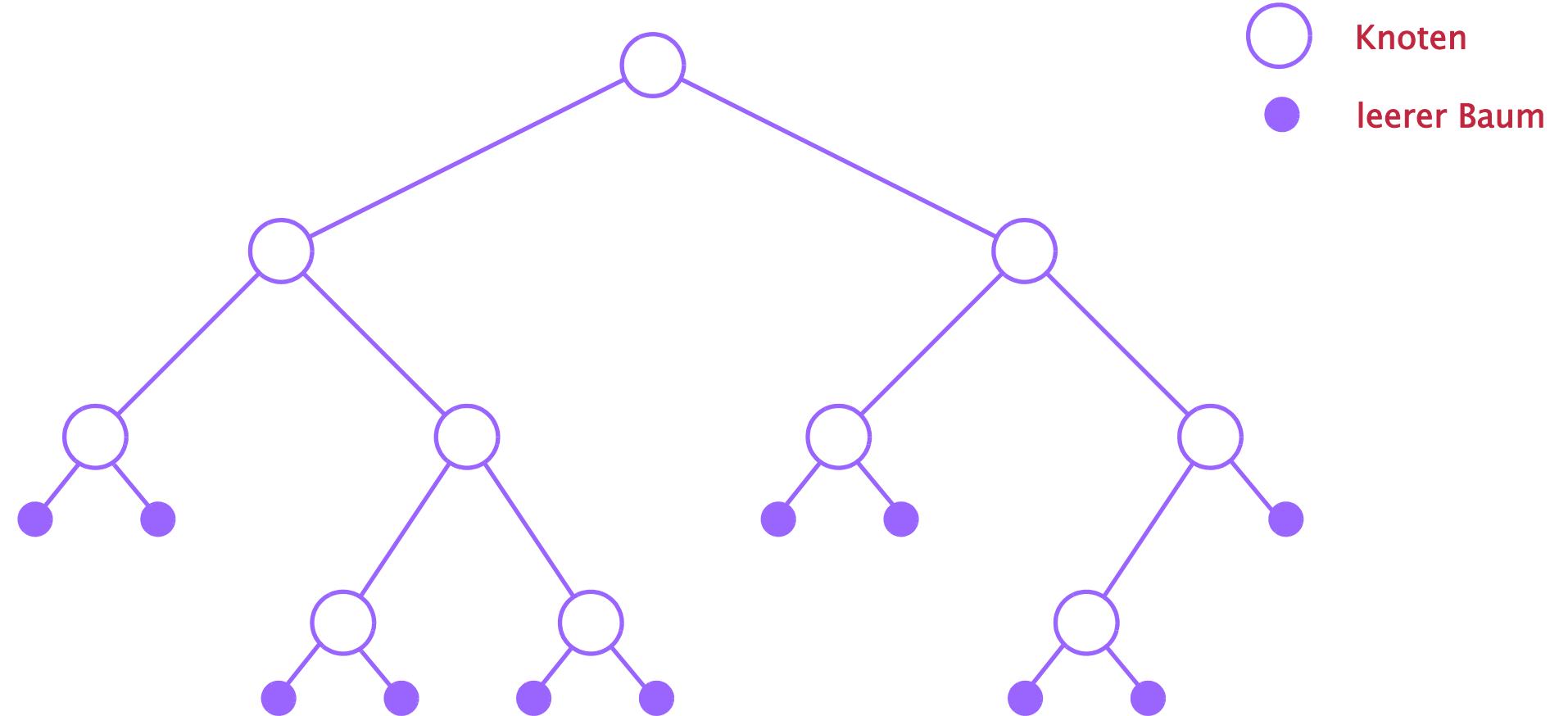
leerer Baum



## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

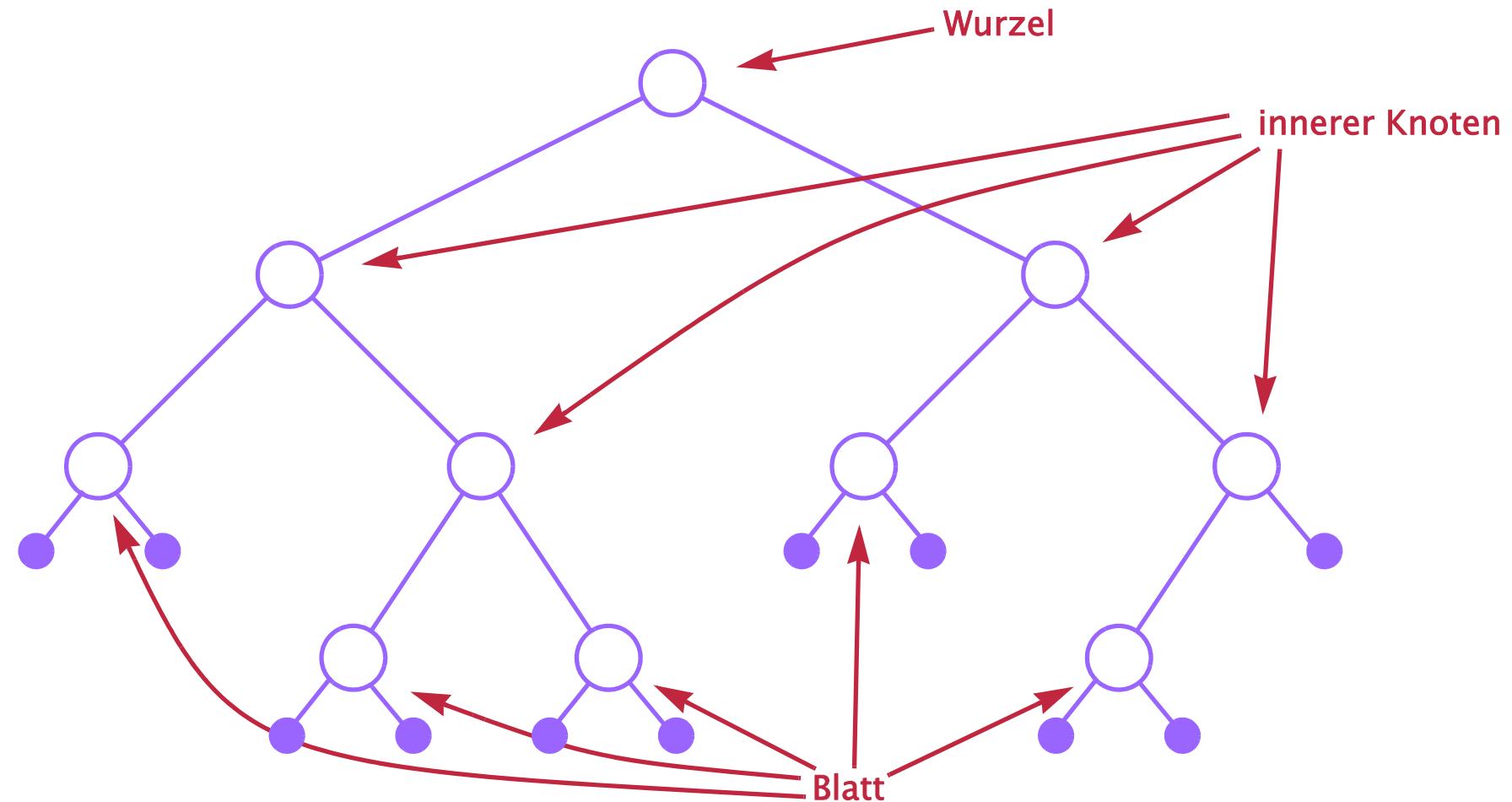
binärer Baum – Beispiel



## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

binärer Baum – **Beispiel** und **Terminologie**

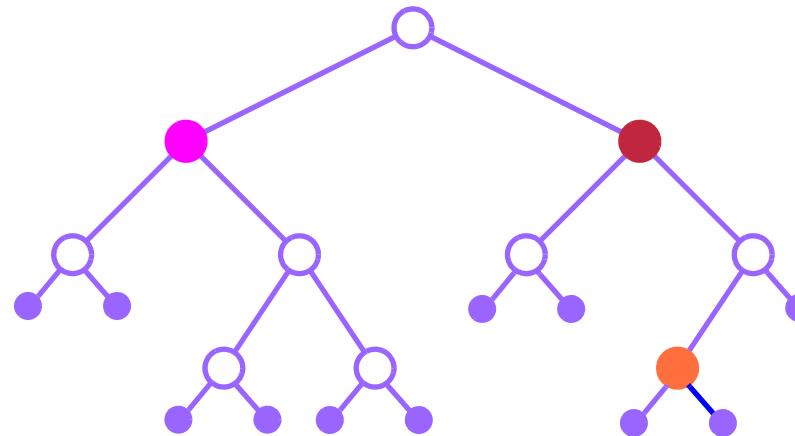


## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

binärer Baum – Terminologie:

- Die *Wurzel* (eines Baums) ist der einzige Knoten ohne Vorgänger (in diesem Baum).  
(auch übliche Bezeichnung: Wurzel eines Teilbaums, z.B. ●)
- Ein *innerer Knoten* besitzt mindestens einen nachfolgenden, nicht leeren Teilbaum.  
(Beispiel: ●)
- Ein *Blatt* ist ein Knoten mit zwei nachfolgenden, leeren Teilbäumen z.B. ●.  
(Besitzt eine Baum nur genau einen Knoten, so ist dieser zugleich Wurzel und Blatt)



## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

Konstruktionsidee:

Aufbau eines *binären Baums* derart, dass  
die in einem Text vorkommenden Zeichen die Blätter bilden,  
die selten vorkommenden Zeichen an langen Ästen platziert werden und  
die häufig vorkommenden Zeichen an kurzen Ästen platziert werden.

Dann lässt sich die Kodierung an den Pfaden durch den Baum ablesen.

Vorgehen:

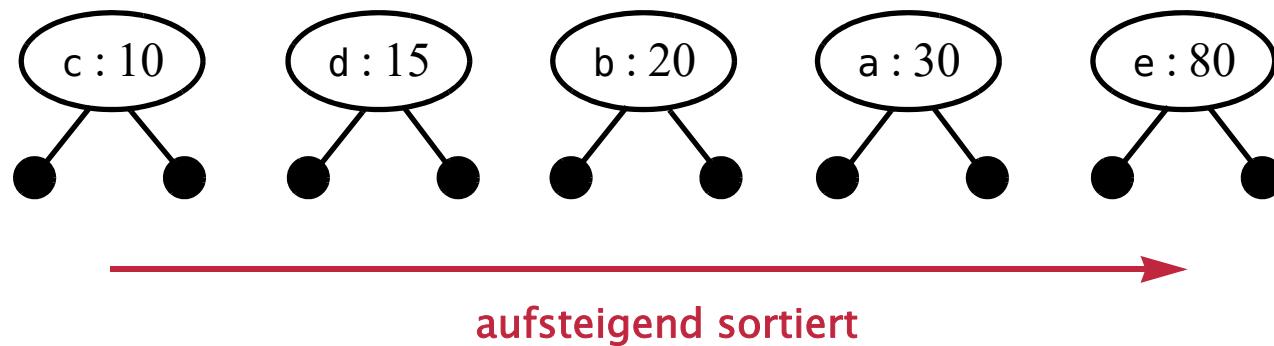
- Lege für jedes zu kodierende Zeichen einen Baum mit genau einem Knoten an.
- Sortiere die Bäume aufsteigend nach der Häufigkeit des Vorkommens.
- Fasse die beiden ersten Bäume – also die mit der geringsten Häufigkeit – unter einem neuen Wurzelknoten zusammen, dessen Häufigkeit sich als Summe der Häufigkeiten der beiden Teilbäume ergibt.
- Wiederhole dieses Vorgehen solange, bis nur noch ein Baum existiert.

## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

Beispiel:

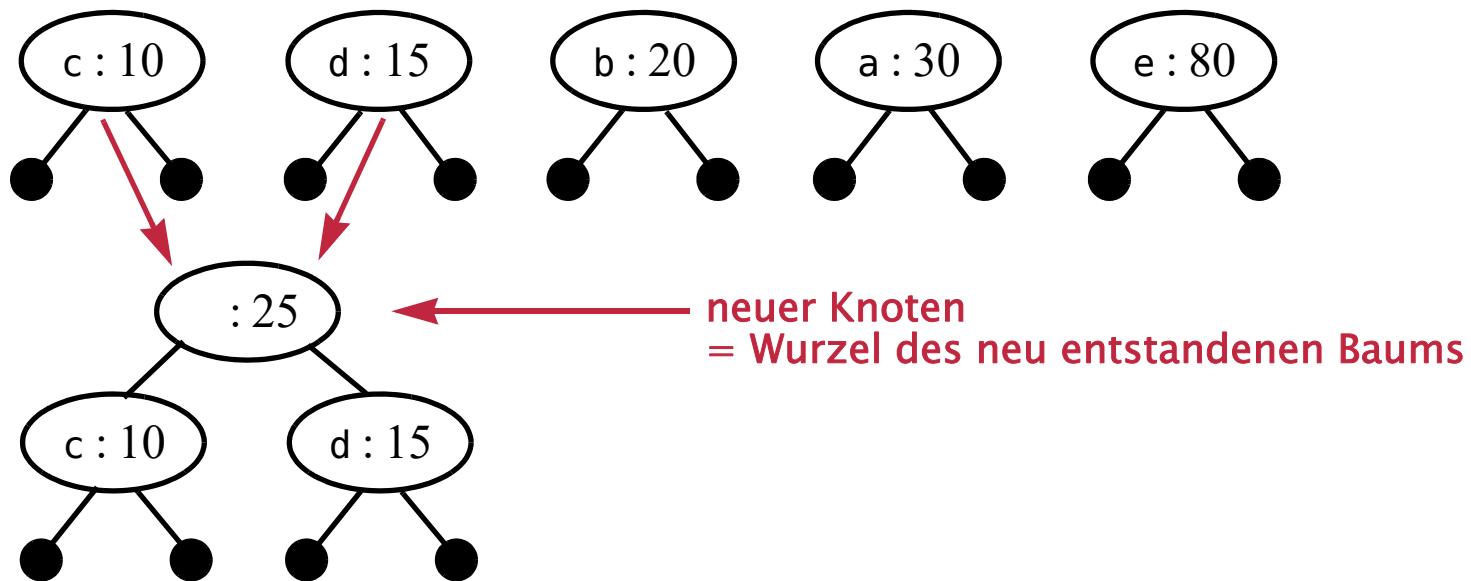
<i>Zeichen</i>	a	b	c	d	e
<i>Häufigkeit</i>	30	20	10	15	80



## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

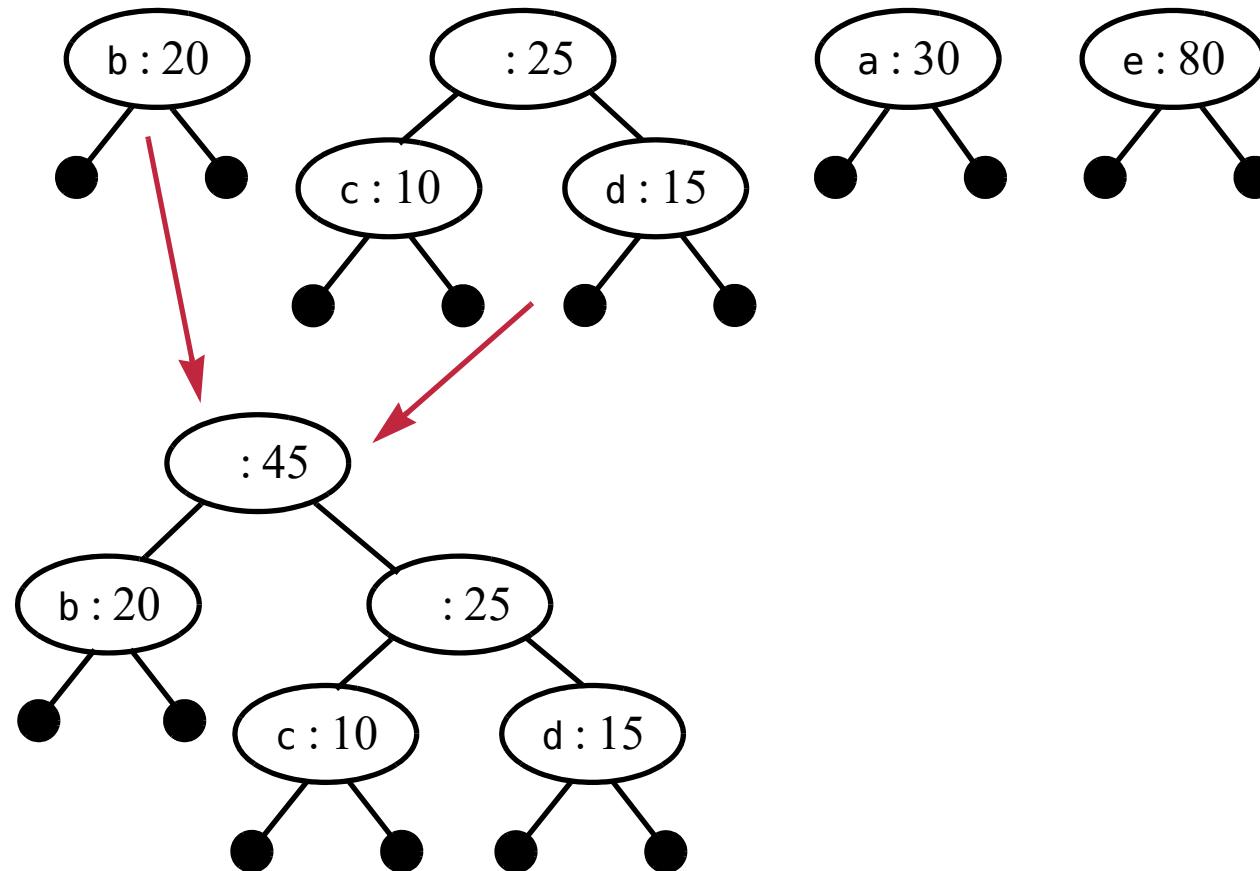
Beispiel:



## Datenkompression – Algorithmus von Huffman

Beispiel:

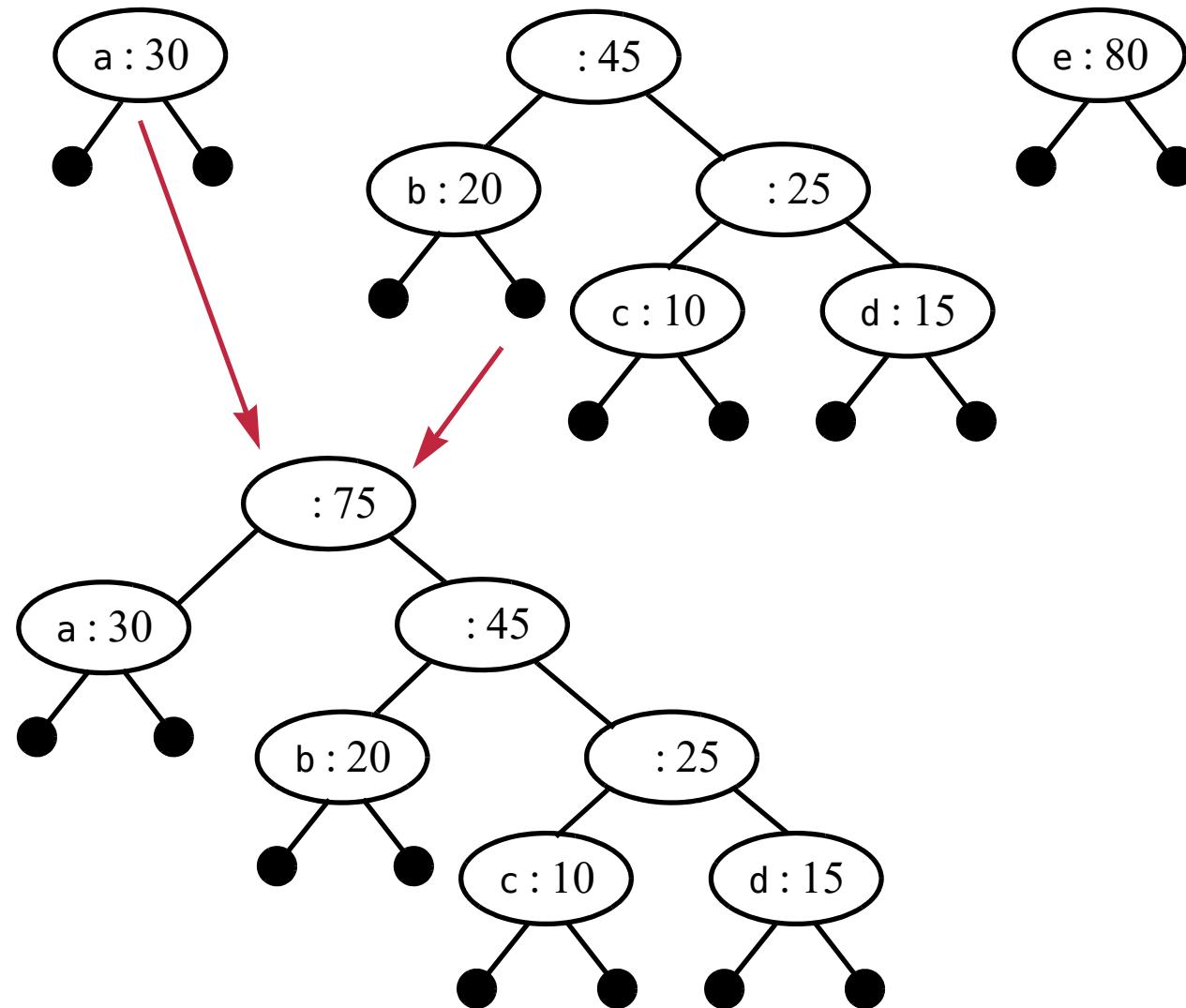
(Fortsetzung)



## Datenkompression – Algorithmus von Huffman

Beispiel:

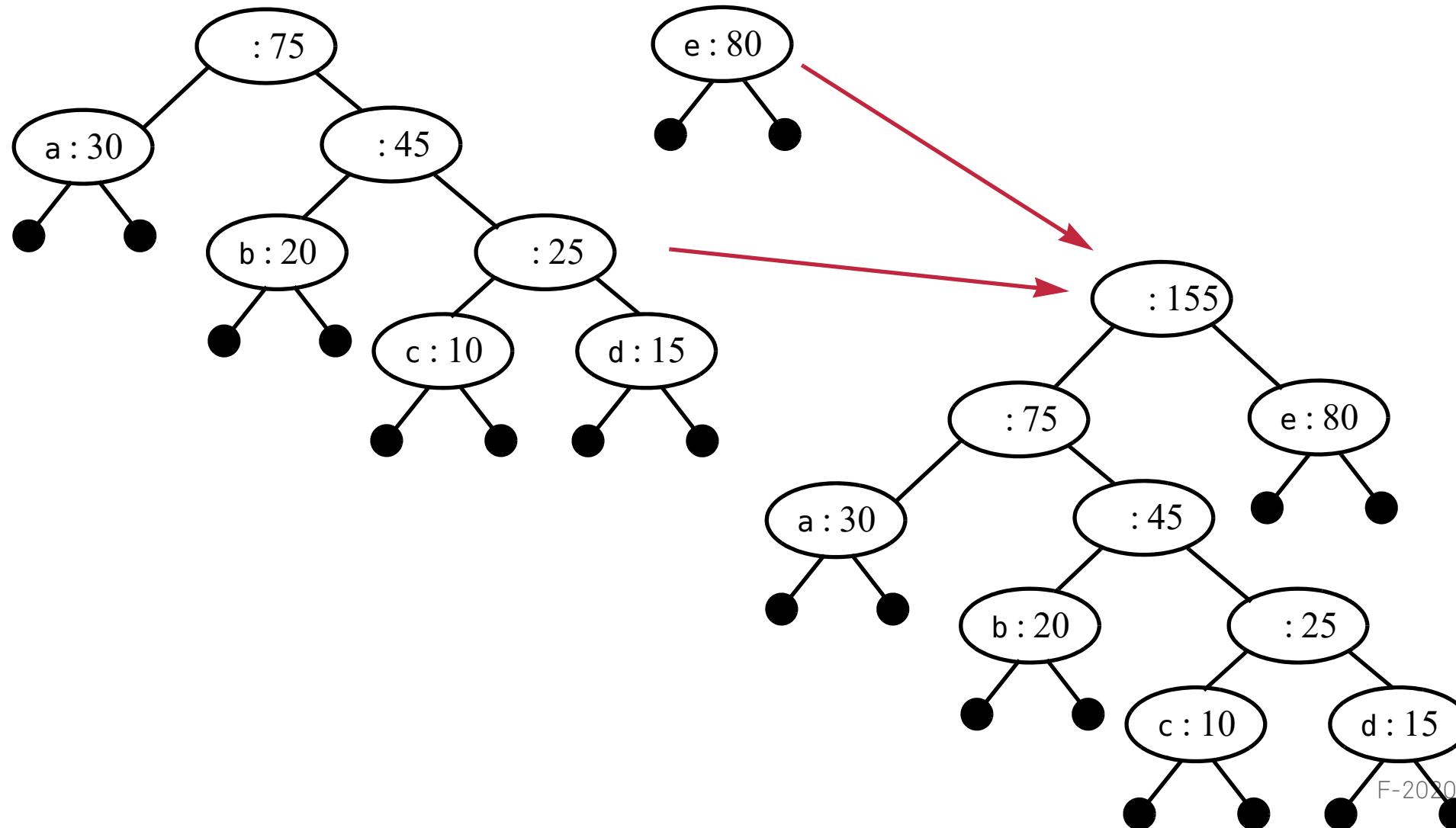
(Fortsetzung)



## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

Beispiel:



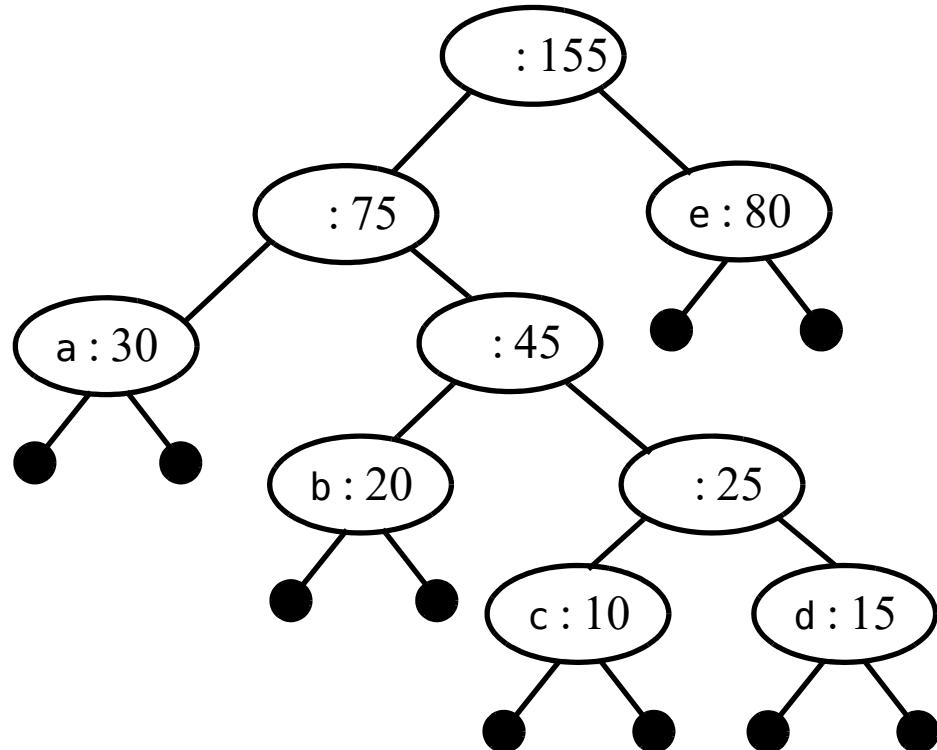
## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

Beispiel:

Ableiten der Kodierung:

- Weg zum linken Teilbaum wird als **0** kodiert.
- Weg zum rechten Teilbaum wird als **1** kodiert.
- Kodierung eines Zeichens ergibt sich aus dem Pfad von der Wurzel zu diesem Zeichen.



Zeichen	Kodierung
a	
b	
c	
d	
e	

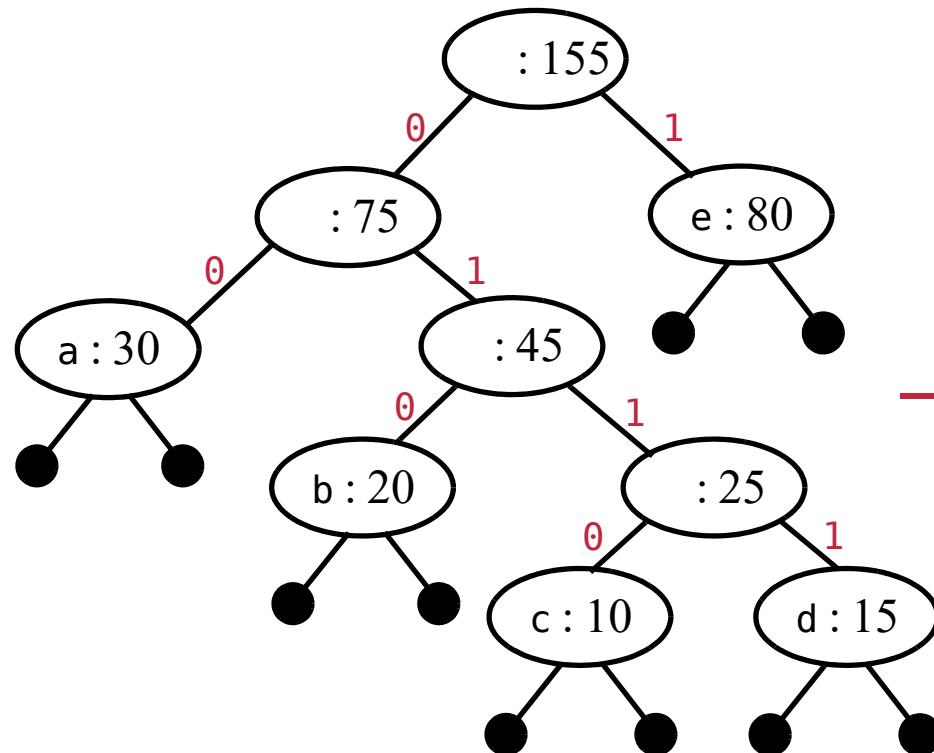
## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

Beispiel:

Ableiten der Kodierung:

- Weg zum linken Teilbaum wird als **0** kodiert.
- Weg zum rechten Teilbaum wird als **1** kodiert.
- Kodierung eines Zeichens ergibt sich aus dem Pfad von der Wurzel zu diesem Zeichen.



Zeichen	Kodierung
a	<b>00</b>
b	<b>010</b>
c	<b>0110</b>
d	<b>0111</b>
e	<b>1</b>

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## **6.2. Datenkompression – Implementierung zur Huffman-Codierung**

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-506

## Datenkompression – Algorithmus von Huffman

(Fortsetzung)

Implementierung der Datenstrukturen für den Kodierungsbaum:

- Die Klasse `HuffmanTriple` mit den für die Kompression notwendigen
  - Attributen für das Zeichen, seine Häufigkeit und seine Kodierung.
- Die Klasse `HuffmanTree` zur Repräsentation eines aus Knoten bestehenden Baums mit
  - einem Attribut für ein `HuffmanTriple`-Objekt,
  - zwei Attributen für den linken und rechten Teilbaum zum Aufbau der Baumstruktur,
  - geeigneten Konstruktoren,
  - einer Methode zur Generierung der komprimierenden Kodierung für den Baum,
  - einer Methode zur Ausgabe der Kodierung eines Baums.
- Die Klasse `HuffmanCoding` auf der Grundlage der Klasse `HuffmanTree` mit
  - einem Feld zum Speichern der schrittweise entstehenden Bäume,
  - einem geeigneten Konstruktor,
  - einer Methode zum Sortieren der Bäume,
  - einer Methoden zum Aufbau des Huffman-Baums,
  - einer Methode zur Ausgabe der Kodierung des Huffman-Baums.

## Datenkompression – Implementierung der Klasse HuffmanTriple

```
public class HuffmanTriple
{
    private char token;
    private String code;
    private int quantity;

    ...
}
```



- benötigte Konstruktoren:
  - Zeichen werden zunächst mit ihrer Häufigkeit bestimmt.
  - Die Kodierung wird später ermittelt, kann also nicht im Konstruktor gesetzt werden.
- benötigte Methoden:
  - Lesen der Attribute
  - Setzen der Kodierung
  - Methode `toString` für Ausgaben
  - Inkrementieren der Häufigkeit \*)

\*) Diese Methode wird später benötigt.

## Datenkompression – Implementierung der Klasse HuffmanTriple

(Fortsetzung)

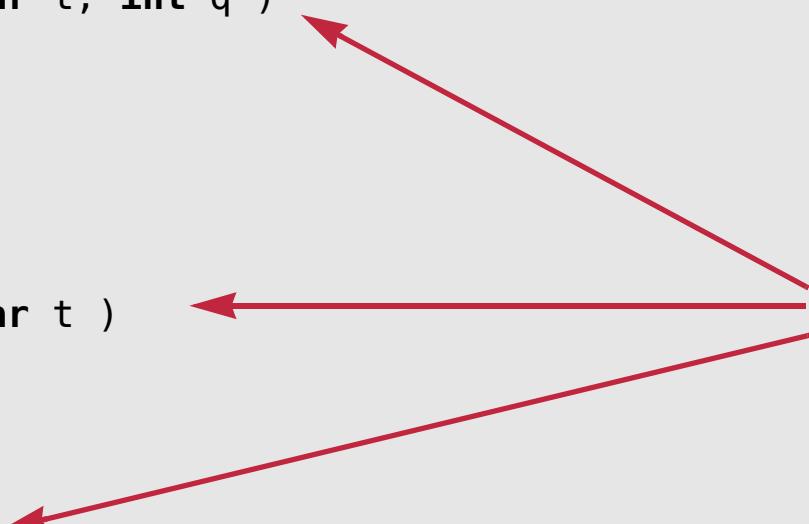
```
public class HuffmanTriple
{
    private char token;
    private String code;
    private int quantity;

    public HuffmanTriple( char t, int q )
    {
        token = t;
        code = "";
        quantity = q;
    }

    public HuffmanTriple( char t )
    {
        this( t, 1 );
    }

    public HuffmanTriple()
    {
        this( ' ', 0 );
    }

    ...
}
```



Konstruktoren

## Datenkompression – Implementierung der Klasse HuffmanTriple

(Fortsetzung)

```
public class HuffmanTriple
{
    ...
    public char getToken()
    {
        return token;
    }

    public String getCode()
    {
        return code;
    }

    public int getQuantity()
    {
        return quantity;
    }

    public void setCode( String c )
    {
        code = c;
    }
    ...
}
```

Zugriff auf die Attribute:  
get-Methoden

Setzen der Kodierung

## Datenkompression – Implementierung der Klasse HuffmanTriple

(Fortsetzung)

```
public class HuffmanTriple
{
    ...
    public void incrementQuantity()
    {
        quantity++;
    }

    public int compareTo( HuffmanTriple other )
    {
        return quantity - other.quantity;
    }

    public String toString()
    {
        return "token (quantity: " + quantity + "): " + token + " -> code: " + code ;
    }
}
```

Inkrementieren der Häufigkeit

Vergleich über Häufigkeitswert

## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

### Überlegungen zur Struktur von Huffman-Bäumen

- Jedes einzelne Zeichen bildet zunächst einen Baum mit genau einem Knoten:  
Hierfür wird ein geeigneter Konstruktor benötigt.
- Jeweils zwei Bäume werden unter einer neuen Wurzel zusammengesetzt:  
Auch hierfür wird ein geeigneter Konstruktor benötigt.
- Jeder innere Knoten eines Huffman-Baums besitzt immer genau zwei Nachfolgeknoten:  
Diese Eigenschaft kann beim Bearbeiten eines Baums ausgenutzt werden.
- Die Blätter eines Huffman-Baums enthalten die zu kodierenden Zeichen.

## Datenkompression – Implementierung der Klasse HuffmanTree

```
public class HuffmanTree
{
    private HuffmanTriple content;
    private HuffmanTree leftChild, rightChild;
    ...
}
```

Inhalt eines Knotens  
Baumstruktur

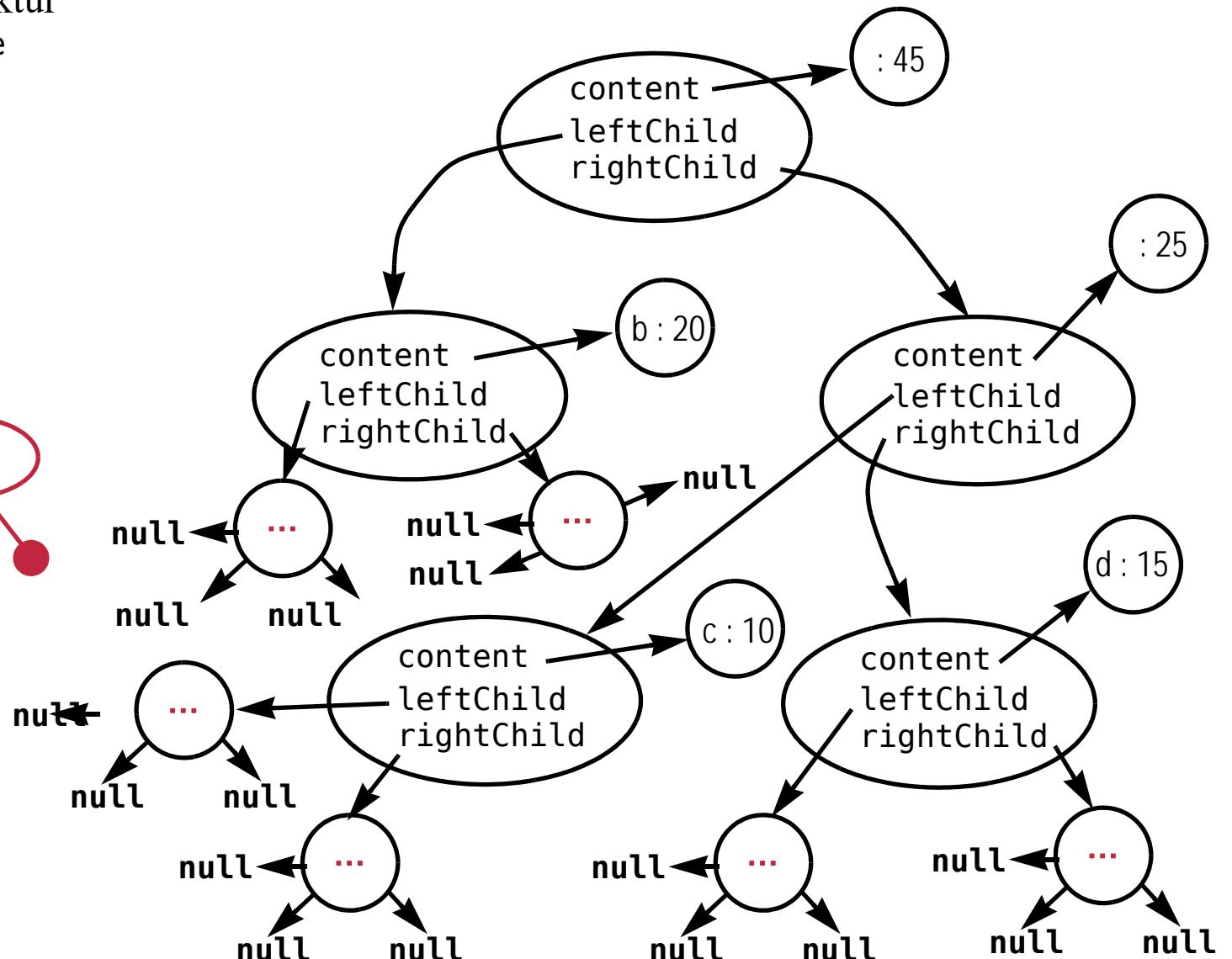
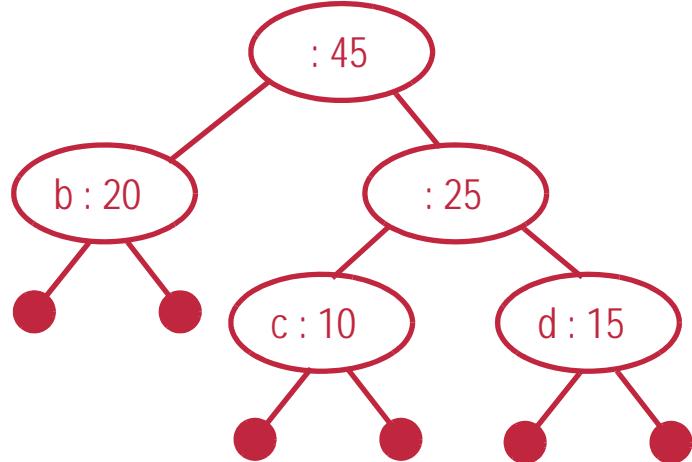
- Die Klasse HuffmanTree ist eine *rekursiv definierte Datenstruktur*, da die Deklaration der Klasse Bezug auf sich selbst nimmt.
- Jedes HuffmanTree-Objekt enthält
  - eine Referenz auf ein HuffmanTriple-Objekt,
  - eine Referenz auf ein HuffmanTree-Objekt, das den linken Nachfolger bildet (`leftChild`),
  - eine Referenz auf ein HuffmanTree-Objekt, das den rechten Nachfolger bildet (`rightChild`).
- Ein *leerer Baum* ist dadurch gekennzeichnet, dass `content` auf `null` verweist.
- Ein *Blatt* ist dadurch gekennzeichnet, dass `leftChild` und `rightChild` auf leere Bäume verweisen.
- Die Datenstruktur hat keine vorgegebene Größe, sondern kann durch das Setzen der Referenzen auf neu erzeugte HuffmanTree-Objekte vergrößert werden.

## Datenkompression – Implementierung der Klasse HuffmanTree

Beispiel für eine Objektstruktur  
mit der Klasse HuffmanTree

(Fortsetzung)

aus Beispiel:



## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

```
public class HuffmanTree
{
    ...
    public HuffmanTree()
    {
        content = null;
        leftChild = null;
        rightChild = null;
    }

    public HuffmanTree( HuffmanTriple t )
    {
        content = t;
        leftChild = new HuffmanTree();
        rightChild = new HuffmanTree();
    }
}
```

Konstruktor:  
erzeugt *leeren Baum*

Konstruktor:  
erzeugt Knoten mit Inhalt

## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

```

public class HuffmanTree
{
    ...
    public HuffmanTree( HuffmanTree lc, HuffmanTree rc ) ← Konstruktor:  

    {                                                 erzeugt Wurzel
        content = new HuffmanTriple
                    (' ', lc.getContent().getQuantity()+rc.getContent().getQuantity());
        leftChild = lc;
        rightChild = rc;
    }

    public boolean isEmpty() ← leeren Baum identifizieren
    {
        return content == null;
    }

    public boolean isLeaf() ← Blatt identifizieren
    {
        return !isEmpty() && leftChild.isEmpty() && rightChild.isEmpty();
    }

    ...
}

```

## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

```
public class HuffmanTree
{
    ...
    public HuffmanTriple getContent() { ... }
        if ( !isEmpty() )
        {
            return content;
        } else {
            throw new IllegalStateException();
        }
    }

    public int compareTo ( HuffmanTree other ) { ... }
        if ( !isEmpty() && !other.isEmpty() )
        {
            return content.compareTo( other.content );
        } else {
            throw new IllegalStateException();
        }
    }
    ...
}
```

Zugriff auf Inhalt

Ausnahmesituation

Vergleich

## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

```
public class HuffmanTree
{
    ...
    public void generateCodes() ← Erzeugen der Kodierung
    {
        if ( !isEmpty() && !isLeaf() )
        {
            leftChild.content.setCode( content.getCode() + "0" );
            rightChild.content.setCode( content.getCode() + "1" );
            leftChild.generateCodes();
            rightChild.generateCodes();
        }
    }
    ...
}
```

## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

```

public class HuffmanTree
{
    ...
    public void generateCodes()
    {
        if ( !isEmpty() && !isLeaf() )
        {
            leftChild.content.setCode( content.getCode() + "0" );
            rightChild.content.setCode( content.getCode() + "1" );
            leftChild.generateCodes();
            rightChild.generateCodes();
        }
    }
    ...
}

```

Erzeugen der Kodierung

rekursive Methode

- Die Methode `generateCodes` arbeitet *rekursiv* auf dem Baum und betrachtet immer zuerst den linken und danach den rechten nachfolgenden Teilbaum.

## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

```
public class HuffmanTree
{
    ...
    public void generateCodes()
    {
        if ( !isEmpty() && !isLeaf() )
        {
            leftChild.content.setCode( content.getCode() + "0" );
            rightChild.content.setCode( content.getCode() + "1" );
            leftChild.generateCodes();
            rightChild.generateCodes();
        }
    }
    ...
}
```

The code is annotated with red arrows and text:

- A horizontal arrow points to the first line of the method definition (`public void generateCodes()`) with the text "Erzeugen der Kodierung".
- A curved arrow points from the text "Generieren für Teilbäume" to the `leftChild.generateCodes()` and `rightChild.generateCodes()` calls.
- A curved arrow points from the text "Generieren für Teilbäume" back to the opening brace of the `if` block.

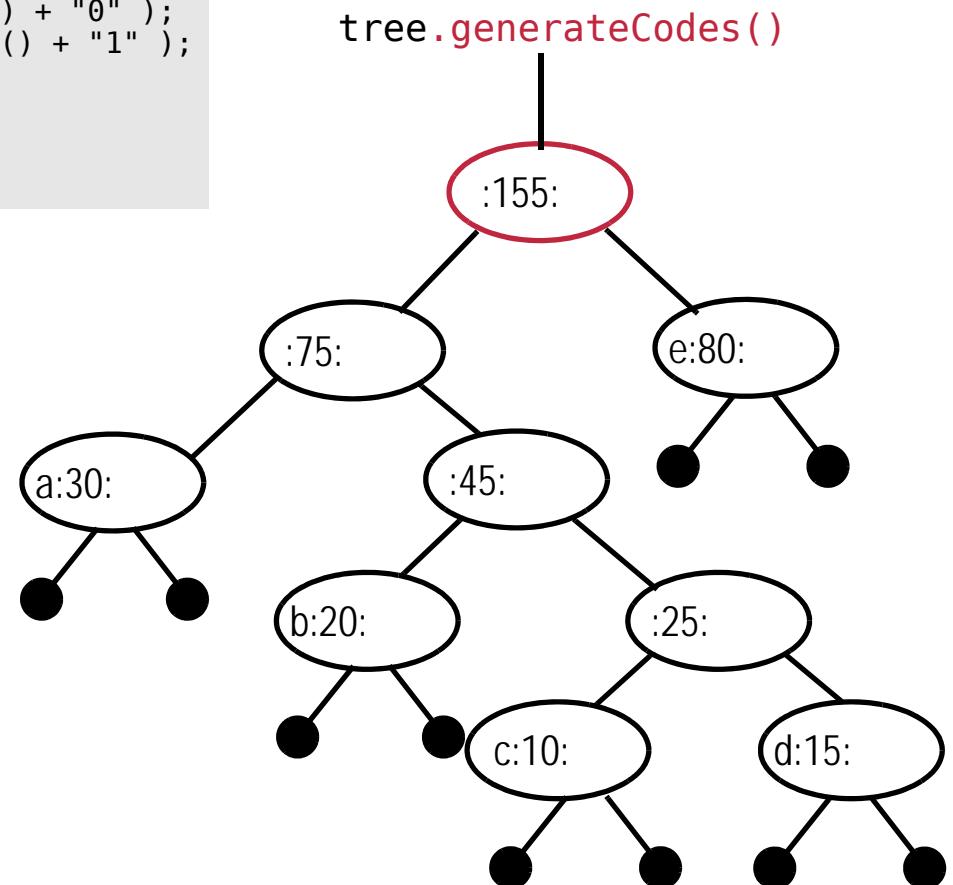
## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

```
public class HuffmanTree
{
    ...
    public void generateCodes()
    {
        if ( !isEmpty() && !isLeaf() ) ← Abbruchkriterium
        {
            leftChild.content.setCode( content.getCode() + "0" );
            rightChild.content.setCode( content.getCode() + "1" );
            leftChild.generateCodes();
            rightChild.generateCodes();
        }
    }
    ...
}
```

## Beispiel – Ausführung von generateCodes

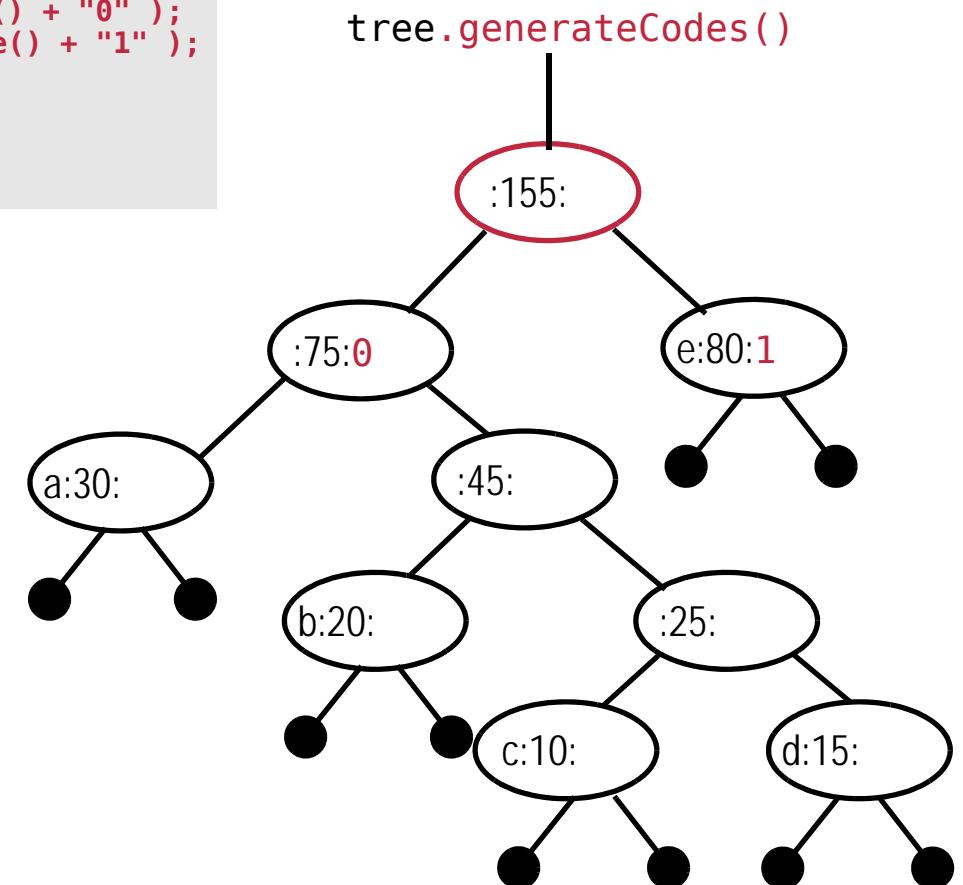
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

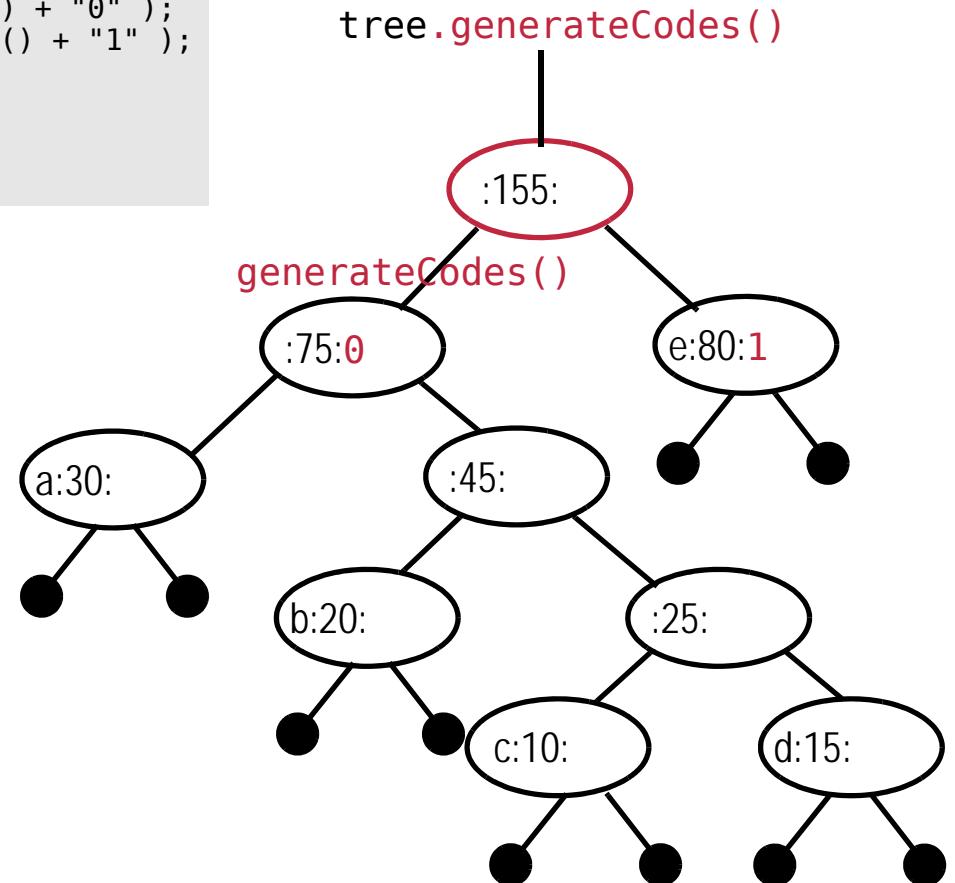
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

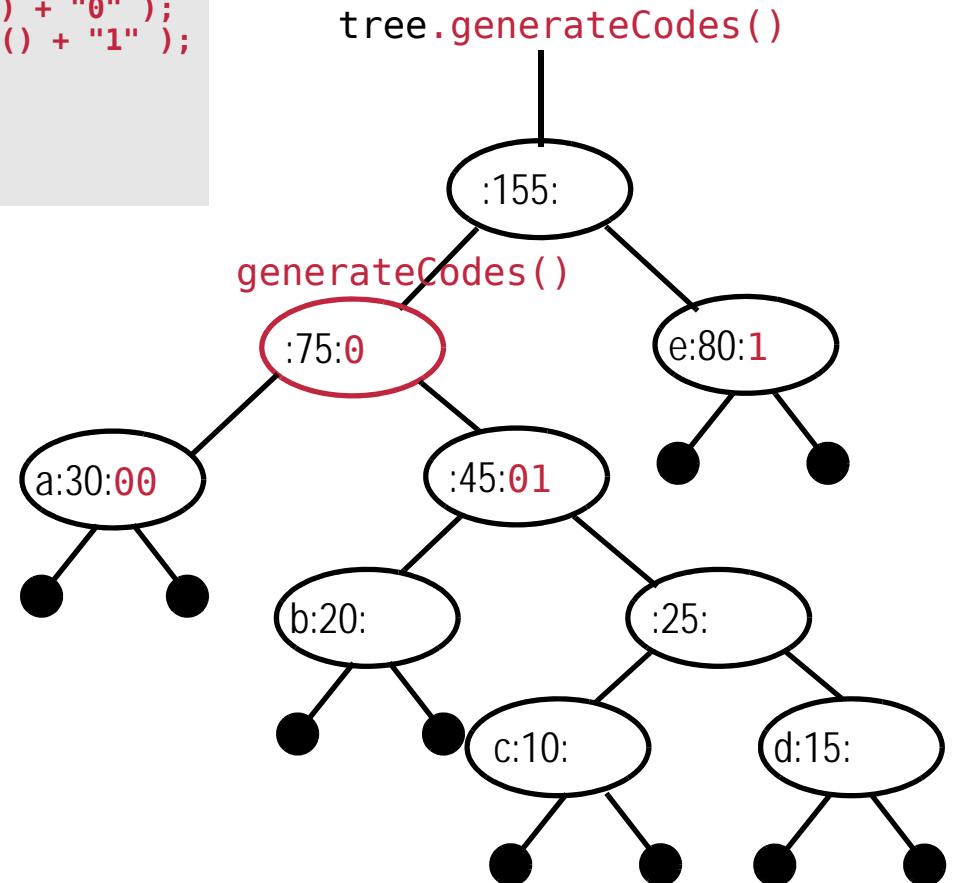
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

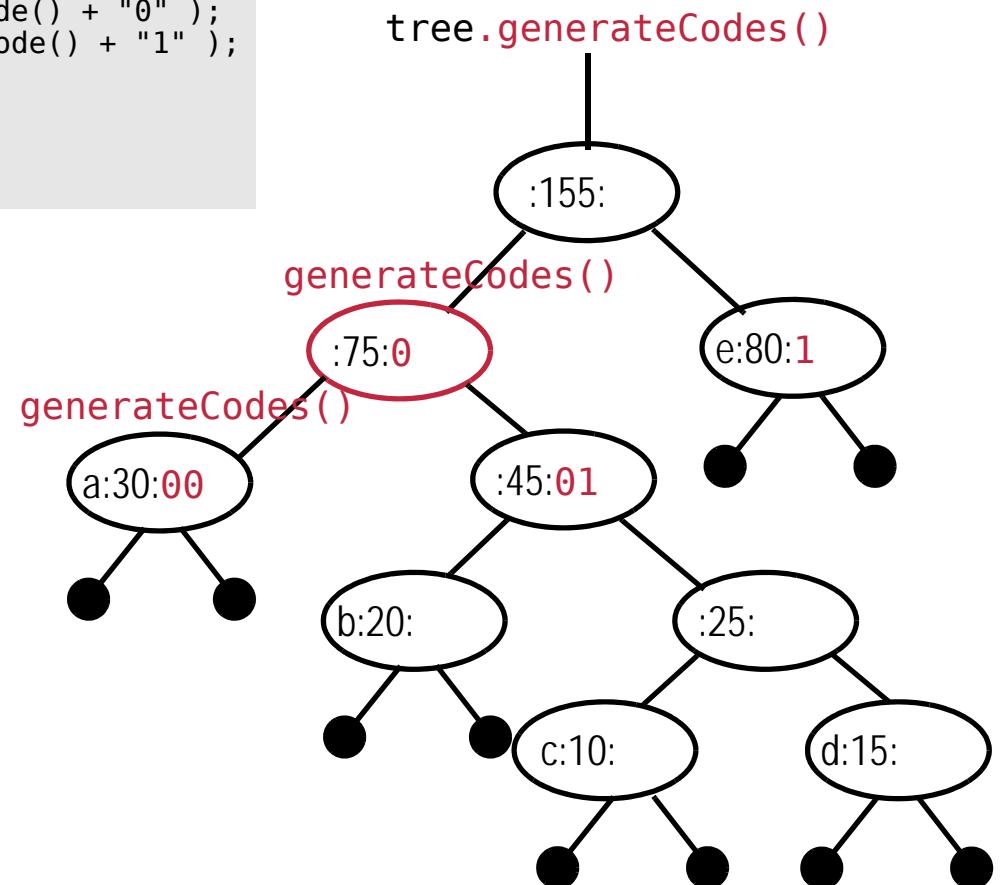
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

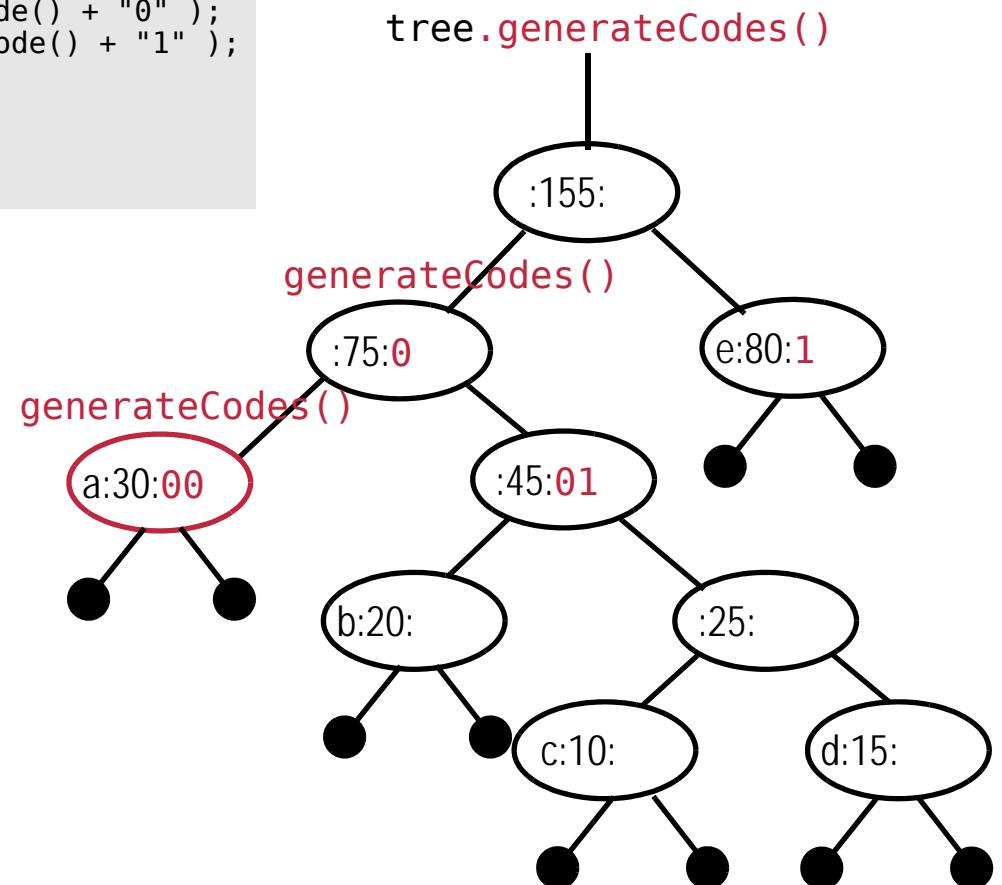
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

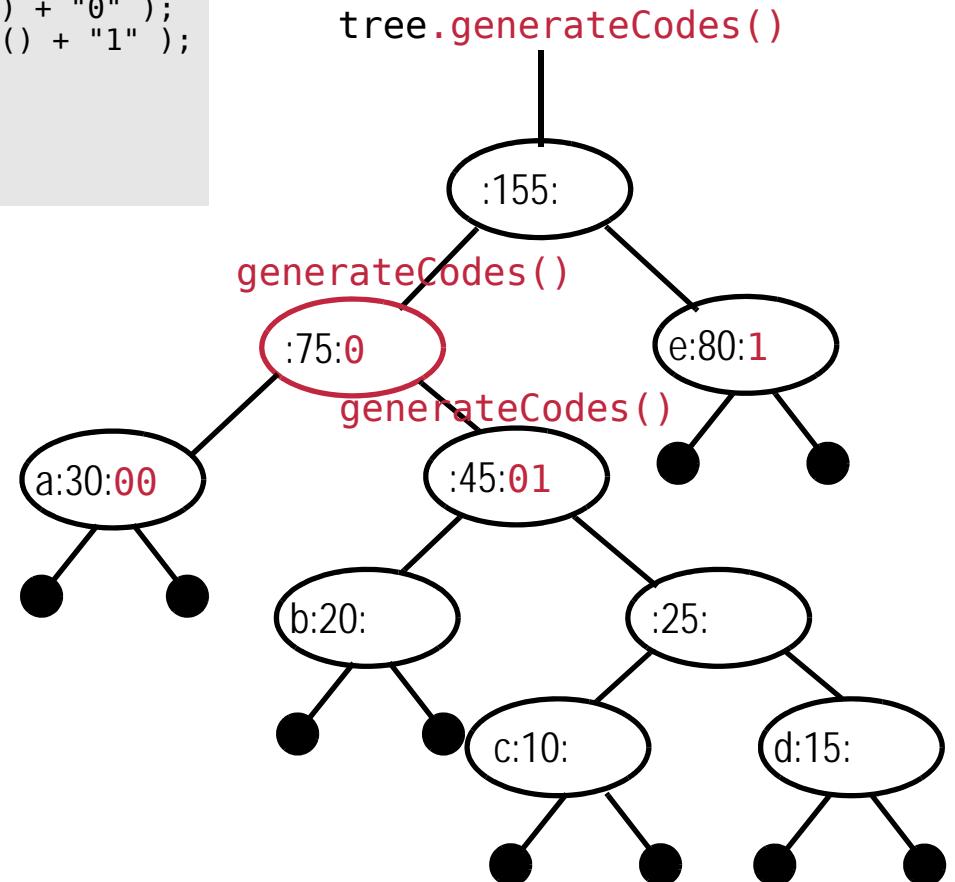
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

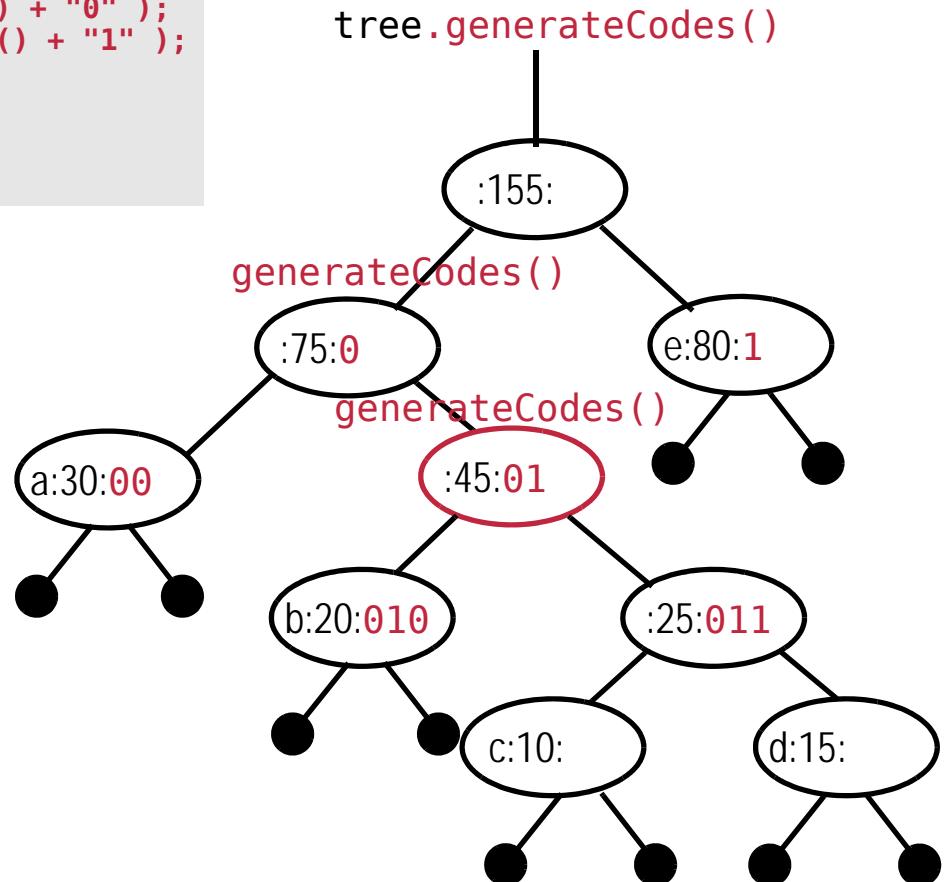
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

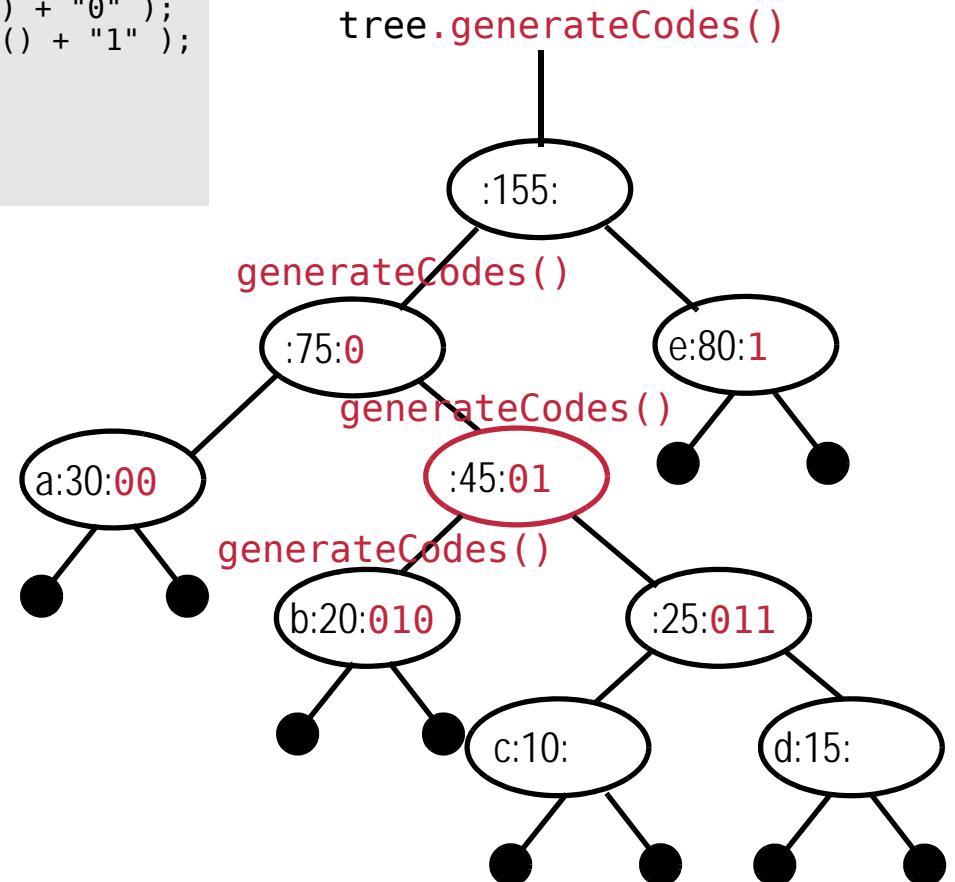
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

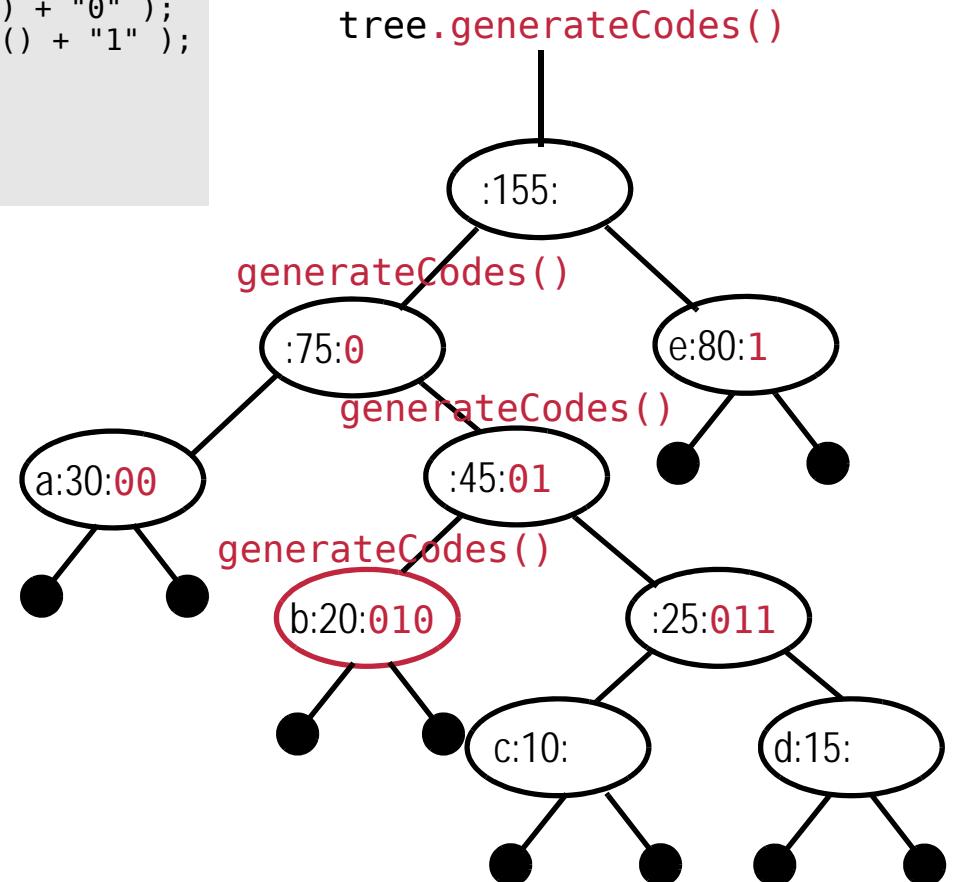
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

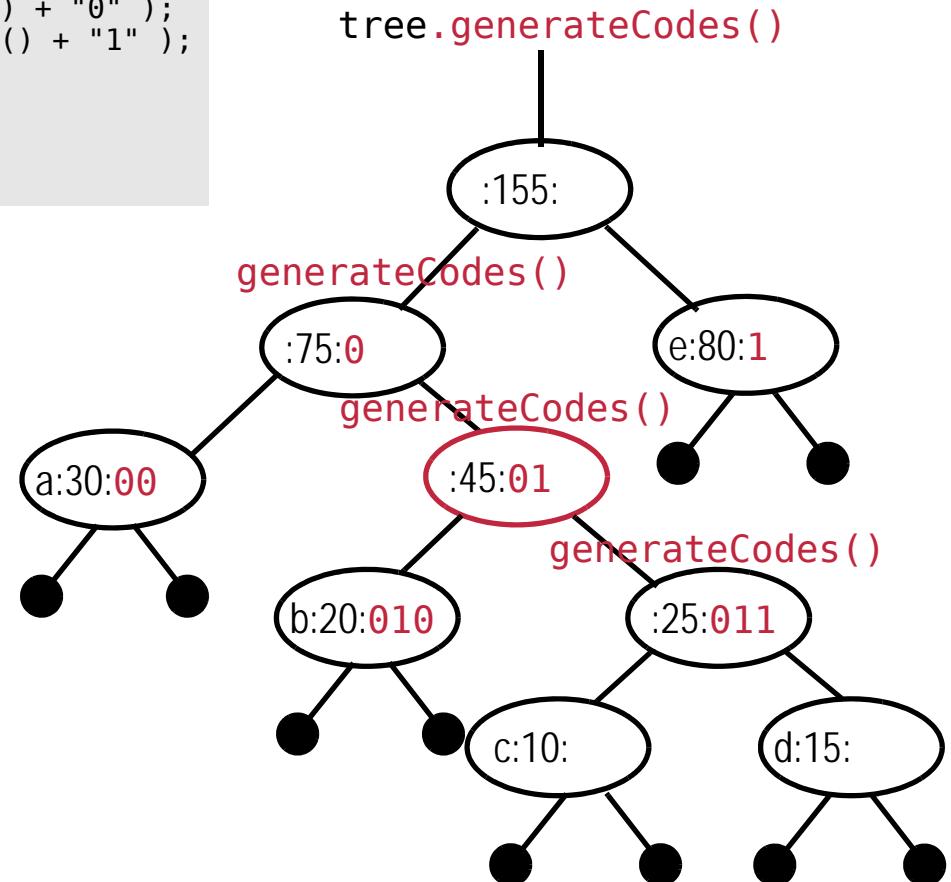
```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



## Beispiel – Ausführung von generateCodes

(Fortsetzung)

```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```



und analog fortsetzen ...

## Datenkompression – Implementierung der Klasse HuffmanTree

(Fortsetzung)

```
public class HuffmanTree
{
    ...
    public void showCodes()
    {
        if ( !isEmpty() )
        {
            if ( isLeaf() )
            {
                System.out.println( content.toString() );
            }
            else
            {
                leftChild.showCodes();
                rightChild.showCodes();
            }
        }
    }
}
```

Ausgabe der Kodierungen  
für die Blätter

rekursive Suche  
nach Blättern

*Erinnerung:* Kodierungen von Zeichen stehen nur in den Blättern des Huffman-Baums.

## Datenkompression – Implementierung der Klasse HuffmanCoding

```
public class HuffmanCoding
{
    private HuffmanTree[] trees; ← Feld mit den Wurzeln  
der Teilbäume
    ...
}
```

Methoden der Klasse HuffmanCoding:

- ❑ Konstruktor, der
  - einzelne Zeichen als Bäume anlegt,
  - die Bäume nach der Größenangabe in ihrem Wurzelknoten aufsteigend sortiert,
  - die beiden kleinsten Bäume unter einem neuen Wurzelknoten zusammenfasst,
  - diesen Vorgang solange wiederholt, bis nur noch ein Baum vorliegt, und
  - abschließend die Kodierung erzeugt.
- ❑ Methode `showCodeTable()`, die die Kodierungstabelle ausgibt.

## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

```

public class HuffmanCoding
{
    private HuffmanTree[] trees;

    public HuffmanCoding( HuffmanTriple[] input ) ← Konstruktor
    {
        // compression only if different tokens appear
        if ( input.length > 1 )
        {
            initializeTrees( input ); ← Bäume anlegen
            buildTree(); ← Bäume zusammenfassen
            trees[trees.length-1].generateCodes(); ← Kodierung eintragen
        }
        else
        {
            throw new IllegalStateException();
        }
    }
    ...
}

```

## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

```
public class HuffmanCoding
{
    private HuffmanTree[] trees;

    public HuffmanCoding( HuffmanTriple[] input )
    {
        // compression only if different tokens appear
        if ( input.length > 1 ) ←
        {
            initializeTrees( input );
            buildTree();
            trees[trees.length-1].generateCodes();
        }
        else
        {
            throw new IllegalStateException(); ←
        }
    }
    ...
}
```

Ausführung auf  
sinnvolle Situationen  
beschränken

sonst: Abbruch

## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

```
public class HuffmanCoding
{
    private HuffmanTree[] trees;

    ...

    private void initializeTrees( HuffmanTriple[] input ) ← Feld von
    {
        trees = new HuffmanTree[input.length];
        for ( int i = 0; i < input.length; i++ )
        {
            trees[i] = new HuffmanTree( input[i] ); ← Feld von Bäumen
        }
    }

    ...
}
```

- ❑ Es werden die Elemente eines Feldes von `HuffmanTriple`-Objekten als Inhalte in die Elemente eines Feldes von `HuffmanTree`-Objekten übernommen.
- ❑ Für jedes Zeichen liegt anschließend eine einelementiger Baum im Feld `trees` vor.

## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

```

public class HuffmanCoding
{
    private HuffmanTree[] trees;

    ...

    private void buildTree()
    {
        for ( int i = 0; i+1 < trees.length; i++ )
        {
            insertionSort( i );
            trees[i+1] = new HuffmanTree( trees[i], trees[i+1] );
        }
    }

    ...
}

```

Sortieren ab Index i

zwei Bäume werden zusammengefasst



- ❑ Die relevanten Bäume (ab Index *i*) werden durch `insertionSort( i )` sortiert.
- ❑ Variation des `insertionSort`-Algorithmus, bei dem das Sortieren mit dem Index *start* beginnt. So kann darauf reagiert werden, dass am Anfang des Feldes immer mehr Bäume zusammengefasst werden.

## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

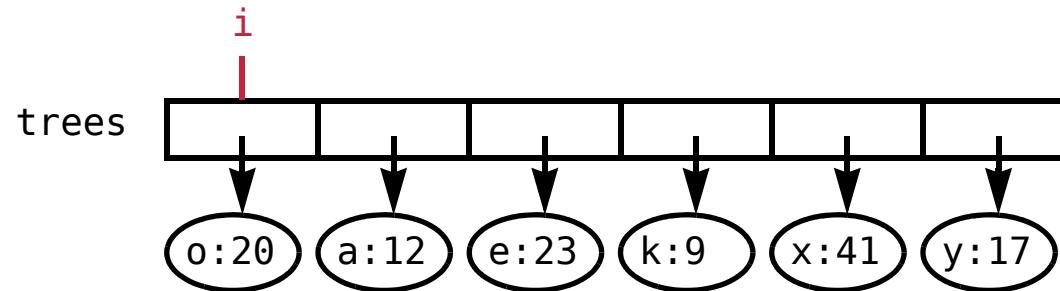
Begründung für die Wahl von *InsertionSort* für als Sortieralgorithmus:

- ❑ im ersten Durchlauf der Schleife muss eine vollständige Sortierung durchgeführt werden
- ❑ ab dem zweiten Durchlauf muss jeweils nur der neu entstandene erste Baum in die bereits sortierte Folge von Bäumen eingeordnet werden:
  - sortierte Folgen führen bei *QuickSort* zu einem überflüssigen Aufbau von rekursiven Aufrufen, die nur bereits sortierte Feldabschnitte überprüfen.
  - *SelectionSort* führt immer sehr viele Vergleiche durch,
  - *InsertionSort* erhält die bestehende Ordnung und verschiebt nur schrittweise den neu entstandenen Baum an seine Position.
- ❑ *InsertionSort* ermöglicht also mit einem Algorithmus ein erstes Sortieren und anschließend ein recht effizientes Einordnen in die sortierte Folge.

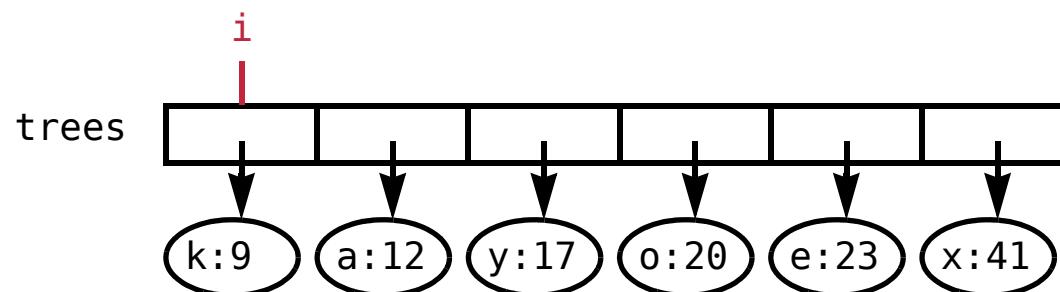
## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

Arbeitsweise der Methode `buildTree()` – Ausgangssituation:



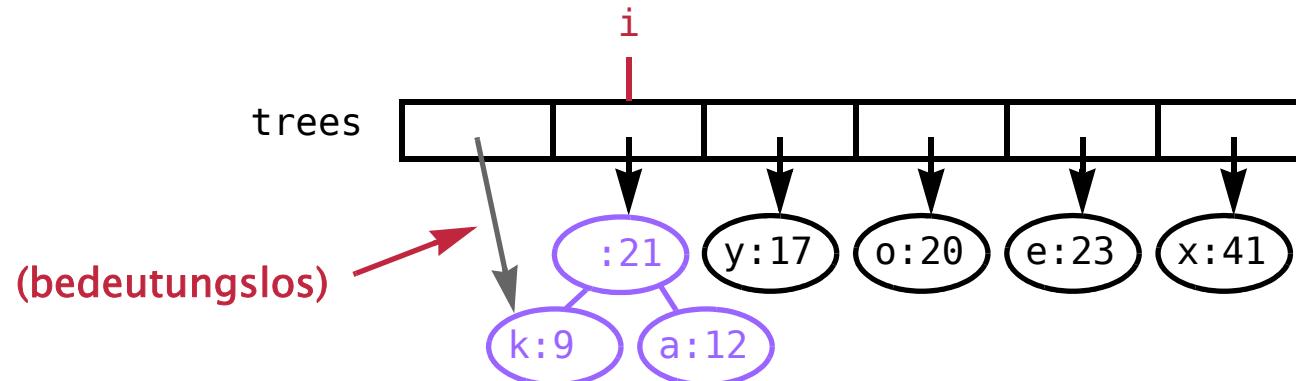
Situation nach dem ersten Aufruf von `insertionSort( 0 )`:



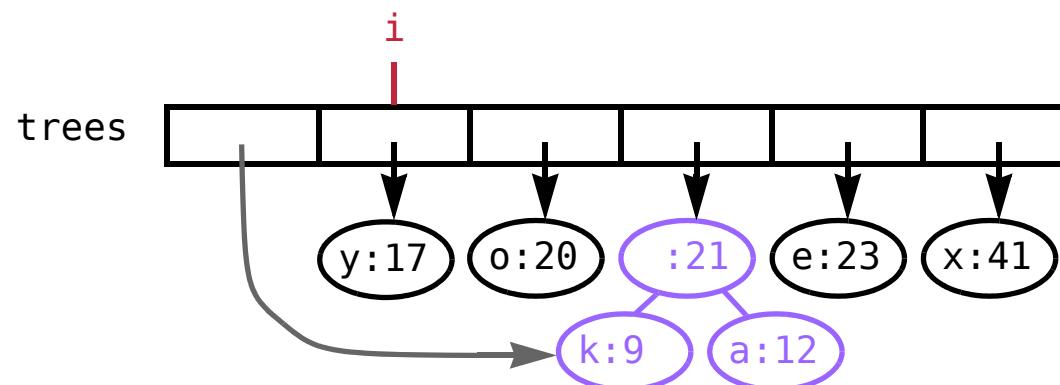
## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

Arbeitsweise der Methode `buildTree()` – Situation nach erstem Durchlauf (`i++` ist ausgeführt):



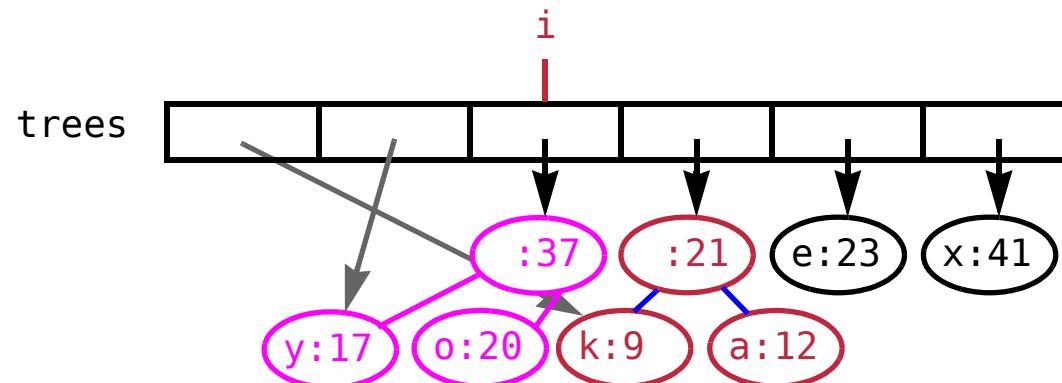
Situation nach dem Aufruf von `insertionSort( 1 )` im zweiten Durchlauf:



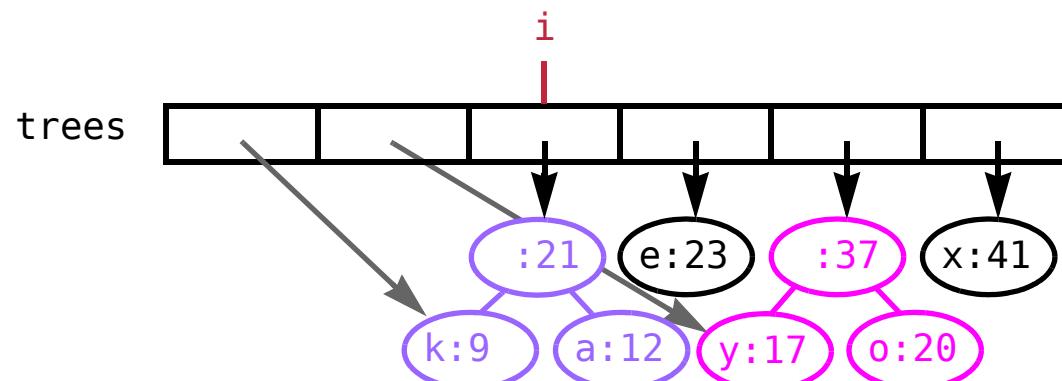
## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

Arbeitsweise der Methode `buildTree()` – Situation nach zweiten Durchlauf (`i++` ist ausgeführt):



Situation nach dem Aufruf von `insertionSort( 2 )` im dritten Durchlauf:



## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

```
public class HuffmanCoding
{
    private HuffmanTree[] trees;

    ...

    private void insertionSort( int start )
    {
        for ( int i = start + 1; i < trees.length; i++ )
        {
            shiftTrees( i );
        }
    }

    ...
}
```

## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

```
public class HuffmanCoding
{
    private HuffmanTree[] trees;

    ...

    private void shiftTrees( int start )
    {
        if ( start < trees.length )
        {
            HuffmanTree toInsert = trees[start];
            int i = start;
            while ( i > 0 &&
                    trees[i-1].compareTo( toInsert ) > 0 )
            {
                trees[i] = trees[i-1];
                i--;
            }
            trees[i] = toInsert;
        }
    ...
}
```

## Datenkompression – Implementierung der Klasse HuffmanCoding

(Fortsetzung)

```
public class HuffmanCoding
{
    private HuffmanTree[] trees;

    ...

    public void showCodeTable()
    {
        trees[trees.length-1].showCodes();
    }
}
```

- ❑ Nach Ausführung des Konstruktors ist der Huffman-Baum aufgebaut und die Kodierung ist ermittelt und eingetragen.
- ❑ Der Baum steht im letzten Element des Feldes `trees`.
- ❑ Die Kodierung ist für jedes Zeichen in dem entsprechenden `HuffmanTriple`-Objekt abgelegt.

## Datenkompression – Beispiel

```
HuffmanTriple[] tokens1 =  
{ new HuffmanTriple( 'a', 30 ),  
  new HuffmanTriple( 'b', 20 ),  
  new HuffmanTriple( 'c', 10 ),  
  new HuffmanTriple( 'd', 15 ),  
  new HuffmanTriple( 'e', 80 )  
};  
HuffmanCoding hc = new HuffmanCoding( tokens1 );  
hc.showCodeTable();
```

## Datenkompression – Beispiel

(Fortsetzung)

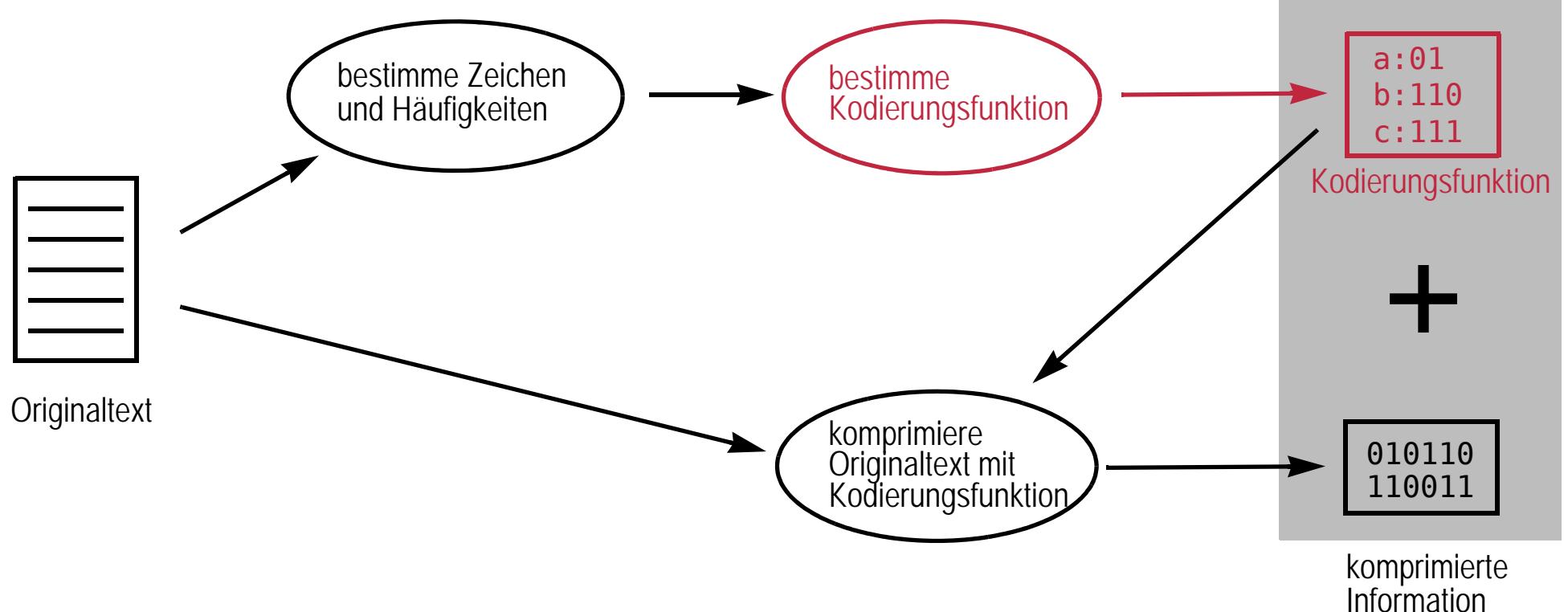
```
HuffmanTriple[] tokens1 =  
{ new HuffmanTriple( 'a', 30 ),  
  new HuffmanTriple( 'b', 20 ),  
  new HuffmanTriple( 'c', 10 ),  
  new HuffmanTriple( 'd', 15 ),  
  new HuffmanTriple( 'e', 80 )  
};  
HuffmanCoding hc = new HuffmanCoding( tokens1 );  
hc.showCodeTable();
```

### Ausgabe:

```
token (quantity: 30): a -> code: 00  
token (quantity: 20): b -> code: 010  
token (quantity: 10): c -> code: 0110  
token (quantity: 15): d -> code: 0111  
token (quantity: 80): e -> code: 1
```

## Datenkompression – Motivation (Erinnerung – Folie 492)

Ablauf einer Textkompression:



# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 7.1. Binärer Suchbaum – Datenstruktur und Implementierung

Dr. Stefan Dissmann  
Fakultät für Informatik



## Lernziele des Kapitels 7. Binärer Suchbaum

Nach Durcharbeiten des Kapitels Datenstruktur Binärer Suchbaum sollen die teilnehmenden Studierenden

- die Definition der Datenstruktur *binärer Suchbaum* kennen,
- einen binären Suchbaum aufbauen können,
- in einem binären Suchbaum suchen können,
- einen binären Suchbaum durchlaufen können,
- einen binären Suchbaum verändern können,
- den Aufwand für Operationen auf einem Suchbaum grob abschätzen können,
- Algorithmen für Tiefendurchläufe (*InOrder*, *PreOrder*, *PostOrder*) anwenden können,
- die Implementierung der Klasse `CharacterSearchTree` verstehen und erweitern können.

## Zählen von Zeichen

Problem:

In einem Text soll die Häufigkeit der vorkommenden Zeichen ermittelt werden. \*)

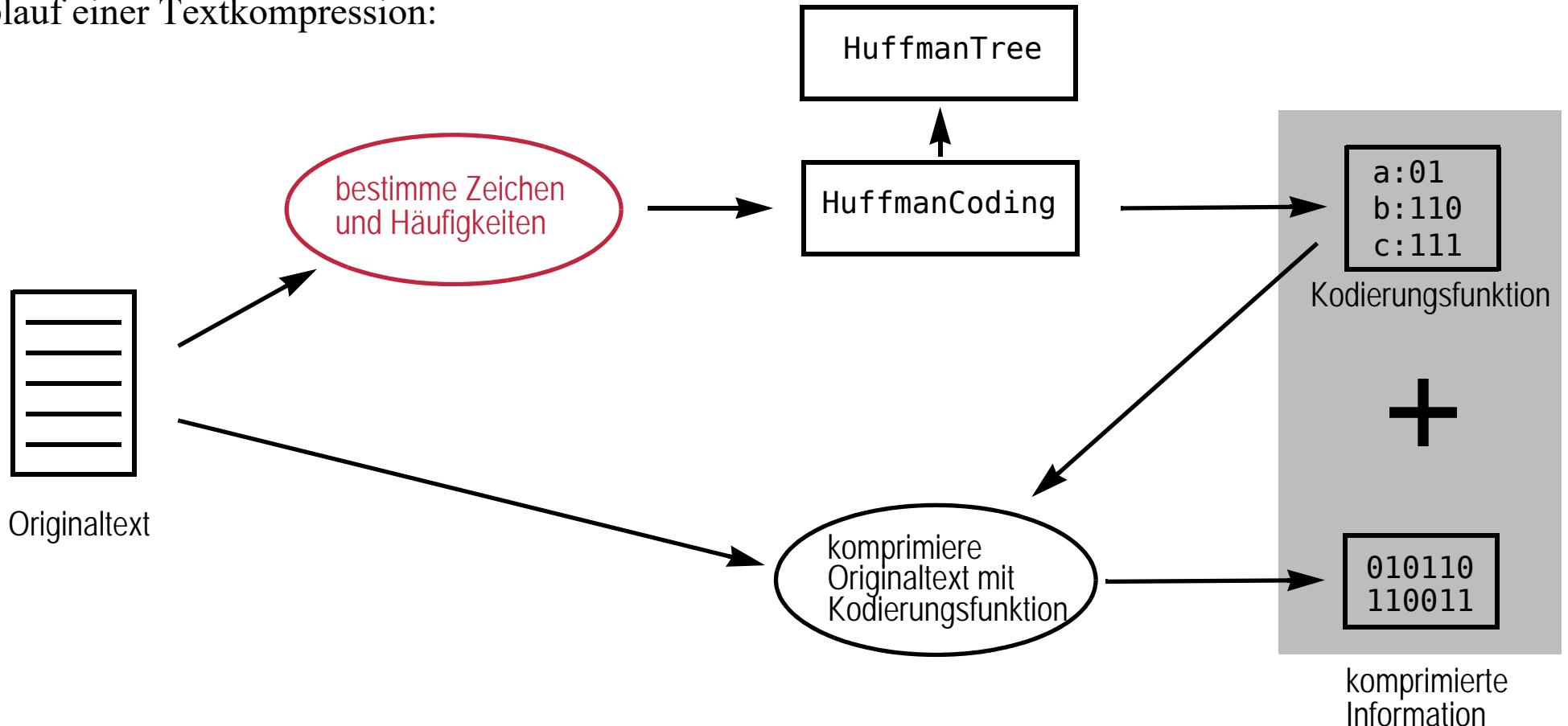
Problemanalyse:

- Die Anzahl der verschiedenen Zeichen ist unbekannt.  
Das muss die eingesetzte Datenstruktur berücksichtigen.
- Tritt ein Zeichen zum ersten Mal auf, muss es in die Datenstruktur aufgenommen werden.
- Tritt ein Zeichen erneut auf, muss lediglich seine Häufigkeit erhöht werden.
- In beiden Fällen muss das Zeichen gesucht (und gefunden) werden.
- Bei einem langen Text muss also sehr häufig gesucht werden. Die Datenstruktur sollte also das schnelle Suchen (und Finden) unterstützen.

\*) Variation der Aufgabenstellung: In einem Bild soll die Pixel-Häufigkeit der vorkommenden Farben ermittelt werden.

## Datenkompression – Motivation (Erinnerung – Folie 548)

Ablauf einer Textkompression:



## Zählen von Zeichen

(Fortsetzung)

Welche Datenstruktur ist geeignet?

- Ein Feld ist eher ungeeignet, da
  - es eine feste Länge hat
    - ungünstig für eine unbekannte Anzahl von verschiedenen Zeichen/Informationen – und
    - es bei der Suche sequentiell durchlaufen werden muss.
- Eine rekursiv definierte Datenstruktur (binärer Baum) kann bei Bedarf wachsen, da einfach weitere Knoten angefügt werden können.
- Eine solche erweiterbare Datenstruktur heißt auch *dynamische Datenstruktur*.  
(Knoten könnten auch gelöscht werden, so dass eine dynamische Anpassung in beide Richtungen möglich wäre.)

## Zählen von Zeichen

(Fortsetzung)

Welche Datenstruktur ist geeignet?

- Ein Feld ist eher ungeeignet, da
  - es eine feste Länge hat
    - ungünstig für eine unbekannte Anzahl von verschiedenen Zeichen/Informationen – und
    - es bei der Suche sequentiell durchlaufen werden muss.
- Eine rekursiv definierte Datenstruktur (binärer Baum) kann bei Bedarf wachsen, da einfach weitere Knoten angefügt werden können.
- Eine solche erweiterbare Datenstruktur heißt auch *dynamische Datenstruktur*.  
(Knoten könnten auch gelöscht werden, so dass eine dynamische Anpassung in beide Richtungen möglich wäre.)

Wie könnten die Zeichen im Baum angeordnet werden?

**Idee aus QuickSort:**

**kleine und große Werte voneinander trennen!**

## Binärer Suchbaum

### Ein *binärer Suchbaum*

ist ein binärer Baum, für den gilt, dass

- die Information in der Wurzel größer ist als jede Information in ihrem linken Teilbaum und
  - die Information in der Wurzel kleiner ist als jede Information in ihrem rechten Teilbaum und
  - der linke Teilbaum auch ein binärer Suchbaum ist und
  - der rechte Teilbaum auch ein binärer Suchbaum ist.
- 
- Der leere Baum ist auch ein binärer Suchbaum.

## Binärer Suchbaum

(Fortsetzung)

### Ein *binärer Suchbaum*

ist ein binärer Baum, für den gilt, dass

- die Information in der Wurzel größer ist als jede Information in ihrem linken Teilbaum und
- die Information in der Wurzel kleiner ist als jede Information in ihrem rechten Teilbaum und
- der linke Teilbaum auch ein binärer Suchbaum ist und
- der rechte Teilbaum auch ein binärer Suchbaum ist.

Konkretisierung:

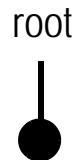
- Die Informationen in einem Baum, in dem nach Zeichen gesucht werden sollen, sind Zeichen des Typs **char**.
- Die Ordnungsrelation für diesen Baum ergibt sich daher aus der Ordnungsrelation, die für die Werte des Typs **char** definiert sind.
- Die Häufigkeiten werden zwar mit den Zeichen zusammen abgelegt. Sie haben aber für die Struktur des Baums keine Bedeutung, da nicht nach Häufigkeiten gesucht werden soll.

## Binärer Suchbaum

(Fortsetzung)

Der Aufbau des binären Suchbaums erfolgt von der Wurzel aus!

Es sollen die Zeichen des folgenden Textes erfasst werden: "halloween"

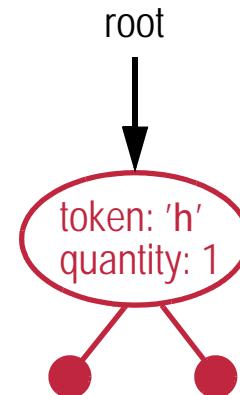


## Binärer Suchbaum

(Fortsetzung)

Der Aufbau des binären Suchbaums erfolgt von der Wurzel aus!

Es sollen die Zeichen des folgenden Textes erfasst werden: "halloween"

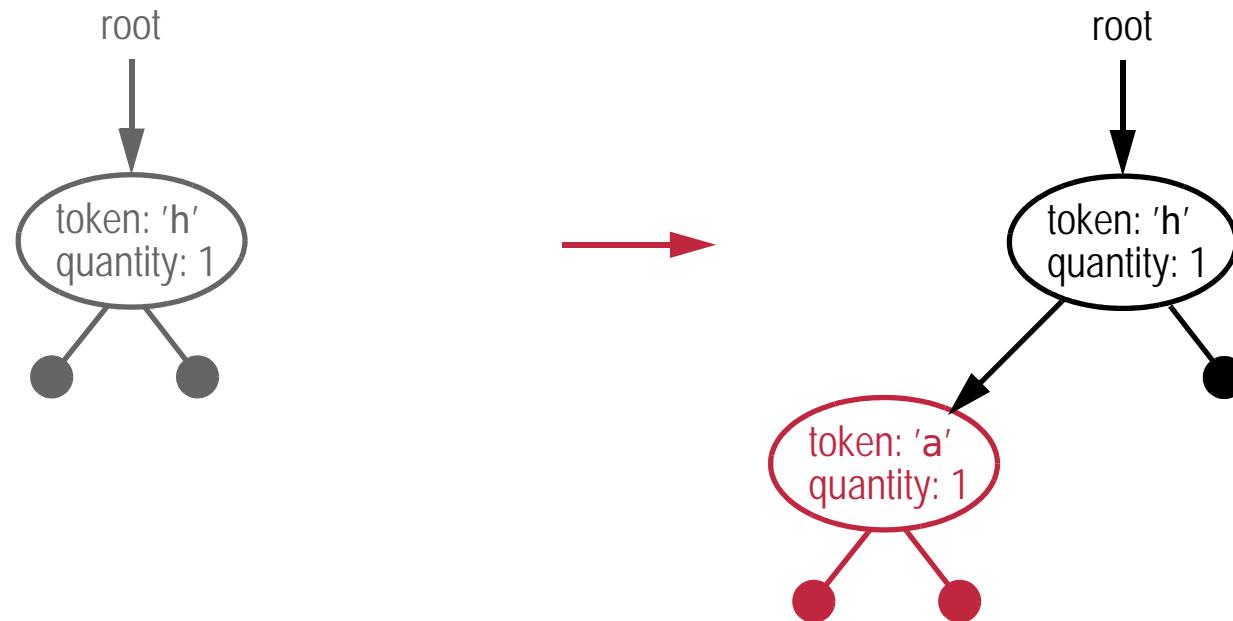


## Binärer Suchbaum

(Fortsetzung)

Der Aufbau des binären Suchbaums erfolgt von der Wurzel aus!

Es sollen die Zeichen des folgenden Textes erfasst werden: "halloween"

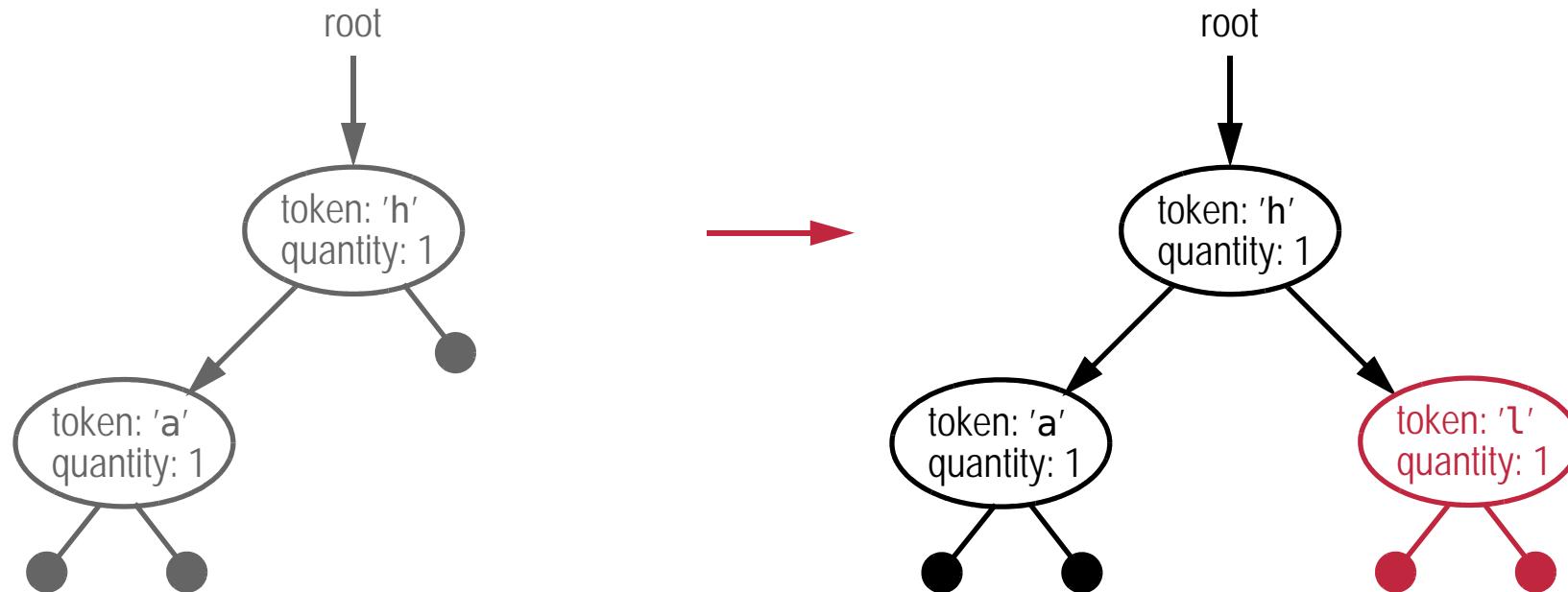


## Binärer Suchbaum

(Fortsetzung)

Der Aufbau des binären Suchbaums erfolgt von der Wurzel aus!

Es sollen die Zeichen des folgenden Textes erfasst werden: "halloween"

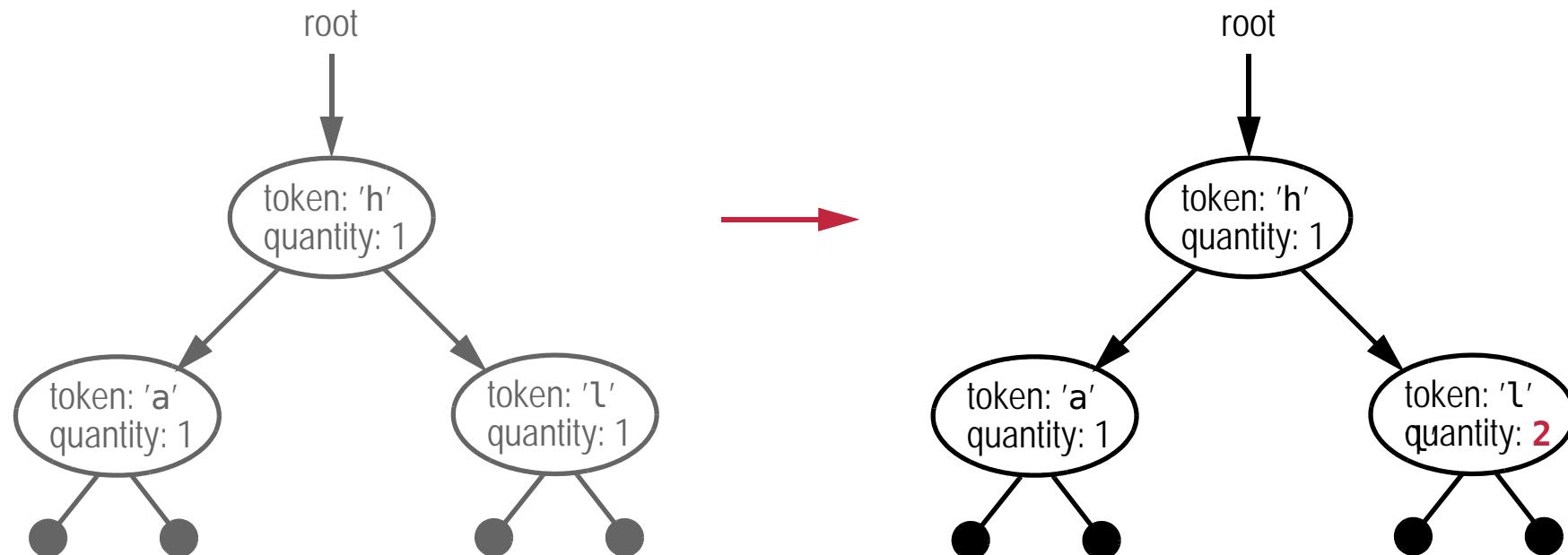


## Binärer Suchbaum

(Fortsetzung)

Der Aufbau des binären Suchbaums erfolgt von der Wurzel aus!

Es sollen die Zeichen des folgenden Textes erfasst werden: "halloween"

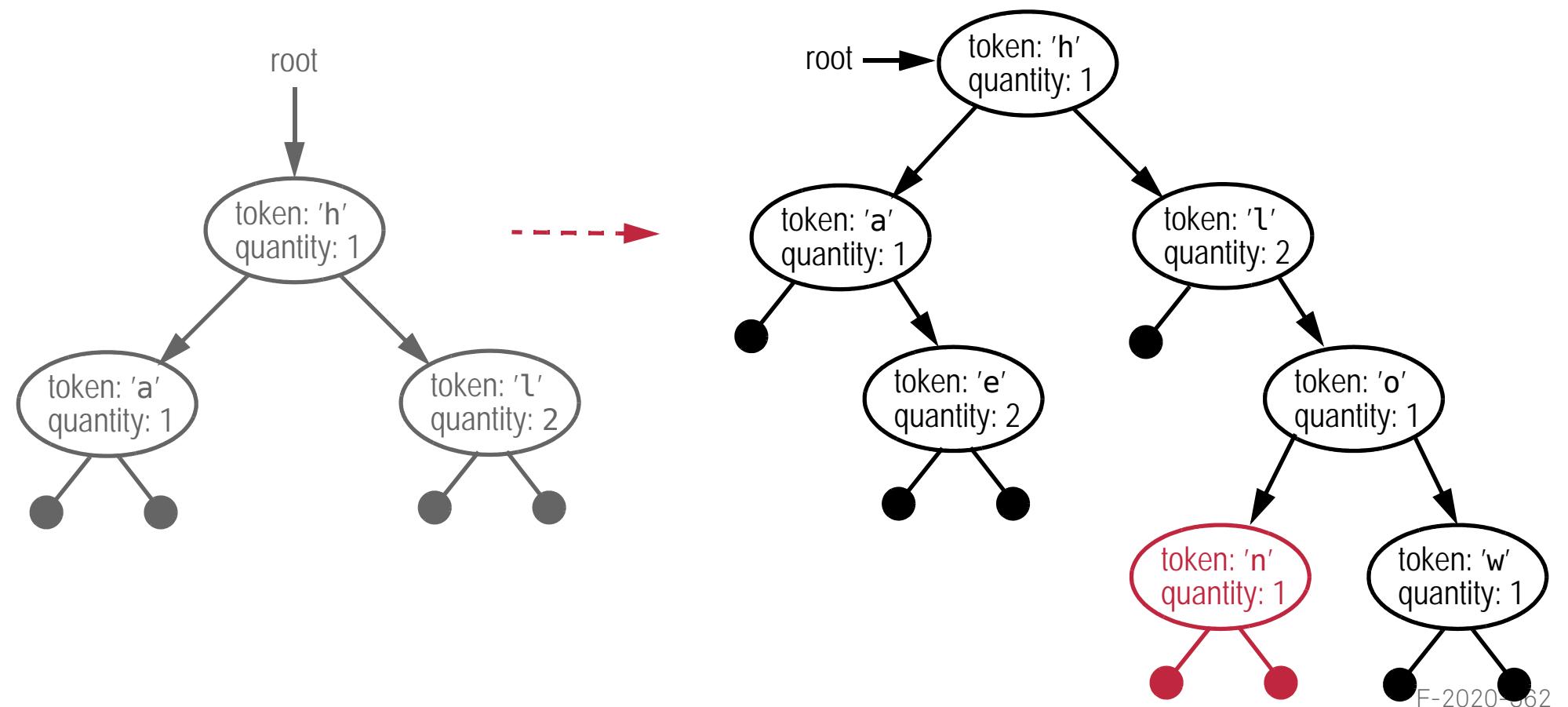


## Binärer Suchbaum

(Fortsetzung)

Der Aufbau des binären Suchbaums erfolgt von der Wurzel aus!

Es sollen die Zeichen des folgenden Textes erfasst werden: "hallowee**n**"

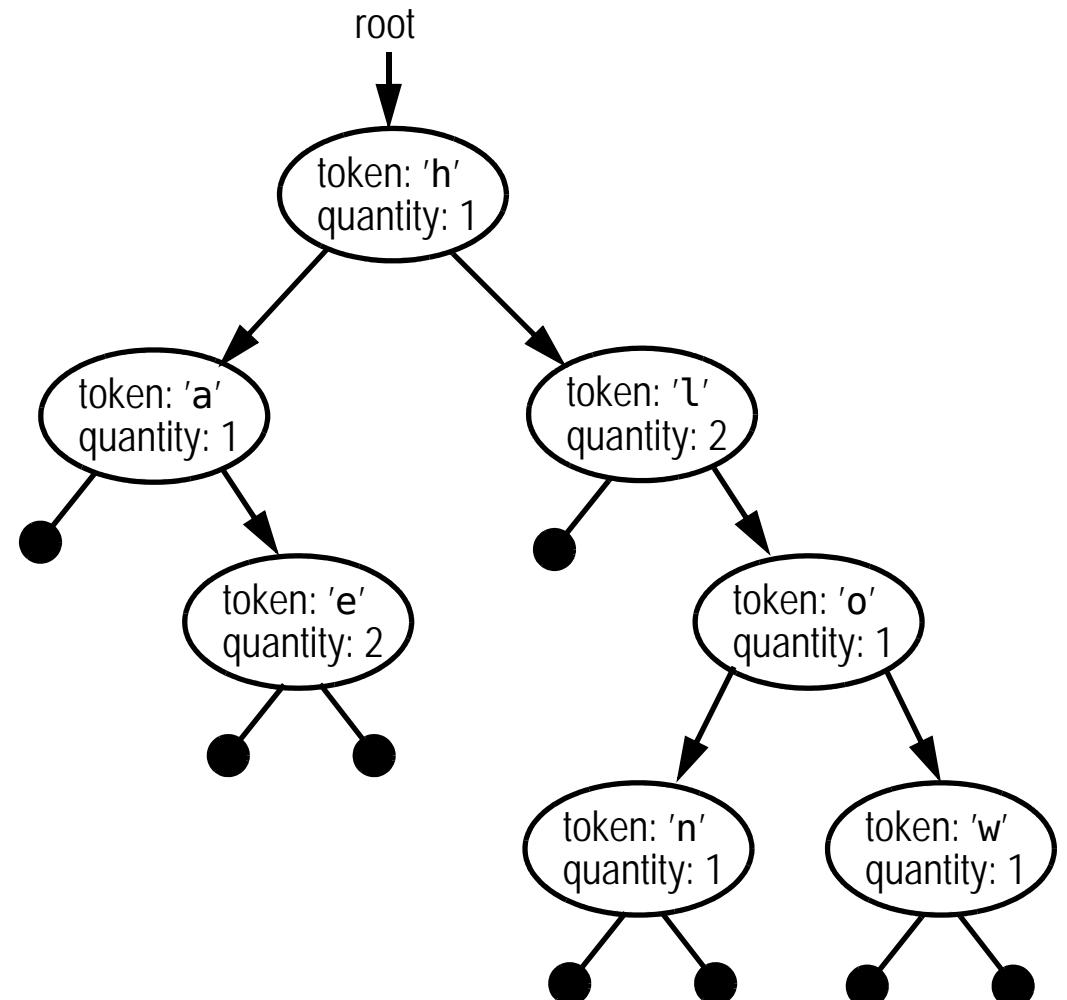


## Binärer Suchbaum

(Fortsetzung)

Problemanalyse (Wiederholung von Folie 551):

- ❑ Tritt ein Zeichen zum ersten Mal auf,  
muss es in die Datenstruktur  
aufgenommen werden.

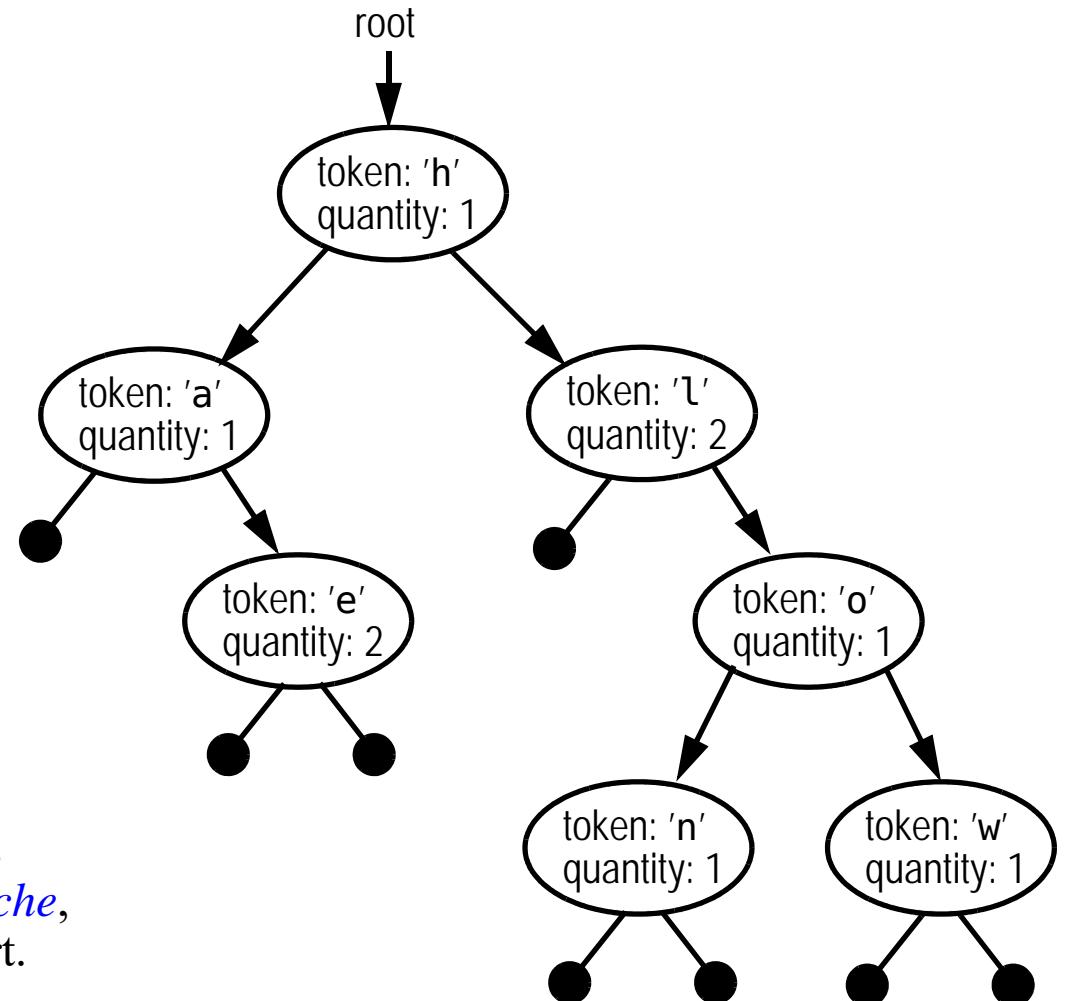


## Binärer Suchbaum

(Fortsetzung)

Problemanalyse (Wiederholung von Folie 551):

- Tritt ein Zeichen zum ersten Mal auf, muss es in die Datenstruktur aufgenommen werden.
- Tritt ein Zeichen erneut auf, muss lediglich seine Häufigkeit erhöht werden.
- Der Aufbau des Baums hängt von der Reihenfolge der hinzugefügten Werte ab:  
Die Buchstabenfolge "ahlloween" führt zu einer *anderen* Baumstruktur!
- Vor dem Einfügen eines neuen Knotens in den Baum erfolgt immer erst *eine Suche*, ob bereits ein passender Knoten existiert.



## Binärer Suchbaum

(Fortsetzung)

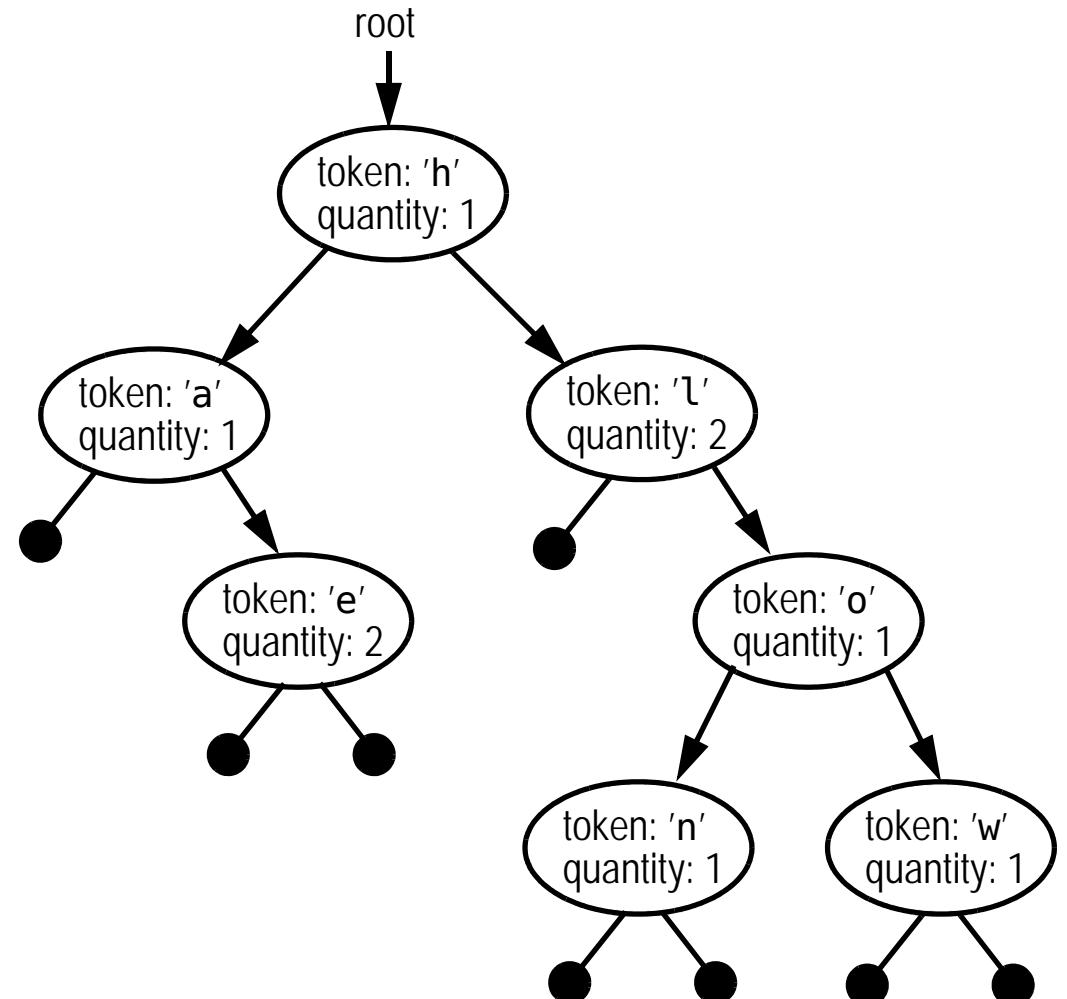
### Suchvorgang:

In jedem Knoten wird überprüft,  
ob das zu suchende Zeichen

- kleiner: dann weiter nach links – oder
  - größer: dann weiter nach rechts – oder
  - gleich: gefunden!
- ist.

### Suchgeschwindigkeit:

- Die Dauer der Suche wird durch die Länge des Wegs von der Wurzel zu dem gesuchten Knoten bestimmt.
- Die Dauer der Suche ist also abhängig von der Struktur des Baums.

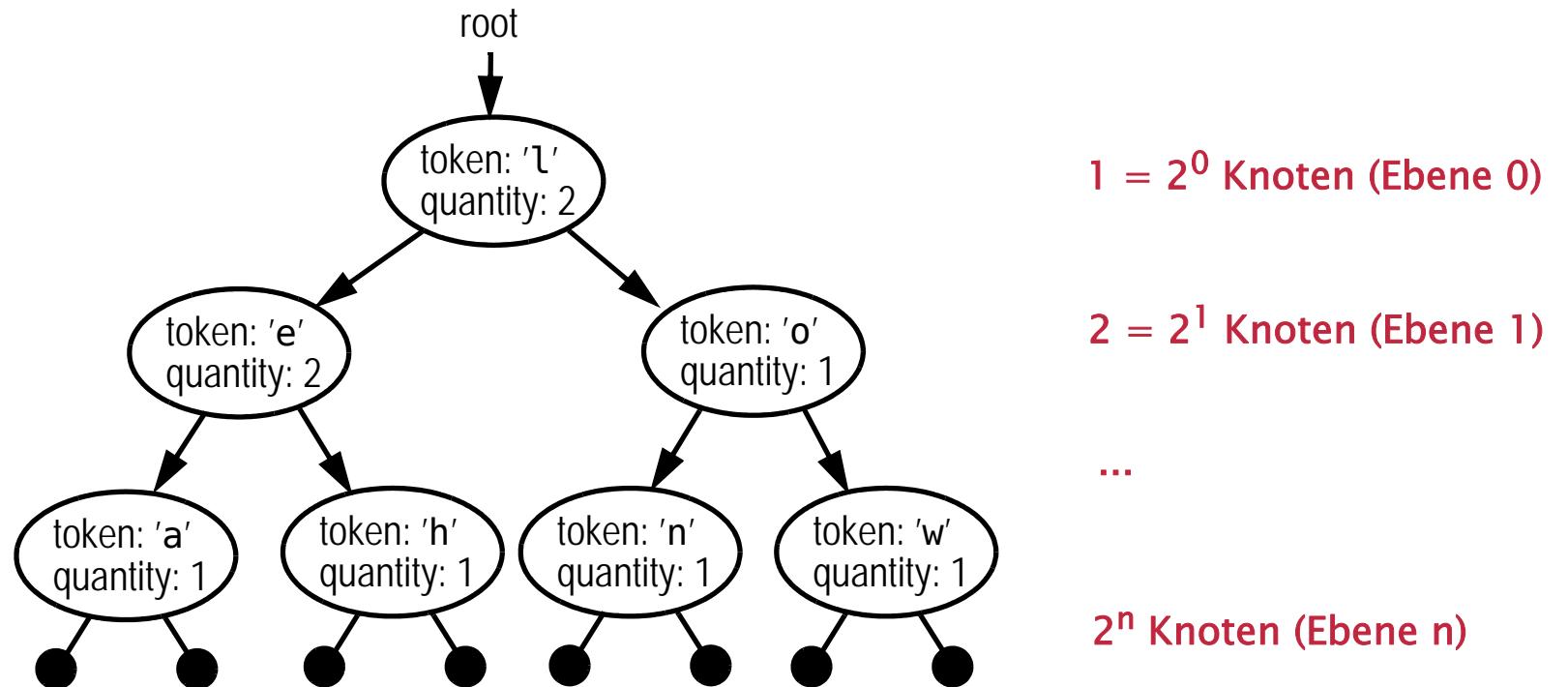


## Binärer Suchbaum

(Fortsetzung)

### Anmerkungen

- Wenn alle Informationen, nach denen gesucht werden soll, mit gleicher Wahrscheinlichkeit auftreten, dann ist die optimale Struktur für die Suche ein gleichmäßig aufgebauter Baum.



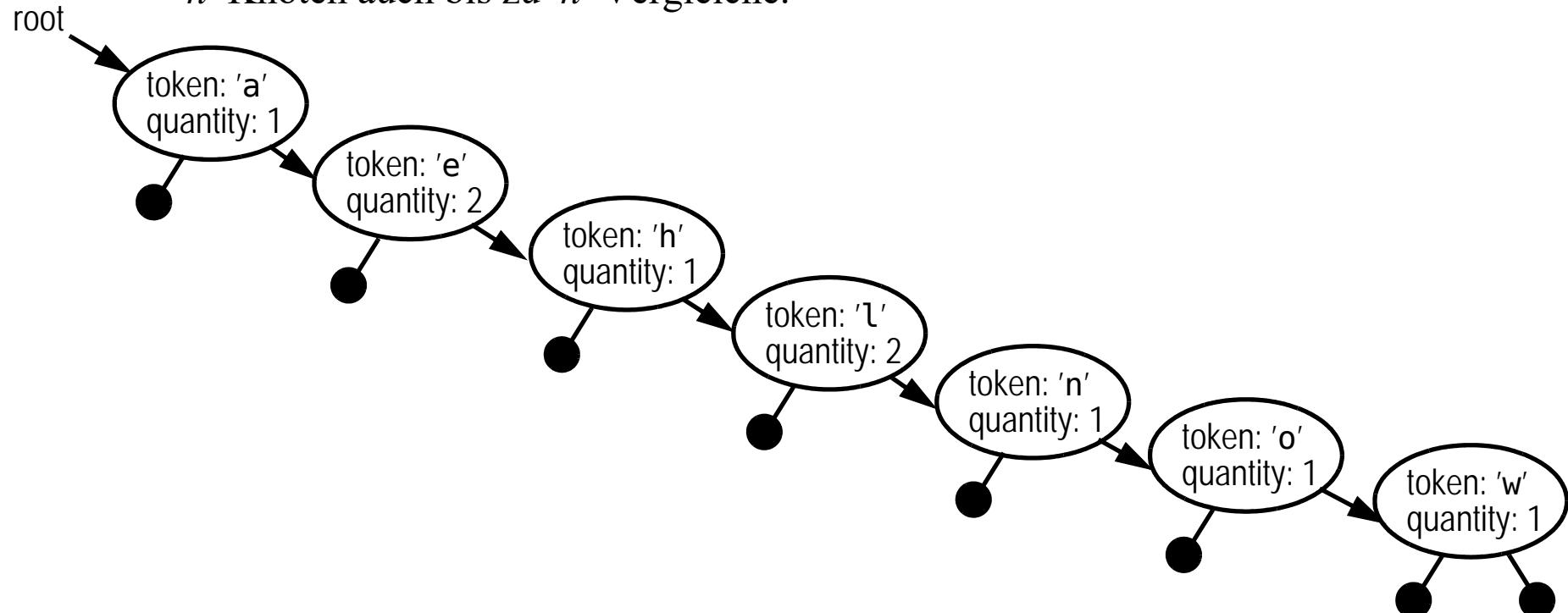
- Ein gleichmäßig aufgebauter mit  $n$  Knoten besitzt eine Höhe von nur  $\log_2(n)$ . Eine Suche erfordert daher im optimalen Fall bei  $n$  Knoten höchstens  $\log_2(n)$  Vergleiche.

## Binärer Suchbaum

(Fortsetzung)

### Anmerkungen

Eine ungünstige Eingabe kann zu einem Baum mit nur wenigen, sehr langen Ästen führen, im Extremfall zu einem einzigen Ast, auf dem alle Knoten liegen. Eine Suche erfordert dann für  $n$  Knoten auch bis zu  $n$  Vergleiche.



- ❑ Falls ein Baum beim Aufbau sehr ungleichmäßig wächst, kann er *ausgeglichen* \*) werden.

\*) Details hierzu werden in DAP 2 vorgestellt.

## Zählen von Zeichen

(Fortsetzung)

```
public class CharacterSearchTree
{
    private HuffmanTriple content;
    private CharacterSearchTree leftChild, rightChild;

    public CharacterSearchTree()
    {
        content = null;
        leftChild = null;
        rightChild = null;
    }
    ...
}
```

Attribute für Zeichen,  
Häufigkeit (und Kodierung)

Attribute für  
Baumstruktur

Konstruktor:  
legt einen leeren Baum an

- Die Klasse `CharacterSearchTree` hat viele Gemeinsamkeiten mit der Klasse `HuffmanTree`: Beide Klassen dienen dem Aufbau binärer Bäume.
- Die Klassen enthalten Attribute für die gleichen Aufgaben: `leftChild` und `rightChild` für die Konstruktion des Baums, `content` für den Inhalt eines Knotens
- Die Konstruktionsregeln des binären Suchbaums drücken sich nicht in seinen Attributen aus. Die Konstruktionsregeln werden von Methoden sichergestellt, die einen Baum erzeugen und diesen verwalten.

## Zählen von Zeichen

(Fortsetzung)

```
public class CharacterSearchTree
{
    ...
    public HuffmanTriple getContent()
    {
        if ( !isEmpty() )
        {
            return content;
        } else {
            throw new IllegalStateException();
        }
    }

    public boolean isEmpty()
    {
        return content == null;
    }

    public boolean isLeaf()
    {
        return !isEmpty() && leftChild.isEmpty() && rightChild.isEmpty();
    }
    ...
}
```

Zugriff auf den Inhalt des Knotens

leeren Baum identifizieren

Blatt identifizieren

## Zählen von Zeichen

(Fortsetzung)

Das Einfügen eines Zeichens in den Baum erfordert

- entweder das Anlegen eines neuen Knotens für dieses Zeichen
- oder das Erhöhen der Häufigkeit für das schon vorhandenen Zeichen:

```
public void add( char t )
{
    if ( leerer Baum )
    {
        erzeuge Baum mit Inhalt t
    }
    else
    {
        suche, ob bereits ein Knoten mit Inhalt t existiert,
        falls dieser gefunden wurde – erhöhe dessen Häufigkeit,
        oder – falls Suche endgültig erfolglos – füge neuen Knoten mit t ein
    }
}
```

## Zählen von Zeichen

(Fortsetzung)

```

public void add( char t )
{
    if ( isEmpty() ) ← leerer Baum liegt vor
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        suche, ob bereits ein Knoten mit Inhalt t existiert,
        falls dieser gefunden wurde – erhöhe dessen Häufigkeit,
        oder – falls Suche endgültig erfolglos – füge neuen Knoten mit t ein
    }
}

```

← **Blatt wird erzeugt.**

- Der leere Baum wird in ein Blatt umgewandelt, indem
  - ein passendes HuffmanTriple-Objekt angelegt wird,
  - die Referenzen für die nachfolgenden Teilbäume auf leere Bäume verweisen.

## Zählen von Zeichen

(Fortsetzung)

```
public void add( char t )
{
    if ( isEmpty() )
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        if ( content.getToken() > t )
        {
            weiteres Einfügen im linken Teilbaum
        }
        else if ( content.getToken() < t )
        {
            weiteres Einfügen im rechten Teilbaum
        }
        else
        {
            erhöhe die Häufigkeit
        }
    }
}
```

## Zählen von Zeichen

(Fortsetzung)

```

public void add( char t )
{
    if ( isEmpty() )
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        if ( content.getToken() > t )
        {
            leftChild.add( t ); ← Rekursion auf
                                Teilbaum
        }
        else if ( content.getToken() < t )
        {
            rightChild.add( t ); ← Rekursion auf
                                Teilbaum
        }
        else
        {
            erhöhe die Häufigkeit
        }
    }
}

```

## Zählen von Zeichen

(Fortsetzung)

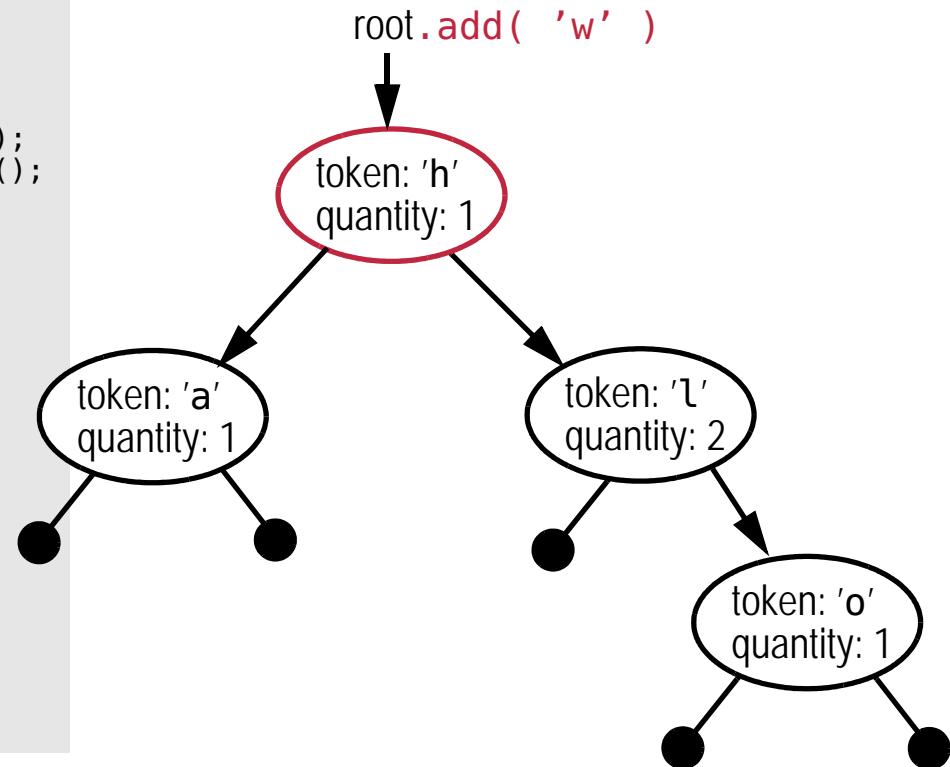
```
public void add( char t )
{
    if ( isEmpty() )
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        if ( content.getToken() > t )
        {
            leftChild.add( t );
        }
        else if ( content.getToken() < t )
        {
            rightChild.add( t );
        }
        else
        {
            content.incrementQuantity();
        }
    }
}
```

## Ausführung von add

```

public void add( char t )
{
    if ( isEmpty() )
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        if ( content.getToken() > t )
        {
            leftChild.add( t );
        }
        else if ( content.getToken() < t )
        {
            rightChild.add( t );
        }
        else
        {
            content.incrementQuantity();
        }
    }
}

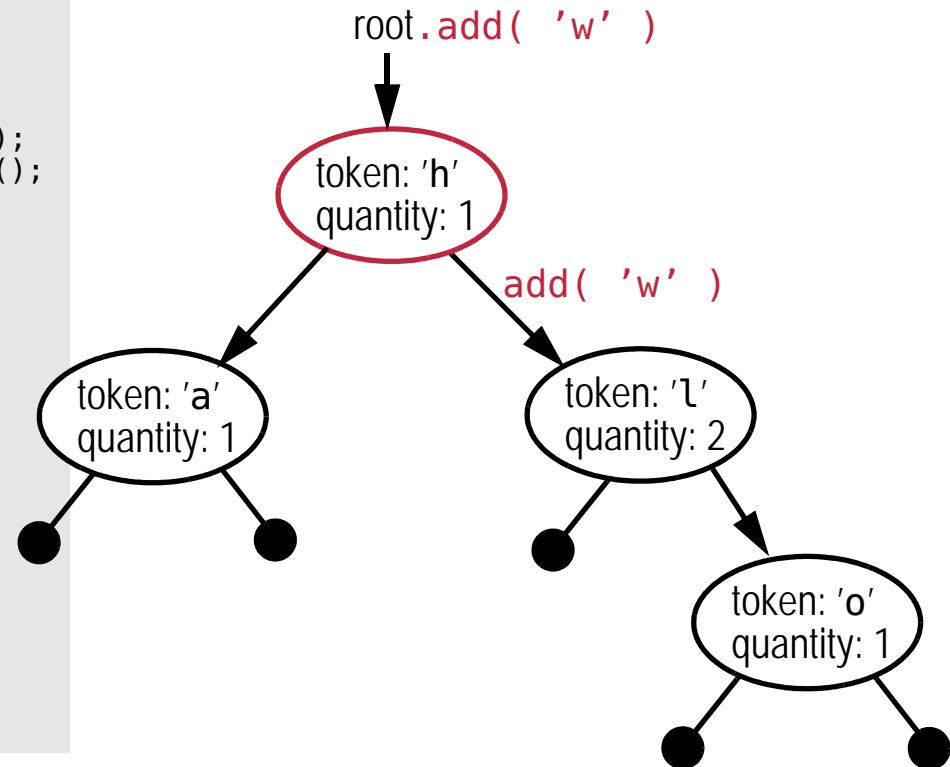
```



## Ausführung von add

(Fortsetzung)

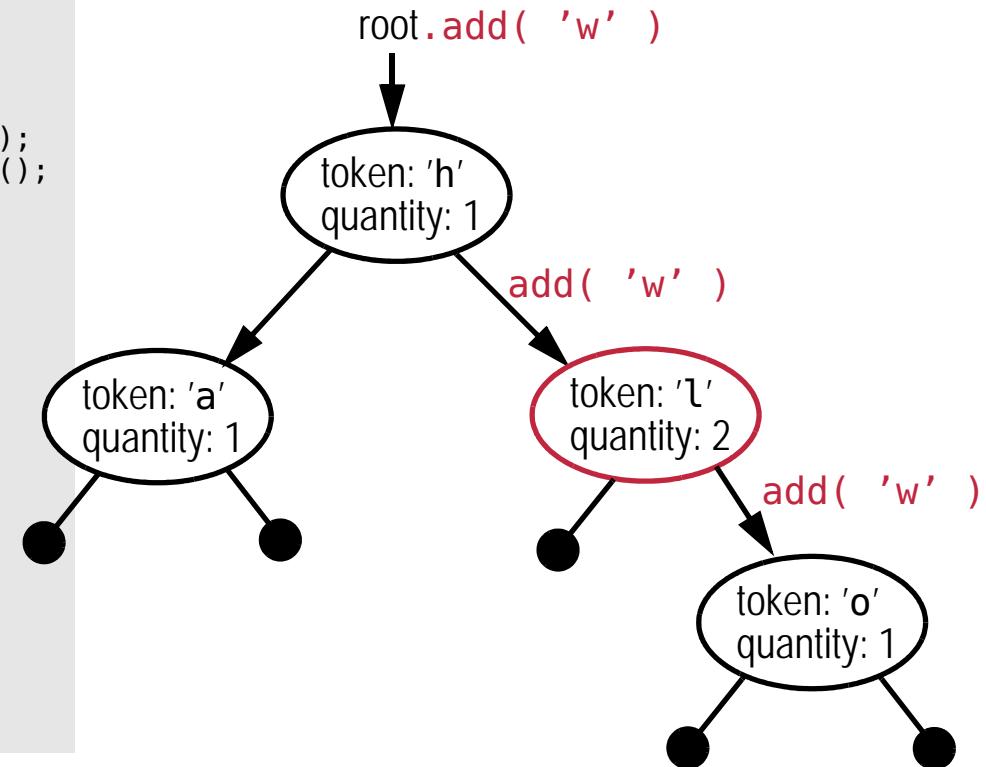
```
public void add( char t )
{
    if ( isEmpty() )
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        if ( content.getToken() > t )
        {
            leftChild.add( t );
        }
        else if ( content.getToken() < t )
        {
            rightChild.add( t );
        }
        else
        {
            content.incrementQuantity();
        }
    }
}
```



## Ausführung von add

(Fortsetzung)

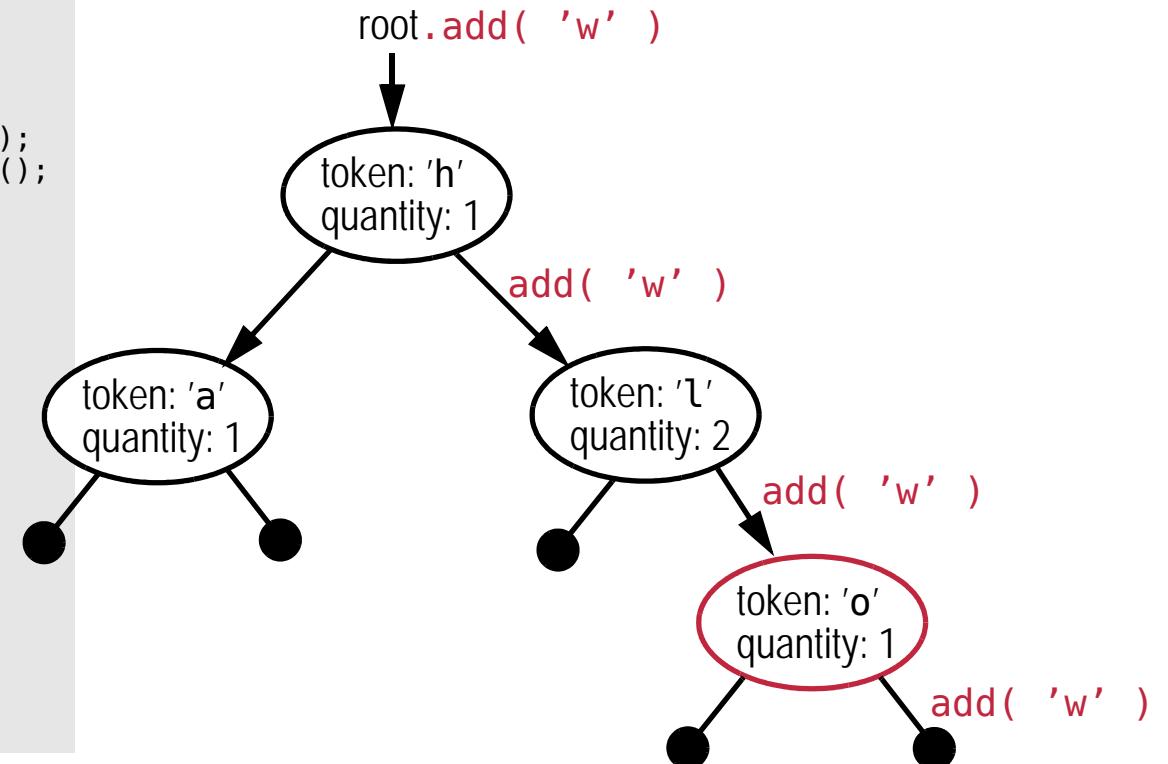
```
public void add( char t )
{
    if ( isEmpty() )
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        if ( content.getToken() > t )
        {
            leftChild.add( t );
        }
        else if ( content.getToken() < t )
        {
            rightChild.add( t );
        }
        else
        {
            content.incrementQuantity();
        }
    }
}
```



## Ausführung von add

(Fortsetzung)

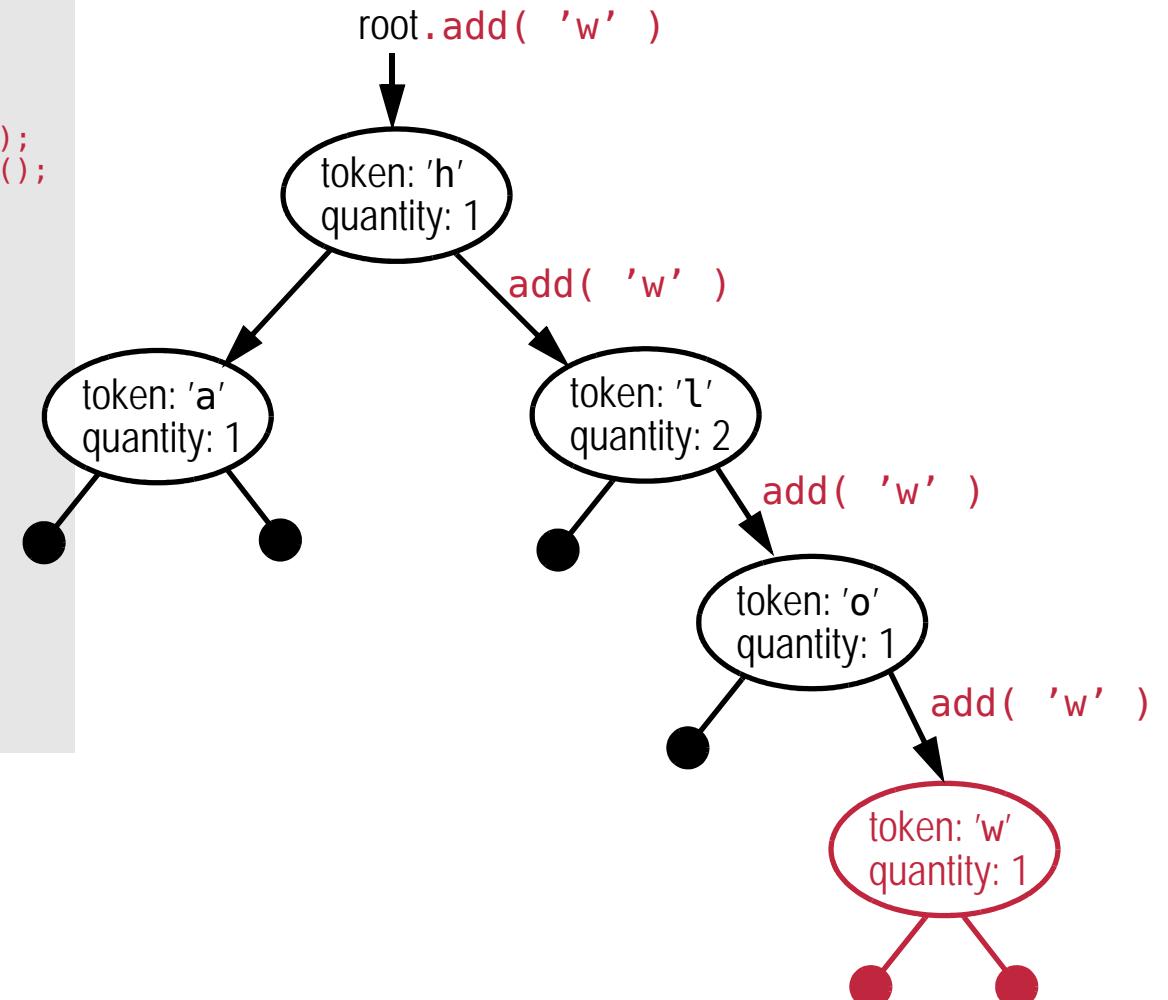
```
public void add( char t )
{
    if ( isEmpty() )
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        if ( content.getToken() > t )
        {
            leftChild.add( t );
        }
        else if ( content.getToken() < t )
        {
            rightChild.add( t );
        }
        else
        {
            content.incrementQuantity();
        }
    }
}
```



## Ausführung von add

(Fortsetzung)

```
public void add( char t )
{
    if ( isEmpty() )
    {
        content = new HuffmanTriple( t );
        leftChild = new CharacterSearchTree();
        rightChild = new CharacterSearchTree();
    }
    else
    {
        if ( content.getToken() > t )
        {
            leftChild.add( t );
        }
        else if ( content.getToken() < t )
        {
            rightChild.add( t );
        }
        else
        {
            content.incrementQuantity();
        }
    }
}
```



## Zählen von Zeichen

(Fortsetzung)

Das Feststellen der Anzahl der abgelegten, unterschiedlichen Zeichen lässt sich ebenfalls an die nachfolgenden Teilbäume übertragen und daher rekursiv formulieren:

```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size() + rightChild.size() + 1;
    }
}
```

leerer Baum:  
enthält kein Zeichen

rekursive Aufrufe

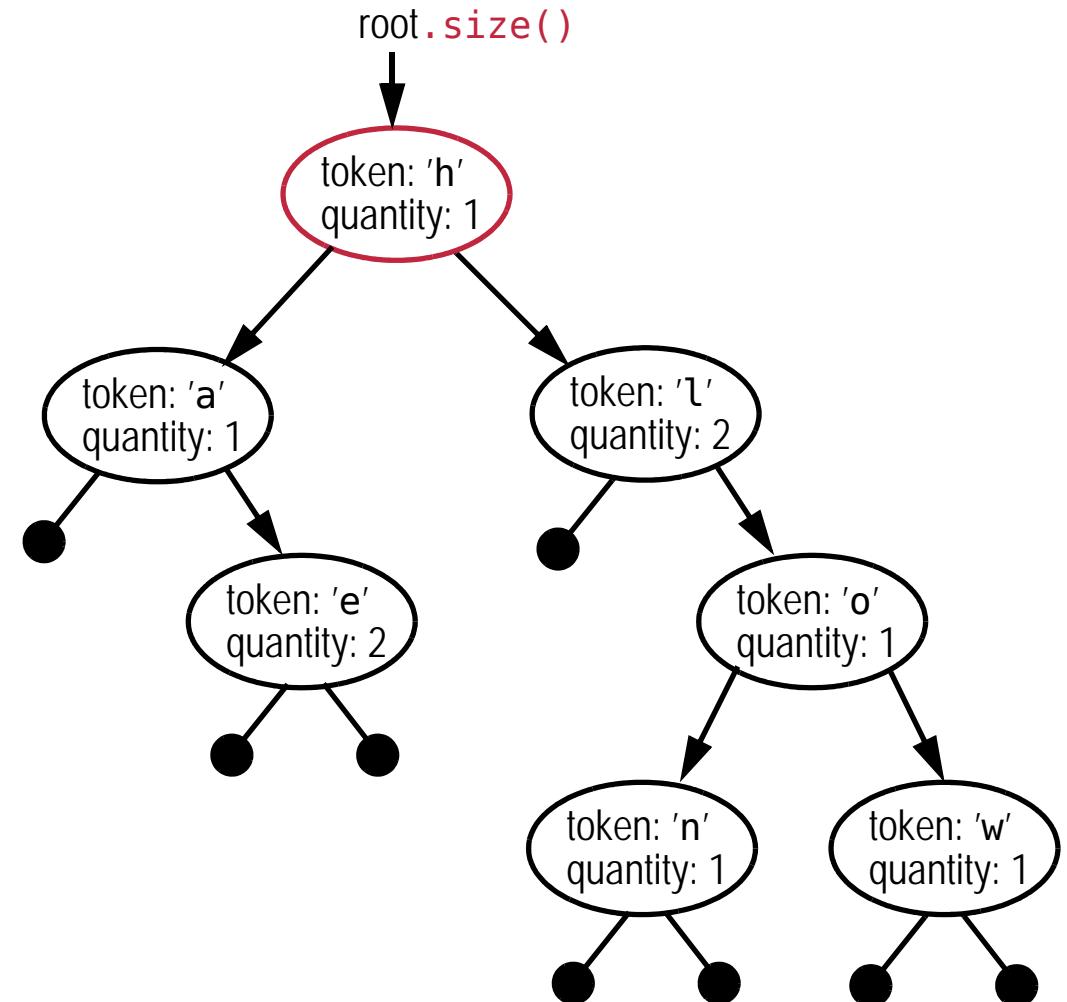
Jeder nicht-leere Teilbaum besteht aus

- der Anzahl der Zeichen, die in den Teilbäumen abgelegt sind
- und aus dem einen Zeichen in seiner Wurzel:

$\text{leftChild.size()} + \text{rightChild.size()}$   
 $+ 1$

## Ausführung von size

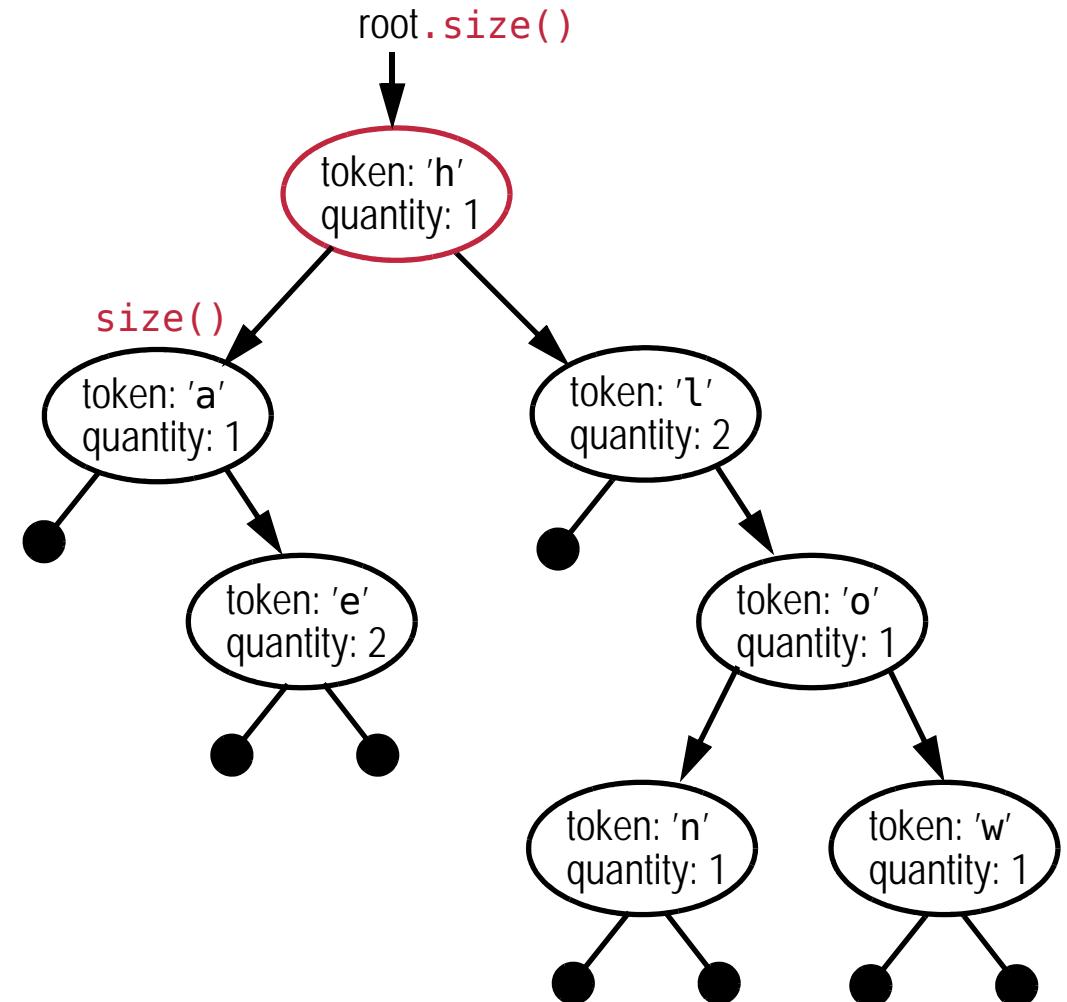
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

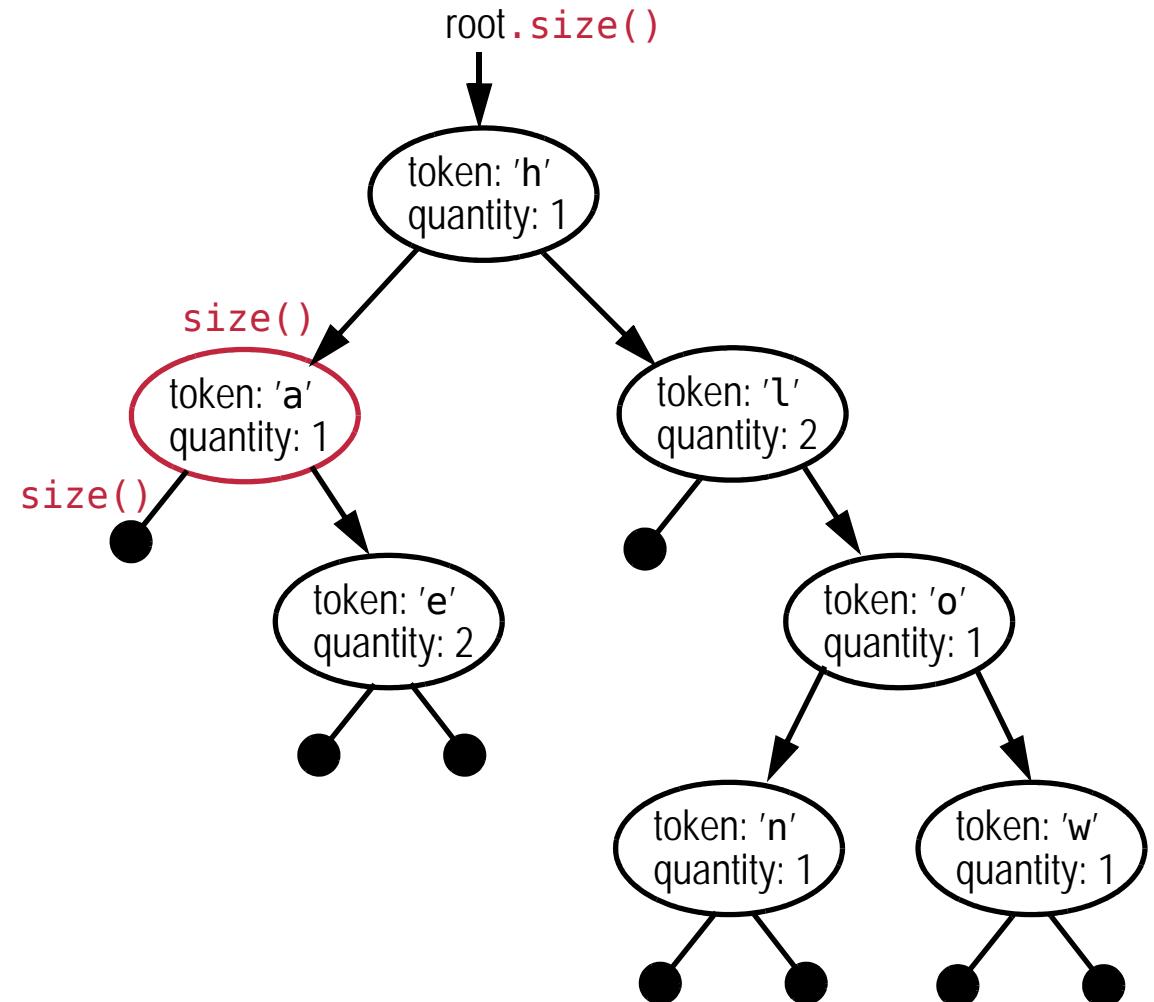
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

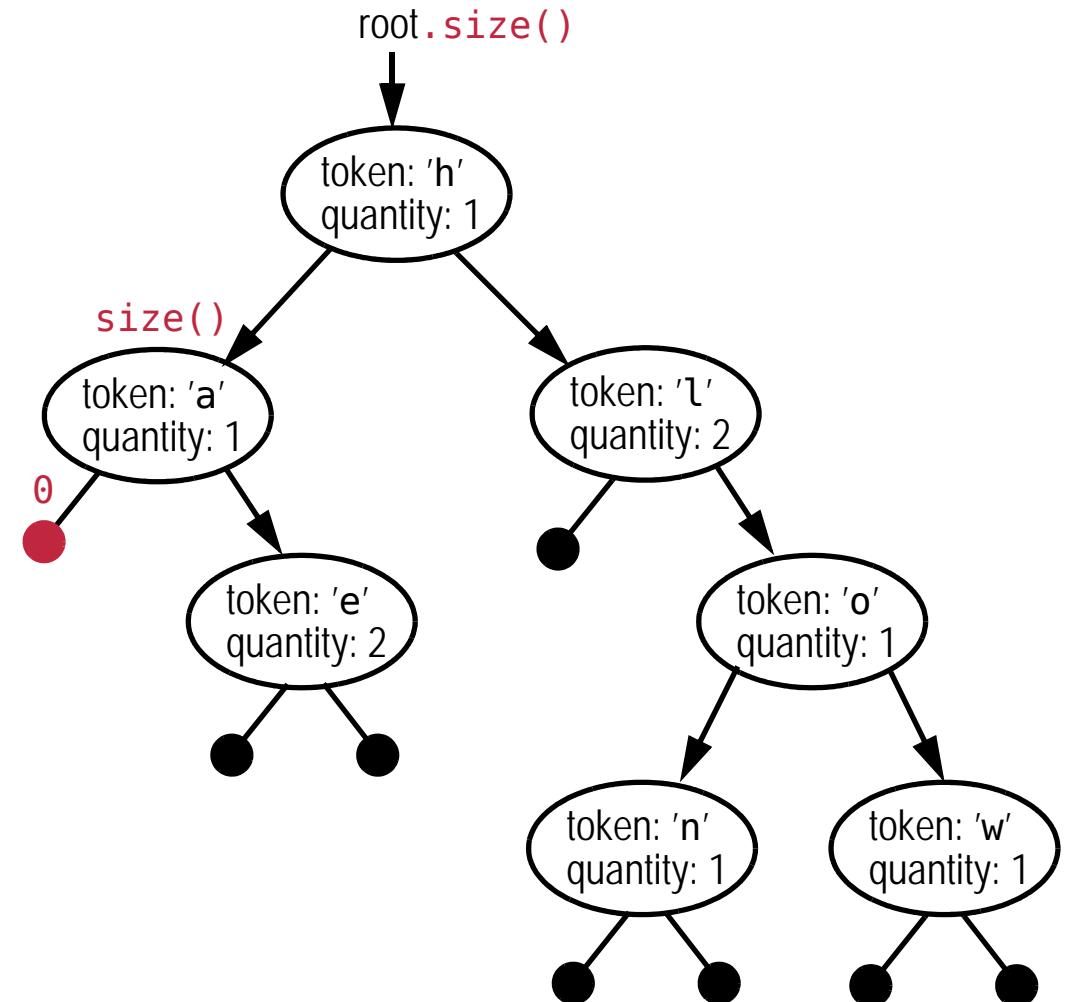
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

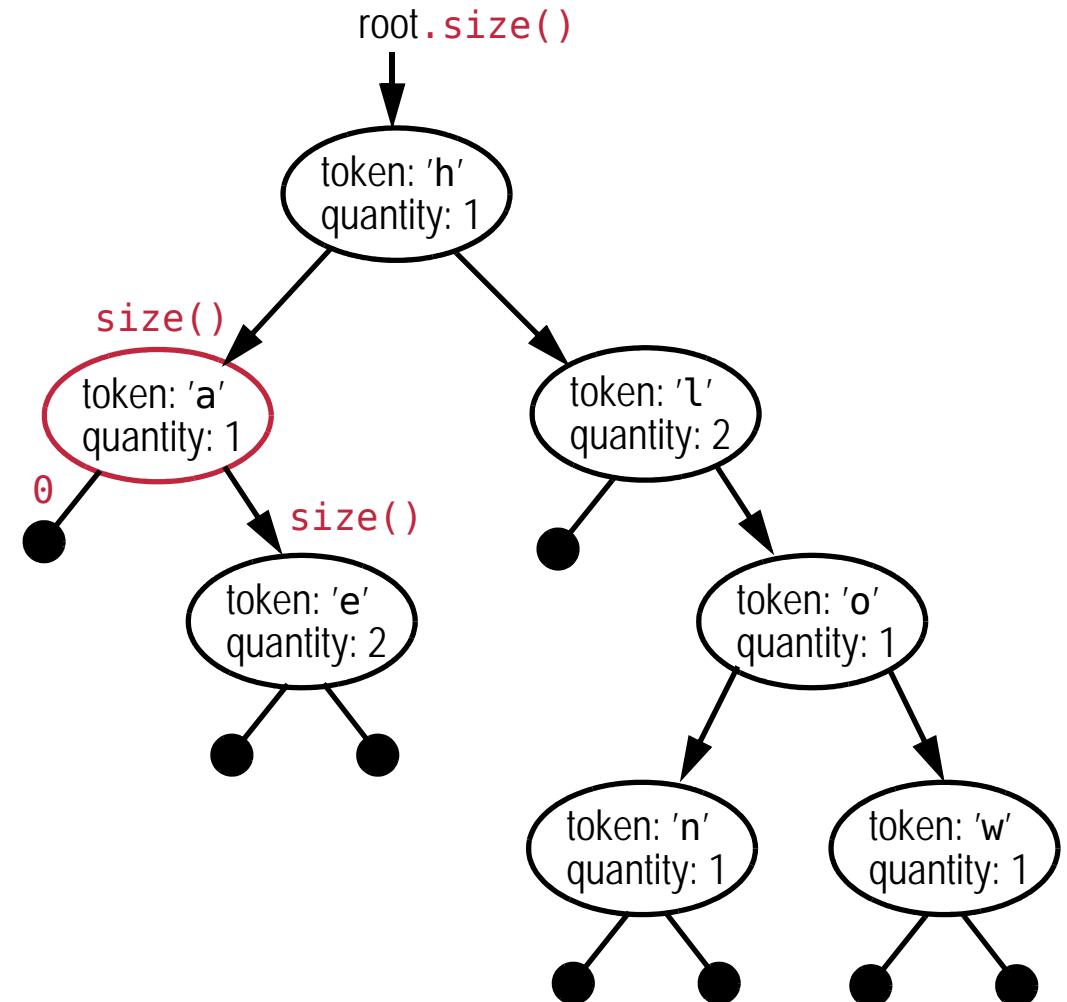
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

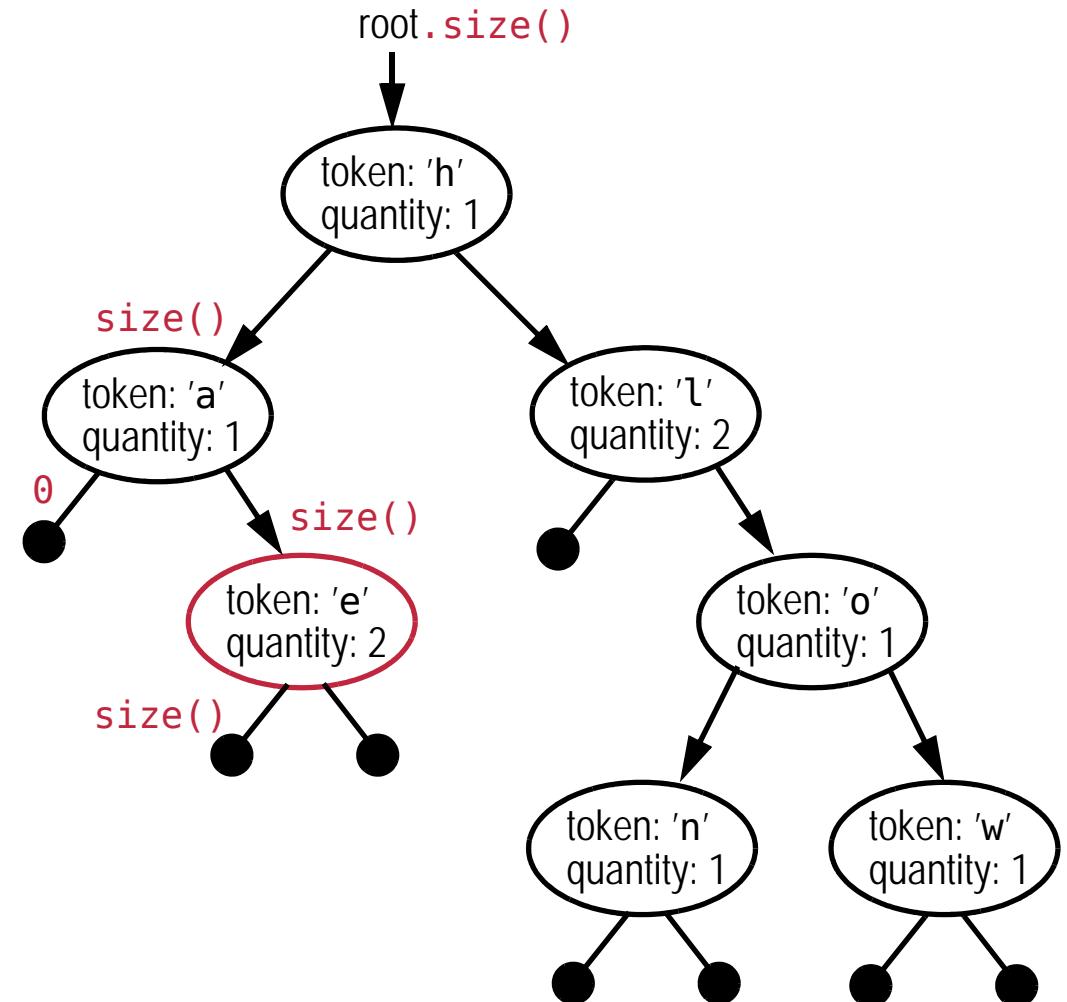
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

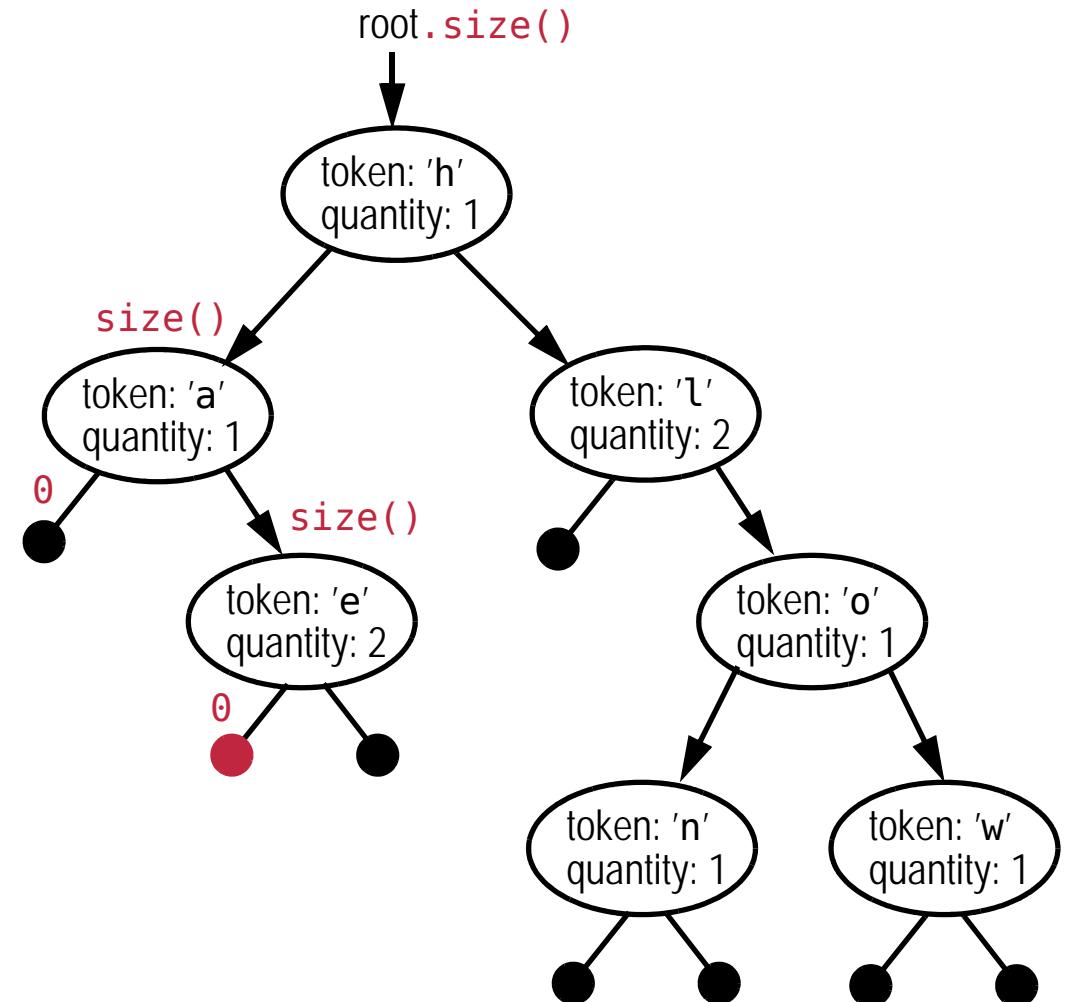
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

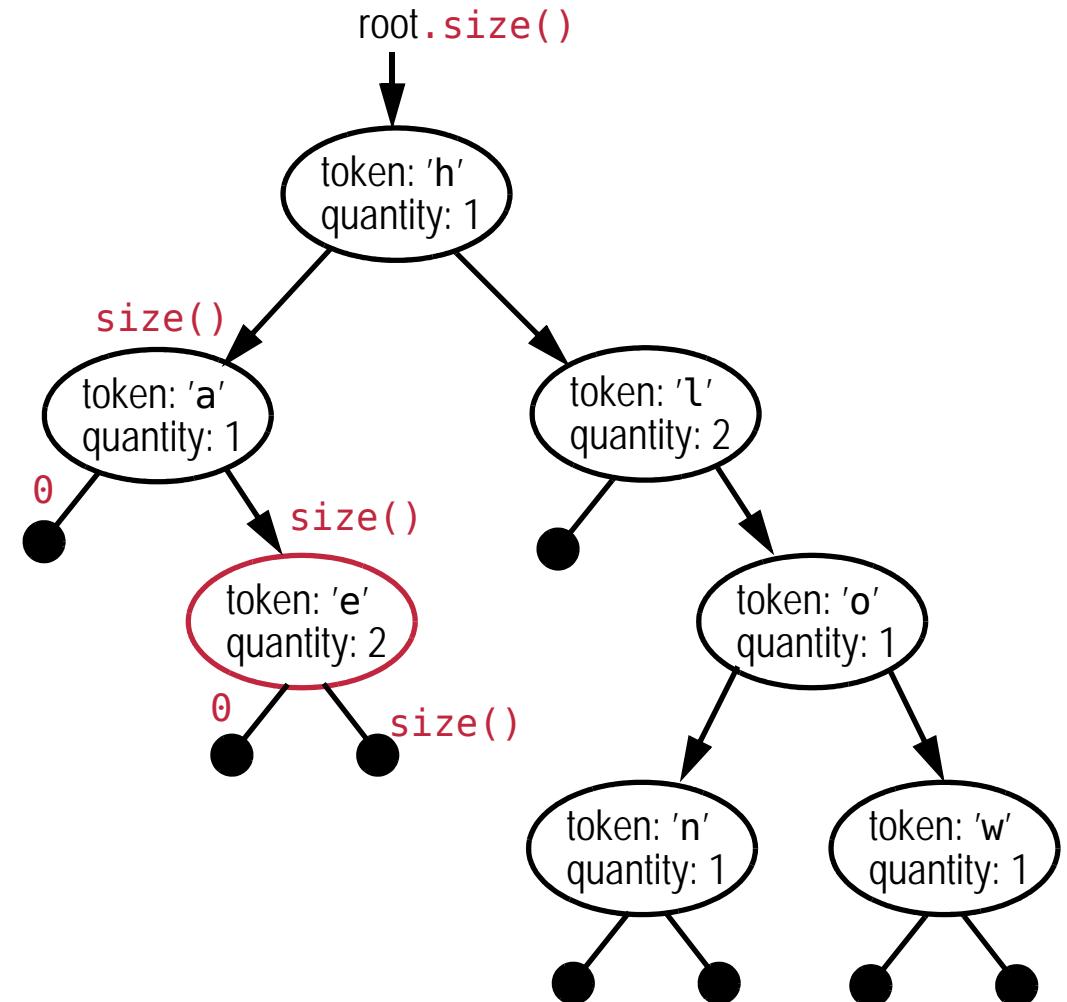
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

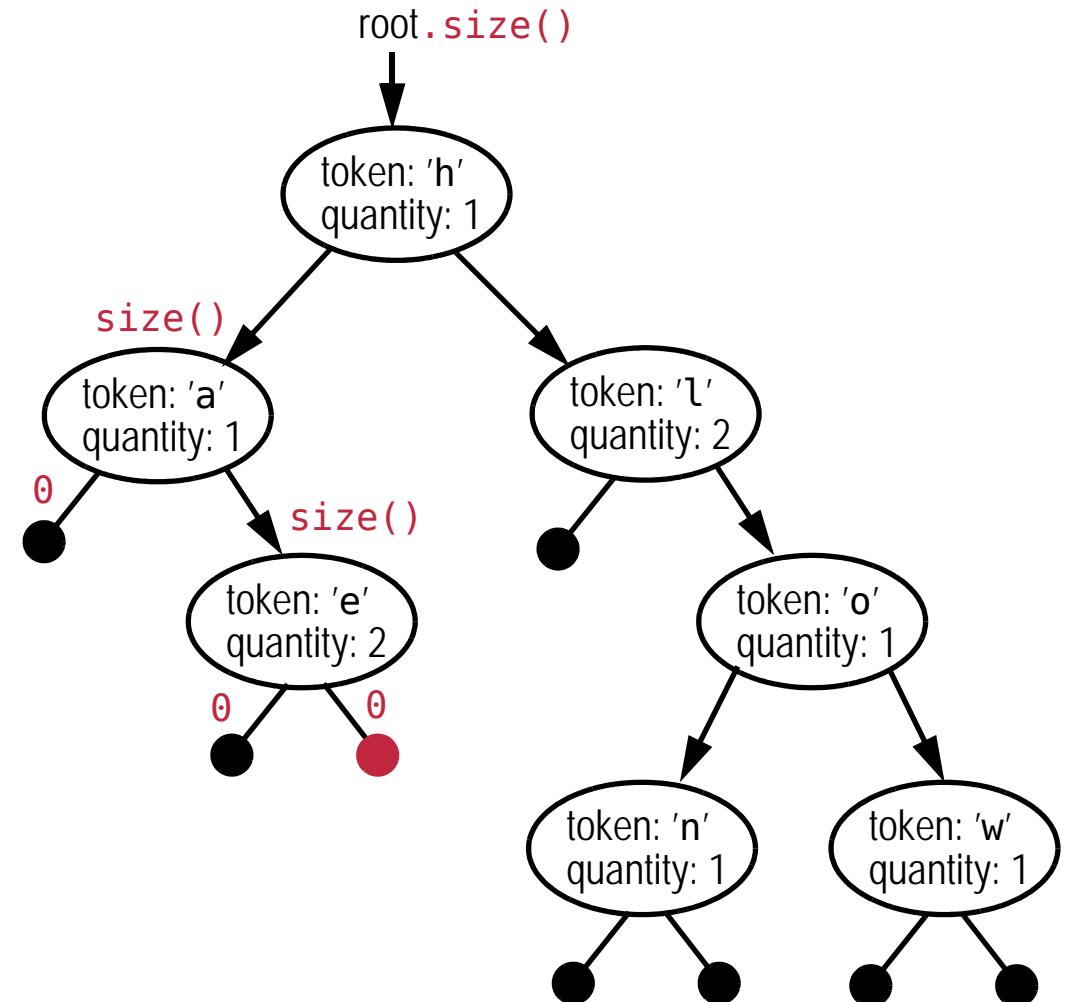
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

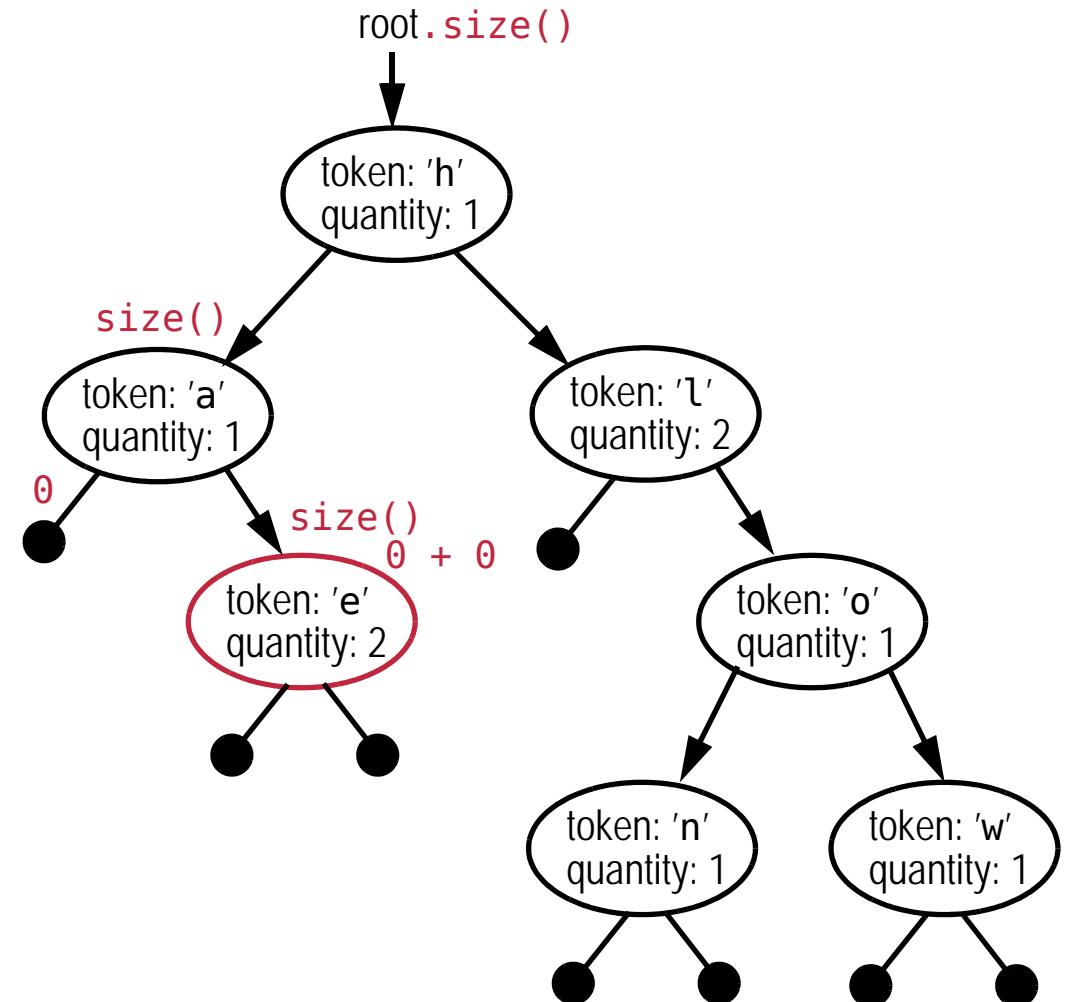
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

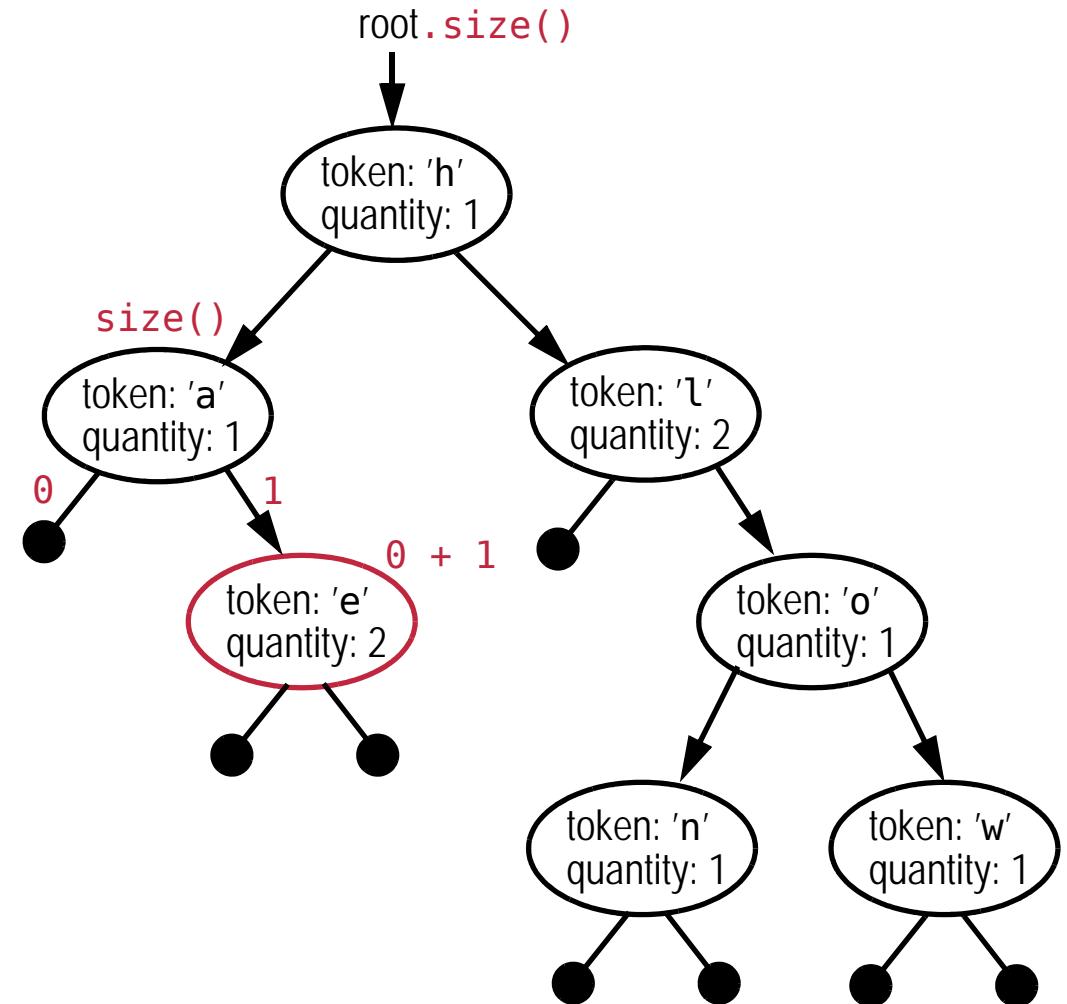
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

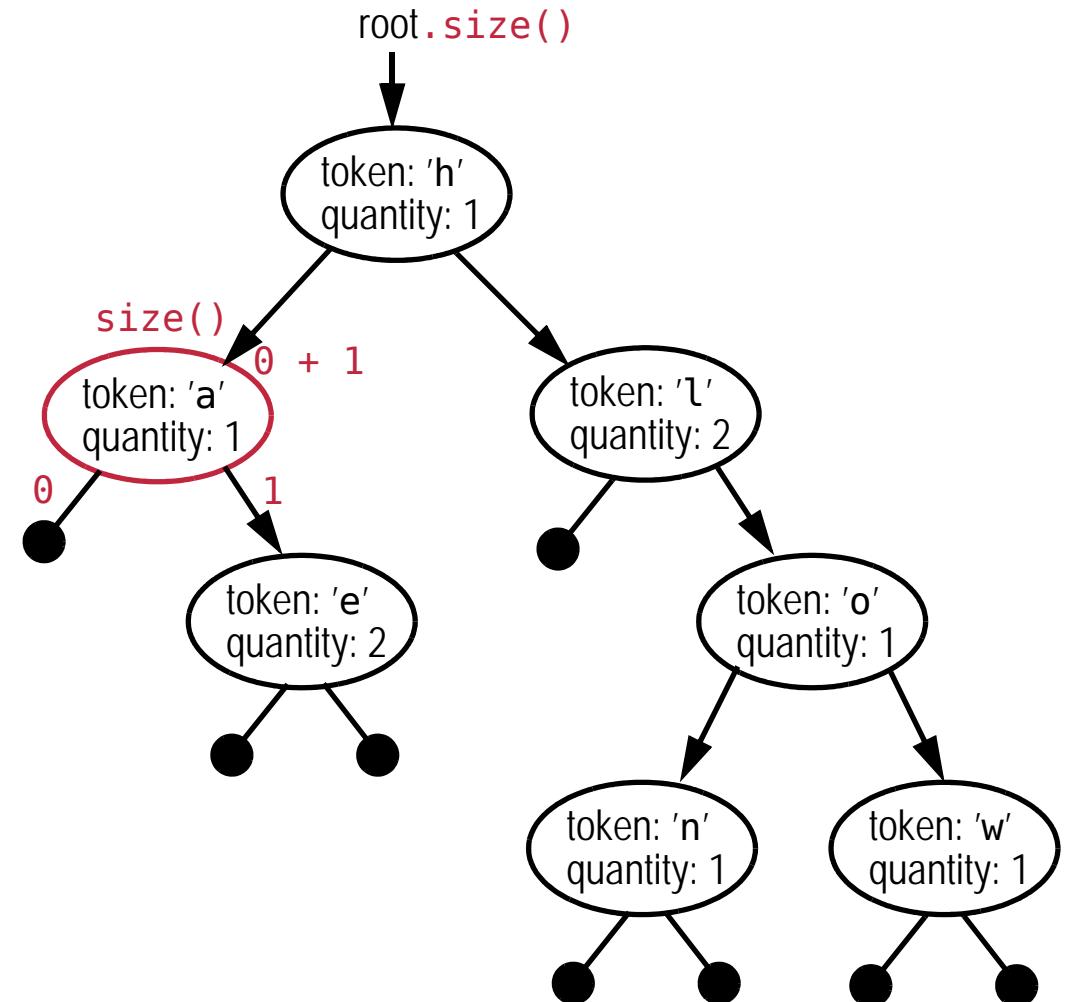
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

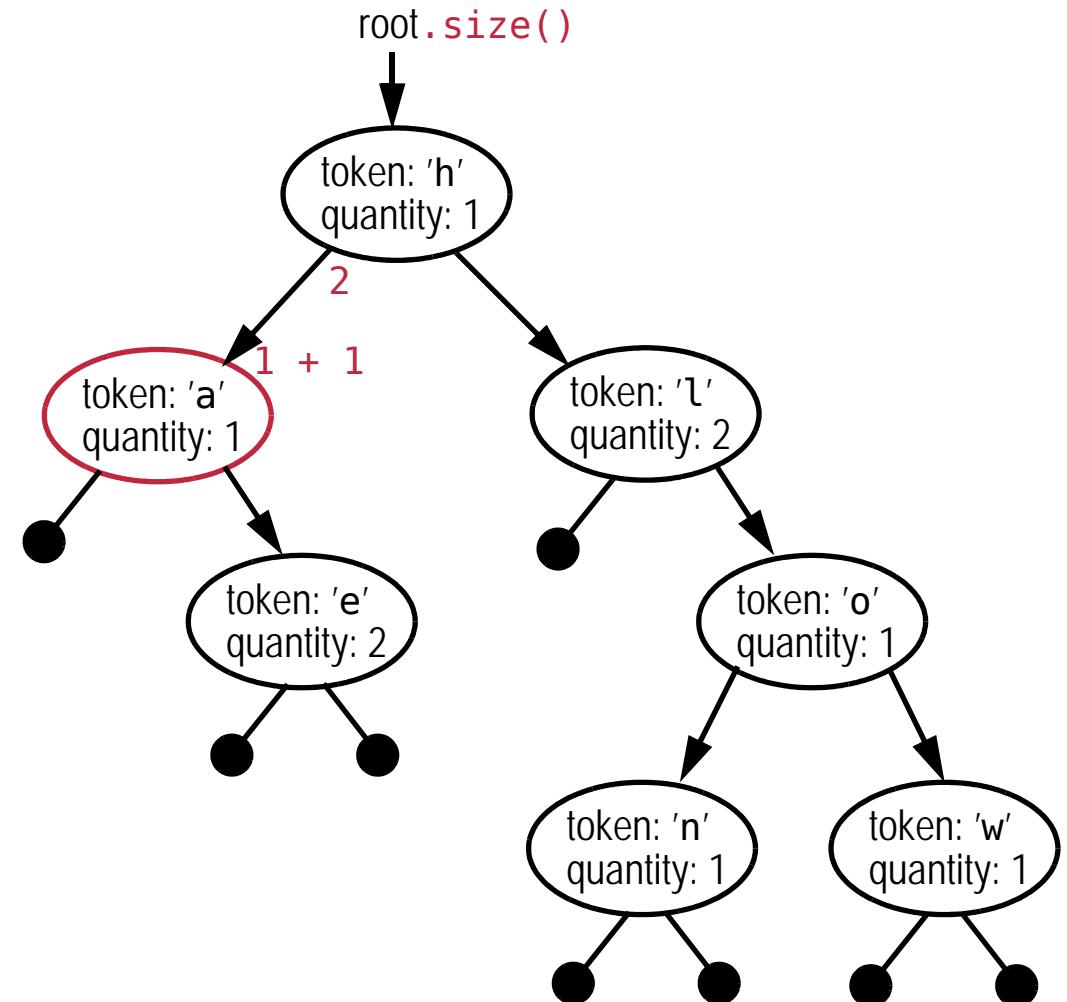
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

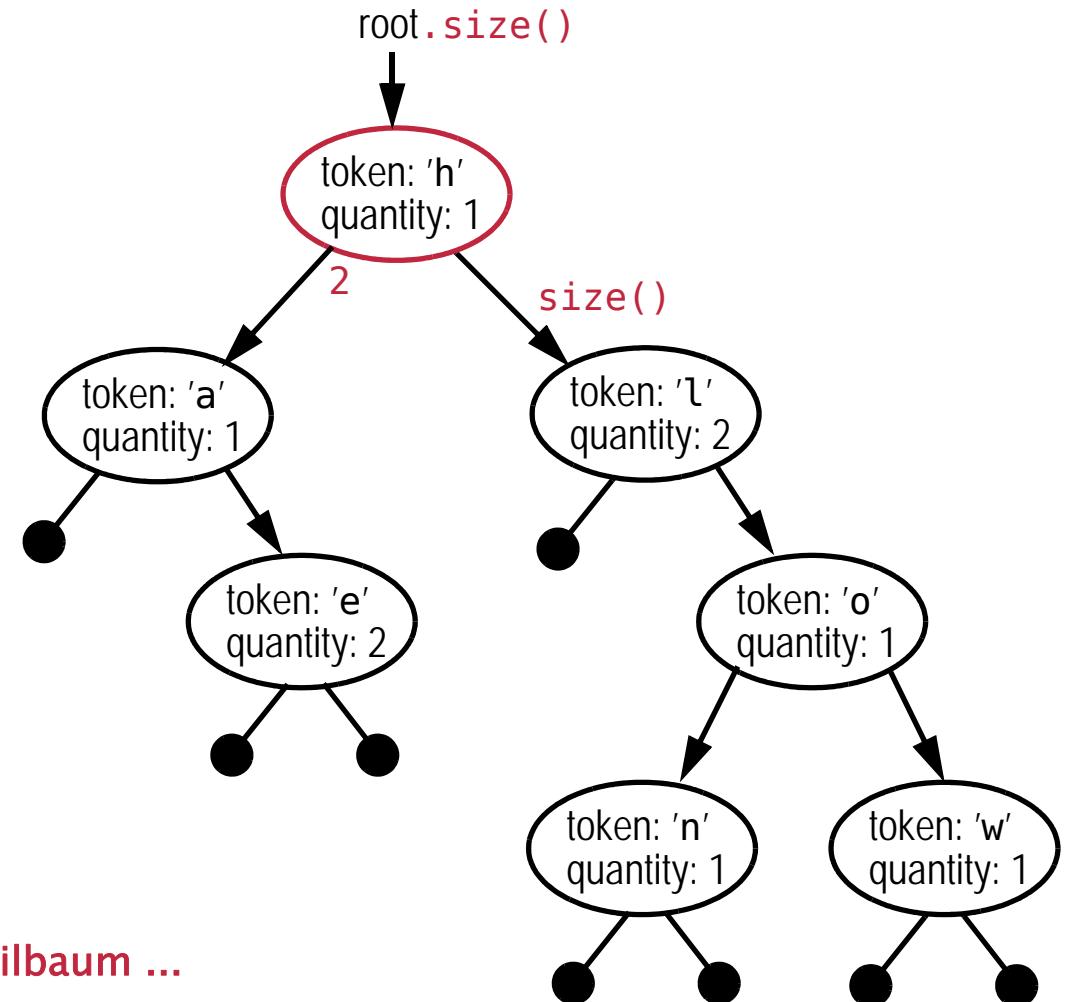
```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



## Ausführung von size

(Fortsetzung)

```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size()
            + rightChild.size()
            + 1;
    }
}
```



und analog fortsetzen für den rechten Teilbaum ...

## Zählen von Zeichen

(Fortsetzung)

Die Ausgabe der abgelegten, unterschiedlichen Zeichen und ihrer Häufigkeiten lässt sich ebenfalls an die nachfolgenden Teilbäume übertragen und daher rekursiv formulieren:

```
public void show()
{
    if ( !isEmpty() ) ←
    {
        leftChild.show();
        System.out.println( content.toString() );
        rightChild.show();
    }
}
```

leerer Baum würde  
kein Zeichen enthalten

- Abbruchkriterium: Aufruf für den leeren Baum

## Zählen von Zeichen

(Fortsetzung)

Die Ausgabe der abgelegten, unterschiedlichen Zeichen und ihrer Häufigkeiten lässt sich ebenfalls an die nachfolgenden Teilbäume übertragen und daher rekursiv formulieren:

```
public void show()
{
    if ( !isEmpty() )
    {
        leftChild.show();
        System.out.println( content.toString() );
        rightChild.show();
    }
}
```

rekursive Aufrufe

- Ausgabe des linken Teilbaums durch rekursiven Aufruf der Methode `show`
- Ausgabe des Inhalts der Wurzel
- Ausgabe des rechten Teilbaums durch rekursiven Aufruf der Methode `show`

*InOrder-Durchlauf:* Die Wurzel wird zwischen (*in*) den beiden Teilbäumen bearbeitet.

## Zählen von Zeichen

(Fortsetzung)

- ❑ Die Übergabe der abgelegten, unterschiedlichen Zeichen und ihrer Häufigkeiten an ein Feld wird als Eingabe für die Klasse `HuffmanCoding` benötigt.
- ❑ Das Erzeugen eines solchen Feldes lässt sich ebenfalls über einen *InOrder-Durchlauf* rekursiv formulieren.
- ❑ Die Methode `toArray()` bereitet den *InOrder-Durchlauf* vor, indem ein passendes Feld für `HuffmanTriple`-Objekte bereitgestellt wird.
- ❑ Die Anzahl der benötigten Elemente wird durch den Aufruf der Methode `size()` ermittelt.

```
public HuffmanTriple[] toArray()
{
    if ( !isEmpty() )
    {
        HuffmanTriple[] collector = new HuffmanTriple[size()];
        toArray( collector, 0 );
        return collector;
    }
    return new HuffmanTriple[0];
}
```



## Zählen von Zeichen

(Fortsetzung)

- ❑ Die Methode `toArray( HuffmanTriple[] collector, int index )` führt den *InOrder-Durchlauf* durch, indem die in den Wurzeln der Teilbäume gefundenen `HuffmanTriple`-Objekte in das Feld eingefügt werden.
- ❑ Die Position für den nächsten Zugriff auf das Feld wird im Parameter `index` übergeben.
- ❑ Der Ablauf der Rekursion wird über die besuchten Teilbäume und nicht über die beiden Parameter der Methode gesteuert.

```
private int toArray( HuffmanTriple[] collector, int index )
{
    if ( !isEmpty() )
    {
        index = leftChild.toArray( collector, index );
        collector[index] = content; ← nach der Zuweisung
        index = rightChild.toArray( collector, index + 1 ); ← wird index erhöht
    }
    return index;
}
```

## Zählen von Zeichen

(Fortsetzung)

Anmerkungen:

- ❑ Der durch die Klasse `CharacterSearchTree` aufgebaute binäre Baum ist eine *rekursive* Datenstruktur.
- ❑ Rekursive Datenstrukturen lassen sich leicht mit *rekursiven* Methoden bearbeiten.
- ❑ Die vorgestellten rekursiven Methoden unterscheiden sich darin, wie die Rekursion eingesetzt wird:
  - Die Methoden `size()`, `show()` und `toArray` nutzen die Rekursion, um alle Teilbäume zu erreichen.
  - Die Methode `add()` folgt immer nur einem bestimmten, durch die Ergebnisse der Vergleiche vorgegebenen Pfad von Knoten durch einen Teilbaum.

Daher kann die Methode `add()` auch iterativ, also durch Nutzung einer Schleife, formuliert werden.

## Zählen von Zeichen

(Fortsetzung)

```
public void iterativeAdd( char t )
{
    CharacterSearchTree current = this;
    while ( !current.isEmpty() && current.content.getToken() != t )
    {
        if ( current.content.getToken() > t )
        {
            current = current.leftChild;
        }
        else
        {
            current = current.rightChild;
        }
    }
    if ( current.isEmpty() )
    {
        current.content = new HuffmanTriple( t );
        current.leftChild = new CharacterSearchTree();
        current.rightChild = new CharacterSearchTree();
    }
    else
    {
        current.content.incrementQuantity();
    }
}
```

## Zählen von Zeichen

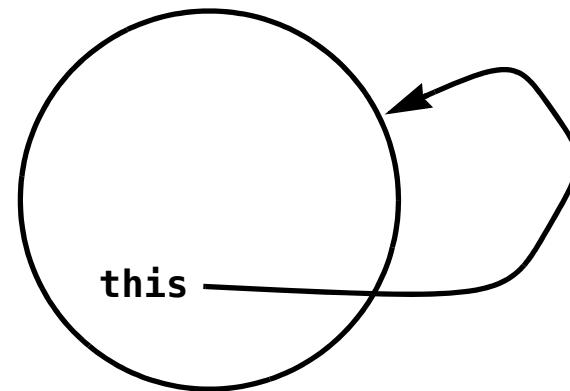
(Fortsetzung)

```
public void iterativeAdd( char t )
{
    CharacterSearchTree current = this; this ist Referenz auf das ausführende Objekt
    while ( !current.isEmpty() && current.content.getToken() != t )
    {
        if ( current.content.getToken() > t )
        {
            current = current.leftChild;
        }
        else
        {
            current = current.rightChild;
        }
    }
    if ( current.isEmpty() )
    {
        current.content = new HuffmanTriple( t );
        current.leftChild = new CharacterSearchTree();
        current.rightChild = new CharacterSearchTree();
    }
    else
    {
        current.content.incrementQuantity();
    }
}
```

## Referenz **this**

### Anmerkungen

- Die Referenz **this** ist für jedes Objekt definiert.
- **this** erlaubt den Zugriff auf das ausführende Objekt:  
Methoden werden so in die Lage versetzt, eine Referenz auf das sie ausführende Objekt zurückzugeben oder das eigene Objekt an andere Methoden als Argument zu übergeben.
- **this** hilft beim Auflösen von Namenskonflikten:  
Namenskonflikte zwischen Variablen und Attributen werden von Java immer zugunsten der Variablen entschieden, da diese immer innerhalb von Methoden und damit «lokaler» deklariert sind. **this** bezeichnet das Objekt, **this**. erlaubt den Zugriff auf verdeckte Attribute.



## Zählen von Zeichen

(Fortsetzung)

```
public void iterativeAdd( char t )
{
    CharacterSearchTree current = this;
    while ( !current.isEmpty() && current.content.getToken() != t )
    {
        if ( current.content.getToken() > t )
        {
            current = current.leftChild;
        }
        else
        {
            current = current.rightChild;
        }
    }
    if ( current.isEmpty() )
    {
        current.content = new HuffmanTriple( t );
        current.leftChild = new CharacterSearchTree();
        current.rightChild = new CharacterSearchTree();
    }
    else
    {
        current.content.incrementQuantity();
    }
}
```

Schleife bestimmt  
Position für Aktion

## Zählen von Zeichen

(Fortsetzung)

```

public void iterativeAdd( char t )
{
    CharacterSearchTree current = this;
    while ( !current.isEmpty() && current.content.getToken() != t )
    {
        if ( current.content.getToken() > t )
        {
            current = current.leftChild;
        }
        else
        {
            current = current.rightChild;
        }
    }
    if ( current.isEmpty() ) ← neues Zeichen anlegen
    {
        current.content = new HuffmanTriple( t );
        current.leftChild = new CharacterSearchTree();
        current.rightChild = new CharacterSearchTree();
    }
    else
    {
        current.content.incrementQuantity(); ← Häufigkeit erhöhen
    }
}

```

## Klasse CharacterSearchTree – abschließender Überblick

```
public class CharacterSearchTree
{
    private HuffmanTriple content;
    private CharacterSearchTree leftChild, rightChild;                                Attribute

    public CharacterSearchTree() { ... }                                              Konstruktor

    public HuffmanTriple getContent() { ... }
    public boolean isEmpty() { ... }
    public boolean isLeaf() { ... }
    public void add( char t ) { ... }
    public void iterativeAdd( char t ) { ... }
    public String getCode( char t ) { ... }
    public int size() { ... }
    public void show() { ... }
    public HuffmanTriple[] toArray() { ... }                                         öffentliche Methoden

    private int toArray( HuffmanTriple[] collector, int index ) { ... }                private Methode
}
```

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 7.2. Binärer Suchbaum – Einsatz bei Datenkompression

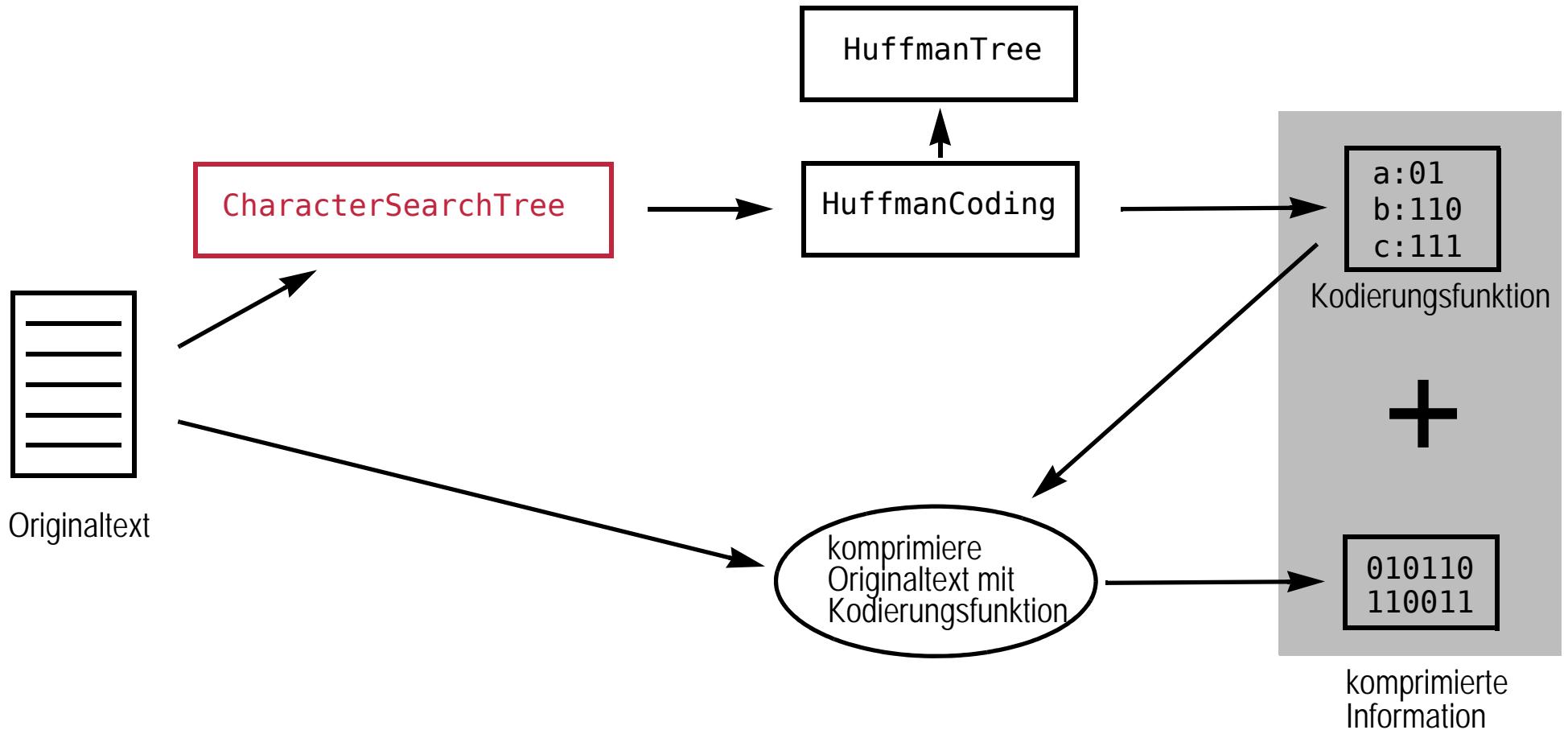
Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-606

## Datenkompression – Motivation (Erinnerung – Folie 492 und Folie 548)

Ablauf einer Textkompression:



## Datenkompression – Beispiele

```
public static void firstTreeTest()
{
    String s = "halloween" ; Eingabetext

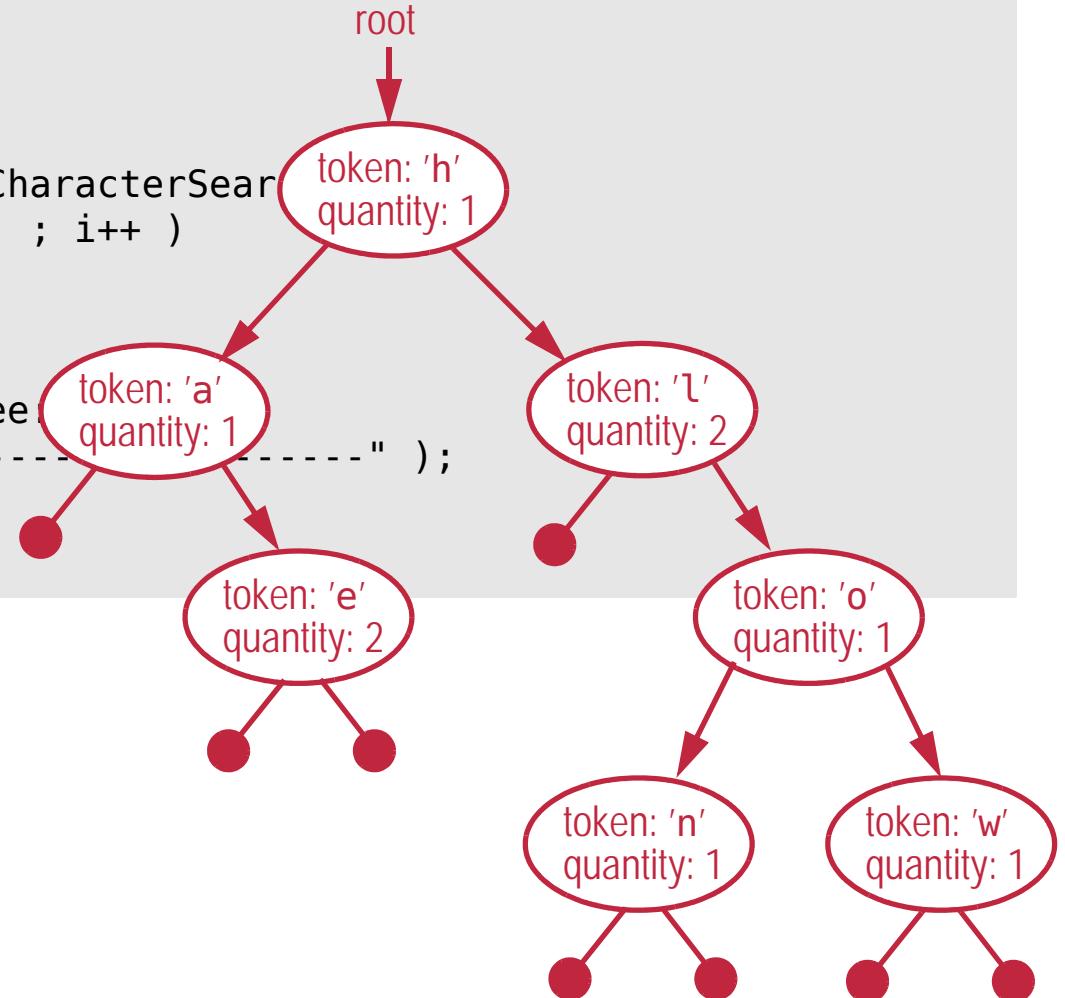
    CharacterSearchTree hal = new CharacterSearchTree(); Erzeugen
    for ( int i = 0; i < s.length() ; i++ )
    {
        hal.add( s.charAt( i ) ); Einfügen
    }
    System.out.println( "binary tree: " );
    System.out.println( "-----" );
    hal.show(); Ausgeben
}
```

## Datenkompression – Beispiele

(Fortsetzung)

```
public static void firstTreeTest()
{
    String s = "halloween" ;

    CharacterSearchTree hal = new CharacterSearchTree();
    for ( int i = 0; i < s.length(); i++ )
    {
        hal.add( s.charAt( i ) );
    }
    System.out.println( "binary tree" );
    System.out.println( "-----" );
    hal.show();
}
```



## Datenkompression – Beispiele

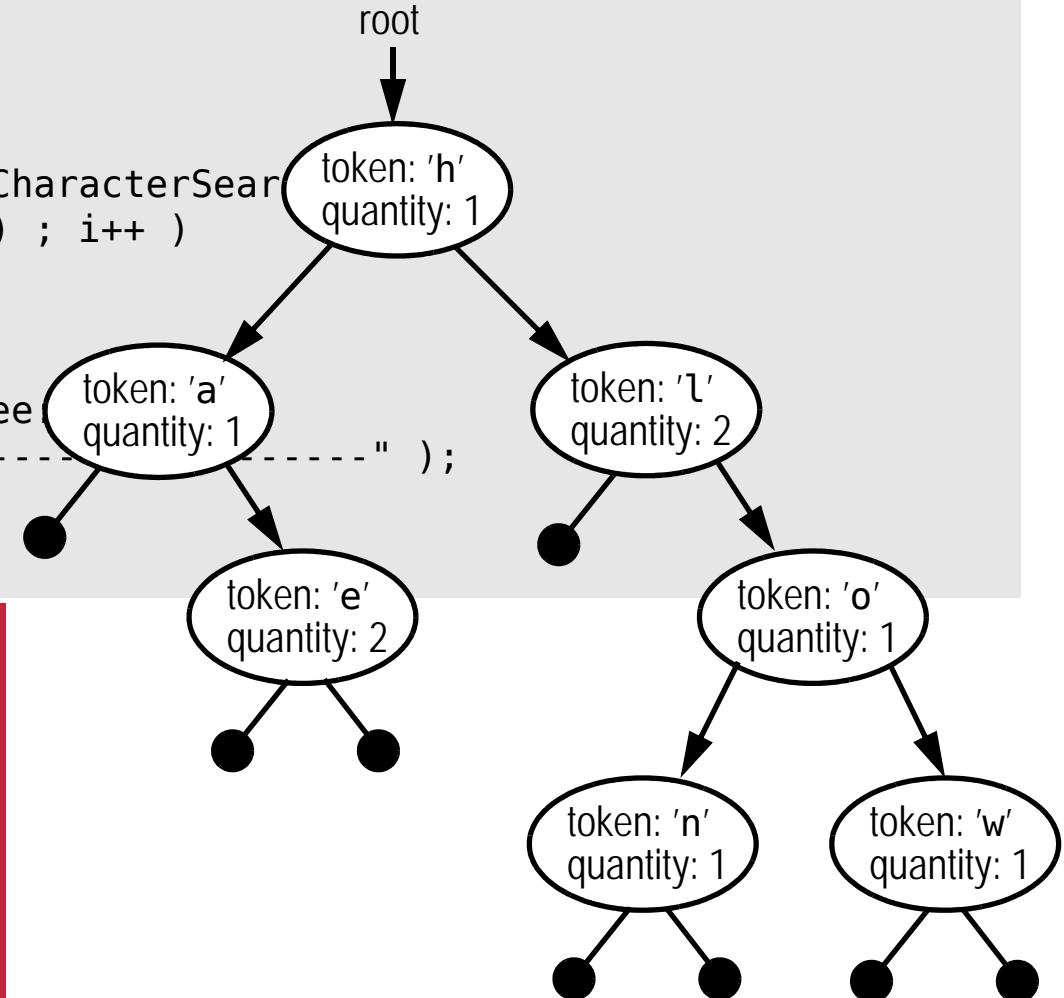
(Fortsetzung)

```
public static void firstTreeTest()
{
    String s = "halloween" ;

    CharacterSearchTree hal = new CharacterSearchTree();
    for ( int i = 0; i < s.length(); i++ )
    {
        hal.add( s.charAt( i ) );
    }
    System.out.println( "binary tree" );
    System.out.println( "-----" );
    hal.show();
}
```

binary tree:

```
-----
token (quantity: 1): a -> code:
token (quantity: 2): e -> code:
token (quantity: 1): h -> code:
token (quantity: 2): l -> code:
token (quantity: 1): n -> code:
token (quantity: 1): o -> code:
token (quantity: 1): w -> code:
```



## Datenkompression – Beispiele

(Fortsetzung)

```
public static void secondTreeTest()
{
    String s = "Die Würde des Menschen ist unantastbar. Sie zu achten " +
               "und zu schützen ist Verpflichtung aller staatlichen Gewalt. " +
               "Das Deutsche Volk bekennt sich darum zu unverletzlichen und " +
               "unveräußerlichen Menschenrechten als Grundlage jeder menschlichen " +
               "Gemeinschaft, des Friedens und der Gerechtigkeit in der Welt. ";

    CharacterSearchTree gg = new CharacterSearchTree();
    for ( int i = 0; i < s.length() ; i++ )
    {
        gg.add( s.charAt( i ) );
    }
    System.out.println( "binary tree: " );
    System.out.println( "-----" );
    gg.show();
}
```

GG Artikel 1



## Datenkompression – Beispiele

(Fortsetzung)

Ausgabe:

```
binary tree:  
-----  
token (quantity: 41):   -> code:  
token (quantity: 1): , -> code:  
token (quantity: 3): . -> code:  
token (quantity: 3): D -> code:  
token (quantity: 1): F -> code:  
token (quantity: 4): G -> code:  
token (quantity: 2): M -> code:  
token (quantity: 1): S -> code:  
token (quantity: 2): V -> code:  
token (quantity: 2): W -> code:  
token (quantity: 13): a -> code:  
token (quantity: 2): b -> code:  
token (quantity: 15): c -> code:  
token (quantity: 12): d -> code:  
token (quantity: 42): e -> code:  
token (quantity: 2): f -> code:  
token (quantity: 3): g -> code:
```

```
token (quantity: 15): h -> code:  
token (quantity: 15): i -> code:  
token (quantity: 1): j -> code:  
token (quantity: 3): k -> code:  
token (quantity: 13): l -> code:  
token (quantity: 3): m -> code:  
token (quantity: 26): n -> code:  
token (quantity: 1): o -> code:  
token (quantity: 1): p -> code:  
token (quantity: 15): r -> code:  
token (quantity: 16): s -> code:  
token (quantity: 18): t -> code:  
token (quantity: 14): u -> code:  
token (quantity: 2): v -> code:  
token (quantity: 1): w -> code:  
token (quantity: 5): z -> code:  
token (quantity: 1): ß -> code:  
token (quantity: 1): ä -> code:  
token (quantity: 2): ü -> code:
```

## Datenkompression – Komprimierungsvorgang

Problem:

Zu jedem Zeichen des Originaltextes muss die Kodierung ermittelt und in den komprimierten Text eingetragen werden.

Problemanalyse:

- Bei einem langen Text muss also sehr häufig gesucht werden. Die Datenstruktur sollte also das schnelle Suchen (und Finden) der Kodierung unterstützen.
- Die passende Datenstruktur ist schon bekannt:

*binärer Suchbaum*

- Aus dem Huffman-Baum, der die Kodierungen der Zeichen des Originaltextes enthält, muss also ein binärer Suchbaum abgeleitet werden, der die gleichen Informationen enthält.

## Datenkompression – Komprimierungsvorgang

Problem:

Zu jedem Zeichen des Originaltextes muss die Kodierung ermittelt und in den komprimierten Text eingetragen werden.

Problemanalyse:

- Bei einem langen Text muss also sehr häufig gesucht werden. Die Datenstruktur sollte also das schnelle Suchen (und Finden) der Kodierung unterstützen.
- Die passende Datenstruktur ist schon bekannt:

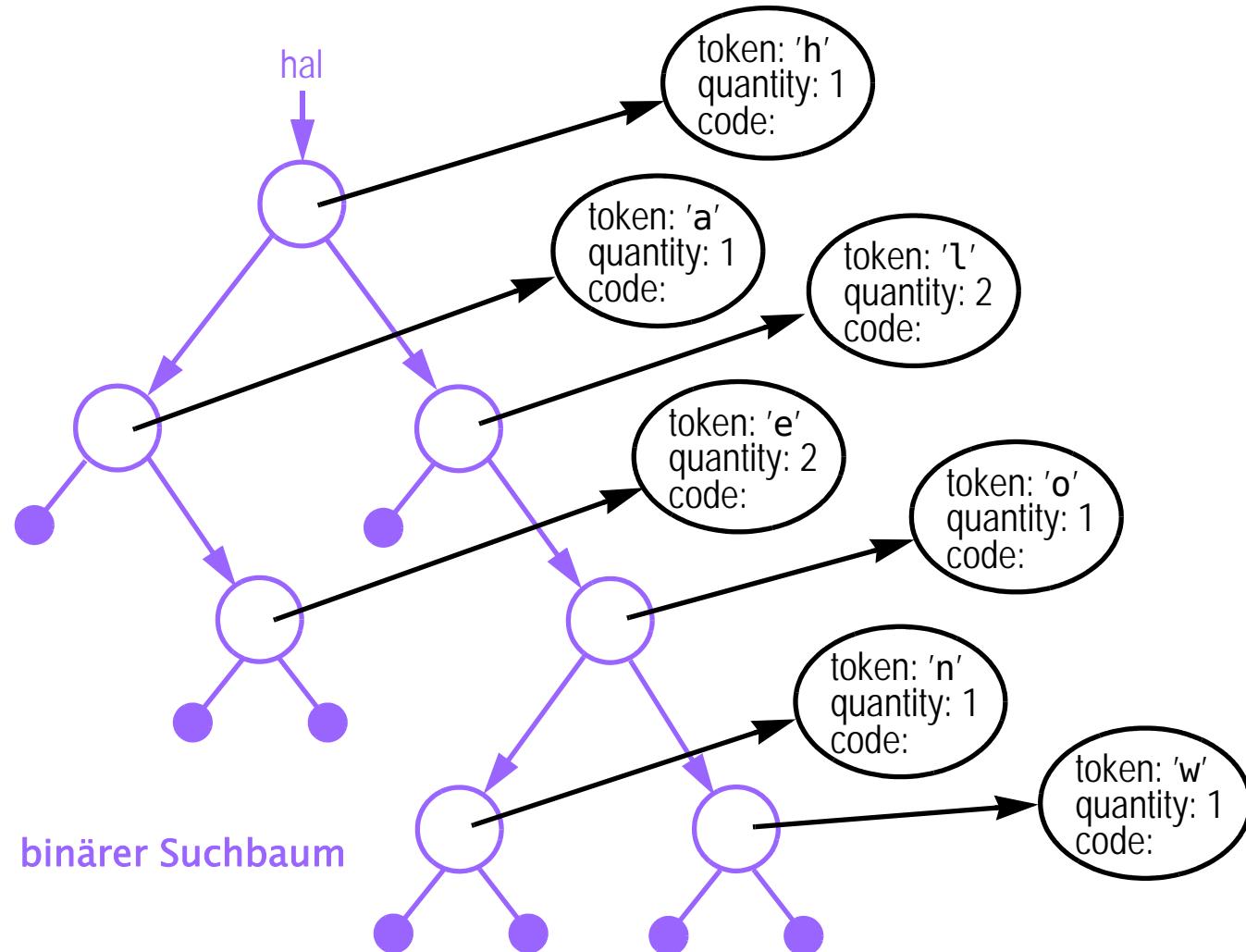
*binärer Suchbaum*

- Aus dem Huffman-Baum, der die Kodierungen der Zeichen des Originaltextes enthält, muss also ein binärer Suchbaum abgeleitet werden, der die gleichen Informationen enthält.
- aber:**
  - Die im Huffman-Baum abgelegten `HuffmanTriple`-Objekte werden beim Einfügen in einen binären Suchbaum erzeugt.
  - Auch bei der Übergabe an den Huffman-Baum können die Referenzen im binären Suchbaum erhalten bleiben.
  - Die Kodierung kann nach ihrer Erzeugung durch den Konstruktor der Klasse `HuffmanCoding` direkt über den binären Suchbaum abgerufen werden.

## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

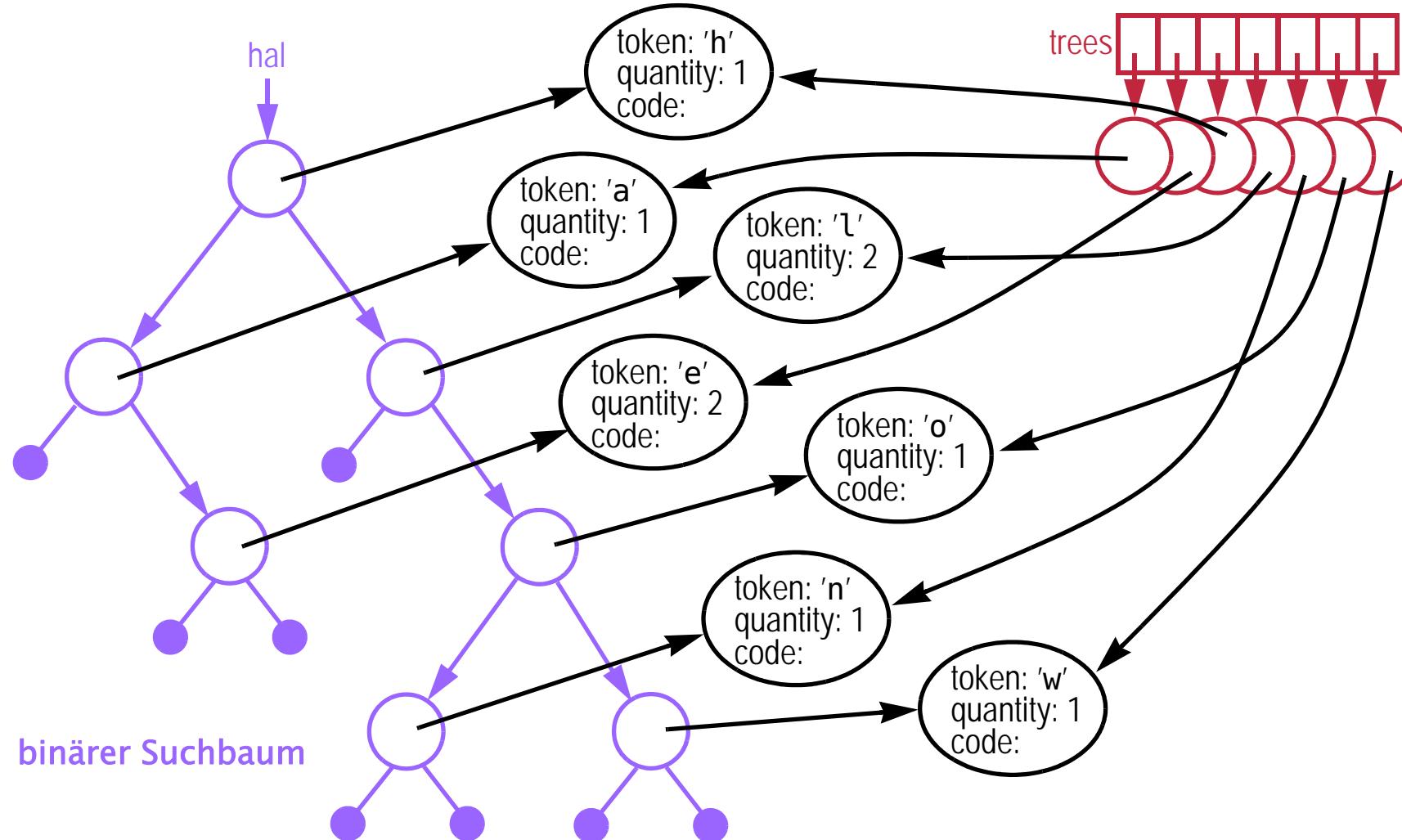
Visualisierung:



## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

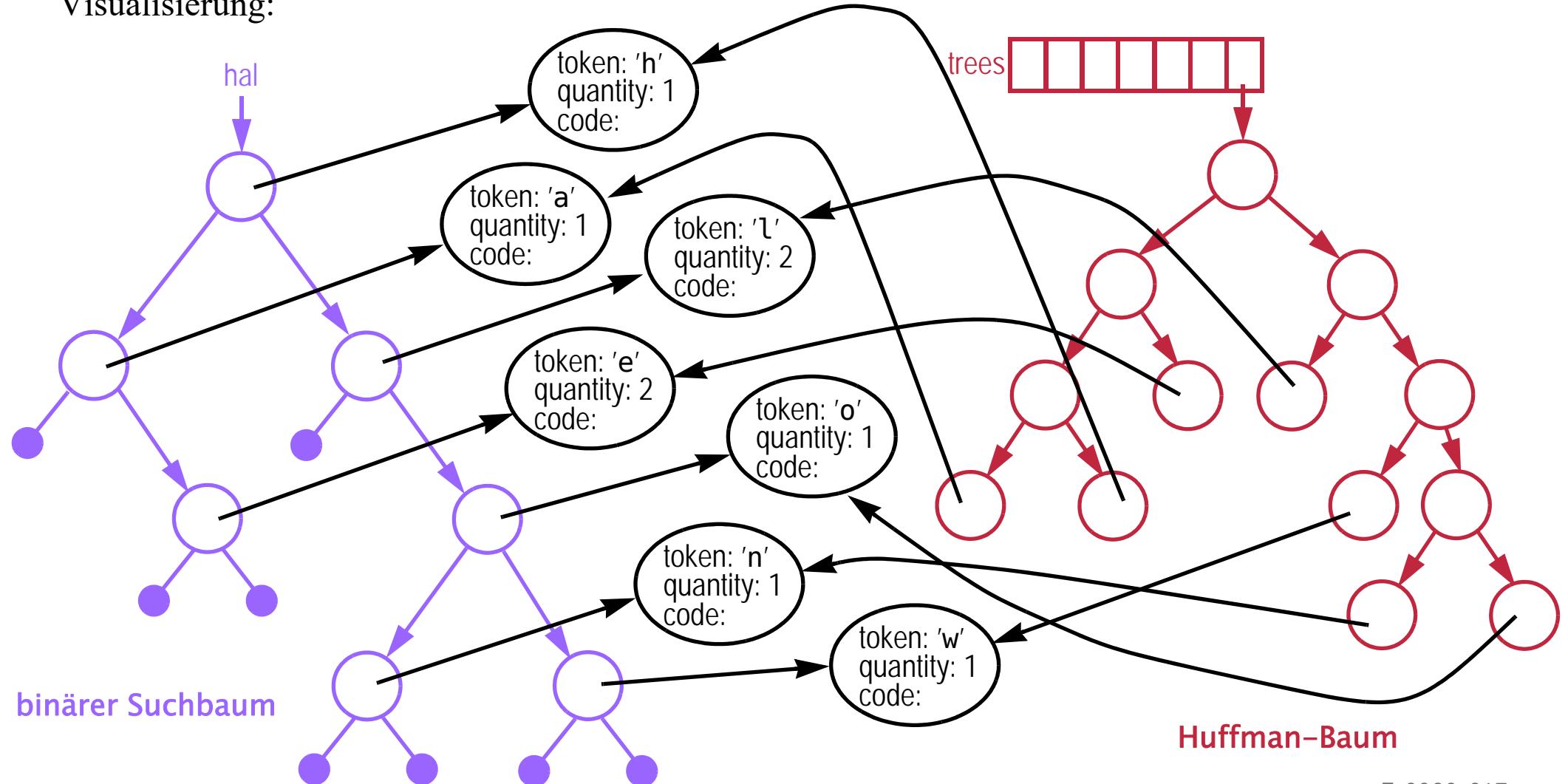
Visualisierung:



## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

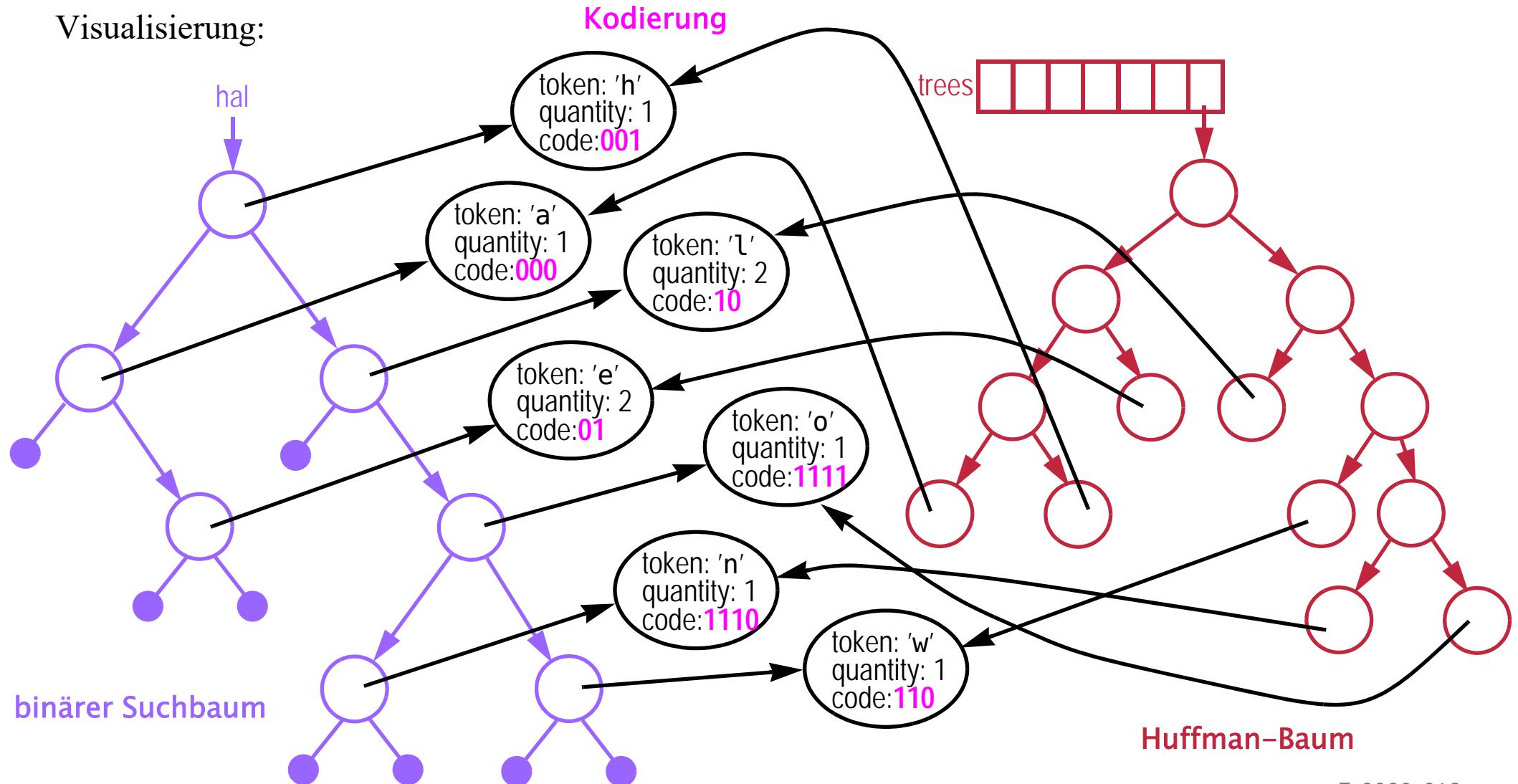
Visualisierung:



## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

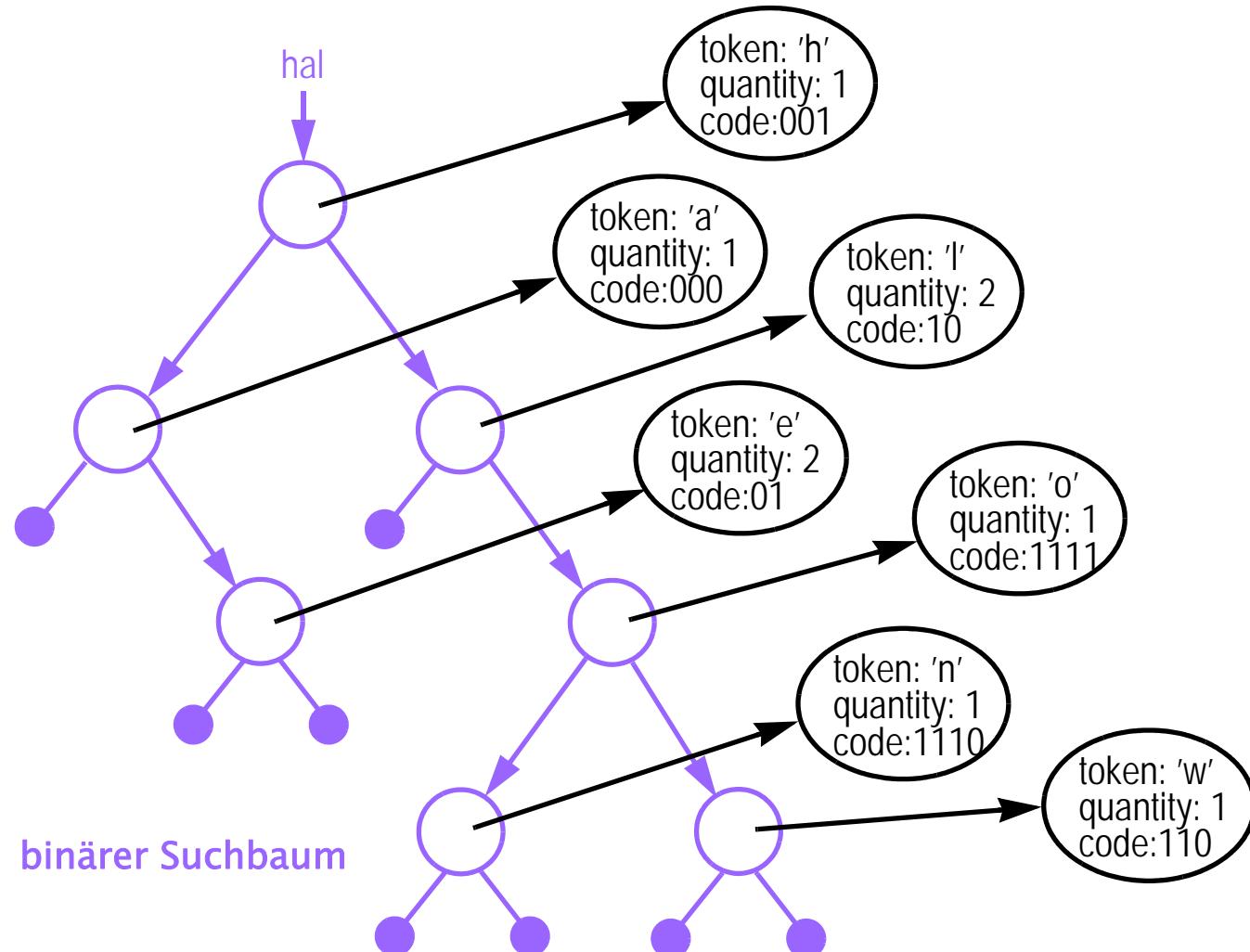
Visualisierung:



## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

Visualisierung:



## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

- ❑ Benötigt wird eine Methode, die ein Zeichen im binären Suchbaum sucht und die Kodierung zurückgibt.

```
public String getCode( char t )
{
    if ( !isEmpty() )
    {
        if ( content.getToken() > t )
        {
            return leftChild.getCode( t );
        } else if ( content.getToken() < t )
        {
            return rightChild.getCode( t );
        } else
        {
            return content.getCode();
        }
    } else
    {
        throw new IllegalStateException();
    }
}
```

Ablauf ist analog  
zur Methode add

Rückgabe der gefundenen  
Kodierung

Fehler: Kodierung nicht  
gefunden

## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

```
public static void firstTreeTest()
{
    String s = "halloween" ;
    CharacterSearchTree hal = new CharacterSearchTree();
    for ( int i = 0; i < s.length() ; i++ )
    {
        hal.add( s.charAt( i ) );
    }
    HuffmanCoding coding = new HuffmanCoding( hal.toArray() );
    String codeOfHal = "";
    for ( int i = 0; i < s.length() ; i++ )
    {
        codeOfHal += hal.getCode( s.charAt( i ) );
    }
    System.out.println( "bit code: " );
    System.out.println( "-----" );
    System.out.println( codeOfHal );
}
```

Einfügen

Kodierung generieren

Komprimieren

Ausgabe

## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

```
public static void firstTreeTest()
{
    String s = "halloween" ;
    CharacterSearchTree hal = new CharacterSearchTree();
    for ( int i = 0; i < s.length() ; i++ )
    {
        hal.add( s.charAt( i ) );
    }
    HuffmanCoding coding = new HuffmanCoding( hal.toArray() );
    String codeOfHal = "";
    for ( int i = 0; i < s.length(); i++ )
    {
        codeOfHal += -----
    }
    System.out.print( codeOfHal );
    System.out.print( "-----" );
    System.out.println();
}
```

binary tree with codes:

-----

token (quantity: 1): a -> code: 000

token (quantity: 2): e -> code: 01

token (quantity: 1): h -> code: 001

token (quantity: 2): l -> code: 10

token (quantity: 1): n -> code: 1110

token (quantity: 1): o -> code: 1111

token (quantity: 1): w -> code: 110

## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

```

public static void firstTreeTest()
{
    String s = "halloween" ;
    CharacterSearchTree hal = new CharacterSearchTree();
    for ( int i = 0; i < s.length() ; i++ )
    {
        hal.add( s.charAt( i ) );
    }
    HuffmanCoding coding = new HuffmanCoding( hal.toArray() );
    String codeOfHal = "";
    for ( int i = 0; i < s.length(); i++ )
    {
        codeOfHal += ...
    }
    System.out.print ...
    System.out.print ...
    System.out.print ...
}

```

binary tree with codes:  
-----  
token (quantity: 1): a -> code: 000  
token (quantity: 2): e -> code: 01  
token (quantity: 1): h -> code: 001  
token (quantity: 2): l -> code: 10  
token (quantity: 1): n -> code: 1110  
token (quantity: 1): o -> code: 1111  
token (quantity: 1): w -> code: 110

bit code:  
-----  
001000101011111001011110

## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

```

public static void firstTreeTest()
{
    String s = "halloween" ;
    CharacterSearchTree hal = new CharacterSearchTree();
    for ( int i = 0; i < s.length() ; i++ )
    {
        hal.add( s.charAt( i ) );
    }
    HuffmanCoding coding = new HuffmanCoding( hal.toArray() );
    String codeOfHal = "";
    for ( int i = 0; i < s.length(); i++ )
    {
        codeOfHal += ...
    }
    System.out.print ...
    System.out.print ...
    System.out.print ...
}

```

binary tree with codes:  
-----  
token (quantity: 1): a -> code: 000  
token (quantity: 2): e -> code: 01  
token (quantity: 1): h -> code: 001  
token (quantity: 2): l -> code: 10  
token (quantity: 1): n -> code: 1110  
token (quantity: 1): o -> code: 1111  
token (quantity: 1): w -> code: 110

bit code:  
-----  
00100010101111100101110

## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

```
public static void secondTreeTest()
{
    String s = "Die Würde .....";

    CharacterSearchTree gg = new CharacterSearchTree();
    for ( int i = 0; i < s.length() ; i++ )
    {
        gg.add( s.charAt( i ) );
    }
    HuffmanCoding coding = new HuffmanCoding( gg.toArray() );
    String codeOfGG = "";
    for ( int i = 0; i < s.length() ; i++ )
    {
        codeOfGG += gg.getCode( s.charAt( i ) );
    }
    System.out.println( "bit code: " );
    System.out.println( "-----" );
    for ( int i = 0; i < codeOfGG.length() ; i++ )
    {
        System.out.print( codeOfGG.charAt( i ) );
        // Formatierung fehlt hier
    }
}
```

## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

binary tree with codes:

```
-----
token (quantity: 41):   -> code: 100
token (quantity: 1): , -> code: 01011100
token (quantity: 3): . -> code: 1101011
token (quantity: 3): D -> code: 1101000
token (quantity: 1): F -> code: 01011101
token (quantity: 4): G -> code: 010101
token (quantity: 2): M -> code: 0101111
token (quantity: 1): S -> code: 01010010
token (quantity: 2): V -> code: 0101100
token (quantity: 2): W -> code: 0101101
token (quantity: 13): a -> code: 11100
token (quantity: 2): b -> code: 1100010
token (quantity: 15): c -> code: 0001
token (quantity: 12): d -> code: 11011
token (quantity: 42): e -> code: 101
token (quantity: 2): f -> code: 1100011
token (quantity: 3): g -> code: 1101001
token (quantity: 15): h -> code: 0010
```

```
token (quantity: 15): i -> code: 0011
token (quantity: 1): j -> code: 01010011
token (quantity: 3): k -> code: 1100110
token (quantity: 13): l -> code: 11101
token (quantity: 3): m -> code: 1100111
token (quantity: 26): n -> code: 1111
token (quantity: 1): o -> code: 01010000
token (quantity: 1): p -> code: 01010001
token (quantity: 15): r -> code: 0100
token (quantity: 16): s -> code: 0110
token (quantity: 18): t -> code: 0111
token (quantity: 14): u -> code: 0000
token (quantity: 2): v -> code: 1100000
token (quantity: 1): w -> code: 110101010
token (quantity: 5): z -> code: 110010
token (quantity: 1): ß -> code: 110101011
token (quantity: 1): ä -> code: 11010100
token (quantity: 2): ü -> code: 1100001
```

## Datenkompression – Komprimierungsvorgang

(Fortsetzung)

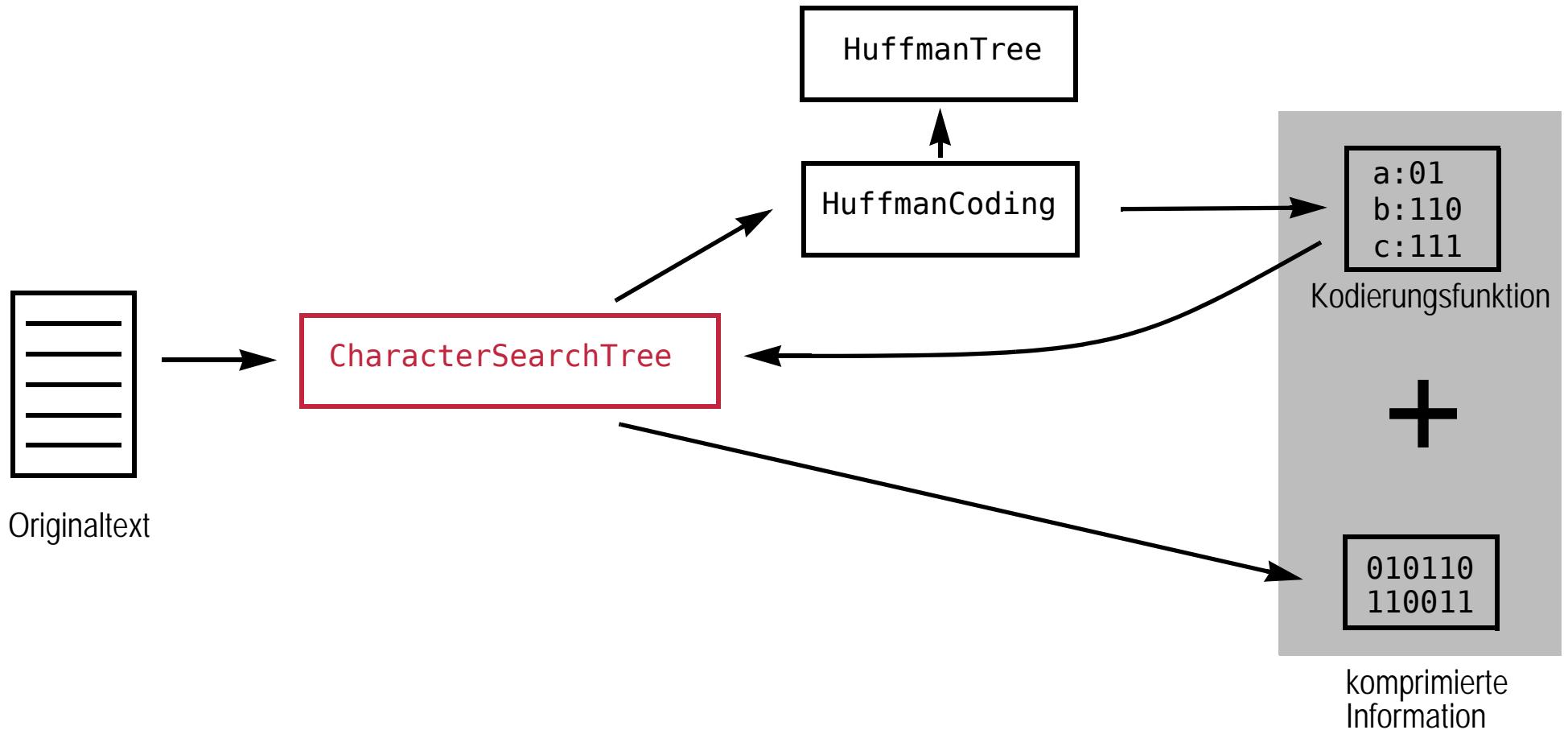
Ausgabe:

```
bit code:  
-----  
1101000001110110001011011000010100110111010011011101011010001011111011111011100  
00100101011111000011011001111000000111111001111011111000110011111000101110001  
001101011100010100100011101100110010000010011100000100100111011111000000111111  
011100110010000100011000010010110000101111100101011111000011011001111000101100  
10101000101000111000111110100110001001001110000111111010011001110011110111010  
100100011001111100111101001100010010101111100010101011111000101011101010111001  
10101111101011100110001110010011010001010000011101100001001010110001011000  
101000011101110011010011000101011100110101111111011110001100011000100101001101  
1111000100000011001111001100100000100000011111100000101010011101101011111001011  
0100110001001010111110000001111110111000000111111000001010100110101000000110101  
0111010100111010011000100101011111000101111101111101100001001010111110100101000  
100100111101111100111001110101000101010100000011111101111101111100110100110111  
000101001110111010010011111011000010010111001111101100010010111010011000100101011110  
001010110111001111010011111101100001001011100110001101110101111001001101110101101  
0001011101010000111011101111011010000001111110111110011011101010100101010010101101  
01001010001001001110011110100111001101001101111000011111100110111010100100010101101  
110110111101011111010111100
```

input chars: 302  
input bits: 2416  
output bits: 1306

## Datenkompression – Motivation (Erinnerung – Folie 492, Folie 548 und Folie 607)

Ablauf einer Textkompression:



## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

Fragestellung:

In welcher Reihenfolge liefert ein *InOrder*-Durchlauf die Werte aus dem binären Suchbaum?

## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

(Fortsetzung)

Fragestellung:

In welcher Reihenfolge liefert ein *InOrder*-Durchlauf die Werte aus dem binären Suchbaum?

Antwoethinweis:

Test mit *InOrder*-Durchlauf der show-Methode!

```
public void show()
{
    if ( !isEmpty() )
    {
        leftChild.show();
        System.out.println( content.toString() );
        rightChild.show();
    }
}
```

## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

(Fortsetzung)

binary tree with codes:

```
-----
token (quantity: 41):   -> code: 100
token (quantity: 1): , -> code: 01011100
token (quantity: 3): . -> code: 1101011
token (quantity: 3): D -> code: 1101000
token (quantity: 1): F -> code: 01011101
token (quantity: 4): G -> code: 010101
token (quantity: 2): M -> code: 0101111
token (quantity: 1): S -> code: 01010010
token (quantity: 2): V -> code: 0101100
token (quantity: 2): W -> code: 0101101
token (quantity: 13): a -> code: 11100
token (quantity: 2): b -> code: 1100010
token (quantity: 15): c -> code: 0001
token (quantity: 12): d -> code: 11011
token (quantity: 42): e -> code: 101
token (quantity: 2): f -> code: 1100011
token (quantity: 3): g -> code: 1101001
token (quantity: 15): h -> code: 0010
```

Buchstaben werden  
in alphabetischer  
Reihenfolge ausgegeben!

```
-----
token (quantity: 15): i -> code: 0011
token (quantity: 1): j -> code: 01010011
token (quantity: 3): k -> code: 1100110
token (quantity: 13): l -> code: 11101
token (quantity: 3): m -> code: 1100111
token (quantity: 26): n -> code: 1111
token (quantity: 1): o -> code: 01010000
token (quantity: 1): p -> code: 01010001
token (quantity: 15): r -> code: 0100
token (quantity: 16): s -> code: 0110
token (quantity: 18): t -> code: 0111
token (quantity: 14): u -> code: 0000
token (quantity: 2): v -> code: 1100000
token (quantity: 1): w -> code: 110101010
token (quantity: 5): z -> code: 110010
token (quantity: 1): ß -> code: 110101011
token (quantity: 1): ä -> code: 11010100
token (quantity: 2): ü -> code: 1100001
```

=> aufsteigende Sortierung?

## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

(Fortsetzung)

Ablauf eines *InOrder*-Durchlaufs für jeden Teilbaum:

- linken Teilbaum durchlaufen
- Bearbeitung für die Wurzel (des Teilbaums)
- rechten Teilbaum durchlaufen

Plausibilitätsbetrachtung:

Da alle Werte im linken Teilbaum kleiner sind als der Wert der Wurzel und alle Werte im rechten Teilbaum größer sind als der Wert der Wurzel, teilt die Wurzel den Durchlauf genau in den Durchlauf der kleineren und den Durchlauf der größeren Werte.

Da diese Eigenschaft für jeden Teilbaum gilt,  
werden die Knoten in aufsteigender Reihenfolge besucht.

## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

(Fortsetzung)

### Satz:

Der *InOrder*-Durchlauf besucht die Knoten des binären Suchbaums in aufsteigender Reihenfolge bezüglich der für den Baum geltenden Ordnungsrelation.

### Beweis:

Der Beweis erfolgt durch *vollständige Induktion* über die Höhe des Baums.

Die Höhe eines Baumes ist definiert als die Anzahl der Knoten auf dem längsten Pfad von der Wurzel bis zu einem Blatt.

## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

(Fortsetzung)

### Satz:

Der *InOrder*-Durchlauf besucht die Knoten des binären Suchbaums in aufsteigender Reihenfolge bezüglich der für den Baum geltenden Ordnungsrelation.

### Beweis:

**Induktionsanfang:** Für den leeren Baum mit der Höhe 0 gilt die Aussage.

**Induktionsvoraussetzung:**

**Induktionsschritt:**

## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

(Fortsetzung)

### Satz:

Der *InOrder*-Durchlauf besucht die Knoten des binären Suchbaums in aufsteigender Reihenfolge bezüglich der für den Baum geltenden Ordnungsrelation.

### Beweis:

**Induktionsanfang:** Für den leeren Baum mit der Höhe 0 gilt die Aussage.

**Induktionsvoraussetzung:** Die Aussage des Satzes gilt für Bäume bis zur Höhe  $n$ .

**Induktionsschritt:**

## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

(Fortsetzung)

### Satz:

Der *InOrder*-Durchlauf besucht die Knoten des binären Suchbaums in aufsteigender Reihenfolge bezüglich der für den Baum geltenden Ordnungsrelation.

### Beweis:

**Induktionsanfang:** Für den leeren Baum mit der Höhe 0 gilt die Aussage.

**Induktionsvoraussetzung:** Die Aussage des Satzes gilt für Bäume bis zur Höhe  $n$ .

**Induktionsschritt:** Sei ein Baum  $T$  der Höhe  $n+1$  gegeben.

$T$  besteht aus der Wurzel  $w$  und dem linken Teilbaum  $T_l$  und rechten Teilbaum  $T_r$ .

Für  $T_l$  und  $T_r$  gilt, dass sie höchstens eine Höhe  $n$  besitzen, so dass der *InOrder*-Durchlauf für sie gemäß der Induktionsvoraussetzung jeweils die Knoten in aufsteigender Reihenfolge besucht.

Da in  $T_l$  alle Werte kleiner als der Wert von  $w$  sind und  $T_l$  vor  $w$  besucht wird und in  $T_r$  alle Werte größer als der Wert von  $w$  sind und  $T_r$  nach  $w$  besucht wird, werden die Knoten von  $T$  in aufsteigender Folge besucht.

## Analyse des *InOrder*-Durchlaufs für einen binären Suchbaum

(Fortsetzung)

Konsequenzen:

- Ein binärer Suchbaum kann zum Sortieren eingesetzt werden.  
Für eine Abschätzung des notwendigen Aufwands müssen die Aufwände für das Einfügen der Werte (bzw. Knoten) und den *InOrder*-Durchlauf zusammengerechnet werden.

Für jeden der  $n$  abgelegten Werte verhält sich der Aufwand zum Einfügen im günstigen Fall proportional zu  $\log_2(n)$ , der Gesamtaufwand also proportional zu  $n \cdot \log_2(n)$ .

Beim *InOrder*-Durchlauf wird jedes Blatt einmal, jeder innere Knoten bis zu drei Mal besucht.  
Der Aufwand für einen *InOrder*-Durchlauf verhält sich also proportional zu  $n$ .

Bei einer ungünstigen Eingabefolge kann sich der Aufwand zum Einfügen allerdings proportional zu  $n$  entwickeln, der Gesamtaufwand also proportional zu  $n^2$ .

## ***PostOrder*-Durchlauf für einen Baum**

Ablauf eines *PostOrder*-Durchlaufs (analog zu *InOrder*-Durchlauf):

- ❑ linken Teilbaum durchlaufen
- ❑ rechten Teilbaum durchlaufen
- ❑ Bearbeitung für die Wurzel

bekanntes Beispiel (aus der Klasse CharacterSearchTree):

```
public int size()
{
    if ( isEmpty() )
    {
        return 0;
    }
    else
    {
        return leftChild.size() + rightChild.size() + 1;
    }
}
```



letzte Operation für einen Knoten erfolgt  
nach dem Durchlaufen beider Teile

## ***PreOrder*-Durchlauf für einen Baum**

Ablauf eines *PreOrder*-Durchlaufs (analog zu *InOrder*-Durchlauf):

- ❑ Bearbeitung für die Wurzel
- ❑ linken Teilbaum durchlaufen
- ❑ rechten Teilbaum durchlaufen

bekanntes Beispiel (aus der Klasse `HuffmanTree`):

```
public void generateCodes()
{
    if ( !isEmpty() && !isLeaf() )
    {
        leftChild.content.setCode( content.getCode() + "0" );
        rightChild.content.setCode( content.getCode() + "1" );
        leftChild.generateCodes();
        rightChild.generateCodes();
    }
}
```

letzte Operation für einen Knoten erfolgt  
vor dem Durchlaufen der beiden Teilbäume

## Tiefendurchläufe durch binäre Bäume

Die Algorithmen

- ❑ *InOrder*,
- ❑ *PostOrder* und
- ❑ *PreOrder*

werden auch unter dem Begriff *Tiefendurchlauf* zusammengefasst,  
da immer ein Teilbaum bis zu seinen Blättern in die Tiefe bearbeitet wird,  
bevor zum anderen Teilbaum gewechselt wird.

Knoten, die auf der gleichen Ebene im Baum liegen wie der aktuell betrachtete Knoten,  
werden dabei immer später behandelt als Knoten,  
die in den Teilbäumen des aktuell betrachteten Knotens weiter unten liegen.

# **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

**Fakultät für Informatik  
Wintersemester 2019/20**

Stefan Dissmann

**Zwischenstand – weitere Ziele**

## Zwischenstand

bisher kennengelernte Datentypen:

- ❑ primitive Typen:
  - Zahlen (**int**, **long**, **float**, **double**)
  - Zeichen (**char**)
  - Wahrheitswerte (**boolean**)
- ❑ Felder (mehrdimensionale Felder in der Übung)
- ❑ Klassen
  - Attribute
  - Methoden
  - Konstruktoren
  - Zugriffsrechte
  - Referenzvariablen
  - Klasse **String**

bisher kennengelernte Ausdrücke und Anweisungen:

- ❑ verschiedene Operatoren
- ❑ Schleifen (**while**, **for**, **for-each**)
- ❑ bedingte Anweisungen (**if**, **if-else**)

## Zwischenstand

(Fortsetzung)

bisher kennengelernte Datenstrukturen:

- ❑ binäre Bäume
  - *Huffman-Baum* (für Kodierung)
  - *binärer Suchbaum*
- ❑ dynamische Datenstrukturen auf Basis rekursiver Deklarationen

bisher kennengelernte Algorithmen:

- ❑ Sortieralgorithmen
  - *SelectionSort*
  - *InsertionSort*
  - *QuickSort*
- ❑ rekursive Algorithmen
  - für Berechnungen
  - für Lösungssuche (*Backtracking*)
  - für die Bearbeitung von rekursiven Datenstrukturen (*InOrder*)

## Zwischenstand

(Fortsetzung)

- ❑ Alle bisher vorgestellten Implementierungen sind spezialisiert und arbeiten nur für *einen* Problembereich
- ❑ Viele der vorgestellten Algorithmen lösen aber allgemeine Problemstellungen:
  - Die Sortieralgorithmen lassen sich auf beliebige Daten anwenden, sofern eine Ordnung für die Daten definiert ist.
  - Ein binärer Suchbaum kann zur Ablage beliebiger Daten eingesetzt werden, sofern eine Ordnung für die Daten definiert ist.
- ❑ Bisher sind Algorithmen und Datenstrukturen gegebenenfalls durch *Copy-and-paste* voneinander abgeleitet worden.  
Dieses Vorgehen führt aber – gerade auch in der Praxis – zu einem Aufwand:
  - Jede Anpassung erfordert Editierarbeiten und kann zu Fehlern an eigentlich nicht betroffenen Abschnitten des Programmtextes führen.
  - Jede Variante muss daher vollständig überprüft werden.
  - Der historische Zusammenhang zwischen den Varianten muss dokumentiert werden, um später erkannte Fehler auch in allen abgeleiteten Varianten beseitigen zu können.

## weitere Ziele

- Vorstellung von Konzepten in der Programmiersprache Java,  
die die Implementierung von allgemein einsetzbaren Klassen (und damit Datenstrukturen)  
ermöglichen:  
*Vererbung* und *Polymorphie*
- Vorstellung von allgemeinen Konzepten für die Konstruktion,  
die eine flexible Anpassung von Algorithmen und Datenstrukturen ermöglichen:  
*Entwurfsmuster*
- Konstruktion allgemein einsetzbarer Datenstrukturen

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 8.1. Vererbung – Grundlagen

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-646

## Lernziele des Kapitels 8. Vererbung und Polymorphie

Nach Durcharbeiten des Kapitels Vererbung und Polymorphie sollen die teilnehmenden Studierenden

- das Konzept der Vererbung kennen und einsetzen können,
- das Konzept polymorpher Methoden kennen und einsetzen können,
- Klassenhierarchien gestalten und implementieren können,  
die eine *ist-ein*-Beziehung umsetzen,
- den *Laufzeittyp* und den *statischen Typ* von Referenzvariablen unterscheiden und bestimmen können.

## Erinnerung: einfache Klassen

Beispiel

```
public class Person
{
    private String firstName;
    private String familyName;
    private String cityOfResidence;
    private int yearOfBirth; Attribute

    public Person( String fi, String fa, String c, int y ) Konstruktor
    {
        firstName = fi;
        familyName = fa;
        cityOfResidence = c;
        yearOfBirth = y;
    }

    public String getFirstName() ...
    public String getFamilyName() ...
    public String getCity() ...
    public void setCity( String c ) ...
    public int getYearOfBirth() ... einfache Methoden
}
```

## Erinnerung: einfache Klassen

(Fortsetzung)

Beispiel

```

public String toString() ← alle Attributwerte als Text
{
    return "family name: " + familyName + ", first name: " + firstName +
           ", born in " + yearOfBirth + ", living in: " + cityOfResidence;
}

public boolean isEqualTo( Person p ) ← «sinnvoller» Vergleich
{
    return ( familyName.compareTo( p.familyName ) == 0 )
           && ( firstName.compareTo( p.firstName ) == 0 )
           && yearOfBirth == p.yearOfBirth;
}

public int compareTo( Person p )
{
    int compFamilyName = familyName.compareTo( p.familyName );
    if ( compFamilyName != 0 ) {
        return compFamilyName;
    } else {
        return firstName.compareTo( p.firstName );
    }
}

```

compareTo über lexikographischen Namensvergleich:  
compareTo aus der Klasse String

## Erinnerung: einfache Klassen

(Fortsetzung)

Beispiel

```
public class Student
{
    private String name;
    private String subject;
    private int registrationNo;

    ...
}
```

- Studierende haben wie Personen einen Namen.
- Streng genommen haben Studierende auch einen Vornamen, ein Geburtsjahr und einen Wohnort.
- Studierende besitzen also auch *alle* Eigenschaften, die Personen haben, da Studierende ja immer auch Personen sind.

## Vererbung

Idee:

Ein Student *ist eine* spezielle Person,  
die *alle* Eigenschaften einer Person besitzt  
und *zusätzliche* Eigenschaften hinzu bekommt, die spezifisch für Studenten sind.

## Vererbung

(Fortsetzung)

Idee:

Ein Student *ist eine* spezielle Person,  
die *alle* Eigenschaften einer Person besitzt  
und *zusätzliche* Eigenschaften hinzu bekommt, die spezifisch für Studenten sind.

Umsetzung in Java:

- Die Klasse `Student` *erbt* von der Klasse `Person`.
- `Person` ist dann die *Oberklasse* von `Student`.
- `Student` ist eine *Unterklasse* von `Person`.

## Vererbung

(Fortsetzung)

Idee:

Ein Student *ist eine* spezielle Person,  
die *alle* Eigenschaften einer Person besitzt  
und *zusätzliche* Eigenschaften hinzu bekommt, die spezifisch für Studenten sind.

Umsetzung in Java:

- Die Klasse `Student` *erbt* von der Klasse `Person`.
- `Person` ist dann die *Oberklasse* von `Student`.
- `Student` ist eine *Unterklasse* von `Person`.

weitere Terminologie:

- Die Klasse `Student` *erweitert* die Klasse `Person`.
- Die Klasse `Student` *spezialisiert* die Klasse `Person`.
- Eine Oberklasse wird auch *Superklasse* bezeichnet.
- Eine Unterklasse wird auch *Subklasse* oder *abgeleitete Klasse* bezeichnet.

## Vererbung

(Fortsetzung)

Umsetzung in Java

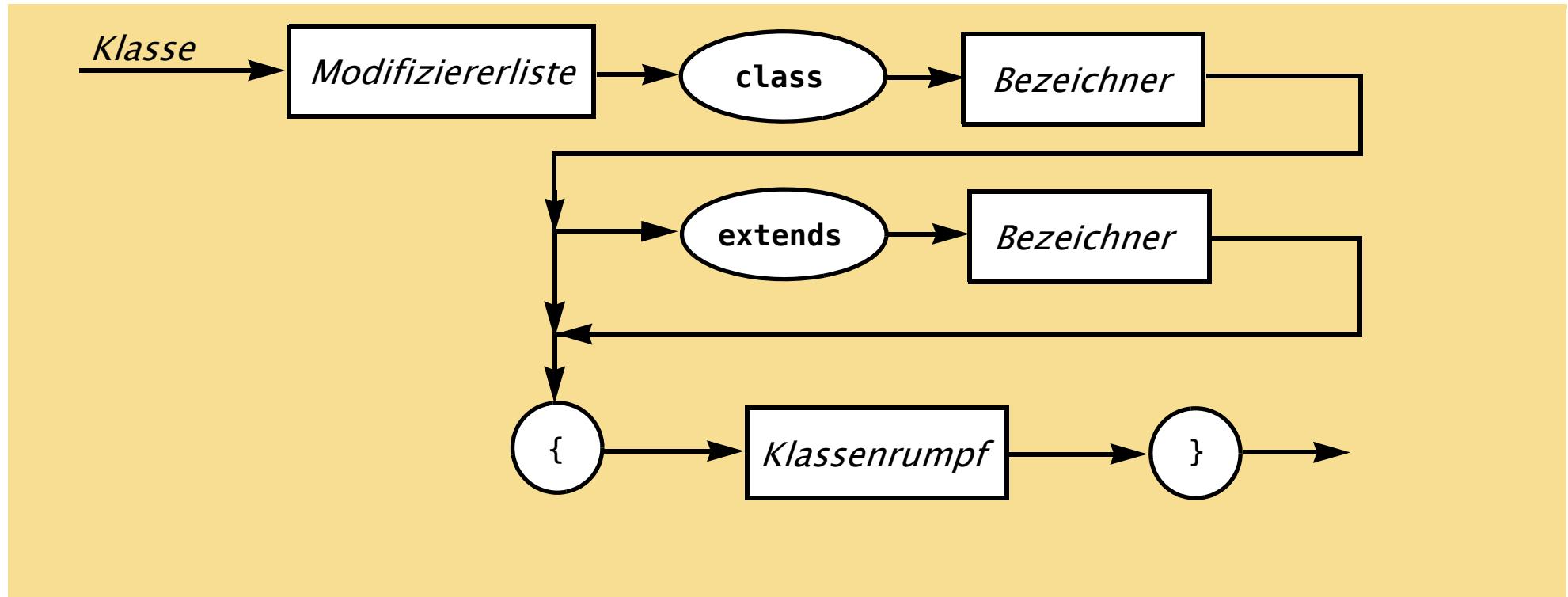
```
public class Student extends Person
{
    private String subject;
    private int registrationNo;
    ...
}
```

Schlüsselwort, hinter dem die Oberklasse folgt

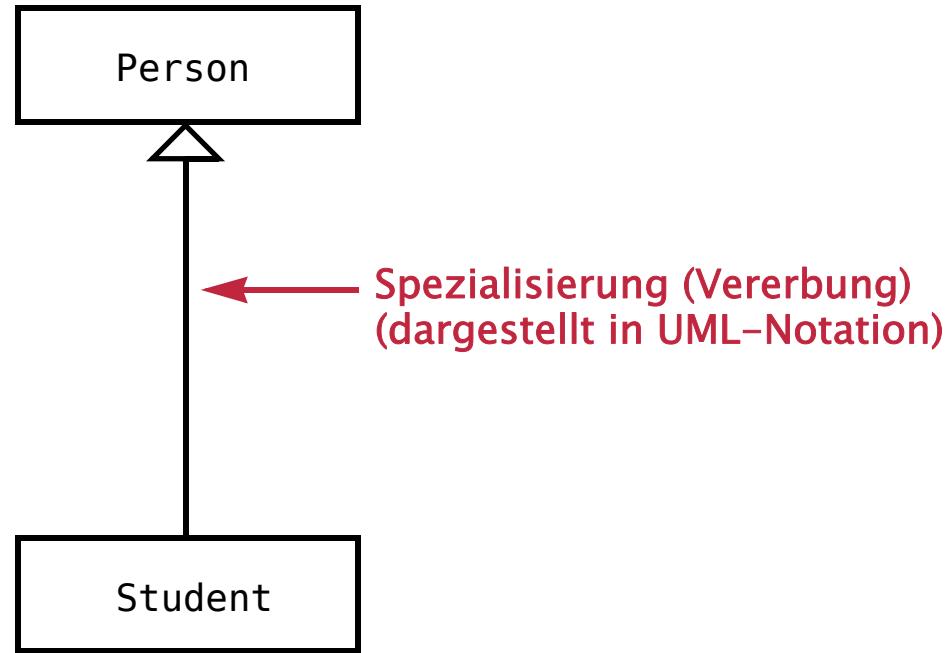
eigene Attribut-Deklarationen

- Die Klasse Student erweitert die Klasse Person.
- Die Klasse Student besitzt jetzt alle Attribute der Klasse Person:
  - firstName, familyName, cityOfResidence, yearOfBirth
- Die Klasse Student stellt auch alle Methoden der Klasse Person bereit:
  - getFirstName, getFamilyName, getCity, ..., compareTo
- Der Konstruktor der Oberklasse wird *nicht* vererbt:  
Die Klasse Person besitzt einen Konstruktor mit vier Parametern,  
die Klasse Student besitzt (zunächst) keinen Konstruktor.
- Die Details der Vererbung werden auf den folgenden Folien geklärt.

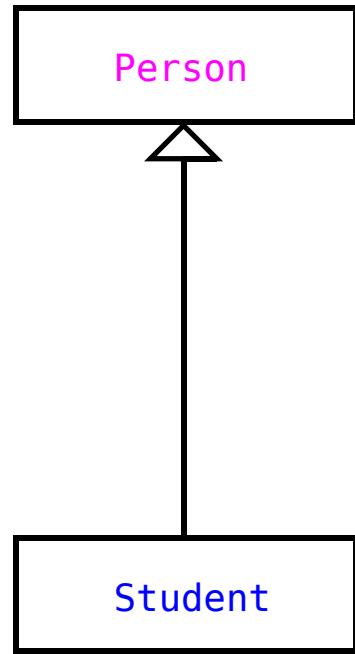
## Syntaxdiagramm zu *Klasse*



## Visualisierung der Klassen



## Visualisierung der Klassen



**mit den Attributen:**  
firstName, familyName, cityOfResidence, yearOfBirth

**mit den Attributen:**  
firstName, familyName, cityOfResidence, yearOfBirth (geerbt)  
subject, registrationNo

## Vererbung

(Fortsetzung)

Umsetzung in Java – Deklaration eines Konstruktors

```
public class Student extends Person
{
    private String subject;
    private int registrationNo;

    public Student( String sub, int no )
    {
        subject = sub;
        registrationNo = no;
    }
}
```

Konstruktor-Deklaration

Versuch, die geerbten Attribute erst einmal zu ignorieren und gar nicht zu setzen

- Fehlermeldung des Compilers:

constructor Person in class Person cannot be applied to given types;

- *Erklärung:*

Da jedes Objekt der Klasse `Student` alle Attribute der Klasse `Person` enthält, versucht der Konstruktor der Klasse `Student` zunächst, diese geerbten Attribute anzulegen und zu initialisieren.

Und dafür nutzt der Konstruktor der Klasse `Student` einen Konstruktor der Klasse `Person`. Ist – wie hier – *kein* anderer Konstruktor *explizit* angegeben, so wird der *Standardkonstruktor* (siehe Folie 271) benutzt, den es aber für die Klasse `Person` *nicht* gibt.

## Vererbung

(Fortsetzung)

Umsetzung in Java – Deklaration eines Konstruktors

```
public class Student extends Person
{
    private String subject;
    private int registrationNo;

    public Student( String fi, String fa, String c, int y, String sub, int no )
    {
        firstName = fi;
        familyName = fa;
        cityOfResidence = c;
        yearOfBirth = y;
        subject = sub;
        registrationNo = no;
    }
}
```

**Konstruktor-Deklaration**

}

**Versuch, alle geerbten Attribute zu setzen**

- Fehlermeldung des Compilers:  
constructor Person in class Person cannot be applied to given types;
- Auch in dieser Implementierung wird versucht, den *Standardkonstruktor* zu nutzen, den es aber für die Klasse Person *nicht* gibt.

## Vererbung

Umsetzung in Java – Deklaration eines Konstruktors

```
public class Student extends Person
{
    private String subject;
    private int registrationNo;

    public Student( String fi, String fa, String c, int y, String sub, int no )
    {
        firstName = fi;
        familyName = fa;
        cityOfResidence = c;
        yearOfBirth = y;
        subject = sub;
        registrationNo = no;
    }
}
```

Standardkonstruktor  
ergänzt:

```
public class Person
{
    public Person(){}
    ...
}
```

Zugriff auf geerbte Attribute nicht möglich,  
da in Person als privat deklariert

(Fortsetzung)

- ❑ Fehlernachricht des Compilers:  
`firstName has private access in Person;`
- ❑ Der Konstruktor der Klasse Student kann – wie auch alle anderen Methoden aus Student *nicht* auf die *privaten* Attribute der Oberklasse Person zugreifen:  
 Mit dem Zugriffsrecht **private** gekennzeichnete Attribute oder Methoden der Klasse Person sind immer *nur* in der Deklaration der Klasse Person sichtbar.

## Vererbung

(Fortsetzung)

Umsetzung in Java – Versuch der Deklaration eines Konstruktors

```
public class Student extends Person
{
    private String subject;
    private int registrationNo;

    public Student( String fi, String fa, String c, int y, String sub, int no )
    {
        super( fi, fa, c, y );
        subject = sub;
        registrationNo = no;
    }
}
```



expliziter Aufruf des  
Konstruktors der Oberklasse

- ❑ In der Klasse `Person` hat der Konstruktor den Namen `Person`.
- ❑ In seinen direkten Unterklassen kann er über das Schlüsselwort `super` aufgerufen werden.
- ❑ Die *erste Anweisung* eines Konstruktors ist immer ein Aufruf eines Konstruktors der Oberklasse:  
*Explizit* durch Angabe von `super` gefolgt von einer Parameterliste oder  
*implizit* – dann ergänzt der Compiler den Aufruf des Standardkonstruktors durch `super()`.
- ❑ Die implizite Form ist nur möglich, wenn die Oberklasse auch einen Standardkonstruktor bereit stellt. (Das ist für die Klasse `Person` nicht der Fall.)

## Vererbung

(Fortsetzung)

Umsetzung in Java – Versuch der Deklaration eines Konstruktors

```
public class Student extends Person
{
    private String subject;
    private int registrationNo;

    public Student( String fi, String fa, String c, int y, String sub, int no )
    {
        super( fi, fa, c, y );
        subject = sub;
        registrationNo = no;
    }
}
```



expliziter Aufruf des Konstruktors der Oberklasse

- Der Konstruktor der Oberklasse ist ein Teil der Oberklasse.
- Der Konstruktor der Oberklasse *kann* auf die privaten Attribute der Oberklasse zugreifen. Durch die Übergabe von Argumenten an den Konstruktor der Oberklasse ist es daher möglich, die privaten Attribute der Oberklasse mit Werten zu belegen.

## Vererbung

(Fortsetzung)

Umsetzung in Java – Methoden in der Klasse Student

```
public String getSubject()
{
    return subject;
}

public int getRegistrationNo()
{
    return registrationNo;
}

public boolean hasGreaterNumber( Student s )
{
    return getRegistrationNo() > s.getRegistrationNo();
}

public boolean hasEqualNumber( Student s )
{
    return getRegistrationNo() == s.getRegistrationNo();
}
```

- Diese Methoden greifen nur auf Attribute zu, die in der Klasse Student deklariert sind.  
Das ist problemlos möglich.

## Vererbung

(Fortsetzung)

Umsetzung in Java – Methoden in der Klasse Student

Implementierung einer `toString`-Methode

```
public String toString()
{
    return "family name: " + getFamilyName() + ", first name: " + getFirstName() +
           ", born in " + getYearOfBirth() + ", living in: " + getCity() +
           ", registration number: " + getRegistrationNo() +
           "(" + getSubject() + ")";
}
```

Ausgabe:

```
family name: Miller, first name: Eva, born in 1979, living in: Bochum,
registration number: 171564(Math)
```

- ❑ Die Werte der geerbten Attribute können über die *geerbten öffentlichen* Methoden abgerufen werden.
- ❑ Die Implementierung scheint aber etwas aufwändig, da ein Teil der Ausgabe bereits in der `toString`-Methode der Klasse `Person` implementiert wurde.

## Vererbung

(Fortsetzung)

Umsetzung in Java – Methoden in der Klasse `Student`

Implementierung einer `toString`-Methode

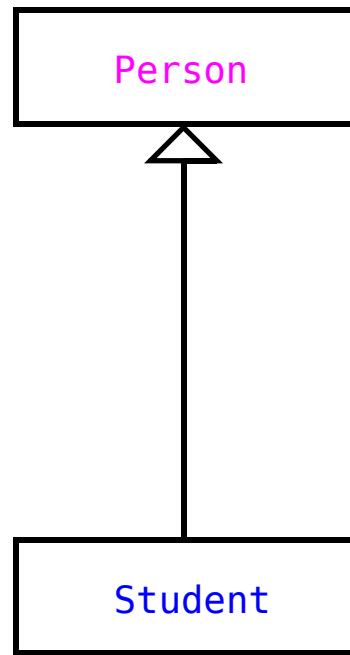
```
public class Student extends Person
{
    ...
    public String toString() {
        return toString() +
            ", registration number: " + getRegistrationNo() + "(" + getSubject() + ")";
    }
}
```

- ❑ Der Compiler erlaubt diese Implementierung.
- ❑ Die Ausführung ist aber *nicht* erfolgreich:  
`toString()` ist jetzt eine rekursive Methode ohne Abbruch-Kriterium,  
da `return toString() + ...`  
sich auf die *eigene* Deklaration in der Klasse `Student` bezieht.

## Vererbung

(Fortsetzung)

Umsetzung in Java – Methoden in der Klasse Student  
Implementierung einer `toString`-Methode



**mit den Attributen:**  
firstName, familyName, cityOfResidence, yearOfBirth  
**mit den Methoden:**  
`toString()`  
...

**mit den Attributen:**  
firstName, familyName, cityOfResidence, yearOfBirth (geerbt)  
subject, registrationNo  
**mit den Methoden:**  
`toString()` und geerbt, aber verdeckt: `toString()`  
...

## Vererbung

(Fortsetzung)

Umsetzung in Java – Methoden in der Klasse Student

Implementierung einer `toString`-Methode

```
public String toString() {  
    return super.toString() + ", registration number: "  
           + getRegistrationNo() + "(" + getSubject() + ")";  
}
```

verweist auf die Methode der Oberklasse

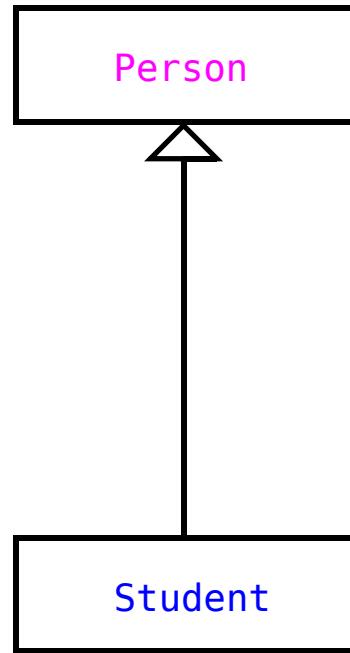
- Methoden der Oberklasse können *überschrieben* werden:  
`toString()` ist dann für die Klasse `Student` die in `Student` deklarierte Methode und nicht die aus der Klasse `Person` geerbte Methode.
- Das Überschreiben der Methode verhindert den Zugriff auf die geerbte Methode.
- Der Zugriff mit `super.` erlaubt den expliziten Zugriff auf die (überschriebene) Methode der direkten Oberklasse.
- Das Schlüsselwort `super` hat also zwei Bedeutungen:
  - `super( ... )` bezeichnet immer den Aufruf eines Konstruktors der Oberklasse und ist darf nur als erste Anweisung eines eigenen Konstruktors verwandt werden.
  - `super.` ist der Zugriff auf ein Element aus der direkten Oberklasse.

## Vererbung

(Fortsetzung)

Umsetzung in Java – Methoden in der Klasse Student

Implementierung einer `toString`-Methode



**mit den Attributen:**

`firstName, familyName, cityOfResidence, yearOfBirth`

**mit den Methoden:**

`toString()`

...

**mit den Attributen:**

`firstName, familyName, cityOfResidence, yearOfBirth (geerbt)`

`subject, registrationNo`

**mit den Methoden:**

`toString()` und geerbt, aber überschrieben: `toString()`

...

`super.toString()`



## Vererbung

(Fortsetzung)

Umsetzung in Java – Methoden in der Klasse Student

Implementierung von Methoden zum Vergleichen von Objekten

```
public boolean hasGreaterName( Student s )
{
    return compareTo( ? ) > 0;
}

public boolean hasEqualName( Student s )
{
    return compareTo( ? ) == 0;
}
```

geerbte Methode erwartet  
Referenz auf Person

- Die geerbte Methode `compareTo` eignet sich gut für die einfache Implementierung von Vergleichsmethoden in der Klasse `Student`.
- aber:*  
Die geerbte Methode `compareTo` erwartet eine Referenz auf `Person` als Argument, `hasGreaterName` und `hasEqualName` sollen aber `Student`-Objekte miteinander vergleichen.

## Vererbung

(Fortsetzung)

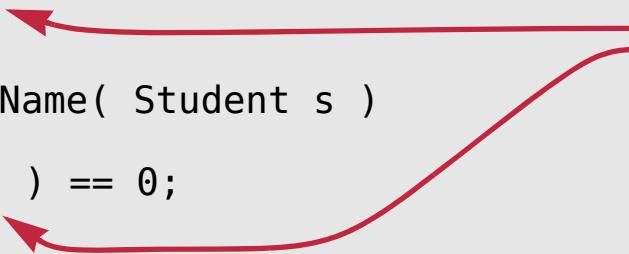
Umsetzung in Java – Methoden in der Klasse Student

Implementierung von Methoden zum Vergleichen von Objekten (Fortsetzung)

```
public boolean hasGreaterName( Student s )
{
    return compareTo( s ) > 0;
}

public boolean hasEqualName( Student s )
{
    return compareTo( s ) == 0;
}
```

Das Student-Objekt an der  
Referenz s ist kompatibel  
zur Klasse Person



- Da die Klasse Student von der Klasse Person erbt, kann ein Objekt der Klasse Student auf *alle* Methodenaufrufe reagieren, die an ein Objekt der Klasse Person gestellt werden können.
- Jedes Objekt der Klasse Student ist daher *typkompatibel mit* der Klasse Person.
- Ein Objekt einer Unterklasse kann also ein Objekt der Oberklasse *ersetzen*.
- Daher darf eine Referenzvariable, die als *deklarierten Typ* die Oberklasse besitzt, zur Laufzeit auf ein Objekt der Unterklasse – das ist dann der *Laufzeittyp* – verweisen.

## Vererbung

(Fortsetzung)

Beispiele:

```
Person x = new Student( "Tina", "Mueller", "Bonn", 1999, M, 190184 );
```

dekliarierter Typ von x ist Person

Laufzeittyp von x ist Student

```
public class Person
{
    ...
    public int compareTo( Person p ) { ... }
}

public class Student extends Person
{
    ...
    public boolean hasGreaterName( Student s )
    {
        return compareTo( s ) > 0;
    }
}
```

dekliarierter Typ von p ist Person

Laufzeittyp von p ist Student

## Vererbung

(Fortsetzung)

Umsetzung in Java – Methoden in der Klasse Student

```
public boolean isEqualTo( Student s )
{
    return super.isEqualTo( s )
        && getRegistrationNo() == s.getRegistrationNo();
}
```

- ❑ Es liegt *kein* Überschreiben der geerbten `isEqualTo`-Methode vor:
  - Geerbt aus `Person` wird eine Methode mit der Signatur `isEqualTo( Person )`,
  - neu definiert in `Student` wird eine Methode mit der Signatur `isEqualTo( Student )`.
- ❑ Die Namensgleichheit der beiden Methoden zusammen mit der Vererbungsbeziehung zwischen den Klassen `Student` und `Person` machen diese Konstruktion möglich.
- ❑ Die geerbte Methode `isEqualTo( Person )` lässt auch die Klasse `Student` als Laufzeittyp für das übergebene Argument zu, da `Student` von `Person` erbt.
- ❑ Die Verwendung von `super.` ist notwendig, da der Compiler sonst wieder von der Deklaration einer rekursiven Methode ausgehen würde.

## Kompatibilität von Referenzvariablen und Klassen

- ❑ Eine Referenzvariable, deren deklarerter Typ eine Oberklasse ist, darf als Laufzeittyp auch auf Objekte der Unterklassen verweisen.
- ❑ Da Objekte der Unterklasse alle Eigenschaften der Oberklasse erben, kann ein Objekt der Unterklasse wie ein Objekt der Oberklasse behandelt werden.
- ❑ Alle Methoden, die für ein Objekt der Oberklasse aufgerufen werden dürfen, sind also auch für ein Objekt der Unterklasse verfügbar und können für dieses aufgerufen werden.
- ❑ Methoden können in Unterklassen überschrieben werden.  
Beim Aufruf stehen verschiedene Versionen der gleichen Methode zur Verfügung:
  - in der Oberklasse: Hier erfolgt eine Deklaration der Methode.
  - in der Unterklasse: Hier sind die geerbte – aber nicht mehr sichtbare – Version und die im Rahmen des Überschreibens vorgenommene eigene Deklaration der Methode verfügbar.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 8.2. Vererbung – Polymorphie

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-674

## Kompatibilität von Referenzvariablen und Klassen

(Fortsetzung)

- Eine Referenzvariable, deren deklarierter Typ eine Oberklasse ist, darf als Laufzeittyp auch auf Objekte der Unterklassen verweisen.
- Da Objekte der Unterklasse alle Eigenschaften der Oberklasse erben, kann ein Objekt der Unterklasse wie ein Objekt der Oberklasse behandelt werden.
- Alle Methoden, die für ein Objekt der Oberklasse aufgerufen werden dürfen, sind also auch für ein Objekt der Unterklasse verfügbar und können für dieses aufgerufen werden.
- Methoden können in Unterklassen überschrieben werden.

Beim Aufruf stehen verschiedene Versionen der gleichen Methode zur Verfügung:

- in der Oberklasse: Hier erfolgt eine Deklaration der Methode.
- in der Unterklasse: Hier sind die geerbte – aber nicht mehr sichtbare – Version und die im Rahmen des Überschreibens vorgenommene eigene Deklaration der Methode verfügbar.

Eine Verallgemeinerung von Algorithmen und Datenstrukturen ist also folgendermaßen möglich:

- Lege eine Vererbungsstruktur aus Oberklasse und Unterklassen an.
- Implementiere gemeinsame Algorithmen und Datenstrukturen für die Oberklasse.
- Überschreibe einzelne Methoden in einzelnen Unterklassen, um das Verhalten der zugehörigen Objekte anzupassen.

## Kompatibilität von Referenzvariablen und Klassen

(Fortsetzung)

- Eine Referenzvariable, deren deklarerter Typ eine Oberklasse ist, darf als Laufzeittyp auch auf Objekte der Unterklassen verweisen.
- Da Objekte der Unterklasse alle Eigenschaften der Oberklasse erben, kann ein Objekt der Unterklasse wie ein Objekt der Oberklasse behandelt werden.
- Alle Methoden, die für ein Objekt der Oberklasse aufgerufen werden dürfen, sind also auch für ein Objekt der Unterklasse verfügbar und können für dieses aufgerufen werden.
- Methoden können in Unterklassen überschrieben werden.

Beim Aufruf stehen verschiedene Versionen der gleichen Methode zur Verfügung:

- in der Oberklasse: Hier erfolgt eine Deklaration der Methode.
- in der Unterklasse: Hier sind die geerbte – aber nicht mehr sichtbare – Version und die im Rahmen des Überschreibens vorgenommene eigene Deklaration der Methode verfügbar.

Eine Verallgemeinerung von Algorithmen und Datenstrukturen ist also folgendermaßen möglich:

- Lege eine Vererbungsstruktur aus Oberklasse und Unterklassen an.
- Implementiere gemeinsame Algorithmen und Datenstrukturen für die Oberklasse.
- Überschreibe einzelne Methoden in einzelnen Unterklassen, um das Verhalten der zugehörigen Objekte anzupassen.
- Für Unterklassen stellt sich noch die Frage: **Welche Version der Methode wird ausgeführt?**

## Kompatibilität von Referenzvariablen und Klassen - Experimente

```
Person p = new Student( "S", "S", "S", 1, "S", 1 );  
System.out.println( p.toString() );
```

Ausgabe:

```
family name: S, first name: S, born in 1, living in: S, registration number: 1(S)
```

- ❑ Ein Student-Objekt an einer **Person**-Referenzvariablen ruft `toString` von `Student` auf:
  - deklarierter Typ von `p`: `Person`
  - Laufzeittyp des von `p` referenzierten Objekts: `Student`
- ❑ **Annahme** aufgrund des Experiments:  
Es wird die passende Methode des Laufzeittyps ausgewählt.

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

- ❑ `toString()` ist eine parameterlose Methode,  
Namensgleichheit und Überschreiben fallen immer zusammen.
- ❑ Was geschieht, wenn namensgleiche Methoden mit unterschiedlichen Signaturen  
(d.h. unterschiedlichen Parameterlisten) auftreten:
  - die Typen der Parameter sind *inkompatibel*:  
Compiler überprüft anhand der übergebenen Argumente die Signatur und wählt die passende Methode aus.
  - die Typen der Parameter sind *kompatibel*:  
Beispiel: Methoden `isEqualTo( Person p )` und `isEqualTo( Student s )`

Ein Objekt der Klasse `Student` kann über eine Referenzvariable sowohl der Klasse `Person` als auch der Klasse `Student` als Argument übergeben werden.

Welche Methode wird wann aufgerufen?

**Experiment**

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)

in der Klasse Person ergänzt:

```
public void whichMethod( Person p )
{
    System.out.println( "Person" );
}
```

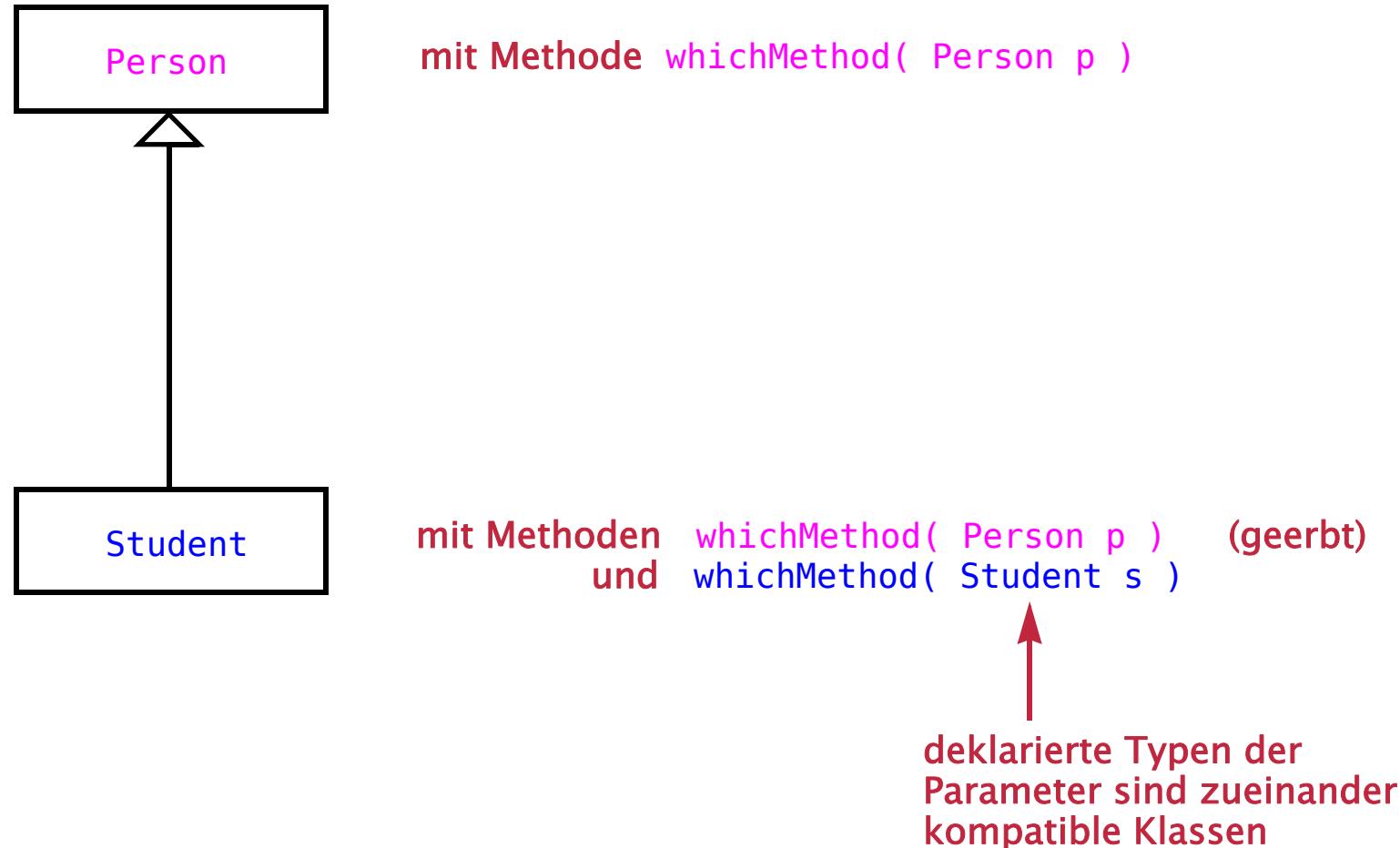
in der Klasse Student ergänzt:

```
public void whichMethod( Student p )
{
    System.out.println( "Student" );
}
```

- Die Methoden sind namensgleich, haben Parameterlisten gleicher Länge, die Parameter besitzen unterschiedlich deklarierte Typen, die aber durch Vererbung zueinander kompatibel sind.
- Die Methoden geben den Namen der Klasse aus, in der sie deklariert sind.

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)



# Kompatibilität von Referenzvariablen und Klassen - Experimente

## Experimentierumgebung

(Fortsetzung)

```
private static void display( String opt, String ort, String pdt, String prt )
{
    System.out.print( opt + "\t\t" + ort + "\t\t" + pdt + "\t\t" + prt + "\t\t" );
}

public static void testTypeCombinations()
{
    Person pWithP = new Person( "P", "P", "C", 1 );
    Student sWithS = new Student( "S", "S", "C", 1, "S", 1 );
    Person pWithS = sWithS;
    System.out.println( "calling object\t\targument" );
    System.out.println( "declared type\truntime type\tdeclared type\truntime type\tmethod from" );
    System.out.println( "-----" );
    display( "Person", "Person", "Person", "Person" );      pWithP.whichMethod( pWithP );
    display( "Person", "Person", "Person", "Student" );     pWithP.whichMethod( pWithS );
    display( "Person", "Person", "Student", "Student" );    pWithP.whichMethod( sWithS );
    display( "Person", "Student", "Person", "Person" );      pWithS.whichMethod( pWithP );
    display( "Person", "Student", "Person", "Student" );    pWithS.whichMethod( pWithS );
    display( "Person", "Student", "Student", "Student" );   pWithS.whichMethod( sWithS );
    display( "Student", "Student", "Person", "Person" );    sWithS.whichMethod( pWithP );
    display( "Student", "Student", "Person", "Student" );   sWithS.whichMethod( pWithS );
    display( "Student", "Student", "Student", "Student" );  sWithS.whichMethod( sWithS );
}
```

## Kompatibilität von Referenzvariablen und Klassen - Experimente

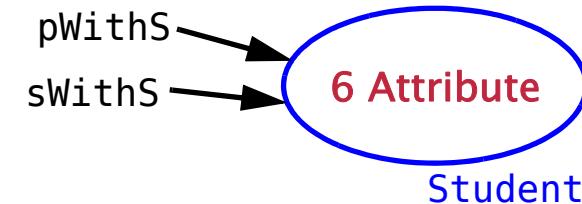
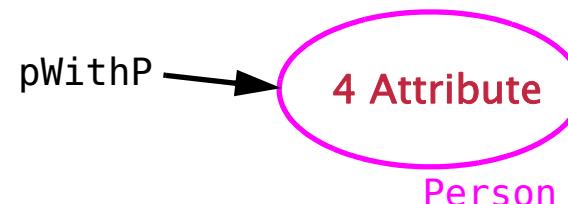
Experimentierumgebung

(Fortsetzung)

```
public static void testTypeCombinations()
{
    Person pWithP = new Person( "P", "P", "C", 1 );
    Student sWithS = new Student( "S", "S", "C", 1, "S", 1 );
    Person pWithS = sWithS;
    ...
}
```

Referenzen und Objekte anlegen

- Es werden drei Referenzvariablen angelegt:
  - Person pWithP verweist auf ein Person-Objekt
  - Person pWithS verweist auf ein Student-Objekt
  - Student sWithS verweist auf ein Student-Objekt (– geht auch nicht anders)
- Über jede der Referenzvariablen wird der Aufruf vorgenommen.
- Jede der Referenzvariablen wird als Argument übergeben.



## Kompatibilität von Referenzvariablen und Klassen - Experimente

Experimentierumgebung

(Fortsetzung)

formatierte Ausgabe

```
System.out.println( "calling object\t\targument" );
System.out.println( "declared type\truntime type\tdeclared type\truntime type\tmethod from" );
System.out.println( "-----" );
display( "Person", "Person", "Person", "Person" );    pWithP.whichMethod( pWithP );
display( "Person", "Person", "Person", "Student" );   pWithP.whichMethod( pWithS );
display( "Person", "Person", "Student", "Student" );  pWithP.whichMethod( sWithS );
display( "Person", "Student", "Person", "Person" );   pWithS.whichMethod( pWithP );
display( "Person", "Student", "Person", "Student" );  pWithS.whichMethod( pWithS );
display( "Person", "Student", "Student", "Student" ); pWithS.whichMethod( sWithS );
display( "Student", "Student", "Person", "Person" );  sWithS.whichMethod( pWithP );
display( "Student", "Student", "Person", "Student" ); sWithS.whichMethod( pWithS );
display( "Student", "Student", "Student", "Student" ); sWithS.whichMethod( sWithS );
```

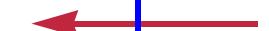
- ❑ Die neun möglichen Kombinationen werden aufgerufen und das Ergebnis wird tabellarisch ausgegeben.
- ❑ '\t' ist *ein* Zeichen des Typs **char**, das Tabulator-Zeichen.
- ❑ manchmal auch hilfreich: '\n' für einen Zeilenumbruch.

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	runtime type	argument declared type	runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
Person	Student	Person	Person	Person
Person	Student	Person	Student	Person
Person	Student	Student	Student	Person
Student	Student	Person	Person	Person
Student	Student	Person	Student	Person
Student	Student	Student	Student	Student



- ☐ Nur bei einer Kombination wird die Methode der Klasse Student aufgerufen.

Warum?

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	runtime type	argument declared type	runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
...				

Der *Compiler* kennt als Eigenschaft der Referenzvariablen:



mit Methode `whichMethod( Person p )`

und wählt zur Ausführung die Methode:

`whichMethod( Person p )`

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	reference runtime type	argument declared type	reference runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
Person	Student	Person	Person	Person
Person	Student	Person	Student	Person
Person	Student	Student	Student	Person
Student	Student	Person	Person	Person
Student	Student	Person	Student	Person
Student	Student	Student	Student	Student

- Die Referenzvariable, über die der Aufruf erfolgt, ist für die Klasse Person deklariert.
- In der Klasse Person ist *eine* Methode whichMethod mit Parameter Person bekannt.
- Der *Compiler* nimmt eine statische Analyse anhand der *deklarierten Typen* vor.
- Der *Compiler* sieht für die Ausführung die Methode *whichMethod( Person )* vor.
- Das *Objekt* der Klasse Person führt die Methode *whichMethod( Person )* aus.

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	runtime type	argument declared type	runtime type	method from
...				
Person	Student	Person	Person	Person
Person	Student	Person	Student	Person
Person	Student	Student	Student	Person
...				

Der *Compiler* kennt als Eigenschaft der Referenzvariablen:



mit Methode `whichMethod( Person p )`

und wählt zur Ausführung die Methode:

`whichMethod( Person p )`

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	reference runtime type	argument declared type	reference runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
Person	Student	Person	Person	Person
Person	Student	Person	Student	Person
Person	Student	Student	Student	Person
Student	Student	Person	Person	Person
Student	Student	Person	Student	Person
Student	Student	Student	Student	Student

- ❑ Die Referenzvariable, über die der Aufruf erfolgt, ist für die Klasse Person deklariert.
- ❑ In der Klasse Person ist *eine* Methode whichMethod mit Parameter Person bekannt.
- ❑ Der *Compiler* nimmt eine statische Analyse anhand der *deklarierten Typen* vor.
- ❑ Der *Compiler* sieht für die Ausführung die Methode `whichMethod( Person )` vor.
- ❑ Die Klasse Student hat die Methode `whichMethod( Person )` von Person *geerbt*.
- ❑ Das *Objekt* der Klasse Student führt die Methode `whichMethod( Person )` aus.

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	runtime type	argument declared type	runtime type	method from
...				
Student	Student	Person	Person	Person
Student	Student	Person	Student	Person
...				

Der *Compiler* kennt als Eigenschaft der Referenzvariablen:

Student	mit Methoden	whichMethod( Person p )	(geerbt)
	und	whichMethod( Student s )	

und wählt zur Ausführung die Methode:

whichMethod( Person p )

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	reference runtime type	argument declared type	reference runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
Person	Student	Person	Person	Person
Person	Student	Person	Student	Person
Person	Student	Student	Student	Person
Student	Student	Person	Person	Person
Student	Student	Person	Student	Person
Student	Student	Student	Student	Student

- Die Referenzvariable, über die der Aufruf erfolgt, ist für die Klasse `Student` deklariert, das Argument ist eine Referenz auf `Person`.
- In `Student` ist nur *eine* Methode `whichMethod` bekannt mit Parameter `Person`.
- Der *Compiler* nimmt eine statische Analyse anhand der *deklarierten Typen* vor und sieht für die Ausführung die Methode `whichMethod( Person )` vor.
- Die Klasse `Student` hat die Methode `whichMethod( Person )` *geerbt*.
- Das *Objekt* der Klasse `Student` führt die Methode `whichMethod( Person )` aus.

## Kompatibilität von Referenzvariablen und Klassen - **Experimente**

(Fortsetzung)

Ausgabe:

calling reference declared type	argument declared type	runtime type	method from
<hr/>			
... <b>Student</b>	<b>Student</b>	<b>Student</b>	<b>Student</b>

Der *Compiler* kennt als Eigenschaft der Referenzvariablen:

<b>Student</b>	<b>mit Methoden</b> <b>whichMethod( Person p )</b> <b>(geerbt)</b> <b>und</b> <b>whichMethod( Student s )</b>
----------------	--

und wählt zur Ausführung die Methode:

**whichMethod( Student s )**

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	reference runtime type	argument declared type	argument runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
Person	Student	Person	Person	Person
Person	Student	Person	Student	Person
Person	Student	Student	Student	Person
Student	Student	Person	Person	Person
Student	Student	Person	Student	Person
Student	Student	Student	Student	Student

- Die Referenzvariable, über die der Aufruf erfolgt, ist für die Klasse `Student` deklariert, das Argument ist eine Referenz auf `Student`.
- In `Student` ist nur *eine* Methode `whichMethod` bekannt mit Parameter `Student`.
- Der *Compiler* nimmt eine statische Analyse anhand der *deklarierten Typen* vor.
- Der *Compiler* sieht für die Ausführung die Methode `whichMethod( Student )` vor.
- Das *Objekt* der Klasse `Student` führt die Methode `whichMethod( Student )` aus.

## Kompatibilität von Referenzvariablen und Klassen - **Experimente**

(Fortsetzung)

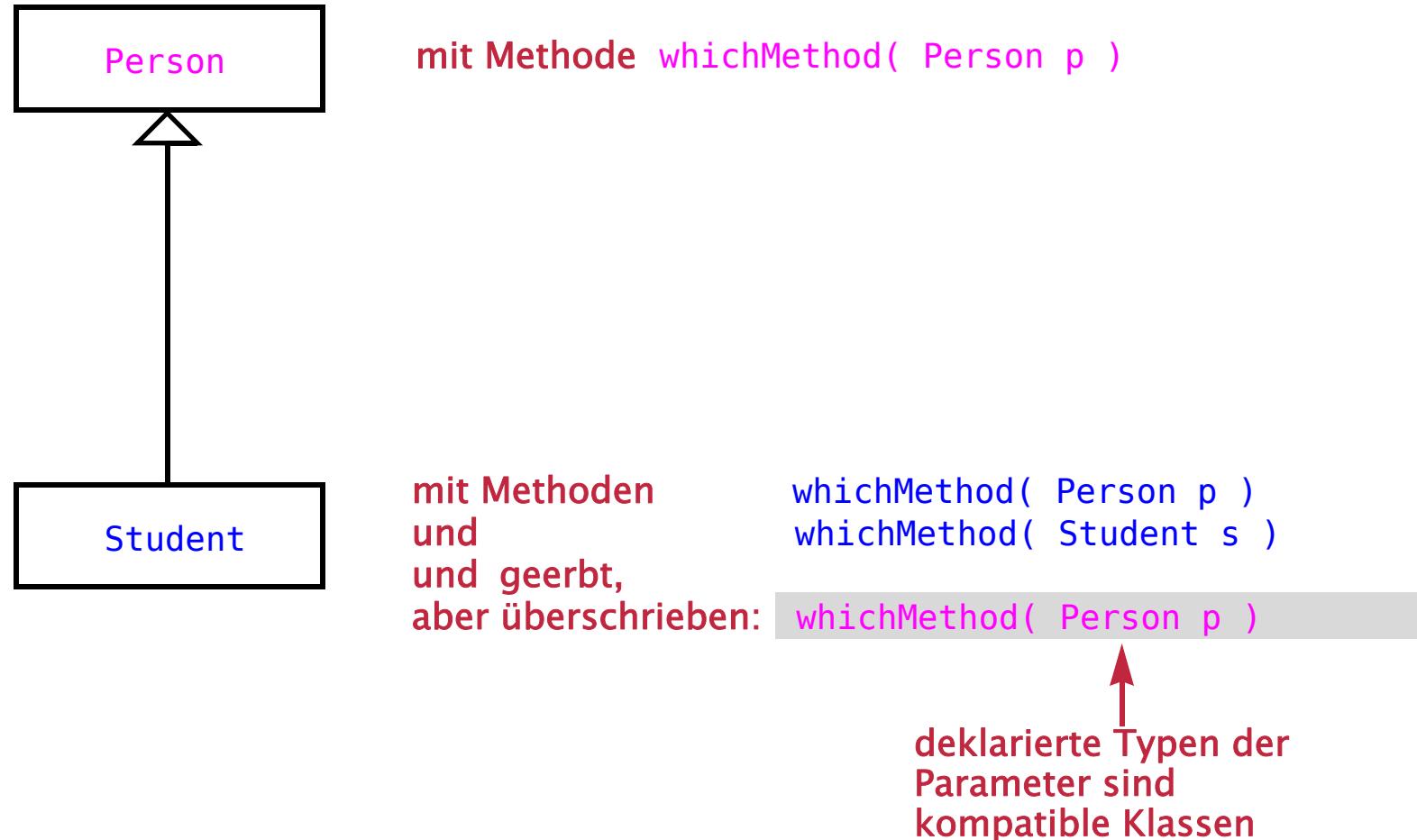
Jetzt wird zusätzlich in `Student` ergänzt:

```
public void whichMethod( Person p )
{
    System.out.println( "overridden in Student" );
}
```

- ❑ Die Klasse `Student` deklariert jetzt zwei Methoden mit gleichem Namen, die unterschiedliche Signaturen besitzen:
  - `whichMethod( Person p )`
  - `whichMethod( Student p )`
- ❑ Die neu hinzugekommene Methode überschreibt die Methode mit der gleichen Signatur, die von der Klasse `Person` geerbt wird, gibt aber den Text "overridden in `Student`" zurück.

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)



## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	reference runtime type	argument declared type	argument runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
Person	Student	Person	Person	overridden in Student
Person	Student	Person	Student	overridden in Student
Person	Student	Student	Student	overridden in Student
Student	Student	Person	Person	overridden in Student
Student	Student	Person	Student	overridden in Student
Student	Student	Student	Student	Student

- Beim Aufruf der bekannten Methode wird nun *häufig* die überschriebene Methode aus der Klasse Student aufgerufen.

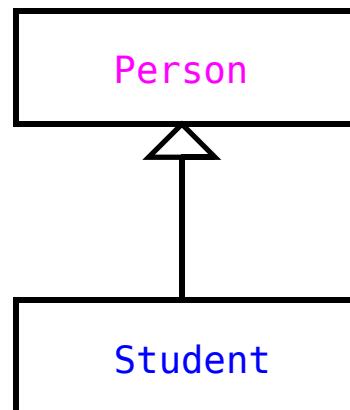
Warum?

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	runtime type	argument declared type	runtime type	method from
...				
Person	Student	Person	Person	overridden in Student
Person	Student	Person	Student	overridden in Student
Person	Student	Student	Student	overridden in Student
...				



mit Methode `whichMethod( Person p )`

mit Methoden  
und  
und geerbt,  
aber überschrieben:

`whichMethod( Person p )`  
`whichMethod( Student s )`

`whichMethod( Person p )`

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	reference runtime type	argument declared type	argument runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
Person	Student	Person	Person	overridden in Student
Person	Student	Person	Student	overridden in Student
Person	Student	Student	Student	overridden in Student
Student	Student	Person	Person	overridden in Student
Student	Student	Person	Student	overridden in Student
Student	Student	Student	Student	Student

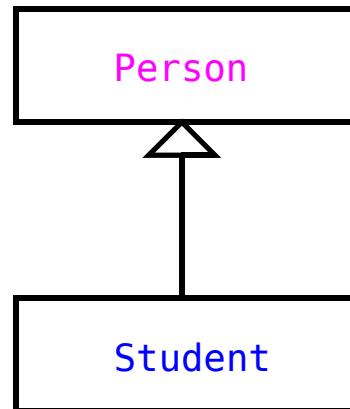
- Da die Referenz für die Klasse Person deklariert ist, wählt der *Compiler* die einzige dort bekannte Methode mit der Signatur `whichMethod( Person )` aus.
- Der Laufzeittyp der Referenz ist gegeben durch ein Objekt der Klasse Student mit der überschriebenen Methode, so dass das *Laufzeitsystem* für die Ausführung diese Implementierung der Methode `whichMethod( Person )` auswählt.

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	runtime type	argument declared type	runtime type	method from
...				
Student	Student	Person	Person	overridden in Student
Student	Student	Person	Student	overridden in Student
...				



mit Methode `whichMethod( Person p )`

mit Methoden  
und  
und geerbt,  
aber überschrieben: `whichMethod( Person p )`

`whichMethod( Person p )`  
`whichMethod( Student s )`

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ausgabe:

calling reference declared type	runtime type	argument declared type	runtime type	method from
Person	Person	Person	Person	Person
Person	Person	Person	Student	Person
Person	Person	Student	Student	Person
Person	Student	Person	Person	overridden in Student
Person	Student	Person	Student	overridden in Student
Person	Student	Student	Student	overridden in Student
Student	Student	Person	Person	overridden in Student
Student	Student	Person	Student	overridden in Student
Student	Student	Student	Student	Student

- Da die Referenz für die Klasse `Student` deklariert ist und das Argument eine Referenz auf `Person` ist, wählt der *Compiler* die dort deklarierte Methode mit der Signatur `whichMethod( Person )` aus.
- Der Laufzeittyp der Referenz ist gegeben durch ein Objekt der Klasse `Student` mit der überschriebenen Methode, so dass das *Laufzeitsystem* für die Ausführung diese Implementierung der Methode `whichMethod( Person )` auswählt.

## Kompatibilität von Referenzvariablen und Klassen - Experimente

(Fortsetzung)

Ergebnis der Experimente:

- In Java bestimmt der *Compiler* ausgehend vom deklarierten Typ der Referenz *immer eindeutig* die Signatur der Methode, die zur Laufzeit ausgeführt wird:
  - Der Compiler richtet sich nach den ihm zur Verfügung stehenden Informationen, also den Angaben, die durch eine *statische* Analyse aus dem Programmtext abgeleitet werden können.
  - Der Compiler wählt daher *nie* eine Methodensignatur aus, die für den deklarierten Typ der Referenz *nicht* deklariert ist.
  - Der Compiler wählt die Signatur aus, die mit den Typen ihrer Parameter am *besten zu* den deklarierten Typen der übergebenen Argumente passt.
- Der Compiler berücksichtigt bei seiner Entscheidung *nie* den Laufzeittyp, also die Klasse eines Objekts, auf das eine Referenz verweist.
- Bei der Ausführung wird in dem Objekt, das die Ausführung vornimmt, die passende Methode zu der *vom Compiler festgelegten Signatur* gesucht und ausgeführt. Hierdurch erfolgt dann eine spezifische Anpassung der Ausführung an den Laufzeittyp.

## Kompatibilität von Referenzvariablen und Klassen – Experimente

(Fortsetzung)

Ergebnis der Experimente:

- In Java bestimmt der *Compiler* ausgehend vom deklarierten Typ der Referenz *immer eindeutig* die Signatur der Methode, die zur Laufzeit ausgeführt wird:
  - Der Compiler richtet sich nach den ihm zur Verfügung stehenden Informationen, also den Angaben, die durch eine *statische* Analyse aus dem Programmtext abgeleitet werden können.
  - Der Compiler wählt daher *nie* eine Methodensignatur aus, die für den deklarierten Typ der Referenz *nicht* deklariert ist.
  - Der Compiler wählt die Signatur aus, die mit den Typen ihrer Parameter am *besten zu* den deklarierten Typen der übergebenen Argumente passt.
- Der Compiler berücksichtigt bei seiner Entscheidung *nie* den Laufzeittyp, also die Klasse eines Objekts, auf das eine Referenz verweist.
- Bei der Ausführung wird in dem Objekt, das die Ausführung vornimmt, die passende Methode zu der *vom Compiler festgelegten Signatur* gesucht und ausgeführt. Hierdurch erfolgt dann eine spezifische Anpassung der Ausführung an den Laufzeittyp.

Um ein verständliches Verhalten von Programmen sicherzustellen, sollten daher keine Methoden mit unterschiedlichen, aber zueinander kompatiblen Signaturen deklariert werden.

## Überschreiben von Methoden – Absicherung

Überschreiben in der Klasse Student:

Annotation

```
@Override ←  
public void whichMethod( Person p )  
{  
    System.out.println( "overridden in Student" );  
}
```

- Die Annotation `@Override` zeigt an, dass die Methode `whichMethod( Person )` eine Methode mit der *gleichen* Signatur aus der Oberklasse überschreiben *soll*.
- Gibt es in der Oberklasse keine Methode mit der gleichen Signatur, meldet der Compiler bei der Verwendung der `@Override`-Annotation einen Fehler.
- Durch die Verwendung von `@Override`-Annotationen können also in Unterklassen Flüchtigkeitsfehler beim Deklarieren von Methoden vermieden werden.
- Die `@Override`-Annotation muss immer unmittelbar vor der Methodendeklaration stehen, die ein Überschreiben vornimmt.

## Anmerkungen

- ❑ Vererbung erfolgt *transitiv*,  
auch geerbte Eigenschaften werden von Klassen an ihre Unterklassen weitergegeben.
- ❑ Auch die auf Vererbung basierende Kompatibilität von Klassen wird auf diese Weise transitiv weitergegeben:
  - Da eine Klasse  $K$  alle von ihrer Oberklasse  $O$  geerbten Eigenschaften an ihre Unterklasse  $U$  weitervererbt, besitzt  $U$  alle Eigenschaften von  $O$ .
  - Referenzen, deren deklarierter Typ die Klasse  $O$  ist, können daher als Laufzeittyp die Klasse  $U$  besitzen, also auf Objekte der Klasse  $U$  verweisen.
- ❑ Auf diese Weise entstehen Klassen- bzw. Vererbungshierarchien.
- ❑ Jede Klasse in Java hat aber immer nur genau eine Oberklasse.
- ❑ Jede Klasse kann viele Unterklassen haben.
- ❑ Die Möglichkeit, dass in einer Klassenhierarchie durch Überschreiben unterschiedliche Implementierungen zu der gleichen Signatur auftreten dürfen, wird als *Polymorphie* (= Vielgestaltigkeit) bezeichnet.

## Vorteile der Vererbung

### Welche Vorteile bringt das Konzept der Vererbung?

- Es wird die Möglichkeit geschaffen, Hierarchien von zueinander kompatiblen Klassen anzulegen.
- Die Klassen in solchen Hierarchien können dazu genutzt werden, allgemein einsetzbare Algorithmen und Datenstrukturen zu implementieren.
- Innerhalb dieser Hierarchien ist eine gewisse Gleichförmigkeit der Signaturen notwendig, damit der Mechanismus der polymorphen Auswahl von Methoden gezielt eingesetzt werden kann. Diese Gleichförmigkeit verbessert auch die Verständlichkeit der Software.
- Der Aufwand zum Erstellen und Ändern in diesen Hierarchien wird vermindert, da geerbte Eigenschaften nicht wiederholt aufgeschrieben werden müssen.
- Die Implementierung von speziellen Varianten von Datenstrukturen oder Algorithmen kann in Unterklassen ausgelagert werden.
- Die Leistungsfähigkeit einer Klasse kann durch das Ableiten einer geeigneten Unterklasse verbessert werden, ohne dass an der Oberklasse Veränderungen vorgenommen werden müssen.

## Deklaration von Konstruktoren

Rückblick auf Folie 661:

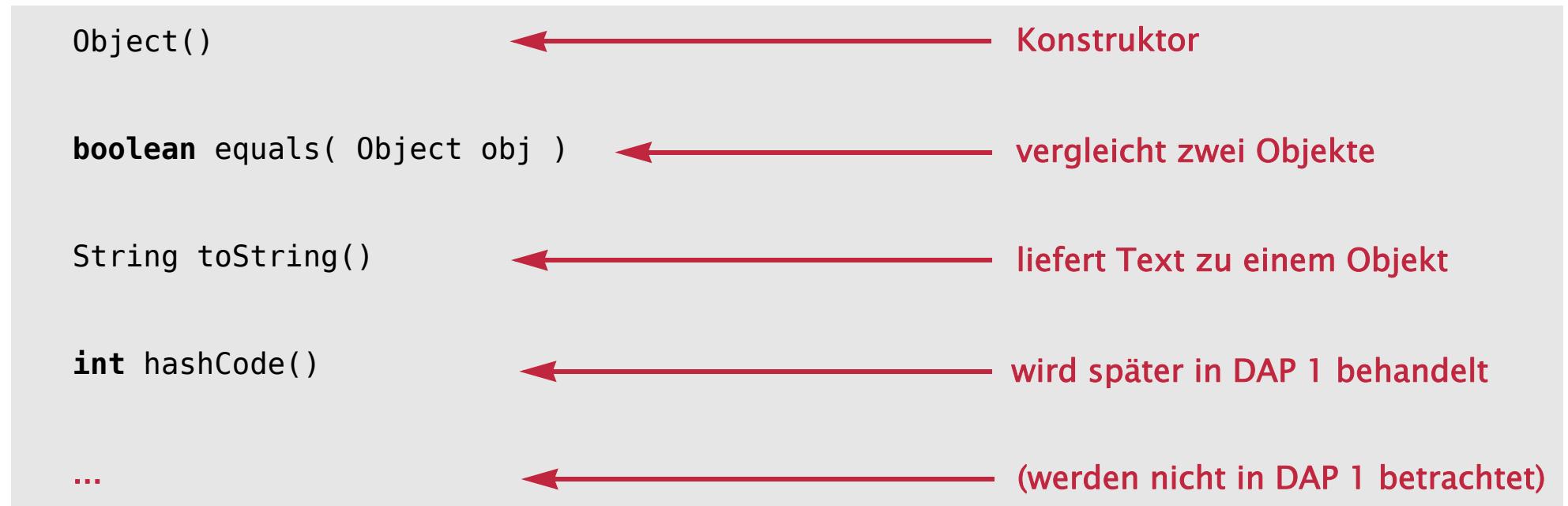
...

- Die *erste Anweisung* eines Konstruktors ist immer ein Aufruf eines Konstruktors der Oberklasse:  
*Explizit* durch Angabe von `super` gefolgt von einer Parameterliste oder  
*implizit* – dann ergänzt der Compiler den Aufruf des Standardkonstruktors durch `super()`.

**Was ist mit der Klasse Person, für die keine Oberklasse angegeben ist?**

- In Java existiert eine Klasse `Object`, von der alle Klassen erben, für die keine Oberklasse explizit durch `extends` angegeben ist.  
Bei diesen Klassen wird automatisch ein `extends Object` verwendet.
- Daraus folgt:
  - Die Klasse `Object` ist die (indirekte) Oberklasse aller anderen Klassen in Java.
  - Referenzen auf `Object` können auf beliebige Objekte verweisen.
  - Alle Klassen bieten die Methoden der Klasse `Object` an,  
da jede Klasse diese Methoden direkt oder indirekt erbt.
- `Object` ist also die Klasse,  
auf deren Grundlage beliebig allgemeine Klassen erstellt werden können.

## Klasse Object



## Klasse Object

(Fortsetzung)

### Anmerkungen

- ❑ Der Konstruktor `Object()` stellt sicher, dass der Konstruktor jeder deklarierten Klasse einen Aufruf des (Standard-)Konstruktors seiner Oberklasse tätigen kann.

## Klasse Object

(Fortsetzung)

### Anmerkungen

- Die in `Object` definierte Form der Methode `equals` entspricht:  
 $x == y$  (*Identität*)
- Diese Form der Gleichheit ist in vielen Situationen zu streng.  
Daher wird die Methode `equals` in der Regel spezifisch für die Aufgabe einer Klasse überschrieben.
- Die Methode `equals` sollte aber immer die folgenden Bedingungen einhalten:
  - Es sollte eine Äquivalenzrelation definiert werden: reflexiv, symmetrisch, transitiv
  - Verschiedene Aufrufe von `equals` mit den gleichen beiden Objekten sollten das gleiche Ergebnis liefern, solange nicht eines der Objekte geändert wurde (Konsistenz).
  - Für jede Referenz `x` auf ein Objekt soll gelten: `x.equals( null ) == false`
  - Das Ergebnis von `equals` muss konsistent zum Ergebnis der – noch unbekannten – Methode `hashCode()` sein:

$$a.equals(b) \Rightarrow a.hashCode() == b.hashCode()$$

## Klasse Object

(Fortsetzung)

### Anmerkungen

- ❑ Die Methode `equals` besitzt die folgende Signatur: `boolean equals( Object obj )`
  - Ein Überschreiben erfolgt *nur dann*, wenn in einer Klasse die Methode `equals` auch genau mit dem ParameterTyp `Object` überschrieben wird: Verwendung von `@Override`.
  - Beim Überschreiben der Methode `equals` muss also immer ein Vergleich mit jedem beliebigen Objekt implementiert werden, da ein Parameter des Typs `Object` beliebige Klassen als Laufzeittypen für das übergebene Argument akzeptiert.
- ❑ Die Methode `toString` soll eine Textrepräsentation des Objekts zurückgeben.  
Die in `Object` definierte Form von `toString` liefert *Klassename@hashCode*.
- ❑ Die `toString`-Methode eines Objekts wird auch immer dann *implizit* aufgerufen, wenn ein `String`-Operand erwartet wird:

```
String s = "!" + new Object();
```

erzeugt den Text:

```
!java.lang.Object@5b77ee02
```

## Nutzung von Vererbung

(Fortsetzung)

Wie kann Vererbung genutzt werden, um allgemein einsetzbare Klassen zu gestalten?

Lösungsansatz:

```
public class BinarySearchTree
{
    private BinarySearchTree leftChild,
                           rightChild;
```

...

```
public class CharacterSearchTree
extends BinarySearchTree
{
    private HuffmanTriple content;
}
```

...

Gemeinsamkeiten in Oberklasse zusammenfassen  
und durch Vererbung weitergeben

```
public class StudentSearchTree
extends BinarySearchTree
{
    private Student content;
}
```

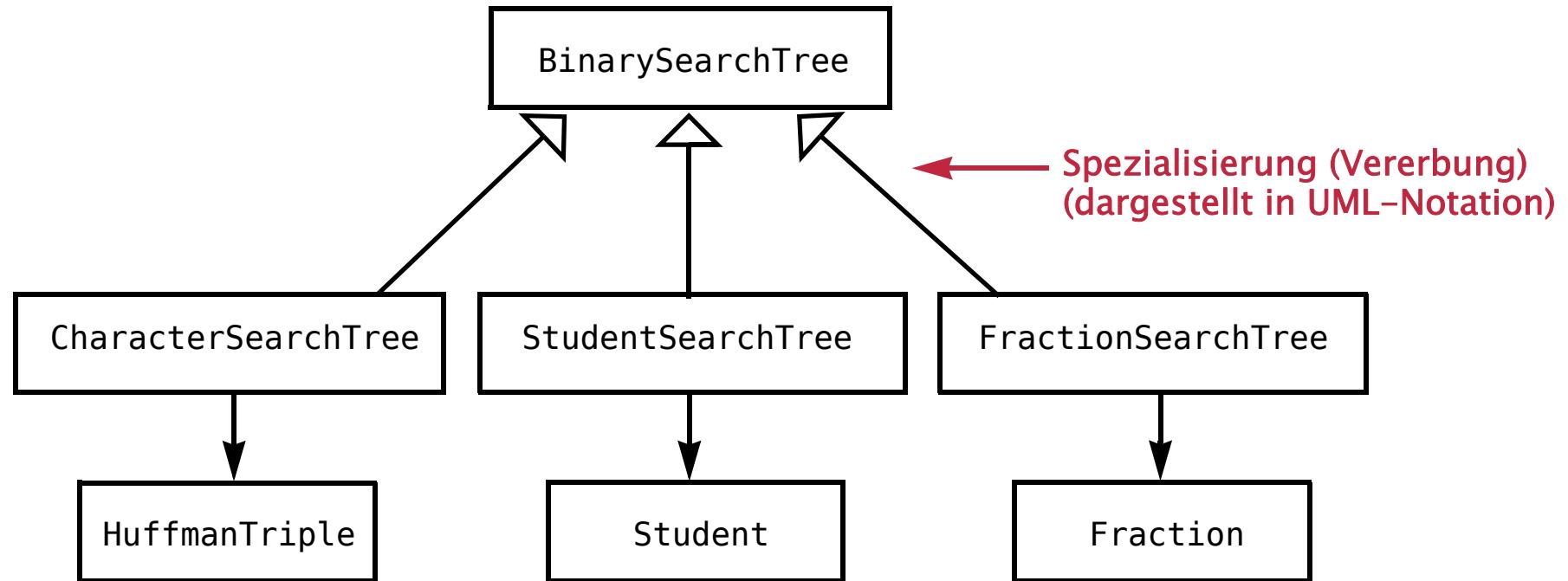
...

## Nutzung von Vererbung

(Fortsetzung)

Wie kann Vererbung genutzt werden, um allgemein einsetzbare Klassen zu gestalten?

Lösungsansatz:

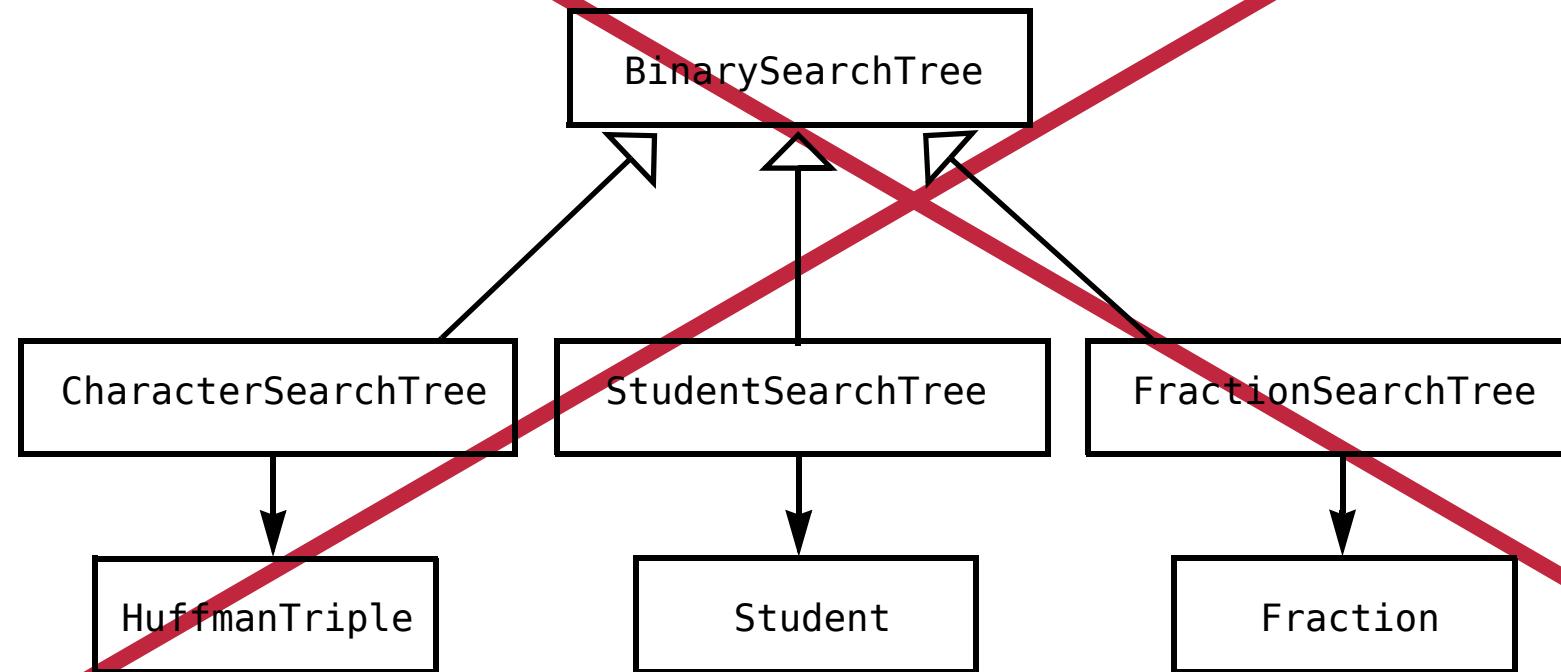


## Nutzung von Vererbung

(Fortsetzung)

Wie kann Vererbung genutzt werden, um allgemein einsetzbare Klassen zu gestalten?

Lösungsansatz:



komplizierte Lösung, für jeden Inhalt muss ein neuer Baum implementiert werden

## Nutzung von Vererbung

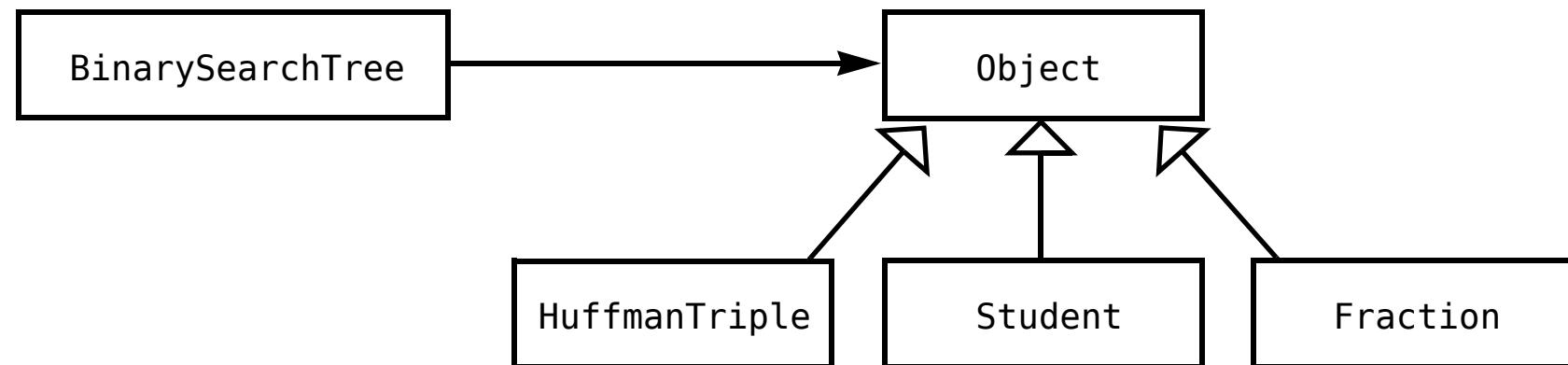
(Fortsetzung)

neuer Lösungsansatz:

nur eine Sorte von Knoten, die flexibel zur Datenhaltung benutzt werden können

```
public class BinarySearchTree
{
    private Object content;
    private BinarySearchTree leftChild, rightChild;
```

Referenz auf Object:  
kann auf jedes Objekt  
verweisen



aber: Object besitzt keine Ordnungsrelation

## Nutzung von Vererbung

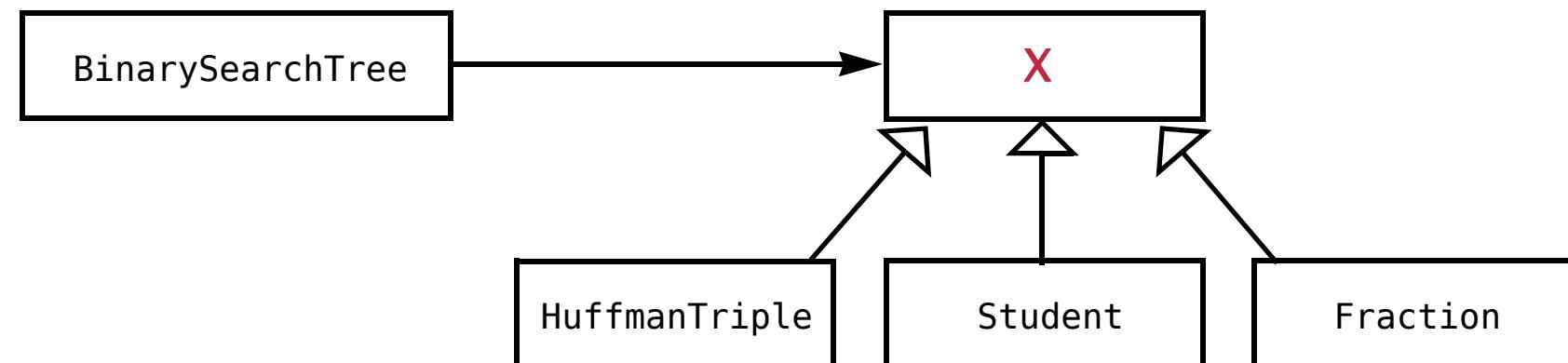
(Fortsetzung)

neuer Lösungsansatz:

nur eine Sorte von Knoten, die flexibel zur Datenhaltung benutzt werden können

```
public class BinarySearchTree
{
    private X content;
    private BinarySearchTree leftChild, rightChild;
```

← Die Klasse X muss eine  
Ordnungsrelation garantieren



## Nutzung von Vererbung

(Fortsetzung)

vor der Umgestaltung des Suchbaums ein einfacheres Beispiel,  
das keine Anforderungen an die abgelegten Inhalte stellt:

### *doppelt verkettete Liste*

Eine Liste ist eine Datenstruktur,

- in der (beliebige) Inhalte sequentiell nacheinander abgelegt werden können,
- die einen Anfang hat,
- die ein Ende hat,
- die sequentiell durchlaufen werden kann,
- in die Daten eingefügt werden können,
- aus der Daten entfernt werden können.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 9. Datenstruktur «Doppelt verkettete Liste»

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-716

## Lernziele des Kapitels 9. Datenstruktur «Doppelt verkettete Liste»

Nach Durcharbeiten des Kapitels Doppelt verkettete Liste sollen die teilnehmenden Studierenden

- die Datenstruktur der doppelt verketteten Liste kennen,
- die Klasse `Element` einsetzen können,
- die Klasse `DoublyLinkedList` erweitern können,
- `Wrapper`-Klassen kennen und einsetzen können
- die Konzepte *Boxing* und *Unboxing* kennen und einsetzen können,
- das Konzept des *Type-Casts* kennen und einsetzen können.

## Idee der Datenstruktur Liste

- ❑ Die Liste dient dem Abspeichern von vielen Werten.
- ❑ Die Werte werden nacheinander – also in einer Sequenz – abgelegt.

Diese Anforderungen ähneln den Möglichkeiten von Feldern.

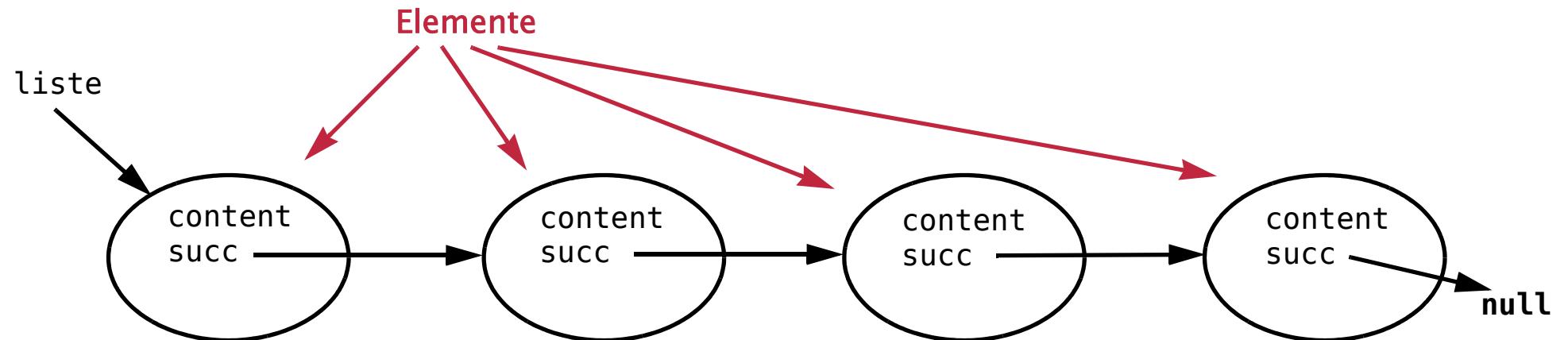
- ❑ Die Anzahl der abgelegten Werte soll variabel und vorab nicht festgelegt sein.
- ❑ Werte sollen an beliebigen Stellen der Sequenz ergänzt werden können.
- ❑ Werte sollen an beliebigen Stellen der Sequenz gelöscht werden können.

Diese Anforderungen lassen sich für Felder nicht oder nur umständlich umsetzen.

## Idee der Datenstruktur Liste

(Fortsetzung)

Visualisierung



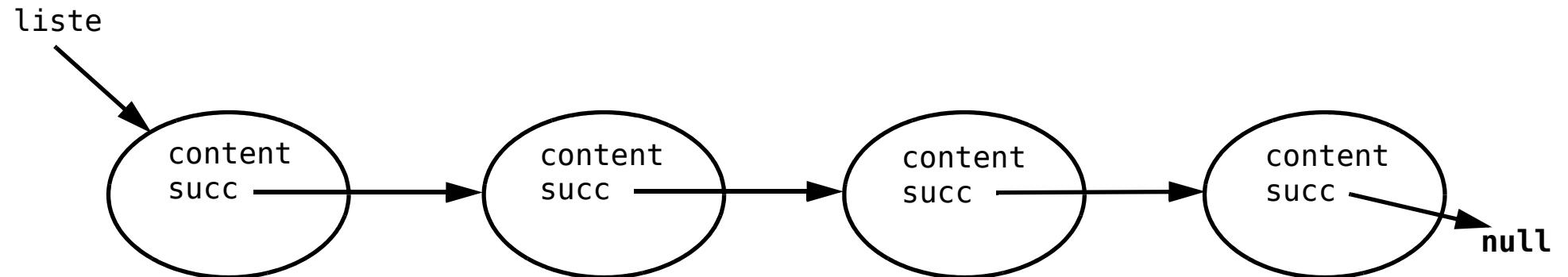
Hinzunehmen und Löschen von Elementen:  
- Elemente erzeugen  
- Referenzen umsetzen

succ = *successor* (Nachfolger)

## Idee der Datenstruktur Liste

(Fortsetzung)

### Visualisierung



#### Hinzunehmen und Löschen von Elementen:

- Elemente erzeugen
- Referenzen umsetzen

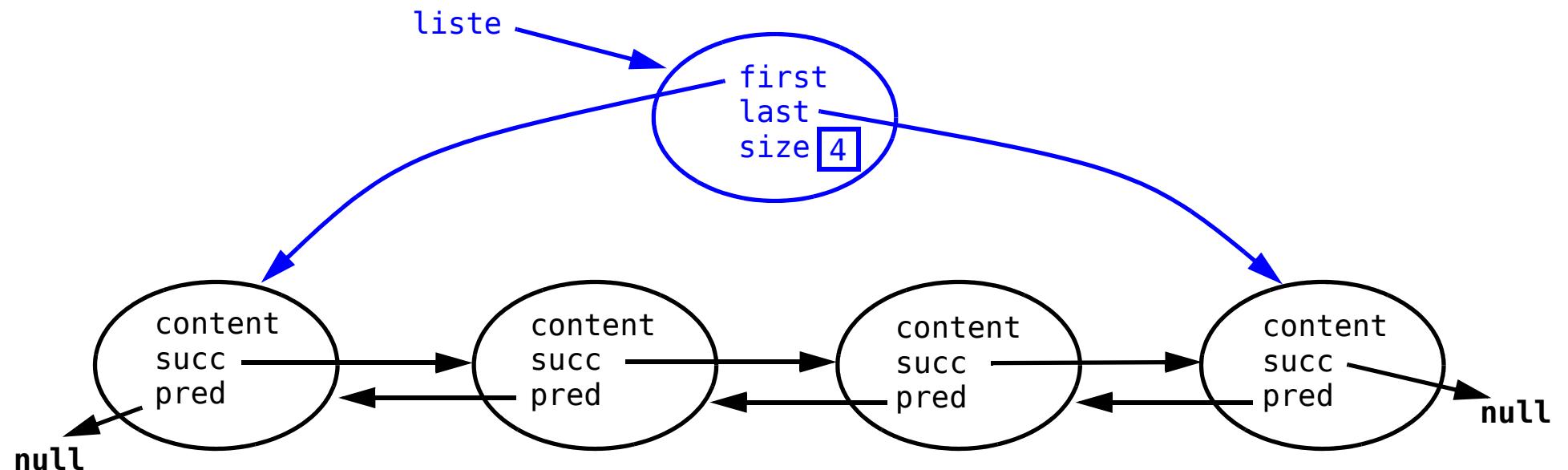
aber:

- Methoden umständlich (z.B. Setzen von Referenzen beim Löschen)
- Ausführung langsam (z.B. Anhängen am Ende erfordert kompletten Durchlauf)

## Idee der Datenstruktur Liste

(Fortsetzung)

Visualisierung



Ergänzungen:

- Nutzung von zwei klassen (verwaltung eines elements, verwaltung der ganzen liste)
- elemente in beide richtungen verbinden, um in beide richtungen navigieren zu können

*pred = predecessor (Vorgänger)*

## Aufbau der Datenstruktur

- Die Klasse `Element` repräsentiert ein einzelnes Element der Liste.
  - Jedes Element kann einen beliebigen Inhalt haben.
  - Jedes Element kann sich mit seinem Vorgänger in der Liste verbinden.
  - Jedes Element kann sich mit seinem Nachfolger in der Liste verbinden.
  
- Die Klasse `DoublyLinkedList` organisiert aus Objekten der Klasse `Element` eine Liste.
  - Die Liste ist leer oder hat ein Anfangs- und ein Endelement.  
Das Anfangselement hat keinen Vorgänger, das Endelement keinen Nachfolger.
  - Die Elemente der Liste sind so verbunden, dass vom Anfangselement ausgehend alle Elemente erreicht werden können.
  - Die Elemente der Liste sind so verbunden, dass vom Endelement ausgehend alle Elemente erreicht werden können.

## Aufbau der Datenstruktur

(Fortsetzung)

```
public class Element
{
    private Object content;
    private Element pred, succ;
    ...
}

public class DoublyLinkedList
{
    private Element first, last;
    private int size;
    ...
}
```

The code shows two annotations with red arrows pointing from text labels to specific attributes:

- An arrow points from the label "beliebiger Inhalt" to the attribute `private Object content;`.
- An arrow points from the label "Verkettung der Elemente" to the attributes `private Element pred, succ;`.

The code shows two annotations with red arrows pointing from text labels to specific attributes:

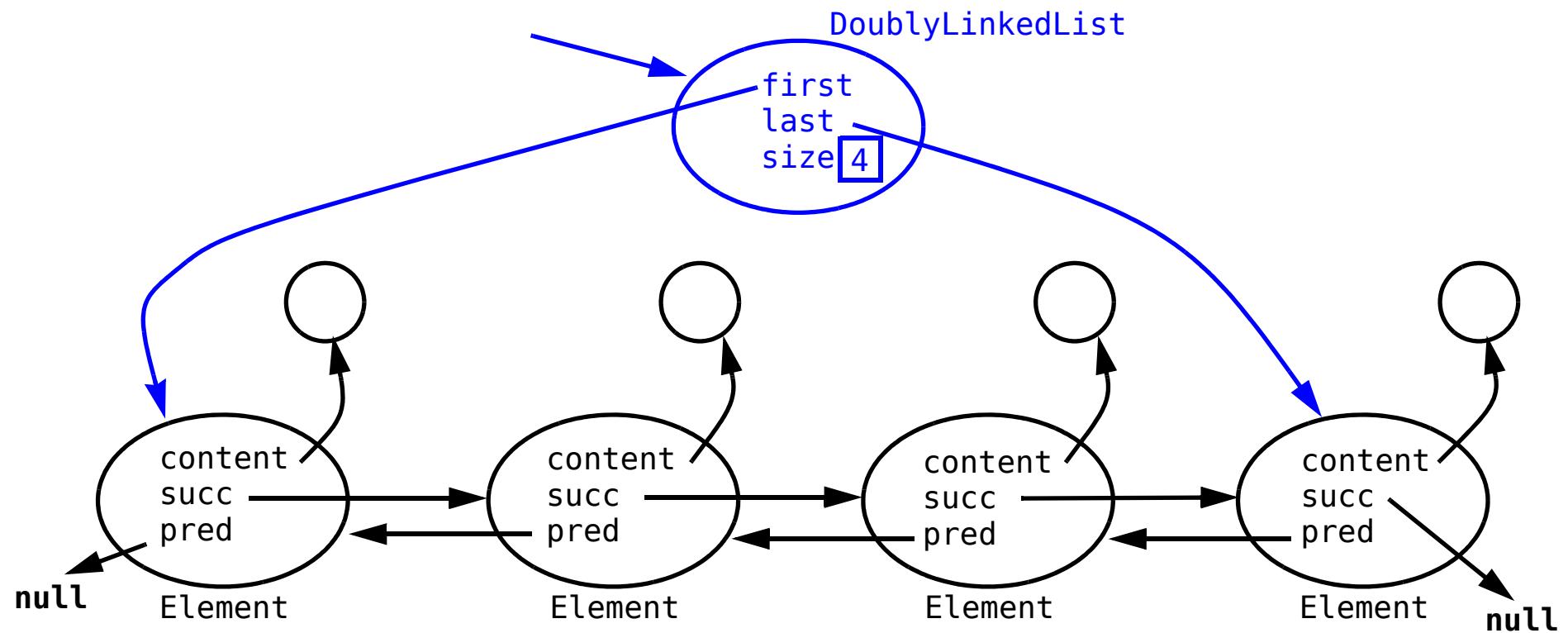
- An arrow points from the label "Anfangs- und Endelement" to the attribute `private Element first, last;`.
- An arrow points from the label "Anzahl der Elemente" to the attribute `private int size;`.

- Die Attribute der Klasse `Element` ähneln denen der Klasse `BinarySearchTree`: Erst die gezielte Verwendung der Attribute `pred` und `succ` in den Methoden führt zu der andersartigen Datenstruktur.
- Auch die Liste ist eine rekursiv deklarierte Datenstruktur, da die `Element`-Objekte Referenzen auf den eigenen Typ enthalten.

## Aufbau der Datenstruktur

(Fortsetzung)

Beispiel: Objekte einer Liste mit vier Elementen



## Klasse Element

```
public class Element
{
    private Object content;
    private Element pred, succ;

    public Element( Object c )
    {
        content = c;
        pred = succ = null;
    }

    public Object getContent()
    {
        return content;
    }

    public void setContent( Object c )
    {
        content = c;
    }
    ...
}
```

The code is annotated with red arrows and text:

- A red arrow points from the word "content" in the first private field declaration to the text "beliebiger Inhalt" (arbitrary content).
- A red arrow points from the declarations of "pred" and "succ" to the text "Verkettung der Elemente" (linking of elements).
- A red arrow points from the constructor header "public Element( Object c )" to the text "Konstruktor" (constructor).
- A red arrow points from the "getContent()" method header to the text "einfache Methoden" (simple methods).

## Klasse Element

(Fortsetzung)

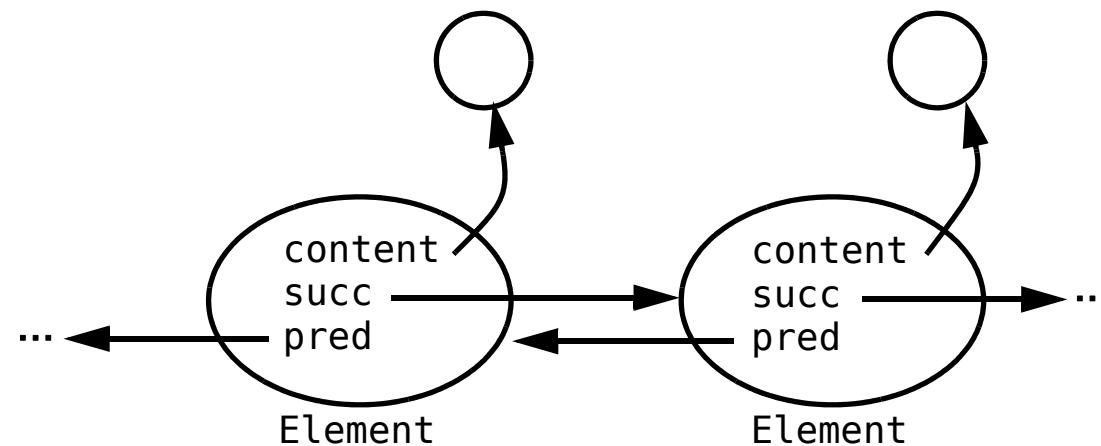
```
...
public boolean hasSucc()
{
    return succ != null;
}

public Element getSucc()
{
    return succ;
}
...
```

Existenz

Methoden, die sich  
mit dem Nachfolger  
beschäftigen

Zugriff



## Klasse Element

(Fortsetzung)

- ❑ Die beiden Referenzen `succ` und `pred` müssen nach der Ausführung einer Methode der Klasse `Element` immer die folgenden Bedingungen einhalten, die die Verbindung von Vorgänger und Nachfolger in beide Richtungen sicherstellen:

```
succ == null || succ.pred == this  
pred == null || pred.succ == this
```

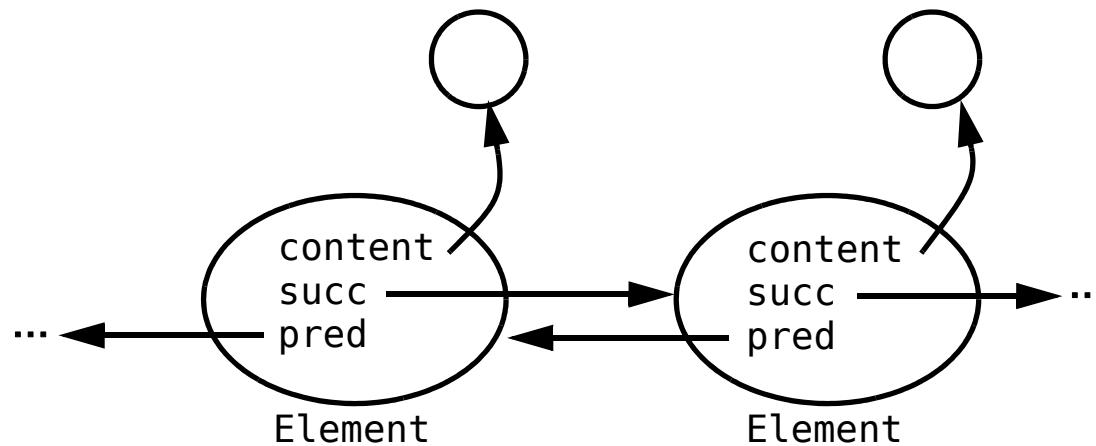
- ❑ Solche Bedingungen werden auch als *Invariante* bezeichnet, da sie für jedes Element einer beliebigen Liste unabhängig von deren Inhalt und deren Anzahl immer gelten.
- ❑ Daraus lassen sich die Implementierungen der Methoden zum Verknüpfen und Trennen von Elementen ableiten.

## Klasse Element

(Fortsetzung)

```
...
public void disconnectSucc()
{
    if ( hasSucc() )
    {
        succ.pred = null;
        succ = null;
    }
}
...
```

Abtrennen, falls vorhanden  
(Einhalten der Invariante)



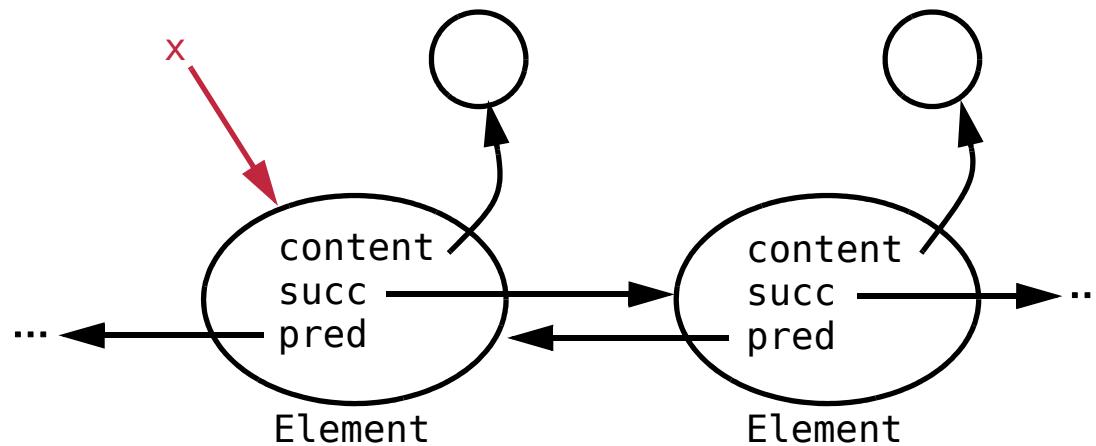
## Klasse Element

(Fortsetzung)

```
...
public void disconnectSucc()
{
    if ( hasSucc() )
    {
        succ.pred = null;
        succ = null;
    }
}
...
```

Abtrennen, falls vorhanden  
(Einhalten der Invariante)

Beispielaufruf: `x.disconnectSucc();`



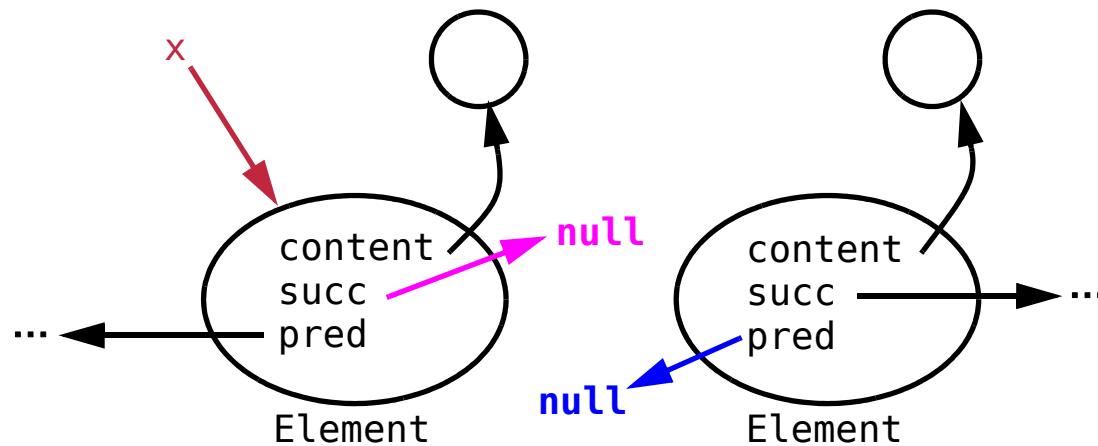
## Klasse Element

(Fortsetzung)

```
...
public void disconnectSucc()
{
    if ( hasSucc() )
    {
        succ.pred = null;
        succ = null;
    }
}
...
```

Abtrennen, falls vorhanden  
(Einhalten der Invariante)

Beispielaufruf: `x.disconnectSucc();`

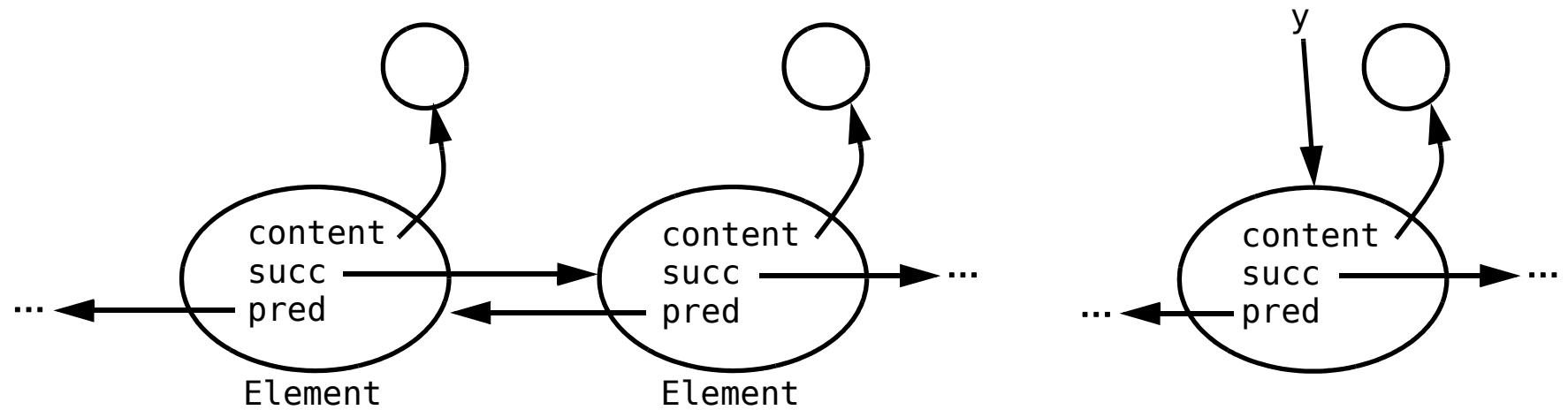


## Klasse Element

(Fortsetzung)

```
...
public void connectAsSucc( Element e )
{
    disconnectSucc();
    if ( e != null )
    {
        e.disconnectPred();
        e.pred = this;
    }
    succ = e;
}
...
```

Hinzufügen (Einhalten der Invariante)



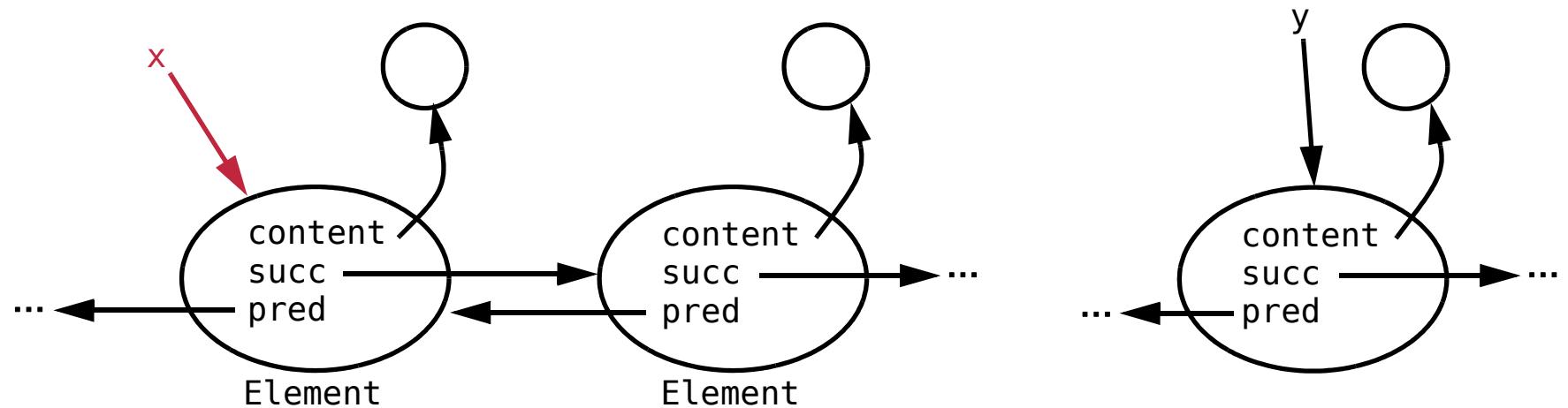
## Klasse Element

(Fortsetzung)

```
...
public void connectAsSucc( Element e )
{
    disconnectSucc();
    if ( e != null )
    {
        e.disconnectPred();
        e.pred = this;
    }
    succ = e;
}
...
```

Einhalten der Invariante

Beispielaufruf: `x.connectAsSucc( y );`



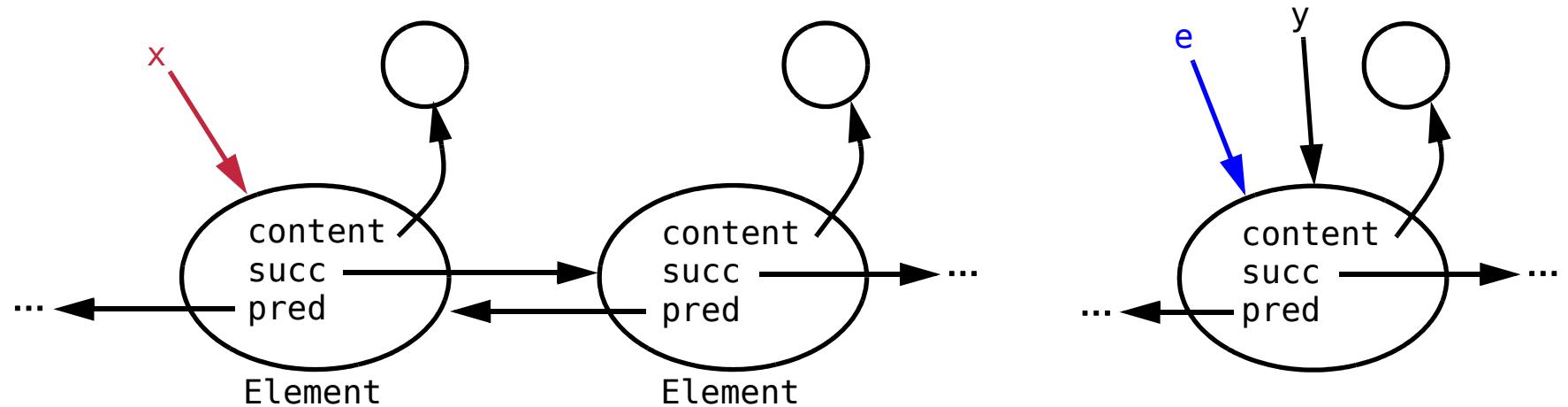
## Klasse Element

(Fortsetzung)

```
...
public void connectAsSucc( Element e )
{
    disconnectSucc();
    if ( e != null )
    {
        e.disconnectPred();
        e.pred = this;
    }
    succ = e;
}
...
```

Einhalten der Invariante

Beispielaufruf: `x.connectAsSucc( y );`



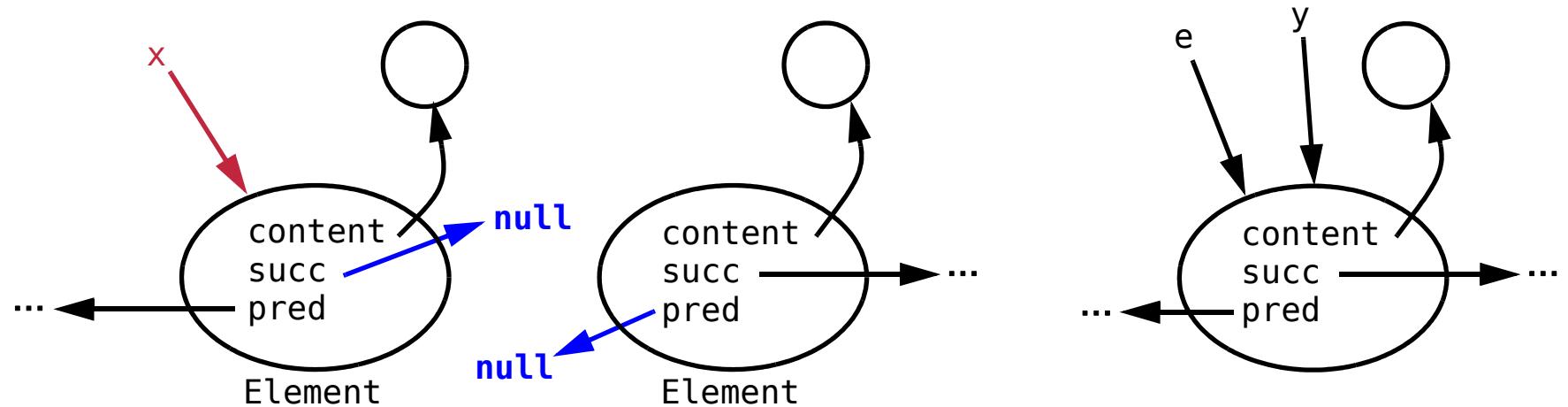
## Klasse Element

(Fortsetzung)

```
...
public void connectAsSucc( Element e )
{
    disconnectSucc();
    if ( e != null )
    {
        e.disconnectPred();
        e.pred = this;
    }
    succ = e;
}
...
```

Einhalten der Invariante

Beispielaufruf: `x.connectAsSucc( y );`



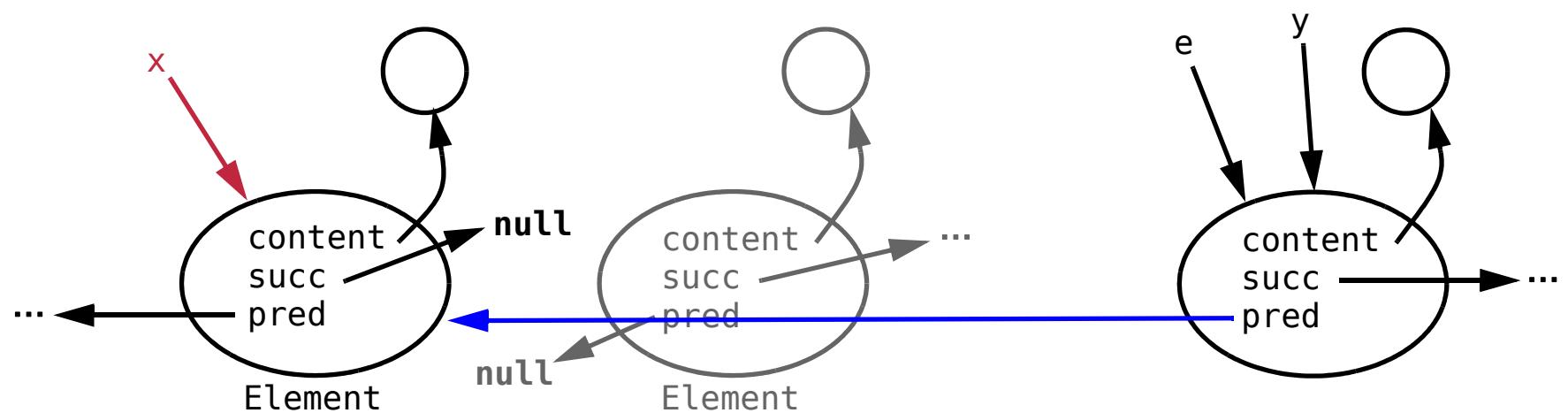
## Klasse Element

(Fortsetzung)

```
...
public void connectAsSucc( Element e )
{
    disconnectSucc();
    if ( e != null )
    {
        e.disconnectPred();
        e.pred = this;
    }
    succ = e;
}
...
```

Einhalten der Invariante

Beispielaufruf: `x.connectAsSucc( y );`



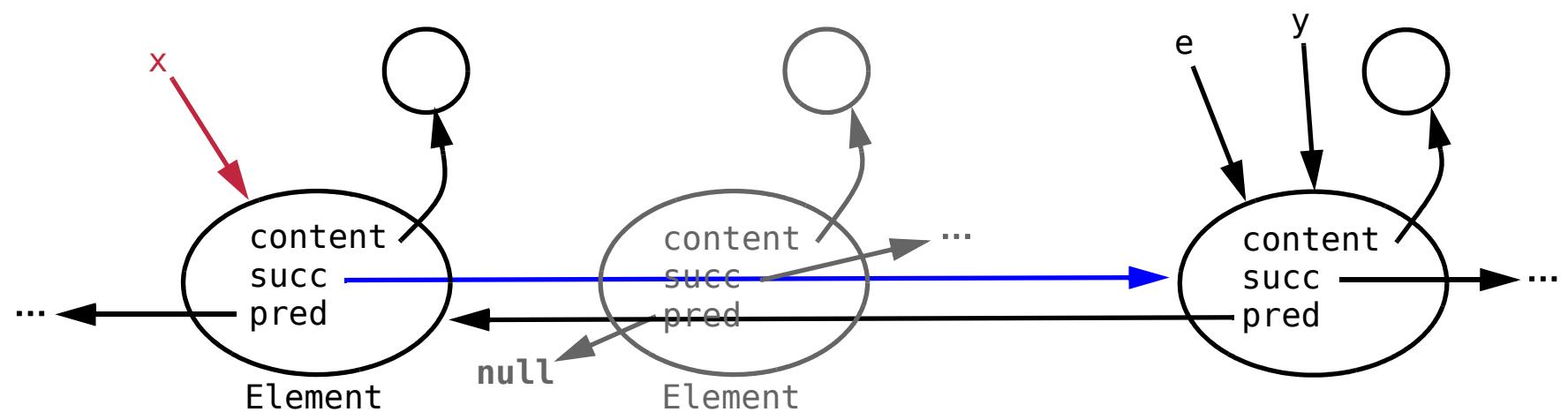
## Klasse Element

(Fortsetzung)

```
...
public void connectAsSucc( Element e )
{
    disconnectSucc();
    if ( e != null )
    {
        e.disconnectPred();
        e.pred = this;
    }
    succ = e;
}
...
```

Einhalten der Invariante

Beispielaufruf: `x.connectAsSucc( y );`



## Klasse Element

(Fortsetzung)

```
...
public void connectAsSucc( Element e )
{
    disconnectSucc();
    if ( e != null )
    {
        e.disconnectPred();
        e.pred = this;
    }
    succ = e;
}
...
```

Einhalten der Invariante

Beispielaufruf: `x.connectAsSucc( y );`



## Klasse Element

(Fortsetzung)

```

public boolean hasPred() {
    return pred != null;
}

public Element getPred() {
    return pred;
}

public void disconnectPred() {
    if ( hasPred() ) {
        pred.succ = null;
        pred = null;
    }
}

public void connectAsPred( Element e ) {
    disconnectPred();
    if ( e != null ) {
        e.disconnectSucc();
        e.succ = this;
    }
    pred = e;
}

```

Existenz

Zugriff

analoge Methoden,  
die sich mit dem  
Vorgänger beschäftigen

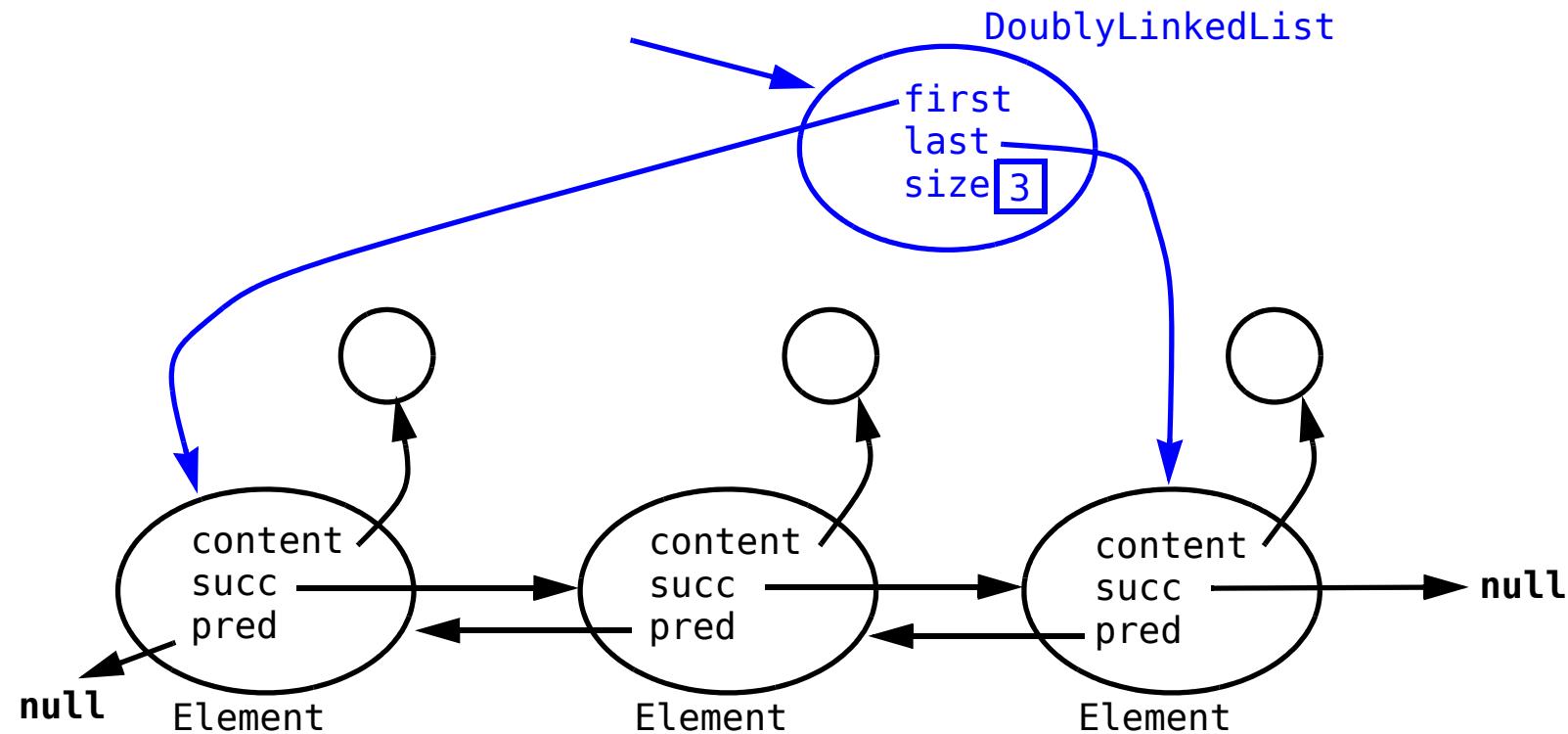
Abtrennen (Einhalten der Invariante)

Hinzufügen (Einhalten der Invariante)

## Erinnerung: Aufbau der Datenstruktur

(Fortsetzung)

Klasse DoublyLinkedList



## Klasse DoublyLinkedList

```
public class DoublyLinkedList
{
    private Element first, last;
    private int size;           ← Anfangs- und Endelement
                                ← Anzahl der Elemente

    public DoublyLinkedList()
    {
        first = last = null;
        size = 0;
    }

    public int size()
    {
        return size;
    }

    public boolean isEmpty()
    {
        return size == 0;
    }

    ...
}
```

Konstruktor für leere Liste

Anzahl der Elemente  
= Länge der Liste

leere Liste identifizieren

- ❑ Das Attribut `size` muss beim Hinzufügen zu oder Löschen in der Liste korrigiert werden.  
Dann liefert die Methode `size()` die Anzahl der Elemente.

## Klasse DoublyLinkedList

(Fortsetzung)

```

...
public void add( Object content )
{
    Element e = new Element( content );
    if ( isEmpty() )           ← Liste ist leer:  

    {                         das einzige Element wird ergänzt
        first = last = e;
    }
    else
    {
        last.connectAsSucc( e ); ← das Element wird angehängt
        last = e;               ← last wird korrigiert
    }
    size++;                  ← size wird korrigiert
}
...

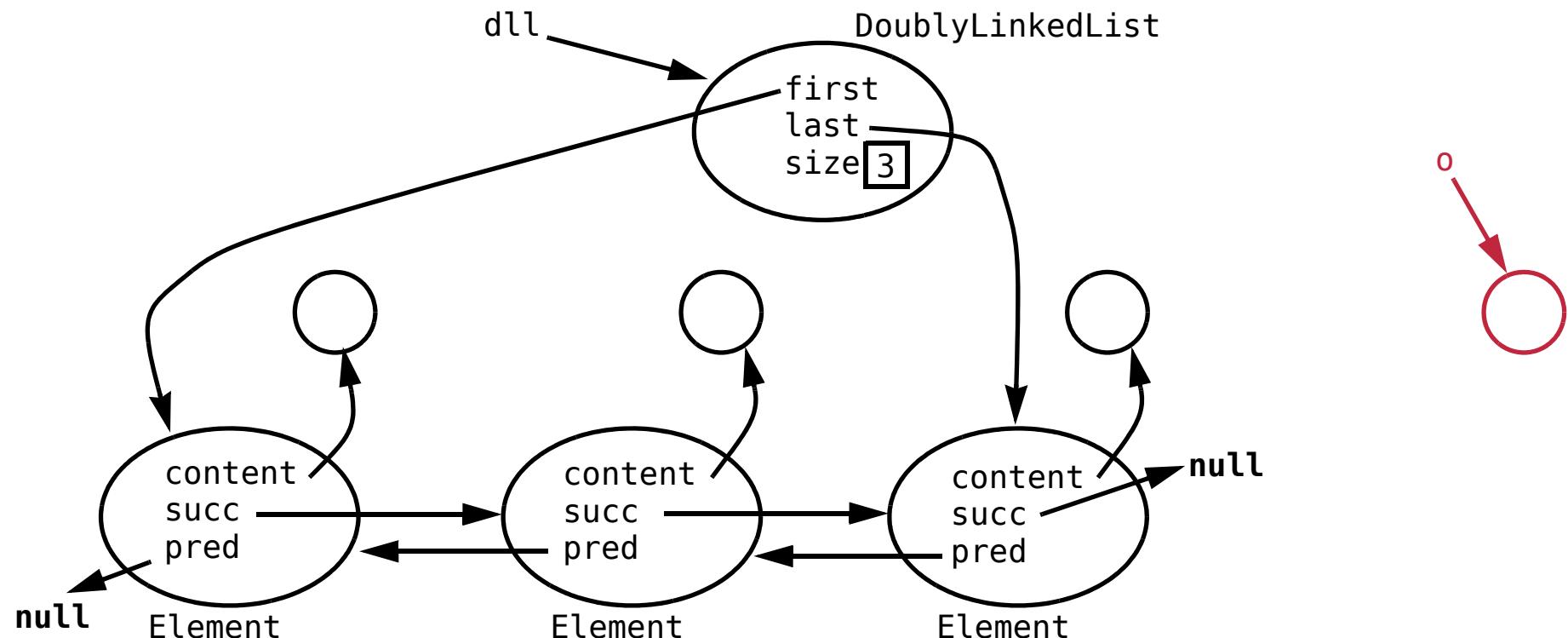
```

- Die Methode `add` ergänzt ein Element am *Ende* der Liste, dessen Inhalt das als Argument übergebene Objekt wird.

## Klasse DoublyLinkedList

(Fortsetzung)

Beispiel für Aufruf: `dll.add( o );`

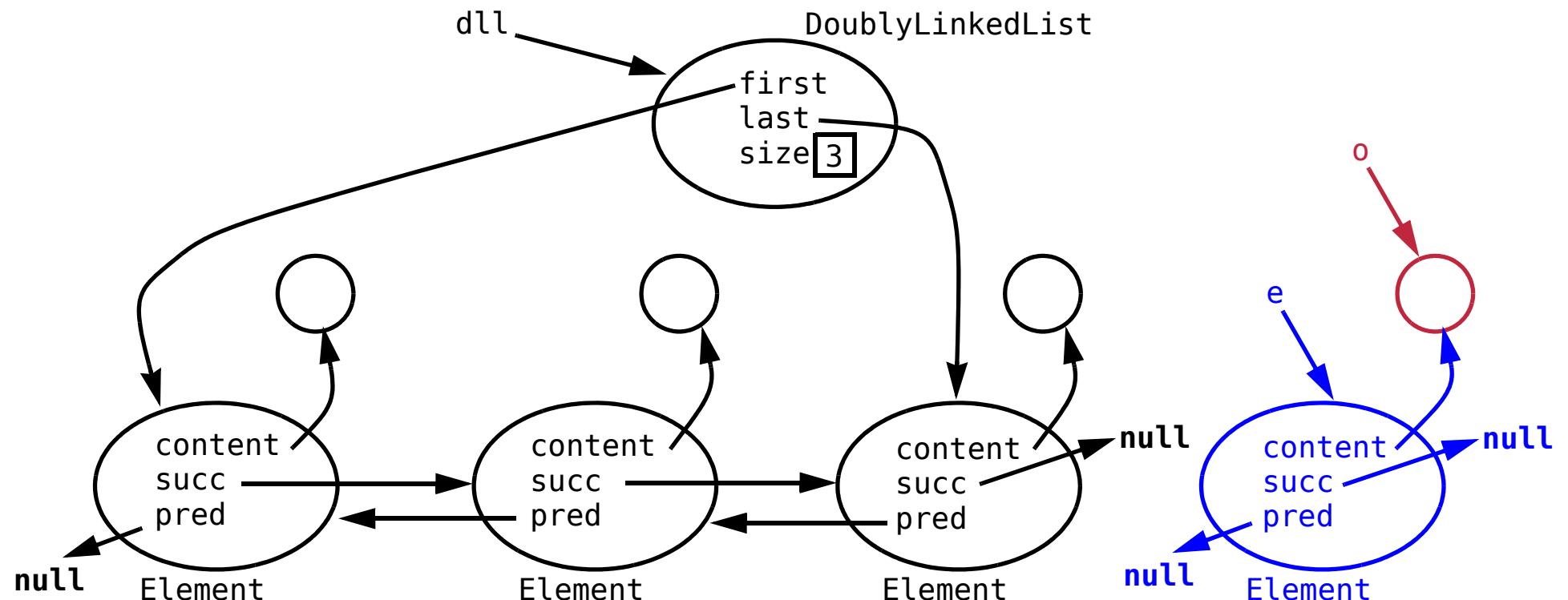


## Klasse DoublyLinkedList

(Fortsetzung)

Beispiel für Aufruf: `dll.add( o );`

`Element e = new Element( content );`

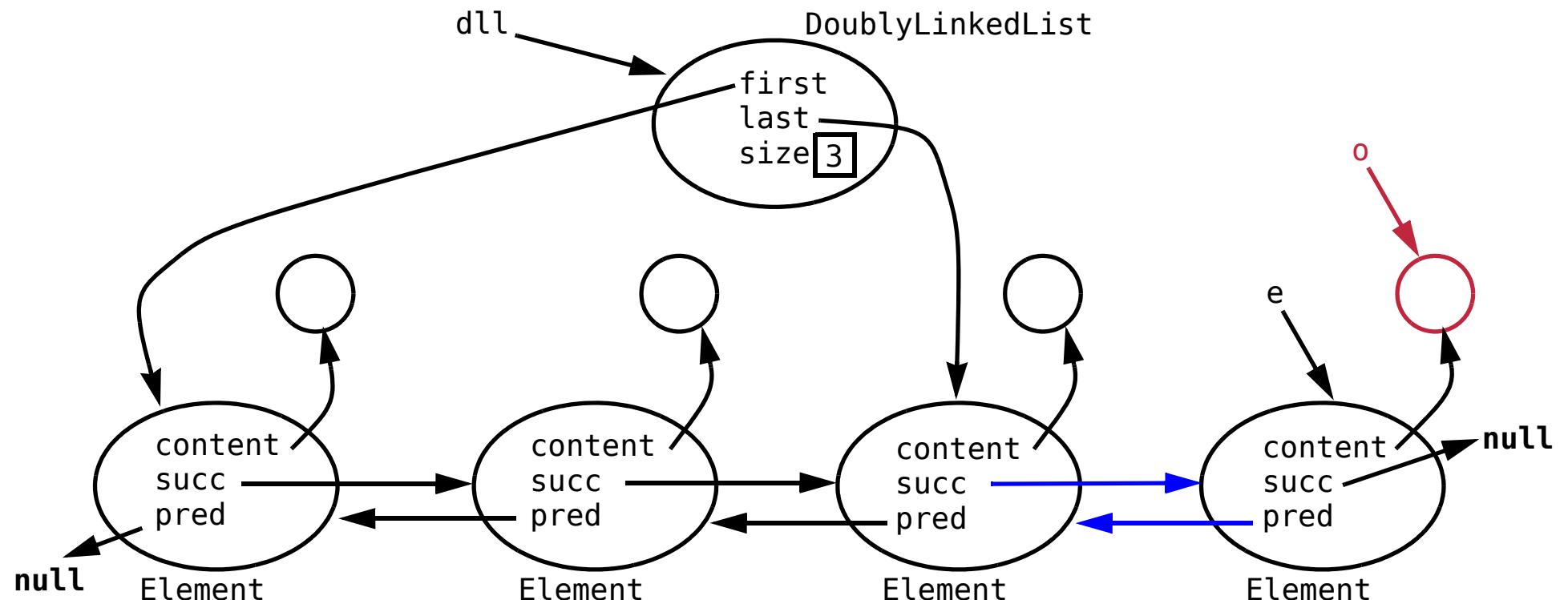


## Klasse DoublyLinkedList

(Fortsetzung)

Beispiel für Aufruf: `dll.add( o );`

`last.connectAsSucc( e );`

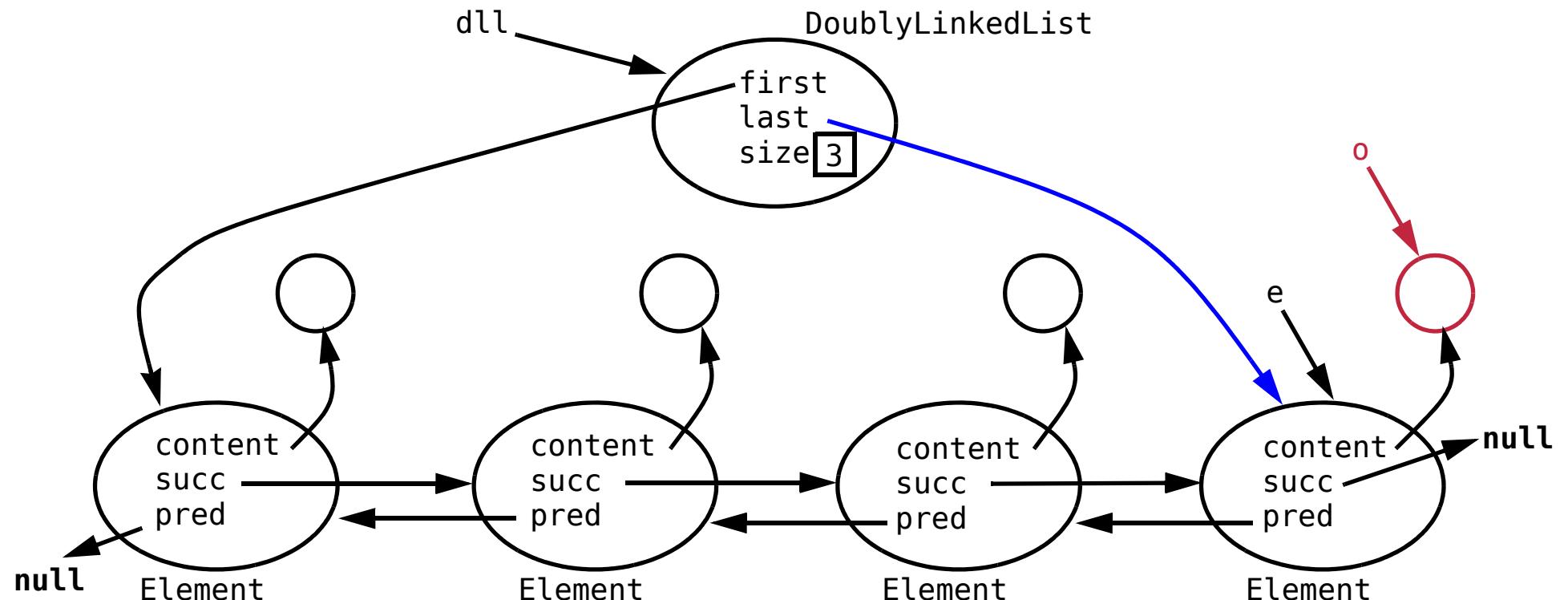


## Klasse DoublyLinkedList

(Fortsetzung)

Beispiel für Aufruf: `dll.add( o );`

`last = e;`

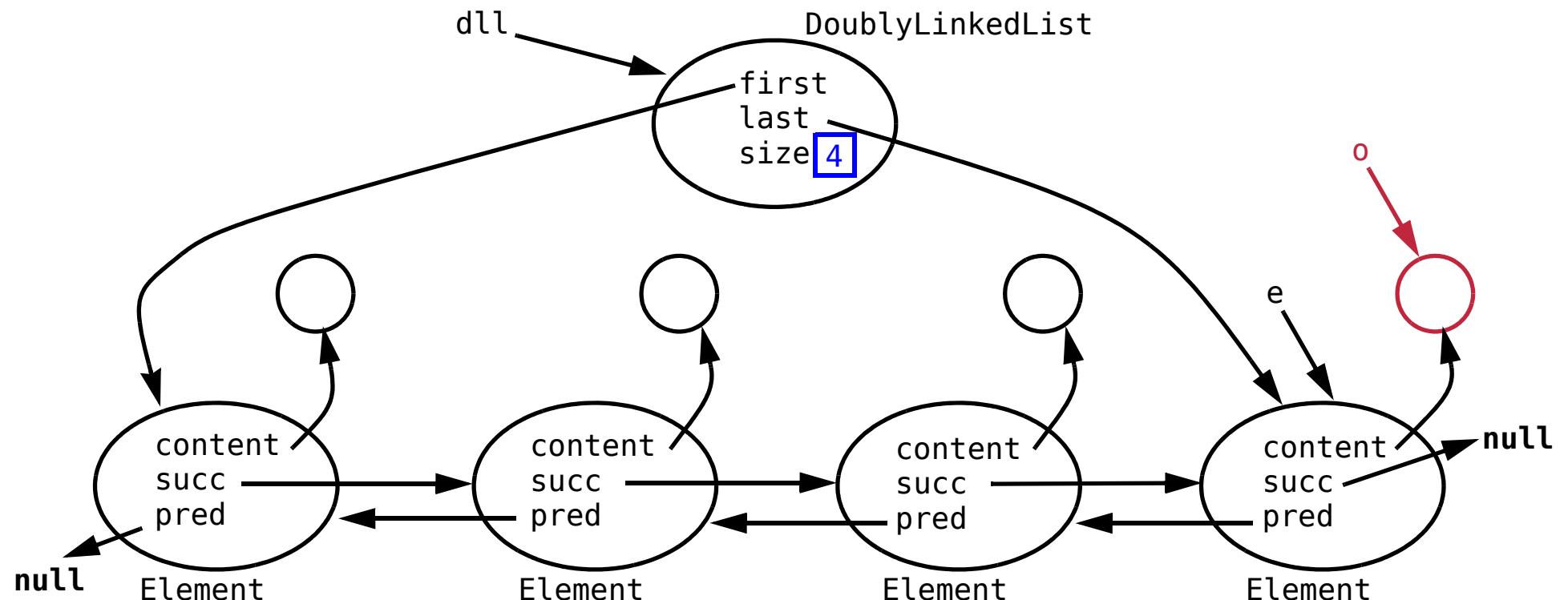


## Klasse DoublyLinkedList

(Fortsetzung)

Beispiel für Aufruf: `dll.add( 0 );`

`size++;`



## Klasse DoublyLinkedList

(Fortsetzung)

```

...
public void addFirst( Object content )
{
    Element e = new Element( content );
    if ( isEmpty() )           ← Liste ist leer:  

    {                         das einzige Element wird ergänzt
        first = last = e;
    }
    else
    {
        first.connectAsPred( e );   ← das Element wird angebunden
        first = e;                ← first wird korrigiert
    }
    size++;                  ← size wird korrigiert
}
...

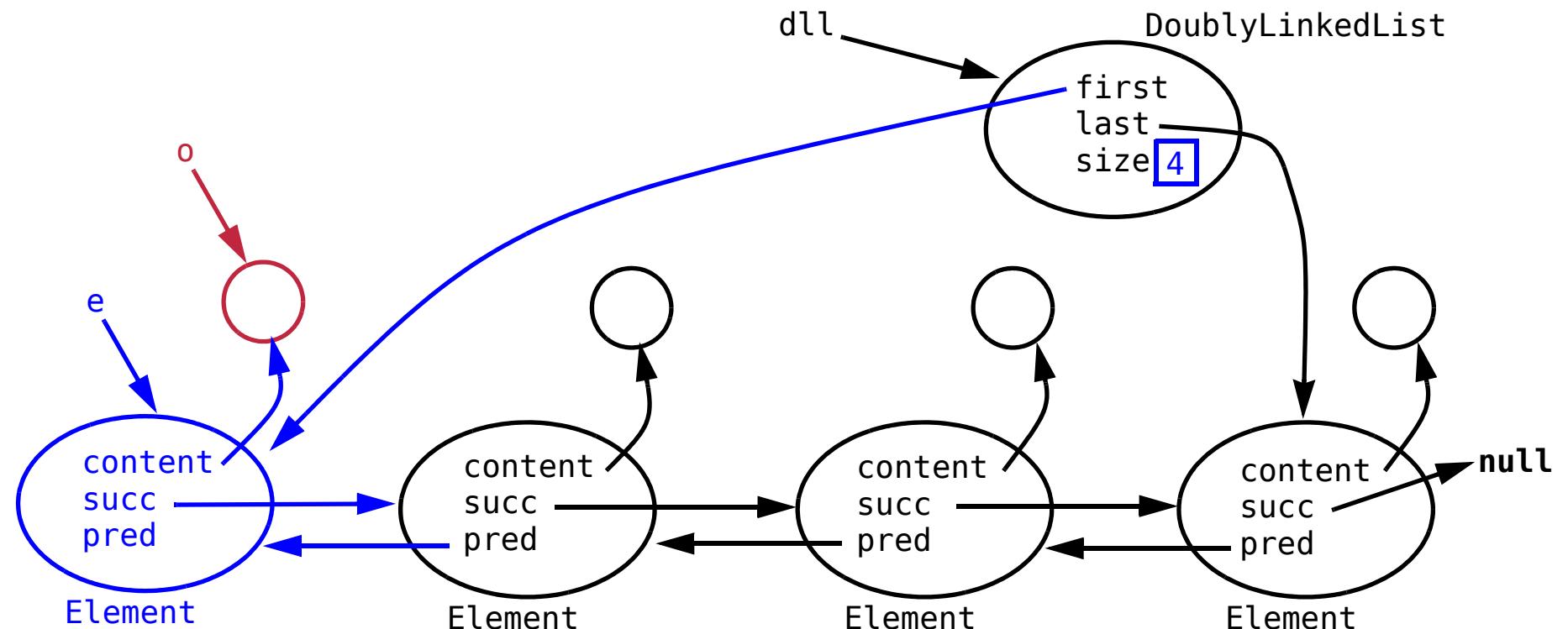
```

- Die Methode `addFirst` ergänzt ein Element am *Anfang* der Liste, dessen Inhalt das als Argument übergebene Objekt wird.

## Klasse DoublyLinkedList

(Fortsetzung)

Beispiel für Aufruf: `dll.addFirst( 0 );`



**Klasse DoublyLinkedList**

(Fortsetzung)

```
...
public Object getFirst()
{
    if ( !isEmpty() )
    {
        return first.getContent();
    }
    else
    {
        throw new IllegalStateException();
    }
}
...
```

Liste ist nicht leer:  
Inhalt des ersten Elements  
wird zurückgegeben

Liste ist leer:  
Abbruch

- Die Methode `getFirst` gibt eine Referenz auf den Inhalt des ersten Elements der Liste zurück, falls die Liste mindestens ein Element enthält.
- Die Liste wird dabei *nicht* verändert.

**Klasse DoublyLinkedList**

(Fortsetzung)

```
...
public Object get( int index )
{
    if ( index >= 0 && index < size )           ← prüft index auf Gültigkeit
    {
        Element current = first;
        for ( int i = 0; i < index; i++ )         ← sucht Element in Liste
        {
            current = current.getsucc();
        }
        return current.getContent();
    }
    else
    {
        throw new IllegalStateException();
    }
}
...
```

- ❑ Jeder Aufruf von `get` führt die Suche vom *Anfang* der Liste aus durch.

## Klasse DoublyLinkedList

(Fortsetzung)

```
public Object removeFirst()
{
    if ( !isEmpty() )
    {
        Object result = first.getContent();
        if ( first.hasSucc() )
        {
            first = first.getSucc();
            first.disconnectPred();
        } else {
            first = last = null;
        }
        size--;
        return result;
    } else {
        throw new IllegalStateException();
    }
}
```

Liste ist nicht leer:

Inhalt des ersten Elements merken

first korrigieren = Löschen

Liste ist nach Löschen nicht leer:  
neues erstes Element korrigieren

Liste ist nach Löschen leer:  
last korrigieren

size korrigieren

- Die Methode `removeFirst` gibt eine Referenz auf den Inhalt des ersten Elements zurück und löscht das Objekt aus der Liste.

## Klasse DoublyLinkedList

(Fortsetzung)

```
...
public void showAll()
{
    Element current = first;
    while ( current != null )
    {
        System.out.print( current.getContent().toString() );
        if ( current != last )
        {
            System.out.print( ", " );
        }
        current = current.getsucc();
    }
    System.out.println();
}
```

toString ist für jedes Objekt  
verfügbar



- Die Methode `showAll` gibt die Inhalte aller Element der Liste aus.

## Klasse DoublyLinkedList – Beispiel

(Fortsetzung)

```
DoublyLinkedList students = new DoublyLinkedList();      eine Liste mit Student-Objekten
students.add( new Student( "A", "Inf", 123433 ) );
students.add( new Student( "B", "Inf", 123456 ) );
students.add( new Student( "C", "Inf", 123457 ) );
students.add( new Student( "D", "Inf", 123458 ) );
students.showAll();
students.addFirst( new Student( "X", "Inf", 123461 ) );
students.showAll();
System.out.println( students.get(2).toString() );
System.out.println( students.removeFirst().toString() );
System.out.println( students.getFirst().toString() );      toString wird in Student
                                                               überschrieben
```

Ausgabe:

```
student: A, registration number: 123433(Inf), student: B, registration number: ...
student: X, registration number: 123461(Inf), student: A, registration number: ...
student: B, registration number: 123456(Inf)
student: X, registration number: 123461(Inf)
student: A, registration number: 123433(Inf)
```

## Klasse DoublyLinkedList – Beispiel

(Fortsetzung)

```
DoublyLinkedList numbers = new DoublyLinkedList();      eine Liste mit Fraction-Objekten
numbers.add( new Fraction( 2, 3 ) );
numbers.add( new Fraction( 1, 7 ) );
numbers.add( new Fraction( 3, 12 ) );
numbers.add( new Fraction( 8 ) );
numbers.showAll();
System.out.println( numbers.removeFirst().toString() );
numbers.showAll();
System.out.println( numbers.getFirst().toString() );    toString wird in Fraction
                                                        überschrieben
```

Ausgabe:

```
2 / 3, 1 / 7, 1 / 4, 8 / 1
2 / 3
1 / 7, 1 / 4, 8 / 1
1 / 7
```

## Klasse DoublyLinkedList – Beispiel

(Fortsetzung)

```
DoublyLinkedList list = new DoublyLinkedList();
list.add( new Student( "A", "Inf", 123433 ) );
list.add( new Fraction( 3, 7 ) );
list.showAll();
```

eine Liste

toString wird in Student und  
Fraction überschrieben

Ausgabe:

```
student: A, registration number: 123433(Inf), 3 / 7
```

- ❑ Da die Referenz content auf beliebige Objekte verweisen kann, können in jeder Liste Objekte verschiedener Klassen eingefügt werden.
- ❑ Eine Liste der Klasse DoublyLinkedList besitzt also *keinen* einheitlichen Grundtyp.

## Klasse DoublyLinkedList – Beispiel

(Fortsetzung)

Die Methode `get` erlaubt es, eine Ausgabe analog zu der der Methode `showAll` auch außerhalb der Klasse `DoublyLinkedList` durch eine Schleife zu erzeugen:

```
for ( int i = 0; i < students.size(); i++ )
{
    System.out.print( students.get(i) );
    if ( i != students.size()-1 )
    {
        System.out.print( ", " );
    }
}
```

- Da jeder Aufruf von `get` die Liste von vorne durchläuft, ist diese Form sehr ineffizient.  
Die Ausgabe mit `showAll` ist sehr viel effizienter.

## Klasse DoublyLinkedList

(Fortsetzung)

### Zusammenfassung der Experimente

Die Datenstruktur `DoublyLinkedList` lässt sich universell verwenden:

- ❑ Eine Liste von `Student`-Objekten konnte ohne Änderungen an den Klassen `Student` oder `DoublyLinkedList` angelegt werden.
- ❑ Eine Liste von `Fraction`-Objekten konnte ohne Änderungen an den Klassen `Fraction` oder `DoublyLinkedList` angelegt werden.
- ❑ Die Verfügbarkeit der Methode `toString` ermöglicht es, eine Methode `showAll` zur Ausgabe in der Klasse `DoublyLinkedList` ohne Kenntnis der tatsächlich abgelegten Objekte zu implementieren.
  
- ❑ aber: Lässt sich auch eine Liste von `int`-Werten anlegen?  
`int` ist *keine* Klasse, Referenzen auf `Object` können nicht auf einfache Typen wie `int`, `double`, `boolean`, `char`, ... verweisen.

## Wrapper-Klassen

### Klasse Integer

Java stellt Klassen (*Wrapper*) bereit, die die einfachen Typen in einem Objekt *einpacken*, das jeweils nichts anderes enthält als genau einen Wert eines einfachen Typs:

- Double (einfacher Typ: **double**)
- Integer (einfacher Typ: **int**)
- Float (einfacher Typ: **float**)
- Long (einfacher Typ: **long**)
- deren gemeinsame Oberklasse Number
  
- Boolean (einfacher Typ: **boolean**)
- Character (einfacher Typ: **char**)

Als Beispiel wird der Einsatz der Klasse Integer gezeigt.

## Wrapper-Klassen

(Fortsetzung)

### Klasse Integer

```
public Integer( int value )
```

**Konstruktor**

```
public int compareTo( Integer anotherInteger )
```

```
public boolean equals( Object obj )
```

```
public int intValue()
```

**liefert Wert als int**

```
public double doubleValue()
```

**liefert Wert als double**

```
public String toString()
```

- ❑ Mit der Klasse können Objekte angelegt werden, die einen **int**-Wert enthalten:
- ❑ Die Klasse besitzt **keine** Methode zum Ändern des Werts eines Objekts!

```
DoublyLinkedList ints = new DoublyLinkedList();
ints.add( new Integer( 4 ) );
ints.add( new Integer( 3 ) );
```

## Wrapper-Klassen

(Fortsetzung)

### Klasse Integer

Da die Klasse Integer fest in Java integriert ist, sind zusätzlich möglich:

- ❑ *Boxing*  
ist die Zuweisung eines **int**-Wertes an eine Integer-Referenz.  
Es wird *implizit* der Konstruktor aufgerufen. («Der **int**-Wert wird einpackt.»)

`Integer ref = 5;`      entspricht      `Integer ref = new Integer( 5 ) ;`

- ❑ *Unboxing*  
ist die Zuweisung einer Integer-Referenz an eine **int**-Variable.  
Es wird *implizit* die **intValue**-Methode aufgerufen. («Der **int**-Wert wird ausgepackt.»)

`int i = ref;`      entspricht      `int i = ref.intValue();`

- ❑ Der Gebrauch von Objekten/Werten der Typen Integer und int kann also einfach gemischt werden.

## Wrapper-Klassen

### Klasse Integer

(Fortsetzung)

Durch die Kompatibilität von `Integer` und `int` ist auch folgender Quelltext möglich:

```
DoublyLinkedList ints = new DoublyLinkedList();
ints.add( 5 );
ints.add( 6 );
```

- Der Parameter der Methode `add` hat den (deklarierten) Typ `Object`.  
Java erkennt, dass Boxing des `int`-Wertes ein zu `Object` kompatibles `Integer`-Objekt erzeugen würde und verpackt daher den `int`-Wert in ein Objekt der Klasse `Integer` und übergibt eine Referenz auf dieses Objekt als Argument.
- Das Programm ist daher compilierbar und ausführbar.
- Die universell deklarierte Liste kann also auch mit einfachen Typen verwendet werden.

## Abfragen von Objekten aus der Liste

Experiment: Lesen von Werten aus der Liste

```
DoublyLinkedList ints = new DoublyLinkedList();
ints.add( new Integer( 5 ) );
ints.add( new Integer( 6 ) );
Integer i = ints.get( 1 );
```

Compiler meldet: incompatible types

- Erinnerung:  
Der Compiler prüft den Programmtext *statisch* und führt diesen nicht aus:  
Die Methode `get` hat als Typ für die Rückgabe `Object` deklariert,  
die Klasse `Object` ist aber Oberklasse von `Integer`, so dass der Compiler nicht garantieren kann, dass das zurückgegebene Objekt tatsächlich zu `Integer` kompatibel ist.

- Ebenso scheitert:

```
int i = ints.get( 1 );
```

`Object` ist grundsätzlich nicht kompatibel zum Typ `int`.  
Der Compiler könnte nur dann ein Unboxing vorsehen, wenn er sicher wäre, dass `ints.get( 1 )` ein `Integer`-Objekt liefern würde.

## Abfragen von Objekten aus der Liste

(Fortsetzung)

### *Lösung des Problems (1)*

Der Compiler akzeptiert nur eine einzige Zuweisung:

```
Object ref = ints.get( 1 );
```

- ❑ Dabei sind der Typ der Referenz und der Typ der Rückgabe gemäß der Deklaration der Methode `get` identisch.
- ❑ Bei der *Ausführung* wird dann in der Methode `get` das zweite in der Liste abgelegte Integer-Objekt ermittelt und eine Referenz auf dieses Objekt zurückgegeben.
- ❑ Die Zuweisung an eine `Object`-Referenz ist möglich, da `Object` als Oberklasse auf Objekte einer seiner Unterklassen – hier also `Integer` – verweisen kann.
- ❑ Allerdings können über die Referenz `ref` *nur solche* Methoden aufgerufen werden, die in der Klasse `Object` deklariert sind. Insbesondere kann nicht auf den im `Integer`-Objekt abgelegten `int`-Wert zugegriffen werden.
- ❑ Diese Lösung hilft also nicht recht weiter.

## Abfragen von Objekten aus der Liste

(Fortsetzung)

### *Lösung des Problems (2)*

Der Compiler *vertraut* aber dem Entwickler. Dieser kann durch eine explizite Typangabe (*Type-Cast*) versichern, dass das zugewiesene Objekt kompatibel zum Typ `Integer` ist. \*)

```
Integer ref = (Integer)ints.get( 1 );
```

- ❑ Der Compiler kann die Richtigkeit der durch den Type-Cast geschaffenen Typkompatibilität *nicht* überprüfen.
- ❑ Liefert die Methode `get` bei der *Ausführung* kein zu `Integer` kompatibles Objekt, tritt ein Laufzeitfehler auf: `ClassCastException`
- ❑ Allerdings können auch bei einem Type-Cast die Vorteile des Boxing und Unboxing für die vordefinierten Wrapper-Klassen genutzt werden. Möglich sind:

```
int i = (Integer)ints.get( 1 );
int k = (int)ints.get( 1 );
Integer ref = (int)ints.get( 1 );
```

\*) Der Type-Cast für Objekte erzeugt kein neues Objekt, sondern gibt nur einen Hinweis zur Typkompatibilität zur Laufzeit.

## Klasse Element – abschließender Überblick

```
public class Element
{
    private Object content;
    private Element pred, succ;

    public Element( Object c ) { ... }

    public void setContent( Object c ) { ... }
    public boolean hasSucc() { ... }
    public Element getSucc() { ... }
    public void connectAsSucc( Element e) { ... }
    public void disconnectSucc() { ... }
    public boolean hasPred() { ... }
    public Element getPred() { ... }
    public void connectAsPred( Element e ) { ... }
    public void disconnectPred() { ... }

}
```

Attribute

Konstruktor

öffentliche Methoden

## Klasse DoublyLinkedList – abschließender Überblick

```
public class DoublyLinkedList
{
    private Element first, last;
    private int size;

    public DoublyLinkedList() { ... }

    public int size() { ... }
    public boolean isEmpty() { ... }
    public void add( Object content ) { ... }
    public Object getFirst() { ... }
    public Object get( int index ) { ... }
    public Object removeFirst() { ... }
    public void showAll() { ... }

}
```

Attribute

Konstruktor

öffentliche Methoden

## nicht vorhersehbare Operationen auf der Liste

- Die Klasse `DoublyLinkedList` enthält *nur* Methoden, die für *jeden* Inhalt ausführbar sind. In `DoublyLinkedList` können keine Annahmen über die in den Elementen gespeicherten Inhalte getroffen werden.
- Schon eine einfache Aufgabenstellung wie das Berechnen der Summe aller in einer Liste von `Integer`-Objekten abgelegten Werte muss über das sehr ineffiziente vielfache Aufrufen der `get`-Methode erfolgen, die bei jedem Aufruf eine Suche vom Anfang der Liste aus beginnt.
- Auch das Verdoppeln aller in der Liste abgelegten Werte ist nur umständlich zu lösen. Da die Klasse `Integer` *keine* Methode zum Ändern ihres `int`-Wertes anbietet, reicht es nicht, die `Integer`-Objekte mit der `get`-Methode auszulesen. Zusätzlich müssten Werte ausgetauscht oder die Liste ab- und wieder aufgebaut werden.

### Lösungsansatz:

Es muss eine Möglichkeit geschaffen werden, die Liste *mit Unterbrechungen* – in denen weitere Methoden aufgerufen werden können – *zu durchlaufen*, ohne bei der Fortsetzung des Durchlaufs jeweils wieder mit dem Anfang der Liste beginnen zu müssen.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 10. Entwurfsmuster – Iterator

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-768

## Lernziele des Kapitels 10. Entwurfsmuster Iterator

Nach Durcharbeiten des Kapitels Entwurfsmuster Iterator sollen die teilnehmenden Studierenden

- Pakete anlegen und nutzen können,
- abstrakte Klassen kennen und einsetzen können,
- innere Klassen kennen und einsetzen können,
- das Entwurfsmuster Iterator kennen,
- die Implementierung mit den Klassen `ListIterator`, `ForwardIterator` und `ReverseIterator` verstehen und einsetzen können,
- das Entwurfsmuster Iterator in anderen Kontexten einsetzen können.

## Wiederholung (Folie 767): nicht vorhersehbare Operationen auf der Liste

- Die Klasse `DoublyLinkedList` enthält *nur* Methoden, die für *jeden* Inhalt ausführbar sind. In `DoublyLinkedList` können keine Annahmen über die in den Elementen gespeicherten Inhalte getroffen werden.
- Schon eine einfache Aufgabenstellung wie das Berechnen der Summe aller in einer Liste von `Integer`-Objekten abgelegten Werte muss über das sehr ineffiziente vielfache Aufrufen der `get`-Methode erfolgen, die bei jedem Aufruf eine Suche vom Anfang der Liste aus beginnt.
- Auch das Verdoppeln aller in der Liste abgelegten Werte ist nur umständlich zu lösen. Da die Klasse `Integer` *keine* Methode zum Ändern ihres `int`-Wertes anbietet, reicht es nicht, die `Integer`-Objekte mit der `get`-Methode auszulesen. Zusätzlich müssten Werte ausgetauscht oder die Liste ab- und wieder aufgebaut werden.

Warum wird die Klasse `DoublyLinkedList` nicht für jeden Spezialfall ergänzt?

## Wiederholung (Folie 767): nicht vorhersehbare Operationen auf der Liste (Fortsetzung)

- Die Klasse `DoublyLinkedList` enthält *nur* Methoden, die für *jeden* Inhalt ausführbar sind. In `DoublyLinkedList` können keine Annahmen über die in den Elementen gespeicherten Inhalte getroffen werden.
- Schon eine einfache Aufgabenstellung wie das Berechnen der Summe aller in einer Liste von `Integer`-Objekten abgelegten Werte muss über das sehr ineffiziente vielfache Aufrufen der `get`-Methode erfolgen, die bei jedem Aufruf eine Suche vom Anfang der Liste aus beginnt.
- Auch das Verdoppeln aller in der Liste abgelegten Werte ist nur umständlich zu lösen. Da die Klasse `Integer` *keine* Methode zum Ändern ihres `int`-Wertes anbietet, reicht es nicht, die `Integer`-Objekte mit der `get`-Methode auszulesen. Zusätzlich müssten Werte ausgetauscht oder die Liste ab- und wieder aufgebaut werden.

Warum wird die Klasse `DoublyLinkedList` nicht für jeden Spezialfall ergänzt?

- Für jede Ergänzung muss der Programmtext von `DoublyLinkedList` verstanden werden.
- Nach jeder Ergänzung muss der Programmtext von `DoublyLinkedList` erneut getestet werden.
- Nach jeder Ergänzung müssen andere Klassen, die `DoublyLinkedList` benutzen, erneut getestet werden.

## Wiederholung (Folie 767): nicht vorhersehbare Operationen auf der Liste (Fortsetzung)

- Die Klasse `DoublyLinkedList` enthält *nur* Methoden, die für *jeden* Inhalt ausführbar sind. In `DoublyLinkedList` können keine Annahmen über die in den Elementen gespeicherten Inhalte getroffen werden.
- Schon eine einfache Aufgabenstellung wie das Berechnen der Summe aller in einer Liste von `Integer`-Objekten abgelegten Werte muss über das sehr ineffiziente vielfache Aufrufen der `get`-Methode erfolgen, die bei jedem Aufruf eine Suche vom Anfang der Liste aus beginnt.
- Auch das Verdoppeln aller in der Liste abgelegten Werte ist nur umständlich zu lösen. Da die Klasse `Integer` *keine* Methode zum Ändern ihres `int`-Wertes anbietet, reicht es nicht, die `Integer`-Objekte mit der `get`-Methode auszulesen. Zusätzlich müssten Werte ausgetauscht oder die Liste ab- und wieder aufgebaut werden.

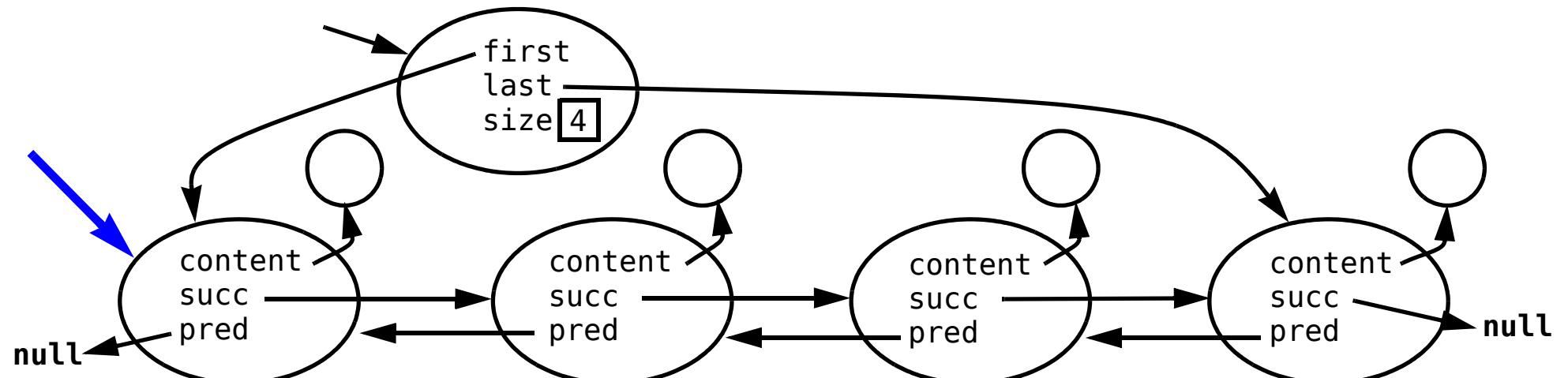
*Lösungsansatz*, um ohne Ergänzung der Klasse `DoublyLinkedList` flexibel zu sein:

Es muss eine Möglichkeit geschaffen werden, die Liste *mit Unterbrechungen* – in denen weitere Methoden aufgerufen werden können – *zu durchlaufen*, ohne bei der Fortsetzung des Durchlaufs jeweils wieder mit dem Anfang der Liste beginnen zu müssen.

## Entwurfsmuster Iterator

Lösungsidee – Konkretisierung:

- ❑ Eine zusätzliche *Referenz* auf Objekte der Klasse Element wird angelegt.
- ❑ Wenn ein Durchlauf beginnt, wird diese Referenz auf das erste Element der Liste gesetzt.
- ❑ Wenn ein Inhalt abgerufen wird, wird er von dem Element bezogen, auf das die Referenz verweist. Gleichzeitig wird die Referenz auf das nachfolgende Element gesetzt.
- ❑ Der letzte Schritt kann nun wiederholt und die Liste so elementweise durchlaufen werden.
- ❑ Voraussetzung ist, dass
  - die Referenz dauerhaft erhalten bleibt und
  - erkannt werden kann, wann ein ein neuer Durchlauf beginnt.

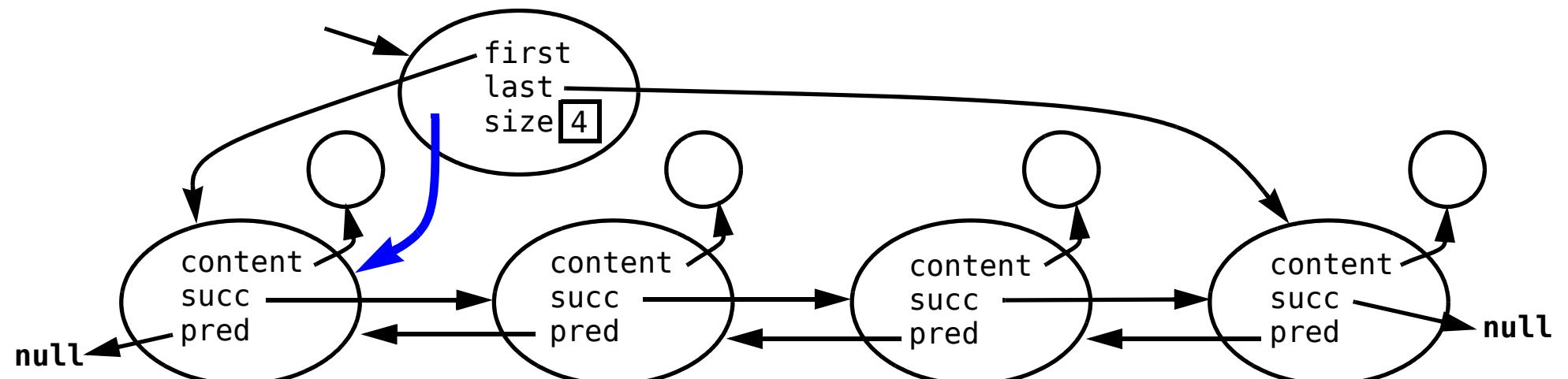


## Entwurfsmuster Iterator

(Fortsetzung)

Wie und wo wird die zusätzliche Referenz deklariert?

- **Vorschlag 1:** als Attribut in der Klasse DoublyLinkedList
  - Dort ist das private Attribut `first` bekannt, das für den Start des Durchlaufs benötigt wird.
  - Methoden übernehmen das Initialisieren und Weitersetzen.

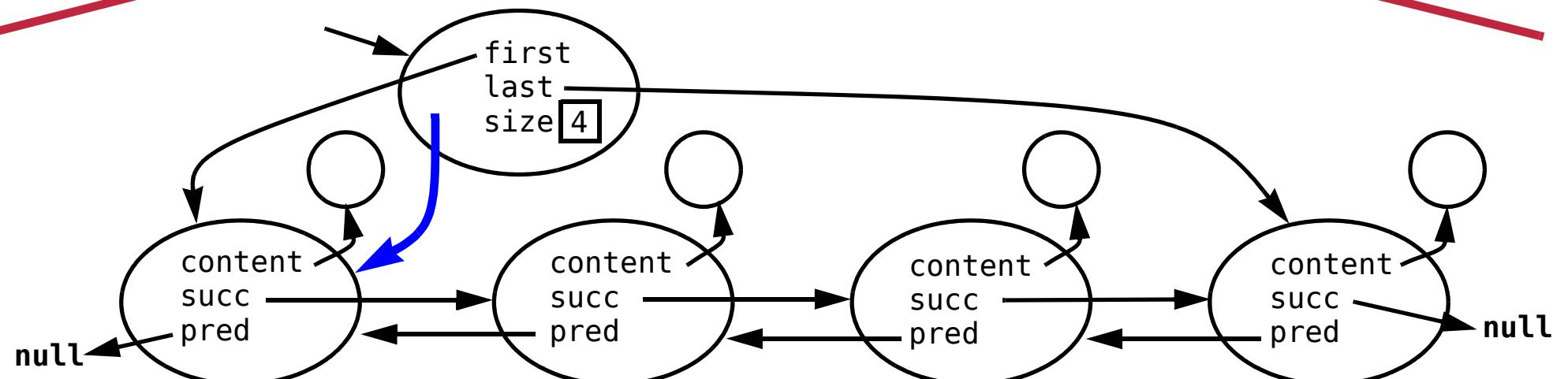


## Entwurfsmuster Iterator

(Fortsetzung)

Wie und wo wird die zusätzliche Referenz deklariert?

- Vorschlag 1:** als Attribut in der Klasse DoublyLinkedList
  - Dort ist das private Attribut `first` bekannt, das für den Start des Durchlaufs benötigt wird.
  - Methoden übernehmen das Initialisieren und Weitersetzen.
  - *aber:* Es kann immer nur *genau ein* Durchlauf zu einem Zeitpunkt erfolgen, da bei dieser Lösung für jede Liste auch nur eine Referenz zur Verfügung steht.  
Bei großen Softwaresystemen kann diese Randbedingung aber nur schwer überprüft und sichergestellt werden.



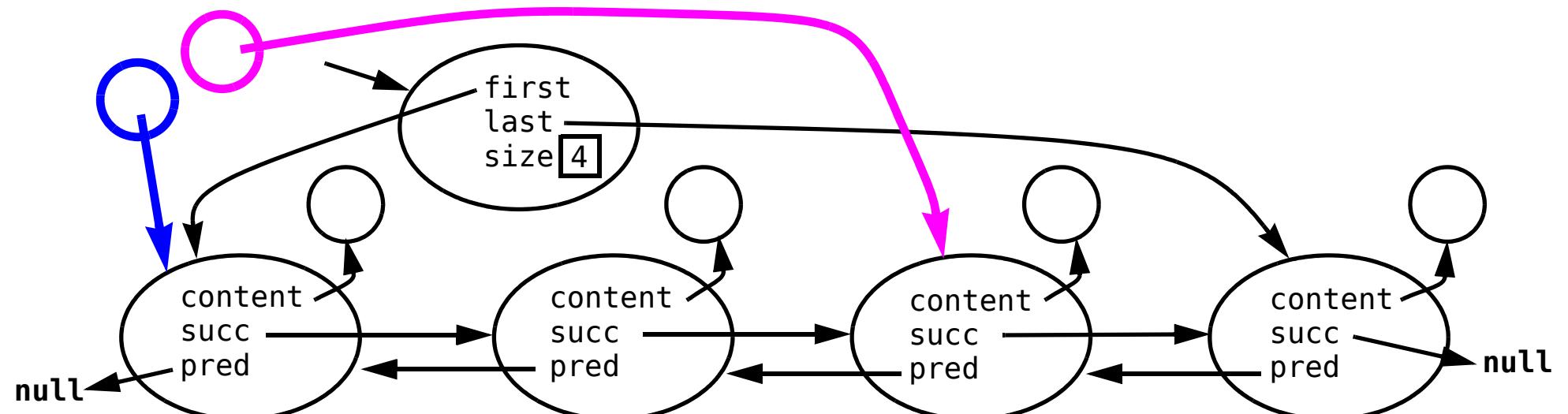
## Entwurfsmuster Iterator

(Fortsetzung)

Wie und wo wird die zusätzliche Referenz deklariert?

□ *Vorschlag 2:* als Attribut einer speziellen Klasse

- Objekte dieser Klasse werden erzeugt, um genau *einen* Durchlauf zu unterstützen.
- Sollen an verschiedenen Stellen in einem Softwaresystem Durchläufe durch die gleiche Liste erfolgen, so wird jeweils ein eigenes Objekt erzeugt.



## Entwurfsmuster Iterator

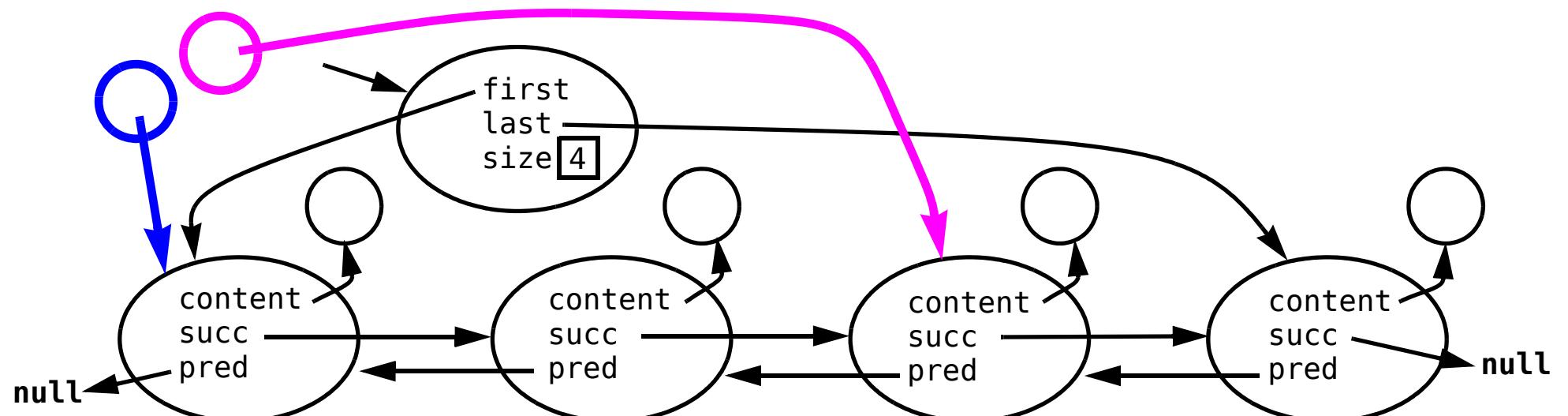
(Fortsetzung)

Wie und wo wird die zusätzliche Referenz deklariert?

□ **Vorschlag 2:** als Attribut einer speziellen Klasse

- Objekte dieser Klasse werden erzeugt, um genau *einen* Durchlauf zu unterstützen.
- Sollen an verschiedenen Stellen in einem Softwaresystem Durchläufe durch die gleiche Liste erfolgen, so wird jeweils ein eigenes Objekt erzeugt.
- *Wie kann auf das private Attribut first zugegriffen werden?*

Die Klasse DoublyLinkedList übernimmt in einer Methode das Initialisieren des Objekts.



## Entwurfsmuster Iterator

(Fortsetzung)

Entwurf der Klasse `ForwardIterator`:

- ❑ Es wird ein Attribut `current` als Referenz auf die Klasse `Element` deklariert, in dem die Position für einen Durchlauf gemerkt wird.
  - ❑ Es wird ein Konstruktor deklariert, der die Referenz `current` initialisiert.
  - ❑ Es wird eine Methode `next()` deklariert, die den Inhalt eines Elements liefert und die Referenz `current` zum nächsten Element bewegt.
  - ❑ Es wird eine Methode `hasNext()` deklariert, die anzeigt, ob es ein weiteres, noch nicht besuchtes Element gibt.
- 
- ❑ Die Klasse `ForwardIterator` ist also sehr einfach gestaltet.

## Entwurfsmuster Iterator

(Fortsetzung)

Implementierung der Klasse ForwardIterator

```
public class ForwardIterator
{
    private Element current; ← Referenz, die die Position merkt

    public ForwardIterator( Element elem )
    {
        current = elem;
    }

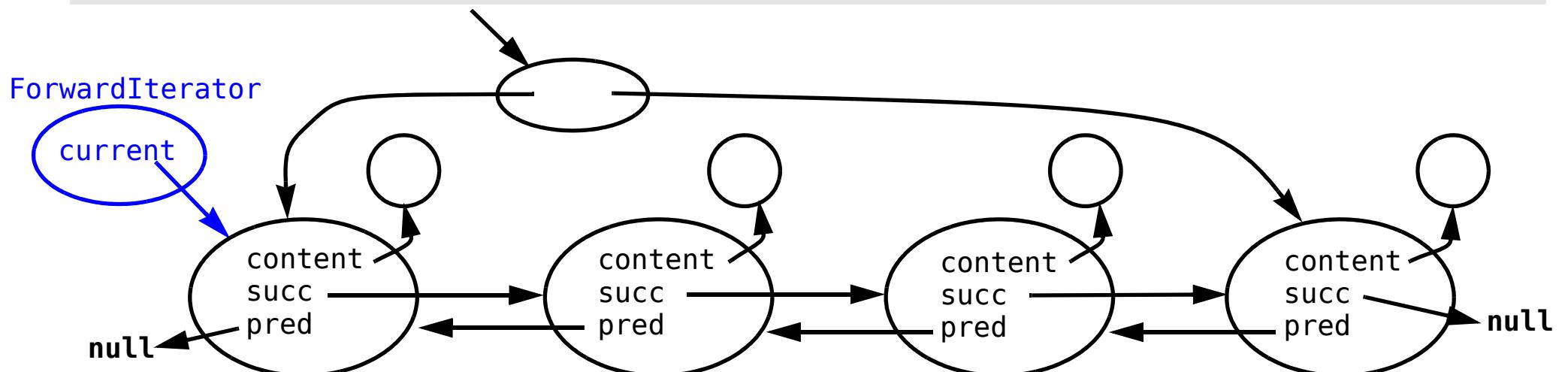
    public boolean hasNext()
    {
        return current != null; ← next verweist auf das Objekt,
    }                                das vom nächsten Aufruf von
    ...                                next() geliefert wird
```

## Entwurfsmuster Iterator

(Fortsetzung)

Implementierung der Klasse ForwardIterator

```
public Object next()
{
    if ( hasNext() ) {
        Object content = current.getContent();
        current = current.getsucc();
        return content;
    } else {
        throw new IllegalStateException();
    }
}
```

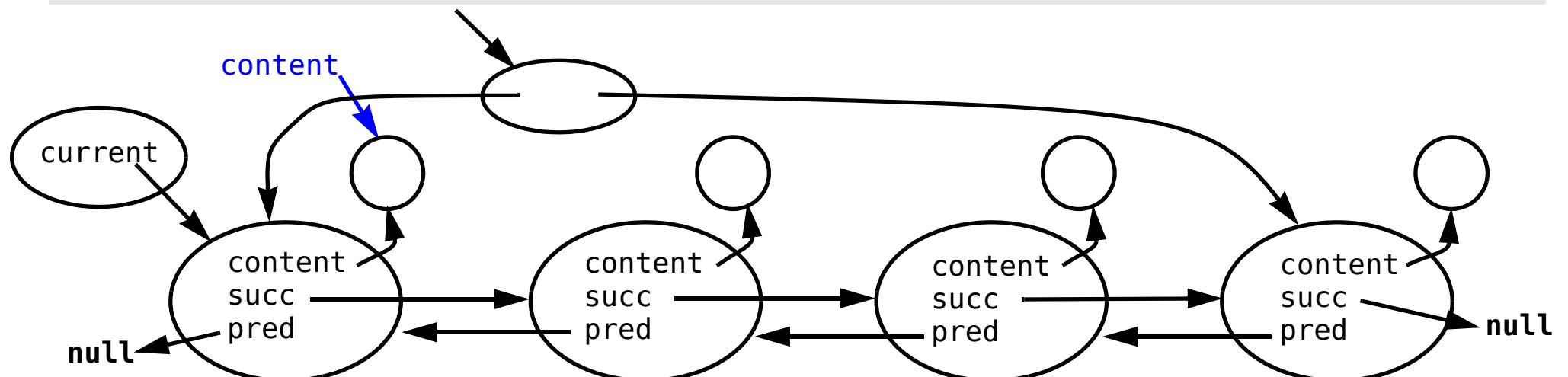


## Entwurfsmuster Iterator

(Fortsetzung)

Implementierung der Klasse ForwardIterator

```
public Object next()
{
    if ( hasNext() ) {
        Object content = current.getContent();
        current = current.getsucc();
        return content;
    } else {
        throw new IllegalStateException();
    }
}
```

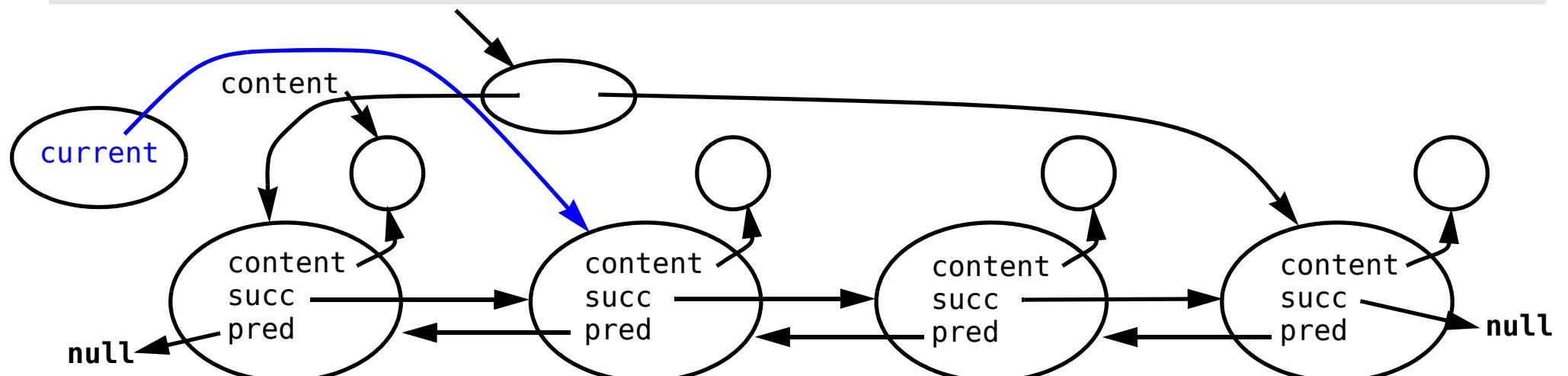


## Entwurfsmuster Iterator

(Fortsetzung)

Implementierung der Klasse ForwardIterator

```
public Object next()
{
    if ( hasNext() ) {
        Object content = current.getContent();
        current = current.getsucc();
        return content;
    } else {
        throw new IllegalStateException();
    }
}
```



## Entwurfsmuster Iterator

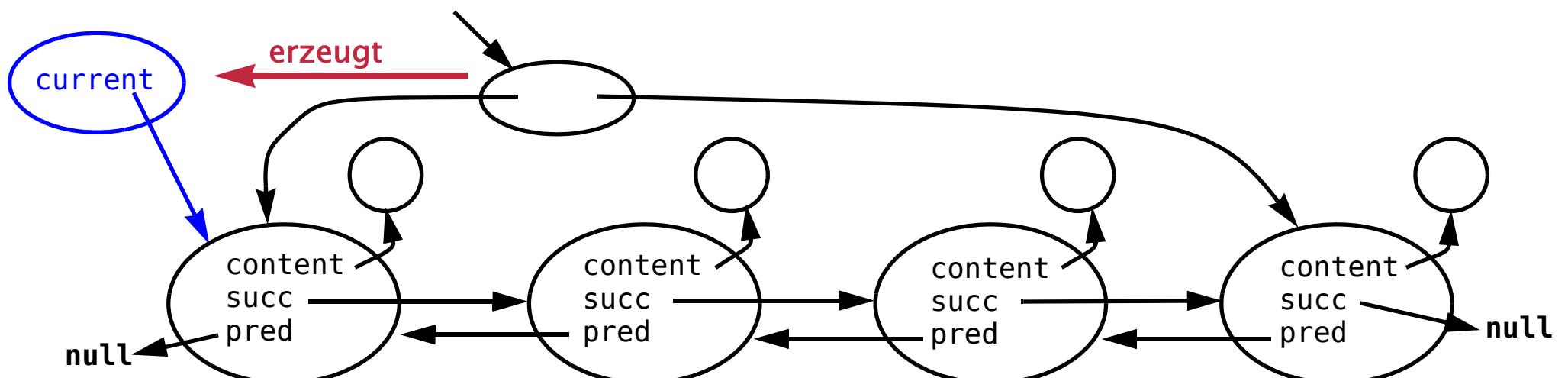
(Fortsetzung)

Ergänzung *in* der Klasse DoublyLinkedList

```
public ForwardIterator iterator()
{
    return new ForwardIterator( first );
}
```

Durchlauf beginnt bei *first*

- Die Methode `iterator()` gibt ein Objekt der Klasse `ForwardIterator` zurück, dessen Attribut `current` auf das erste Element der Liste verweist.  
*Innerhalb* von `DoublyLinkedList` kann auf das private Attribut `first` zugegriffen werden!



## Entwurfsmuster Iterator

(Fortsetzung)

Beispiel für den Einsatz eines Iterators

- Nun kann die Summe der Werte einer Liste von `Integer`-Werten in einer einfachen Schleife berechnet werden.

```
DoublyLinkedList ints = new DoublyLinkedList();
...
int sum = 0;
ForwardIterator intsIterator = ints.iterator();
while ( intsIterator.hasNext() )
{
    sum += (int)intsIterator.next();
}
```

## Entwurfsmuster Iterator

(Fortsetzung)

weitere Beispiele für den Einsatz eines Iterators

- Summe der Quadrate aller Werte von `ints`:

```
int sum = 0;
int value = 0;
ForwardIterator intsIterator = ints.iterator();
while ( intsIterator.hasNext() )
{
    value = (int)intsIterator.next();
    sum += value * value;
}
```

Vorsicht: `next()` setzt weiter!

- Summe der bis zu 10 ersten Werte von `ints`:

```
int sum = 0;
int count = 0;
ForwardIterator intsIterator = ints.iterator();
while ( count < 10 && intsIterator.hasNext() )
{
    sum += (int)intsIterator.next();
    count++;
}
```

## Entwurfsmuster Iterator

(Fortsetzung)

weitere Beispiele für den Einsatz eines Iterators

- Summe der bis zu 10 letzten Werte von ints:  
Mit ForwardIterator etwas umständlich zu lösen.
- Aber mit zusätzlicher Deklaration einer Klasse ReverseIterator mit gleicher Schleife:

```
public class ReverseIterator {  
    private Element current;  
    public ReverseIterator( Element elem ) {  
        current = elem;  
    }  
    public boolean hasNext() {  
        return current != null;  
    }  
    public Object next() {  
        if ( hasNext() ) {  
            Object content = current.getContent();  
            current = current.getPred(); ← current wandert nach vorne  
            return content;  
        } else {  
            throw new IllegalStateException();  
        }  
    }  
}
```

## Entwurfsmuster Iterator

(Fortsetzung)

Ergänzung in der Klasse DoublyLinkedList

```
public ReverseIterator reverseIterator()
{
    return new ReverseIterator( last );
}
```

Durchlauf beginnt bei last

- Die Methode `reverseIterator()` gibt ein Objekt der Klasse `ReverseIterator` zurück, dessen Attribut `current` auf das letzte Element der Liste verweist.  
Innerhalb von `DoublyLinkedList` kann auf das private Attribut `last` zugegriffen werden!
- Die Implementierung der Klassen `ForwardIterator` und `ReverseIterator` weisen Gemeinsamkeiten auf, die nun in einer Klasse `ListIterator` *zusammengefasst* werden, von der die Klassen `ForwardIterator` und `ReverseIterator` erben.

Vorteile:

- Es kann der gleiche Programmtext mit verschiedenen Iteratoren ausgeführt werden, wenn der Iterator unter einer Referenz des Typs `ListIterator` abgelegt wird.
- Der Schreibaufwand wird vermindert.

## Klassenhierarchie für Iteratoren

```
public class ListIterator
{
    private Element current;

    public ListIterator( Element elem )
    {
        current = elem;
    }

    public boolean hasNext()
    {
        return current != null;
    }

    public Object next()
    {
        throw new IllegalStateException();
    }
}
```

identisch übernommen aus den  
beiden bekannten Klassen  
ForwardIterator und  
ReverseIterator

## Klassenhierarchie für Iteratoren

(Fortsetzung)

```
public class ListIterator
{
    private Element current;

    public ListIterator( Element elem )
    {
        current = elem;
    }

    public boolean hasNext()
    {
        return current != null;
    }

    public Object next()
    {
        throw new IllegalStateException();
    }
}
```

Problem 1:  
`private` verhindert einen  
Zugriff in den Unterklassen

## Klassenhierarchie für Iteratoren

(Fortsetzung)

```
public class ListIterator
{
    private Element current;

    public ListIterator( Element elem )
    {
        current = elem;
    }

    public boolean hasNext()
    {
        return current != null;
    }

    public Object next()
    {
        throw new IllegalStateException();
    }
}
```

Problem 1:  
`private` verhindert einen  
Zugriff in den Unterklassen

Problem 2:  
Signatur muss in der Oberklasse  
für eine Nutzung bekannt sein,  
aber so ist der Rumpf sinnlos

- Beide Probleme werden nun gelöst.

## Klassenhierarchie für Iteratoren

(Fortsetzung)

```
public abstract class ListIterator
{
    Element current;  Zugriffsrecht: package

    public ListIterator( Element elem )
    {
        current = elem;
    }

    public boolean hasNext()
    {
        return current != null;
    }

    public abstract Object next();
}
```

## Zugriffsrecht *package*

- ❑ Das Zugriffsrecht **package** für Attribute oder Methoden erlaubt den Zugriff
  - in der Klasse, in der das Attribut oder die Methode deklariert wurde, und
  - in dem *Paket*, dem die Klasse zugeordnet ist.
- ❑ Durch **package** werden also die Zugriffsmöglichkeiten gegenüber **public** beschränkt.
- ❑ Der Zugriff ist in ausgewählten Klassen möglich und damit weiter gefasst als bei **private**.
- ❑ Die Klassen im Paket werden dadurch «besser gestellt» als außerhalb liegende Klassen.

Anmerkung:

Für das Zugriffsrecht **package** existiert *kein Schlüsselwort*.

Das Fehlen einer Angabe zum Zugriffsrecht für ein Attribut oder eine Methode deklariert das Zugriffsrecht **package**.

## Paket

Bisher bekannte strukturierende Einheiten in Java sind Klassen und Methoden.

Diese Strukturierungsmöglichkeiten sind für große Projekte nicht ausreichend, da getrennte Namensräume nur über das Einrichten von Klassen geschaffen werden können. Die *konzeptionelle Aufgabe* einer Klasse ist aber die *Typisierung* von Objekten und nicht das Schaffen eines Namensraums.

Daher bietet Java mit dem Konzept *Paket* eine zusätzliche Strukturierungseinheit, mit dem ein Namensraum abgegrenzt werden kann.

- ❑ Pakete besitzen in der Regel kleingeschriebene Namen.
- ❑ Pakete können geschachtelt werden.
- ❑ Paketschachtelungen entsprechen in der Regel Verzeichnisstrukturen.

## Pakete

(Fortsetzung)

Zuordnung einer Klasse zu einem Paket:

- ❑ Die Datei mit der Klasse muss *im Betriebssystem* in dem Verzeichnis abgelegt werden, das dem Paket entspricht.
- ❑ Die Datei mit dem Programmtext der Klasse muss als *ersten* Eintrag eine **package**-Deklaration enthalten:

```
package dap1list;  
public class DoublyLinkedList  
{ ... }
```

- ❑ Jede Datei kann nur höchstens einem Paket zugeordnet werden, also höchstens eine **package**-Deklaration enthalten.
- ❑ Enthält eine Datei keine **package**-Deklaration, so wird sie implizit dem *default package* zugeordnet. Somit gehört jede Datei zu einem Paket.

- ❑ Die Struktur geschachtelter Pakete wird durch eine **.**-Notation beschrieben.

dap1list	Klassen des Pakets liegen im Verzeichnis dap1list
dap1.trees	Klassen des Pakets liegen im Verzeichnis trees innerhalb des Verzeichnis dap1

## Pakete

(Fortsetzung)

Nutzung von Klassen aus anderen Paketen:

- Um eine Klasse aus einem anderen Paket zu nutzen, muss deren Name mit der Angabe des zugehörigen Pakets kombiniert werden.

Eine Klasse des Pakets `dap1list` kann also eine Liste folgendermaßen genutzt werden:

```
dap1list.DoublyLinkedList allPersons =  
    new dap1list.DoublyLinkedList();
```

- Klassen aus dem *default package* können nicht importiert werden, da das *default package* nicht über eine solche Angabe angesprochen werden kann.

## Pakete

(Fortsetzung)

Nutzung von Klassen aus anderen Paketen:

- ❑ Da diese Schreibweise bei häufiger Nutzung aus anderen Paketen etwas aufwändig ist, kann abkürzend eine **import**-Anweisung erfolgen.

Diese steht am Anfang einer Datei – aber hinter der Paketzuordnung mit **package**. Nach dem Importieren kann eine Klasse so genutzt werden, als wäre sie innerhalb des importierenden Pakets deklariert, also ohne die vollständige Qualifikation.

```
import dap1list.DoublyLinkedList;  
...  
DoublyLinkedList allPersons = new DoublyLinkedList();
```

- ❑ Auch ein Import aller Klassen eines Pakets ist möglich:  
**import dap1list.\*;**
- ❑ Treten durch das Importieren von Klassen *Namenskonflikte* mit Klassen aus dem importierenden Paket auf, so müssen diese durch vollständige Angabe des Pakets gelöst werden.

## Klassenhierarchie für Iteratoren

(Fortsetzung)

```
package dap1list;
public abstract class ListIterator
{
    Element current;

    public ListIterator( Element elem )
    {
        current = elem;
    }

    public boolean hasNext()
    {
        return current != null;
    }

    public abstract Object next();
}
```



Modifizierer **abstract**

## Abstrakte Klasse

- ❑ Eine abstrakte Klasse wird durch den Modifizierer **abstract** deklariert.
- ❑ Eine abstrakte Klasse wird *ausschließlich* deshalb deklariert,  
um in Vererbungsstrukturen als Oberklasse zu dienen.
- ❑ Objekte einer abstrakten Klasse können *nicht* erzeugt werden,  
der **new**-Operator kann nicht angewandt werden (Compiler-Fehler).
- ❑ Eine abstrakte Klasse kann aber Konstruktoren besitzen, die dann in den Konstruktoren von  
Unterklassen über **super( ... )** angesprochen werden können.
- ❑ Besitzt eine abstrakte Klasse *keinen* Standardkonstruktor,  
*müssen* die Unterklassen eigene Konstruktoren implementieren.

## Klassenhierarchie für Iteratoren

(Fortsetzung)

```
package dap1list;
public abstract class ListIterator
{
    Element current;

    public ListIterator( Element elem )
    {
        current = elem;
    }

    public boolean hasNext()
    {
        return current != null;
    }

    public abstract Object next();
}
```

Modifizierer **abstract**

ist Voraussetzung für

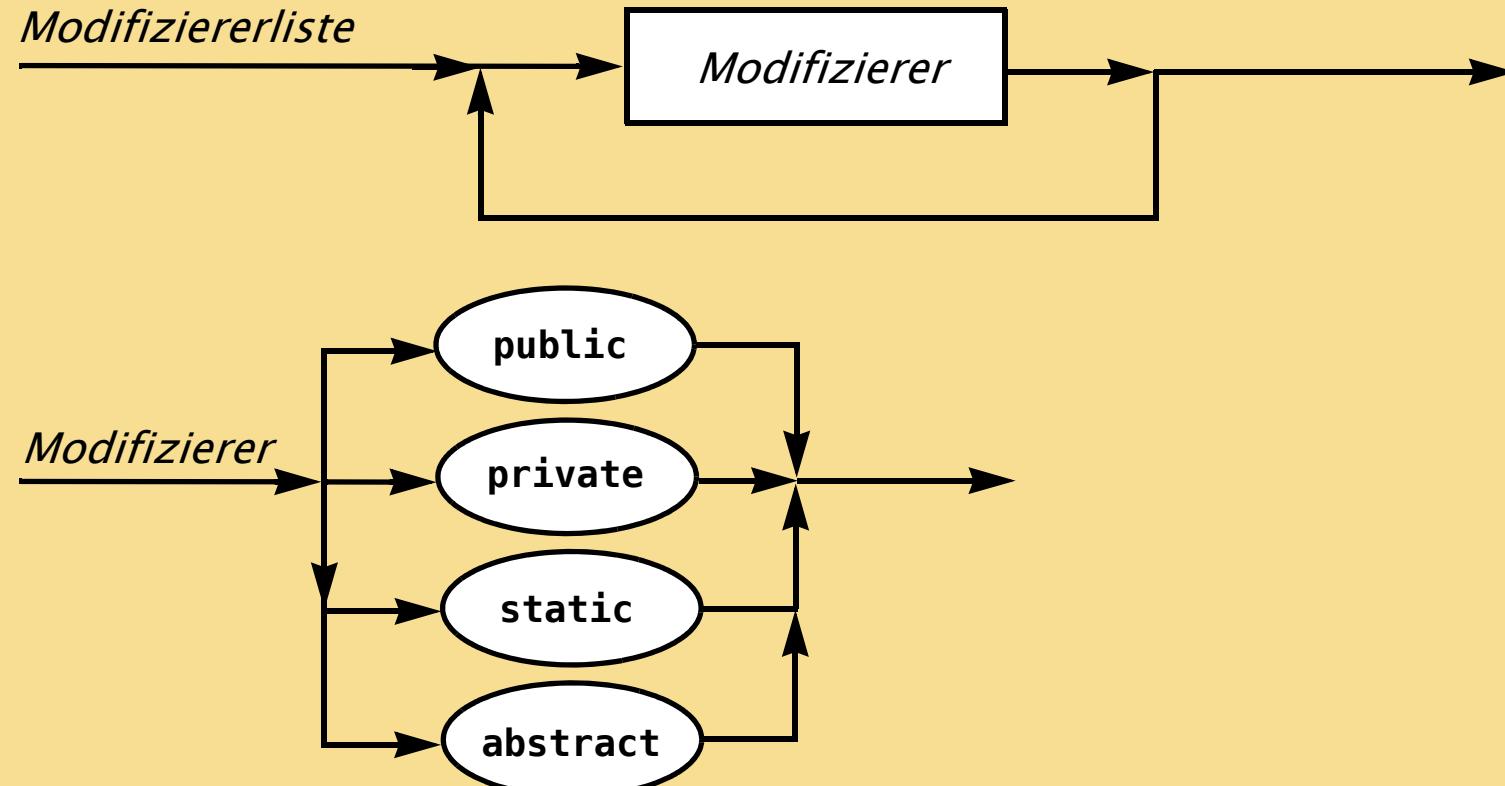
Modifizierer **abstract**  
kein Methodenrumpf notwendig

## Abstrakte Methode

- ❑ Eine abstrakte Klasse *kann* – muss aber nicht – abstrakte Methoden besitzen.
- ❑ Eine abstrakte Methode wird durch den Modifizierer **abstract** deklariert.
- ❑ Eine abstrakte Methode besteht nur aus einer Signatur,  
die durch ; abgeschlossen wird.
- ❑ Eine abstrakte Methode stellt eine Vorgabe für ihre Unterklassen dar:  
In Unterklassen müssen entweder
  - alle abstrakten Methoden implementiert werden oder
  - die Unterklasse muss auch wieder als abstrakte Klasse deklariert werden:  
Dann wird eine Implementierung auf tiefer liegende Unterklassen verschoben.
- ❑ Die Klasse `ListIterator` sorgt also dafür,  
dass *jede* ihrer nicht abstrakten Unterklassen
  - einen Konstruktor und
  - eine Methode mit der folgenden Signatur implementieren muss:

```
public Object next()
```

## Syntaxdiagramm zu *Modifizierern*



## Klassenhierarchie für Iteratoren

(Fortsetzung)

```
package dap1list;
public class ForwardIterator extends ListIterator
{
    public ForwardIterator( Element elem )
    {
        super( elem );
    }

    public Object next()
    {
        if ( hasNext() )
        {
            Object content = current.getContent();
            current = current.getSuccessor();
            return content;
        }
        else
        {
            throw new IllegalStateException();
        }
    }
}
```

spezialisiert abstrakte Oberklasse

notwendig, da kein Standardkonstruktor in der Oberklasse

notwendig, da Signatur in Oberklasse vorgegeben

## Klassenhierarchie für Iteratoren

(Fortsetzung)

```

package dap1list;
public class ForwardIterator extends ListIterator
{
    public ForwardIterator( Element elem )
    {
        super( elem );
    }

    public Object next()
    {
        if ( hasNext() )
        {
            Object content = current.
            current = current.getSuccessor();
            return content;
        }
        else
        {
            throw new IllegalStateException();
        }
    }
}

```

spezialisiert abstrakte Klasse

notwendig, da kein Standardkonstruktor in der Oberklasse

notwendig, da Signatur in Oberklasse vorgegeben

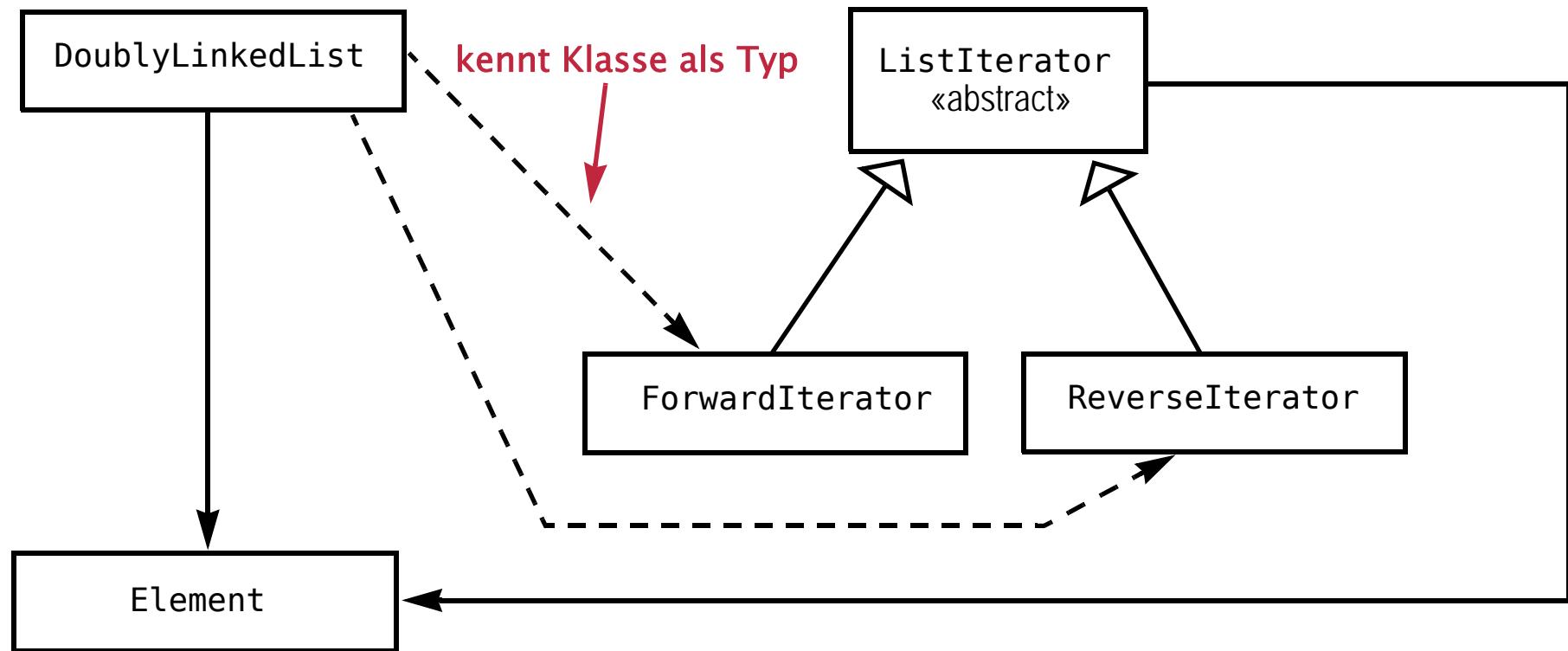
```

package dap1list;
public class ReverseIterator extends ListIterator
{
    public ReverseIterator( Element elem )
    {
        super( elem );
    }
    public Object next()
    {
        if ( hasNext() )
        {
            Object content = current.getContent();
            current = current.getPred();
            return content;
        }
        else
        {
            throw new IllegalStateException();
        }
    }
}

```

## Klassenhierarchie für Iteratoren (im Paket `dap1list`)

(Fortsetzung)



## Entwurfsmuster Iterator

- ❑ Hier vorgestellt wurde als Beispiel ein Iterator für eine doppelt verkettete Liste.
- ❑ Allgemein bietet das Muster *Iterator* eine Lösung an, wie auf Elemente einer zusammengesetzten Struktur sequenziell zugegriffen werden kann, *ohne* die technischen Details der Struktur zu enthüllen und *ohne* einen Zugriff von außen auf die Struktur zu erlauben.
- ❑ Das Muster *Iterator* schlägt dazu vor, spezielle Klassen anzulegen, deren Objekte nur dazu dienen, die Daten einer Struktur *genau einmal* zu durchlaufen. Genau so sind die hier vorgestellten Iteratoren aufgebaut.
- ❑ Ein besonderes Problem beim Einsatz von Iteratoren ist das Ändern der Inhalte der durchlaufenden Datenstruktur und insbesondere das Löschen von Elementen während des Durchlaufs.

Es ist zulässig, sich darum – mit geeigneter Dokumentation – nicht zu kümmern. Die hier vorgestellten Iteratoren arbeiten fehlerhaft, wenn das Element aus der Liste gelöscht wird, auf das `current` verweist.

## Entwurfsmuster

- Ein *Entwurfsmuster (design pattern)* ist ein geeigneter Lösungsansatz für ein in der Softwareentwicklung wiederkehrendes Problem.
- Entwurfsmuster sind aus praktischen Erfahrungen abgeleitet und 1995 erstmals vorgestellt worden. \*)
- Entwurfsmuster bieten nur einen Ansatz, der für jede konkret gegebene Situation geeignet realisiert werden muss.
- Das Muster *Iterator* ist ein *Verhaltensmuster*, da ein bestimmter Ablauf vorgeschlagen wird.
- Das Entwurfsmuster *Iterator* gibt eine Lösungsidee vor, wie beliebige Datenstrukturen gleichförmig durchlaufen werden können.
- Das Entwurfsmuster *Iterator* ist in seinen Einsatzmöglichkeiten nicht auf Listen beschränkt: Beispielsweise lassen sich auch binäre Bäume sequentiell durchlaufen.
- Strukturelle Änderungen können als zusätzliche Methoden durch einen Iterator angeboten werden. Die wesentliche Aufgabe des Iterators bleibt aber das Durchlaufen der Daten.

\*) E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1995 (Das Buch ist sehr schwer zu lesen.)

## Klassenhierarchie für Iteratoren – weitere Abstraktionen

```
package dap1list;
public class ForwardIterator extends ListIterator
{
    public ForwardIterator( Element elem )
    {
        super( elem );
    }

    public Object next()
    {
        if ( hasNext() )
        {
            Object content = current.getNext().getContent()
            current = current.getSuccessor();
            return content;
        }
        else
        {
            throw new IllegalStateException();
        }
    }
}
```

nur eine Zuweisung unterscheidet sich

```
package dap1list;
public class ReverseIterator extends ListIterator
{
    public ReverseIterator( Element elem )
    {
        super( elem );
    }

    public Object next()
    {
        if ( hasNext() )
        {
            Object content = current.getContent();
            current = current.getPred();
            return content;
        }
        else
        {
            throw new IllegalStateException();
        }
    }
}
```

## Klassenhierarchie für Iteratoren – weitere Abstraktionen

(Fortsetzung)

### Analyse:

Die Implementierungen der Methode `next()` in den beiden Unterklassen unterscheiden sich nur in einer Zuweisung.

### Ziel:

Die Klassenhierarchie wird derart umgestaltet, dass der Ablauf der Methode `next()` in der Oberklasse `ListIterator` formuliert werden kann.

## Klassenhierarchie für Iteratoren – weitere Abstraktionen

(Fortsetzung)

### Analyse:

Die Implementierungen der Methode `next()` in den beiden Unterklassen unterscheiden sich nur in einer Zuweisung.

### Ziel:

Die Klassenhierarchie wird derart umgestaltet, dass der Ablauf der Methode `next()` in der Oberklasse `ListIterator` formuliert werden kann.

### Lösungsansatz:

- Die Bestimmung des nächsten Elements wird in eine eigene Methode `step()` ausgelagert.
- Die Methode `step()` wird in der Klasse `ListIterator` als abstrakte Methode deklariert. In der Klasse `ListIterator` ist also noch nicht festgelegt, was `step()` tun wird.
- Die Methode `step()` kann aber in der Klasse `ListIterator` bereits verwendet werden. Ein Objekt, für das die Methode `next()` aufgerufen werden kann, setzt voraus, dass alle abstrakten Methoden – also auch `step()` implementiert sind.

Während der Ausführung wird dann die zum Objekt gehörende Implementierung von `step()` ausgeführt.

## Klassenhierarchie für Iteratoren – weitere Abstraktionen

(Fortsetzung)

```

package dapllist;
public abstract class ListIterator
{
    Element current; ←

    ListIterator( Element elem ) { current = elem; } ← bereits bekannt

    public boolean hasNext() { return current != null; } ←

    public Object next()
    {
        if ( hasNext() )
        {
            Object content = current.getContent();
            current = step(); ← Aufruf von step()
            return content;
        }
        else
        {
            throw new IllegalStateException();
        }
    }

    abstract Element step(); ← abstrakte Deklaration von step()
}

```

## Klassenhierarchie für Iteratoren – weitere Abstraktionen

(Fortsetzung)

```
package dap1list;
public class ForwardIterator extends ListIterator
{
    public ForwardIterator( Element elem ) { super( elem ); }

    Element step()
    {
        return current.getSucc();
    }
}
```

Implementierung  
in den Unterklassen

```
package dap1list;
public class ReverseIterator extends ListIterator
{
    public ReverseIterator( Element elem ) { super( elem ); }

    Element step()
    {
        return current.getPred();
    }
}
```

## Innere Klasse

- ❑ Die Klassen für die beiden Iteratoren und die Elemente der Liste sind öffentlich.
- ❑ Allerdings können diese Klassen sinnvoll nur im Zusammenhang mit der Klasse `DoublyLinkedList` benutzt werden, für die sie gestaltet worden sind.
- ❑ Die Benutzbarkeit der Klasse `DoublyLinkedList` kann verbessert werden, wenn diese Klassen gar nicht öffentlich verfügbar sind.

## Innere Klasse

(Fortsetzung)

- ❑ Die Klassen für die beiden Iteratoren und die Elemente der Liste sind im Paket `dap1list` enthalten. Da sie nur innerhalb des Pakets genutzt werden, können sie mit dem Zugriffsrecht **package** deklariert werden.
- ❑ Die Benutzbarkeit der Klasse `DoublyLinkedList` kann weiter verbessert werden, wenn diese Klassen gar nicht einzeln verfügbar sind.
- ❑ Java bietet die Möglichkeit, die Deklaration von Klassen *innerhalb* von anderen Klassen vorzunehmen. Solche Klassen werden als *innere Klassen* bezeichnet.
- ❑ Innere Klassen können mit Zugriffsrechten versehen werden.
- ❑ Innere Klassen haben Zugriff auf Attribute und Methoden der sie umgebenden Klasse.
- ❑ Die umgebende Klasse hat *keinen* besonderen Zugriff auf die Inhalte der inneren Klasse.
- ❑ Innere Klassen treten auf als:
  - *statische innere Klasse*
    - Die innere Klasse steht gemeinsam mit der umgebenden Klasse zur Verfügung.
    - Ein Objekt der umgebenden Klasse muss nicht erzeugt werden.
  - *Instanzklasse*
    - Die innere Klasse ist nur über ein Objekt der umgebenden Klasse verfügbar.
    - Für jedes Objekt wird eine eigene innere Klasse angelegt, was gelegentlich zu Kompatibilitätsproblemen führen kann.

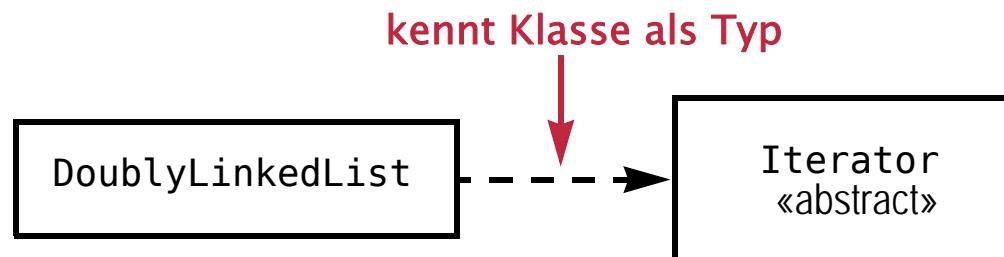
## Innere Klasse

(Fortsetzung)

- ❑ Um die Implementierung der inneren Klassen vollständig verbergen zu können, wird eine öffentliche, abstrakte Klasse `Iterator` angelegt, die lediglich zwei abstrakte Methoden vorschreibt.
- ❑ Referenzen auf `Iterator` können dann auf beliebige Iteratoren verweisen, wenn alle Iteratoren als Unterklasse von `Iterator` vereinbart werden.

```
public abstract class Iterator
{
    public abstract boolean hasNext();
    public abstract Object next();
}
```

gibt zwei Methoden vor



## Innere Klasse

(Fortsetzung)

```
public class DoublyLinkedList
{
    private Element first, last;
    private int size;
    ...
    private abstract class ListIterator extends Iterator
    {
        Element current;
        public boolean hasNext() { ... }
        public Object next() { ... }
        abstract Element step();
    }
}
```

stellt Typkompatibilität her

Zugriffsrecht *package* um Spezialisierung zu ermöglichen

public um Nutzung von außen zu ermöglichen

step() in Unterklassen

- Die Klasse ListIterator wird außerhalb der Klasse DoublyLinkedList nicht benötigt, da der Zugriff auf die Iteratoren ausschließlich über Referenzen auf die abstrakte Klasse Iterator erfolgen.  
Daher kann die Klasse ListIterator mit dem Zugriffsrecht **private** versehen werden.

## Innere Klasse

(Fortsetzung)

```
...
private class ForwardIterator extends ListIterator
{
    public ForwardIterator()
    {
        current = first;
    }

    Element step()
    {
        return current.getsucc();
    }

}
```

- ❑ Die *Instanzklasse* ForwardIterator kann auf das (*Instanz*-)Attribut first der umgebenden Klasse DoublyLinkedList zugreifen.

## Innere Klasse

(Fortsetzung)

```
...
public Iterator iterator()
{
    return new ForwardIterator();
}

...

private class Element
{
    private Object content;
    private Element pred, succ;

    ...
}

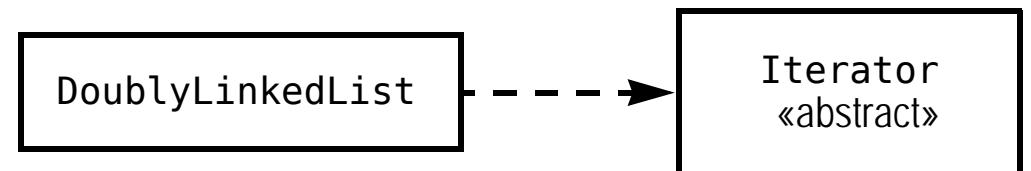
}
```

← auch die Klasse Element kann nun verborgen werden

## Innere Klasse

(Fortsetzung)

von außen sichtbare und  
benutzbare Klassenstruktur:



interner Aufbau der Klasse DoublyLinkedList:

```
public class DoublyLinkedList
{
    ...
    private abstract class ListIterator extends Iterator
    {
        ...
    }
    private class ForwardIterator extends ListIterator
    {
        ...
    }
    private class Element
    {
        ...
    }
}
```

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 11. Entwurfsmuster – Strategie

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-819

## Lernziele des Kapitels 11. Entwurfsmuster Strategie

Nach Durcharbeiten des Kapitels Entwurfsmuster Strategie sollen die teilnehmenden Studierenden

- das Entwurfsmuster *Strategie* kennen,
- Beispiele für Strategien kennen,
- eigene Strategien für vorhandene Strategieklassen implementieren können,
- eigene Strategieklassen implementieren können,
- den Einsatzbereich des Entwurfsmusters *Strategie* gegen den des Entwurfsmusters *Iterator* abgrenzen können.

## Entwurfsmuster Strategie

Der vorgestellte Iterator besitzt noch einige Nachteile:

- Es ist mit dem Iterator *nicht* möglich, strukturelle Änderungen an der Liste vorzunehmen:
  - Der über den Iterator gefundene Inhalt kann geändert, aber nicht ausgetauscht werden.
  - Das über den Iterator erreichte Element kann nicht entfernt werden.
- Die Nutzung des Iterators benötigt immer eine durch `hasnext()-next()`-Aufrufe kontrollierte Schleife.

Wie kann eine Lösung aussehen, die diese Nachteile vermeidet?

- Strukturelle Änderungen müssen *innerhalb* der Klasse `DoublyLinkedList` ausgeführt werden, da nur dort die Klasse `Element` bekannt ist.
- Die Art der Änderung soll jedoch flexibel *außerhalb* der Klasse `DoublyLinkedList` deklariert werden können.
- Ein *innerhalb* von `DoublyLinkedList` deklarierter Ablauf – eine Methode – muss also auf einen beliebigen *außerhalb* deklarierten Ablauf – auch eine Methode – zugreifen.
- Als *Transportbehälter* dient dabei ein Objekt einer Klasse, die eine abstrakte Klasse realisiert, die innerhalb und außerhalb von `DoublyLinkedList` bekannt ist.
- Diese Art des Aufbaus eines Algorithmus ist eine Umsetzung des *Entwurfsmusters Strategie*.

## Entwurfsmuster Strategie – SubstitutionStrategy

```
public class DoublyLinkedList
{
    ...
    public static abstract class SubstitutionStrategy
    {
        public abstract Object substitute( Object ref );
    }
    ...
}
```

statische Deklaration notwendig

- ❑ Die abstrakte Klasse SubstitutionStrategy gibt vor, wie die Methoden deklariert werden können, die der Klasse DoublyLinkedList zur Ausführung übergeben werden können.
- ❑ Die Signatur der Methode substitute ermöglicht die Rückgabe eines Objekts an die Methode, die die Methode substitute aufruft.
- ❑ Daher können nun Änderungen an den Inhalten der Liste vorgenommen werden, die in der Ausführung von substitute bestimmt werden.
- ❑ Die Deklaration erfolgt als öffentliche statische innere Klasse – **public static**, damit außerhalb von DoublyLinkedList Unterklassen angelegt werden können, *ohne* zuvor ein Objekt von DoublyLinkedList angelegt haben zu müssen.

## Entwurfsmuster Strategie – SubstitutionStrategy

(Fortsetzung)

```
public class DoublyLinkedList
{
    ...
    public void substituteAll( SubstitutionStrategy s )
    {
        Element current = first;
        while ( current != null )
        {
            current.setContent( s.substitute( current.getContent() ) );
            current = current.getSuccessor();
        }
    }
    ...
}
```

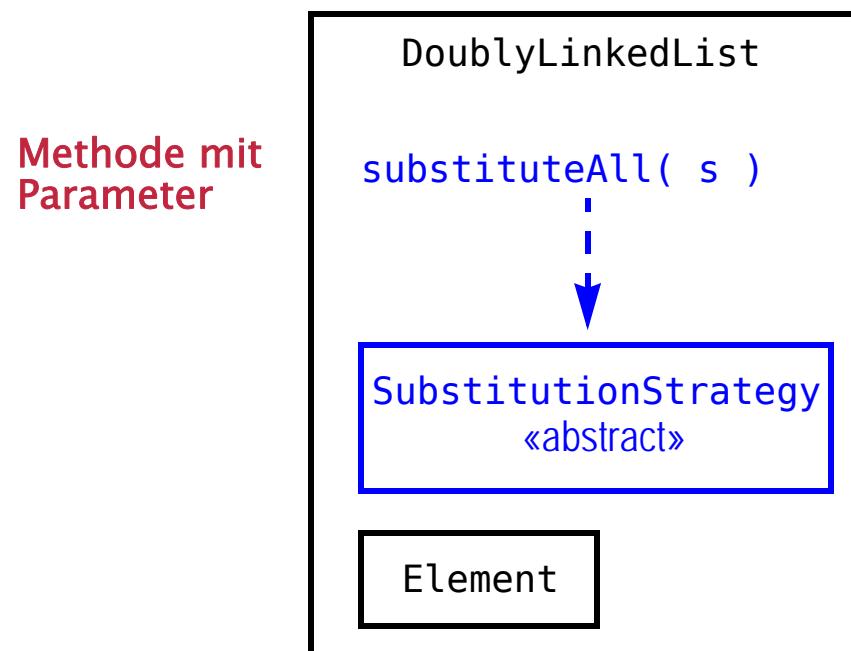
- Die Methode `substituteAll` ruft eine Implementierung der Methode `substitute` nacheinander für alle Elemente der Liste auf.
- Dabei wird `substitute` der Inhalt eines Elements als Argument übergeben.
- `substitute` liefert ein Objekt zurück, das den Inhalt der Liste an der Position der Referenz `current` ersetzt.

## Entwurfsmuster Strategie – SubstitutionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der der Wert jedes einzelnen Elements einer Liste von Integer-Objekten verdoppelt werden kann:

```
public class DoubleIntegersStrategy  
extends DoublyLinkedList.SubstitutionStrategy
```

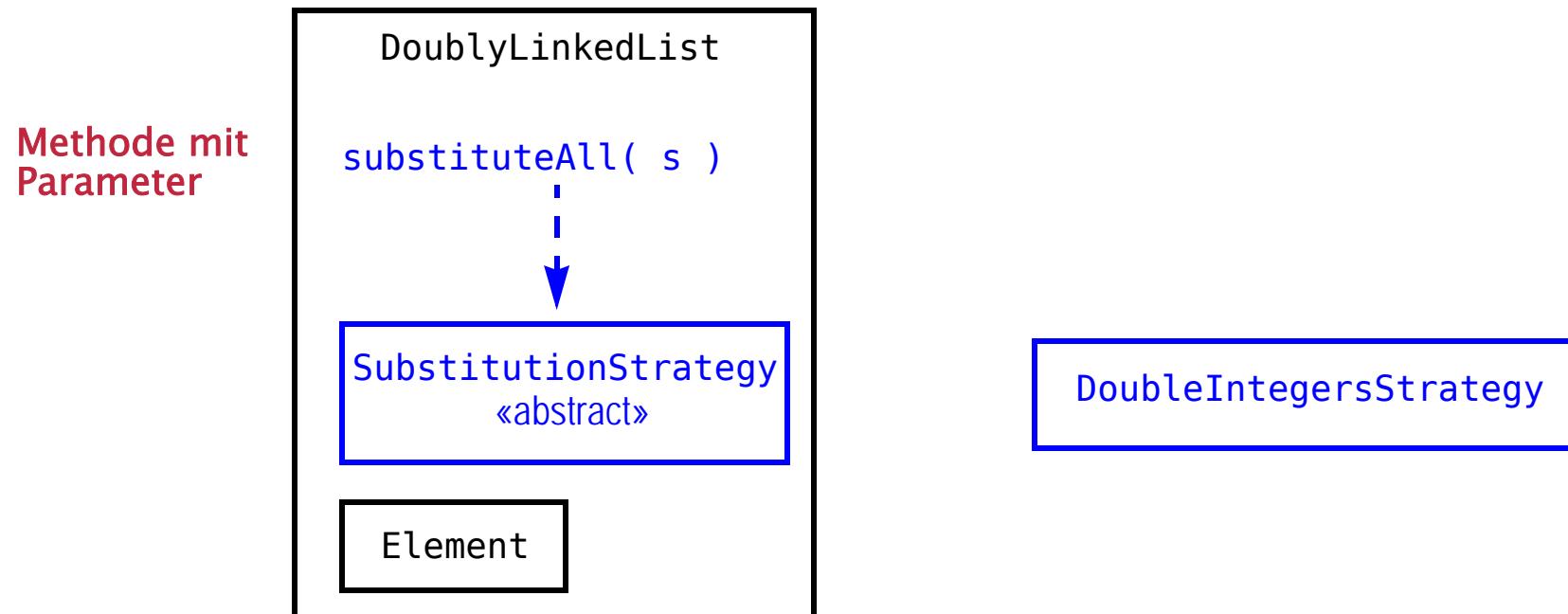


## Entwurfsmuster Strategie – SubstitutionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der der Wert jedes einzelnen Elements einer Liste von Integer-Objekten verdoppelt werden kann:

```
public class DoubleIntegersStrategy  
extends DoublyLinkedList.SubstitutionStrategy
```

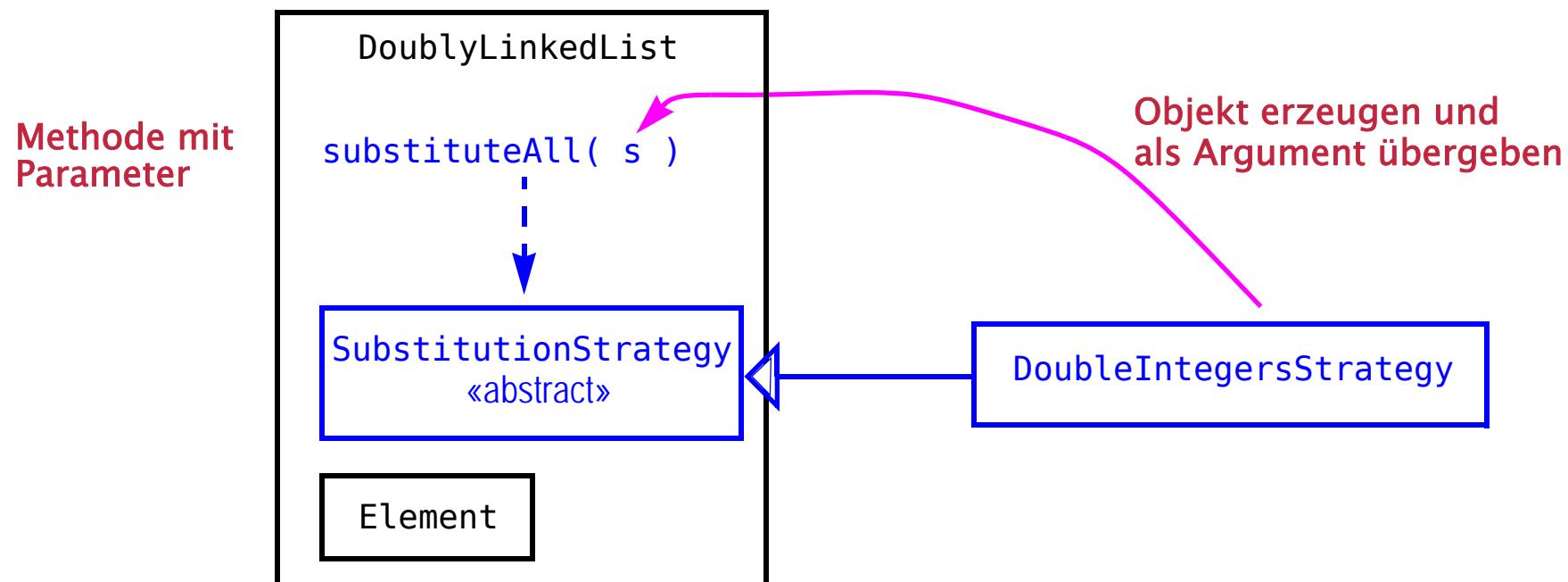


## Entwurfsmuster Strategie – SubstitutionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der der Wert jedes einzelnen Elements einer Liste von Integer-Objekten verdoppelt werden kann:

```
public class DoubleIntegersStrategy
extends DoublyLinkedList.SubstitutionStrategy
```



## Entwurfsmuster Strategie – SubstitutionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der der Wert jedes einzelnen Elements einer Liste von Integer-Objekten verdoppelt werden kann.

```
public class DoubleIntegersStrategy
extends DoublyLinkedList.SubstitutionStrategy
{
    public Object substitute( Object ref )
    {
        return 2 * (int)ref;
    }
}
```

- Erinnerung:  
Aufgrund des Typs `Object` der Rückgabe wird zu dem in der `return`-Anweisung berechneten `int`-Wert durch Boxing in ein passendes `Integer`-Objekt erzeugt, das zurückgegeben wird.

## Entwurfsmuster Strategie – SubstitutionStrategy

(Fortsetzung)

Nutzung der DoubleIntegersStrategy

```
DoublyLinkedList ints = new DoublyLinkedList();
ints.add( 5 );
ints.add( 6 );
ints.add( 11 );
ints.add( 2 );
ints.add( 9 );
DoubleIntegersStrategy manip = new DoubleIntegersStrategy();
System.out.print("in: ");
ints.showAll();
ints.substituteAll( manip );
System.out.print("substituted: ");
ints.showAll();
```

Ausgabe:

in: 5, 6, 11, 2, 9  
substituted: 10, 12, 22, 4, 18

## Entwurfsmuster Strategie – SubstitutionStrategy

(Fortsetzung)

```
public class AddNStrategy
extends DoublyLinkedList.SubstitutionStrategy
{
    private int n;

    public AddNStrategy( int increment )
    {
        n = increment;
    }

    public Object substitute( Object ref )
    {
        return (int)ref + n;
    }
}
```

- Die Methode `substitute` erhöht der Wert jedes einzelnen Elements einer Liste von Integer-Objekten um das im Konstruktor angegebene Inkrement.

## Entwurfsmuster Strategie – InspectionStrategy

- ❑ Der Aufruf der Methode `substitute` eines `SubstitutionStrategy`-Objekts durch die Methode `substituteAll` einer Liste führt immer zu einer Zuweisung an den Inhalt eines Elements der Liste.
- ❑ Sollen nur die Inhalte betrachtet – aber nicht ersetzt – werden, vermeidet eine eingeschränkte Strategie unbeabsichtigte Ersetzungen.
- ❑ Die abstrakte Klasse `InspectionStrategy` gibt vor, wie die Methoden deklariert werden können, die der Klasse `DoublyLinkedList` zur Ausführung übergeben werden können.

Die Signatur der Methode `inspect` schränkt die Realisierung ein:  
Der Parameter ist eine Referenz auf `Object`, es erfolgt *keine* Rückgabe,  
der Inhalt eines Elements kann also nicht ersetzt werden.

## Entwurfsmuster Strategie – InspectionStrategy

```
public class DoublyLinkedList
{
    ...
    public static abstract class InspectionStrategy
    {
        public abstract void inspect( Object ref );
    }
    ...
}
```

- ❑ Die abstrakte Klasse `InspectionStrategy` gibt vor, wie die Methoden deklariert werden können, die der Klasse `DoublyLinkedList` zur Ausführung übergeben werden können.
- ❑ Die Signatur der Methode `inspect` schränkt die Realisierung ein:  
Der Parameter ist eine Referenz auf `Object`, es erfolgt keine Rückgabe.

## Entwurfsmuster Strategie – InspectionStrategy

(Fortsetzung)

```
public class DoublyLinkedList
{
    ...
    public void inspectAll( InspectionStrategy s )
    {
        Element current = first;
        while ( current != null )
        {
            s.inspect( current.getContent() );
            current = current.getSuccessor();
        }
    }
    ...
}
```

- Die Methode `inspectAll` ruft eine Implementierung der Methode `inspect` nacheinander für alle Elemente der Liste auf.
- Dabei wird `inspect` der Inhalt eines Elements als Argument übergeben.
- Das Ergebnis der in der Methode `inspect` vorgenommenen Untersuchung muss die Methode selbst in geeigneter Form in einem Objekt ablegen, beispielsweise in ihrem Strategie-Objekt.

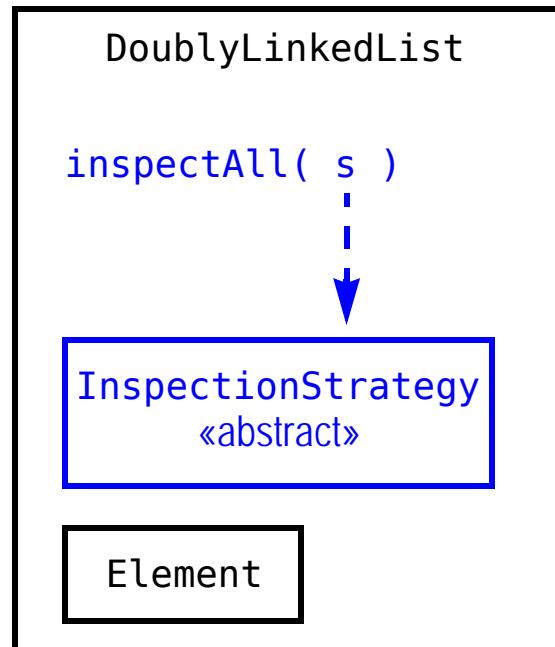
## Entwurfsmuster Strategie – InspectionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der die Summe aller Werte einer Liste von Integer-Objekten berechnet werden kann:

```
public class IntegerSummationStrategy
```

Methode mit  
Parameter

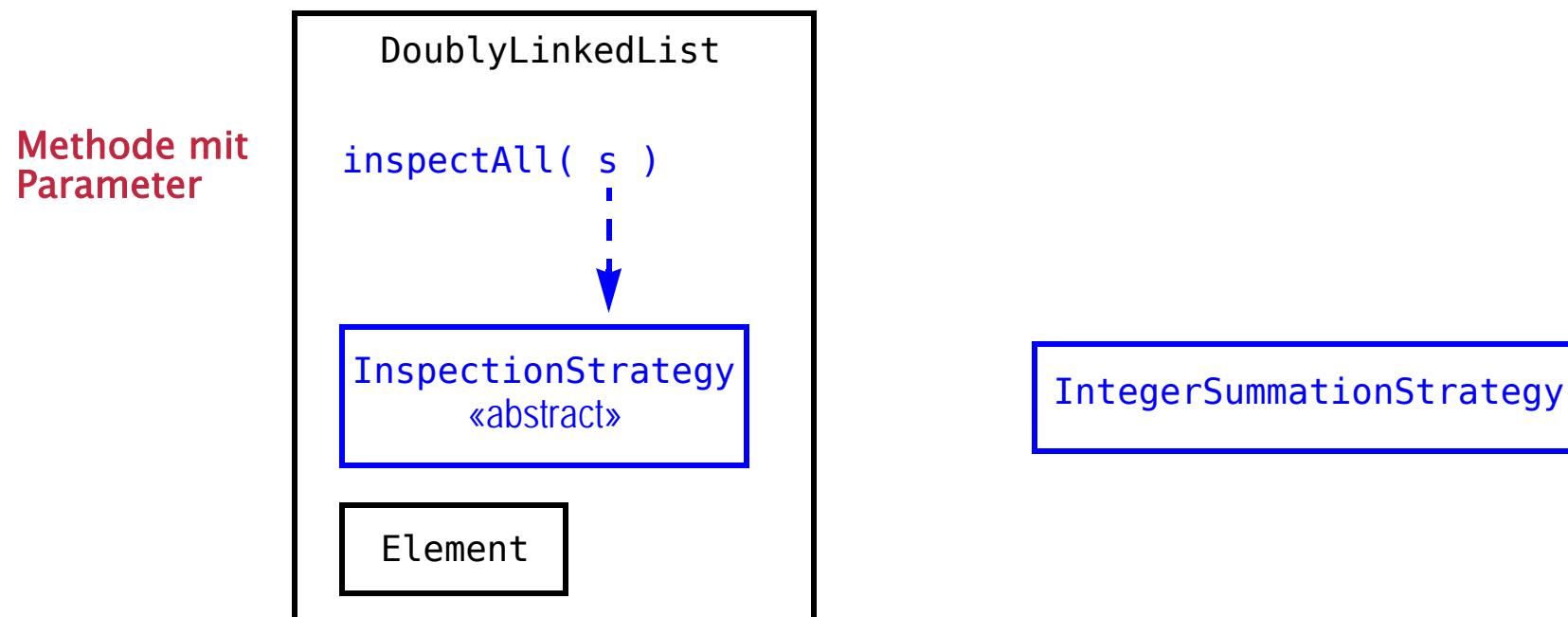


## Entwurfsmuster Strategie – InspectionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der die Summe aller Werte einer Liste von Integer-Objekten berechnet werden kann:

```
public class IntegerSummationStrategy  
extends InspectionStrategy
```

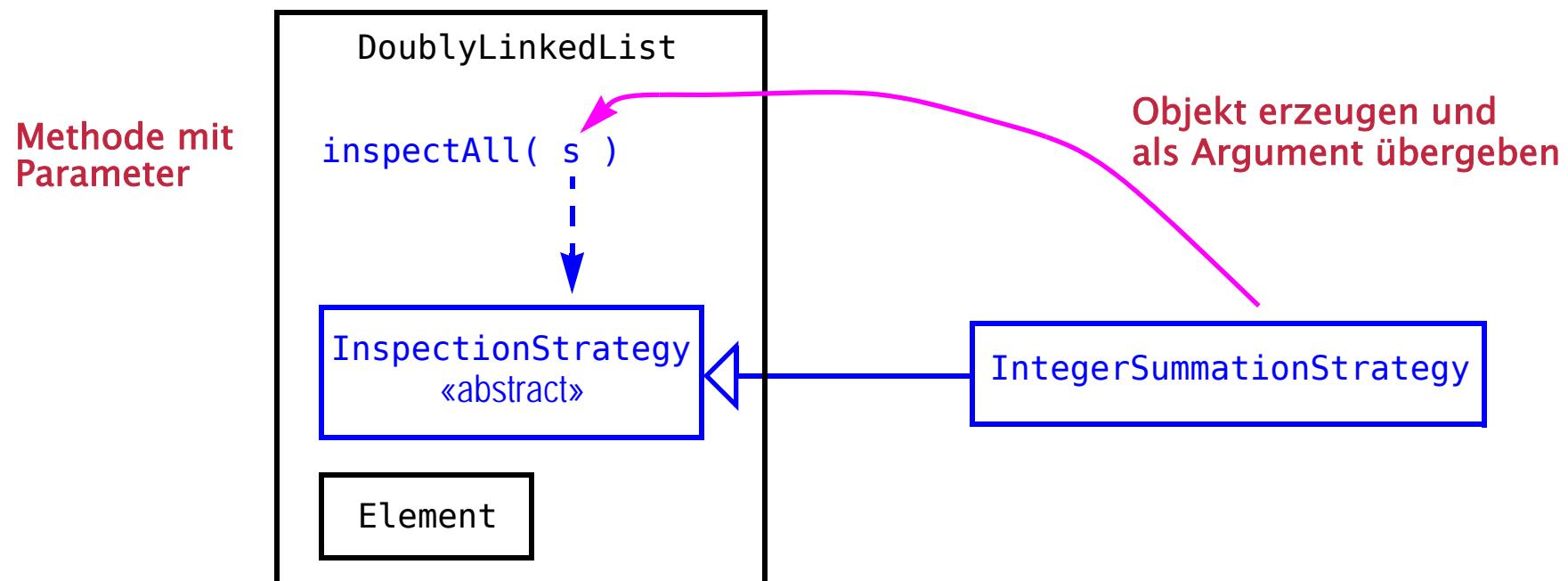


## Entwurfsmuster Strategie – InspectionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der die Summe aller Werte einer Liste von Integer-Objekten berechnet werden kann:

```
public class IntegerSummationStrategy  
extends InspectionStrategy
```



## Entwurfsmuster Strategie – InspectionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der die Summe aller Werte einer Liste von Integer-Objekten berechnet werden kann.

```
public class IntegerSummationStrategy
extends DoublyLinkedList.InspectionStrategy
{
    private int sum;

    public IntegerSummationStrategy() { sum = 0; }

    public void inspect( Object ref )
    {
        sum += (int)ref;
    }

    public int getSum()
    {
        return sum;
    }
}
```

Attribut sum für Zwischenergebnisse

berechnet die Summe aller Werte

Abruf des Ergebnisse der Summation

Hinweis: Das Attribut sum wird nie auf den Wert 0 zurückgesetzt. Mehrfaches Benutzen eines IntegerSummationStrategy-Objekts führt daher zu fehlerhaften Ergebnissen.

## Entwurfsmuster Strategie – InspectionStrategy

(Fortsetzung)

Nutzung der IntegerSummationStrategy

```
DoublyLinkedList ints = new DoublyLinkedList();
ints.add( 5 );
ints.add( 6 );
ints.add( 11 );
ints.add( 2 );
ints.add( 9 );
ints.showAll();

IntegerSummationStrategy sumUp = new IntegerSummationStrategy();
ints.inspectAll( sumUp );
System.out.println( sumUp.getSum() );
```

Ausgabe:

in: 5, 6, 11, 2, 9  
sum: 33

## Entwurfsmuster Strategie – InspectionStrategy

(Fortsetzung)

Beispiel für eine Strategie, mit der die Anzahl aller Studierenden mit einer Matrikelnummer größer 100 in einer Liste von Student-Objekten bestimmt werden kann.

```
public class CountStudentGt100Strategy
extends DoublyLinkedList.InspectionStrategy
{
    private int quantity;

    public CountStudentGt100Strategy() { quantity = 0; }

    public void inspect( Object ref )
    {
        if ( ((Student)ref).getRegistrationNo() > 100 )
        {
            quantity++;
        }
    }

    public int getQuantity() { return quantity; }
}
```

Attribut quantity für Zwischenergebnisse

prüft die Studierenden und zählt

Abruf des Ergebnisses der Zählung

Hinweis: Das Attribut quantity wird nie auf den Wert 0 zurückgesetzt. Mehrfaches Benutzen eines CountStudentGt100Strategy-Objekts führt daher zu fehlerhaften Ergebnissen.

## Entwurfsmuster Strategie – InspectionStrategy

(Fortsetzung)

### Nutzung der CountStudentGt100Strategy

```
DoublyLinkedList students = new DoublyLinkedList();
students.add( new Student( "A", "Inf", 45 ) );
students.add( new Student( "B", "Inf", 167 ) );
students.add( new Student( "C", "Inf", 22 ) );
students.add( new Student( "D", "Inf", 124 ) );
students.add( new Student( "E", "Inf", 123 ) );
students.add( new Student( "F", "Inf", 12 ) );
students.showAll();
CountStudentGt100Strategy counter = new CountStudentGt100Strategy();
students.inspectAll( counter );
System.out.println( counter.getQuantity() );
```

Ausgabe:

```
in: student: A, registration number: 45(Inf), student: B, ...
quantity: 3
```

## Entwurfsmuster Strategie – DeletionStrategy

```
public class DoublyLinkedList
{
    ...
    public static abstract class DeletionStrategy
    {
        public abstract boolean select( Object ref );
    }
    ...
}
```

- ❑ Die abstrakte Klasse `DeletionStrategy` gibt vor, wie die Methoden deklariert werden können, die der Klasse `DoublyLinkedList` zur Ausführung übergeben werden können.
- ❑ Die Signatur der Methode `select` gibt einen `boolean`-Wert an die Methode zurück, die die Methode `select` aufruft.
- ❑ Aufgrund des zurückgegebenen Wertes können nun Änderungen an der Struktur der Liste vorgenommen werden.

## Entwurfsmuster Strategie – DeletionStrategy

(Fortsetzung)

```
public void deleteSelected( DeletionStrategy s )
{
    Element current = first;
    while ( current != null )
    {
        Element candidate = current;
        current = current.getsucc();
        if ( s.select( candidate.getContent() ) )
        {
            remove( candidate );
        }
    }
}
```

- ❑ Die Methode `deleteSelected` prüft alle Elemente der Liste durch einen Aufruf der Methode `select`.
- ❑ Liefert `select` für den Inhalt eines Elements das Ergebnis `true`, wird das Element (und damit auch die Referenz auf den Inhalt) aus der Liste gelöscht.
- ❑ Die Methode `deleteSelected` nutzt die private Methode `remove`, die das als Argument übergebene Element aus der Liste löscht.

## Entwurfsmuster Strategie – DeletionStrategy

(Fortsetzung)

```

private void remove( Element e )
{
    if ( e != null )
    {
        if ( e.hasSucc() && e.hasPred() )
        {
            e.getPred().connectAsSucc( e.getSucc() );
        } else if ( e == first && e.hasSucc() )
        {
            first = first.getSucc();
            first.disconnectPred();
        } else if ( e == last && e.hasPred() )
        {
            last = last.getPred();
            last.disconnectSucc();
        } else {
            first = last = null;
        }
        size--;
    }
}

```

innen liegendes Element

erstes, aber nicht  
einziges Element

letztes, aber nicht  
einziges Element

einziges Element

## Entwurfsmuster Strategie – DeletionStrategy

(Fortsetzung)

```
public class RemoveEvenIntegersStrategy
extends DoublyLinkedList.DeletionStrategy
{
    public boolean select( Object ref )
    {
        return (int)ref % 2 == 0;
    }
}
```

- ❑ Die Methode `select` liefert `true` für jedes Element aus einer Liste von `Integer`-Objekten, dessen Inhalt eine gerade Zahl ist.
- ❑ Im Zusammenwirken mit der Methode `deleteSelected` werden so genau diese Elemente aus der Liste gelöscht.

## Entwurfsmuster Strategie

Zusammenfassung:

- ❑ Das *Entwurfsmuster Strategie* skizziert einen Lösungsweg, wie eine «Familie» von Algorithmen implementiert werden kann:
  - Ein vorgegebener Rahmen – `inspectAll`, `substituteAll`, `deleteSelected` – wird durch den Aufruf austauschbarer Methoden – `inspect`, `substitute`, `select` – konkretisiert.
  - Der Rahmen bestimmt die «gemeinsamen familiären Eigenschaften».
- ❑ Die austauschbaren Methoden werden in geeigneten Klassen deklariert und durch Erzeugen und Übergabe eines Objekts als Argument in den Rahmen eingebracht.
- ❑ Die Deklarationen von Objekt und austauschbarer Methode erfolgen außerhalb der Klasse, in der die Methode ihre Wirkung entfalten soll.
- ❑ Die Implementierung der Methode muss daher ohne Zugriff auf private Inhalte der Klasse erfolgen.
- ❑ Die vorgestellten Beispiele zeigen, dass der Umfang der in einer Familie möglichen Änderungen/Abläufe eingeschränkt werden kann:
  - Unterklassen von `SubstitutionStrategy` können die Inhalte der Elemente ersetzen.
  - Unterklassen von `InspectionStrategy` können die Inhalte der Elemente nur lesen.
  - Unterklassen von `DeletionStrategy` können Elemente löschen.

## Entwurfsmuster Strategie

(Fortsetzung)

Vorteile des Einsatzes:

- ❑ In Klassen festgelegte Rahmen für Abläufe können durch Strategien angepasst werden, ohne dass ein Zugriff auf die Implementierungen der Klassen erfolgen muss.
- ❑ Die Implementierungen der den Rahmen bildenden Klassen müssen für Anpassungen nicht verstanden werden.
- ❑ Die zum Einsatz kommenden Strategien müssen bei der Implementierung der den Rahmen bildenden Klassen nicht bekannt sein.
- ❑ Strategien können während der Ausführung gewechselt werden, wenn dieses von den Klassen, die den Rahmen bilden, vorgesehen ist: Dazu muss nur ein neues Strategie-Objekt eingefügt werden.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 12.1. Generische Klassen – Grundlagen

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-846

## Lernziele des Kapitels 12. Generische Klassen

Nach Durcharbeiten des Kapitels Generische Klassen sollen die teilnehmenden Studierenden

- die Vorteile generischer Klassen kennen und erklären können,
- generische Klassen nutzen können,
- generische Klassen erstellen können.

## Allgemeine Implementierungen mit der Klasse Object

- ❑ Die Klasse `DoublyLinkedList` kann beliebige Inhalte verwalten.
- ❑ Die Methoden dieser Klasse können für ihre Inhalte nur die wenigen Methoden der Klasse `Object` voraussetzen.
- ❑ Werden Objekte zurückgegeben, so muss die aufrufende Methode zunächst einen Type-Cast (Folie 764) vornehmen, um den Compiler davon zu überzeugen, dass das Objekt den erwarteten Typ hat.

Dieses Vorgehen hat den Nachteil, dass für jeden Inhalt meist Varianten von Methoden implementiert werden müssen, bei denen eventuell (*nur*) der Type-Cast ersetzt werden muss.

- ❑ Da die Klassen beliebige Inhalte verwalten können, können Inhalte auch beliebig gemischt werden (Folie 755). Alle Beispiele mit Listen, die (*ausschließlich*) mit Integer-, Person- oder Fraction-Objekten arbeiten, basieren auf der Disziplin des Entwicklers, immer auch nur Objekte *einer* Klasse in einer Liste abzulegen.

Der Compiler kann hierbei keine Unterstützung leisten.

## Allgemeine Implementierungen mit der Klasse Object

(Fortsetzung)

- ❑ Die Klasse `DoublyLinkedList` kann beliebige Inhalte verwalten.
- ❑ Die Methoden dieser Klasse können für ihre Inhalte nur die wenigen Methoden der Klasse `Object` voraussetzen.
- ❑ Werden Objekte zurückgegeben, so muss die aufrufende Methode zunächst einen Type-Cast (Folie 764) vornehmen, um den Compiler davon zu überzeugen, dass das Objekt den erwarteten Typ hat.

Dieses Vorgehen hat den Nachteil, dass für jeden Inhalt meist Varianten von Methoden implementiert werden müssen, bei denen eventuell (*nur*) der Type-Cast ersetzt werden muss.

- ❑ Da die Klassen beliebige Inhalte verwalten können, können Inhalte auch beliebig gemischt werden (Folie 755). Alle Beispiele mit Listen, die (*ausschließlich*) mit Integer-, Person- oder Fraction-Objekten arbeiten, basieren auf der Disziplin des Entwicklers, immer auch nur Objekte *einer* Klasse in einer Liste abzulegen.

Der Compiler kann hierbei keine Unterstützung leisten.

**Problem:**

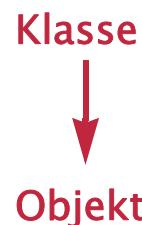
Referenzen auf `Object` bieten keine Möglichkeit, die Überwachung der Typzugehörigkeit durch den Compiler vornehmen zu lassen.

Eine sicherere Lösung bieten *generische Klassen* an.

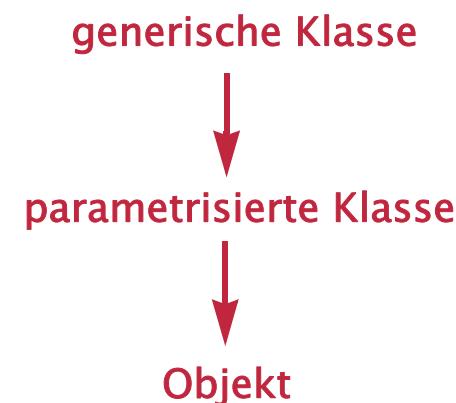
## Grundlegende Idee generischer Klassen

- ❑ Zuerst wird eine Klasse deklariert, die für verschiedene Inhalte nutzbar ist und die Parameter als Platzhalter für Typen enthält. Diese Klasse wird als *generische Klasse* bezeichnet.
- ❑ Für eine Nutzung wird die generische Klasse durch die Angabe eines (Typ-)Arguments in eine *parametrisierte Klasse* transformiert, die nur für den als Argument übergebenen Typ nutzbar ist.
- ❑ Von dieser parametrisierten Klasse können dann *Objekte* erzeugt werden, die Attributwerte speichern und auf denen Methoden ausgeführt werden können.

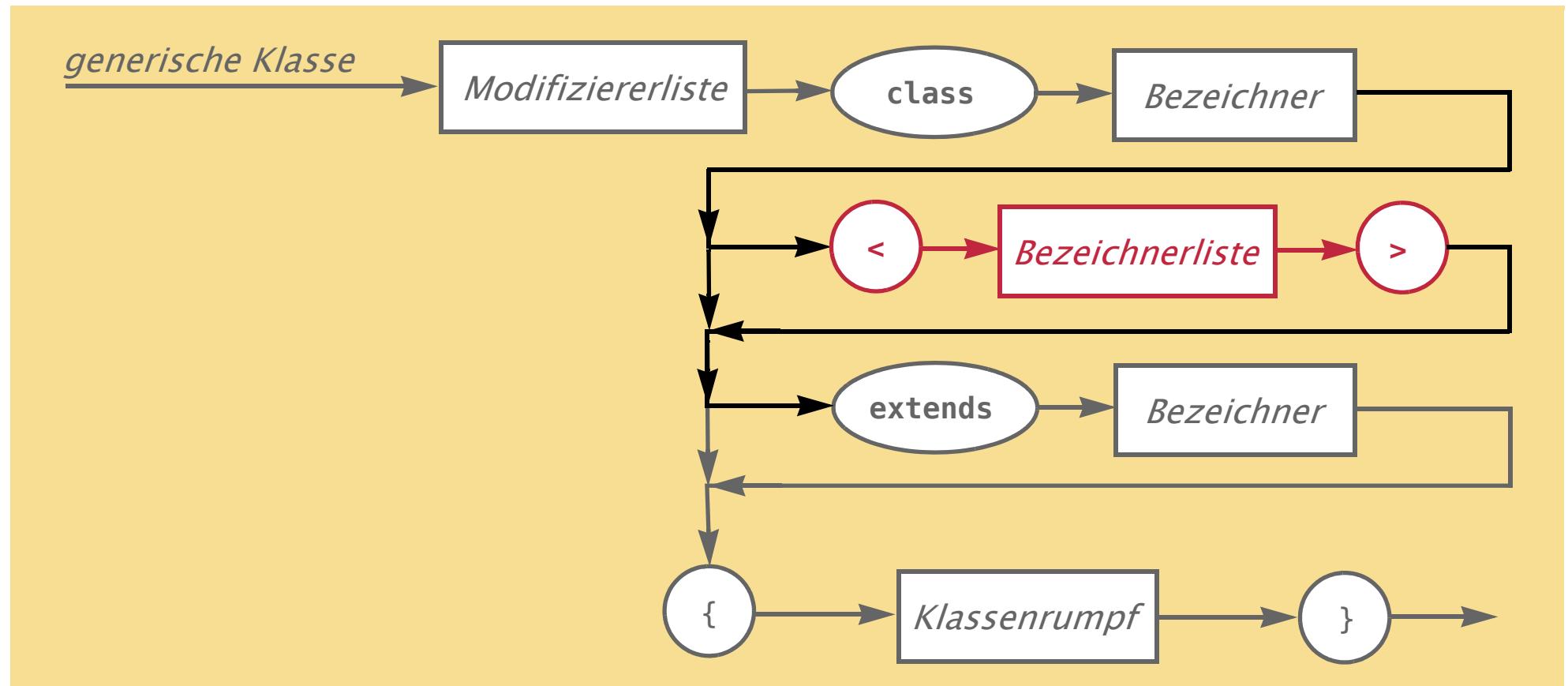
bisher:



jetzt:



## Deklaration einer generischen Klasse



- ❑ Die Bezeichner zwischen `< ... >` werden *Typparameter* der Klassendeklaration genannt.
- ❑ In Java werden Typparameter in der Regel als einzelne Großbuchstaben (z.B. `T`) deklariert.

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

```
public class DoublyLinkedList<T>
{
    private Element first, last;
    private int size;

    ... // Methoden der Liste

    private class Element
    {
        private T content;
        private Element pred, succ;

        ... // Methoden der Klasse Element
    }
}
```



Typparameter T  
Deklaration einer  
generischen Klasse

- T ist Platzhalter für *immer den selben* Typ.

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

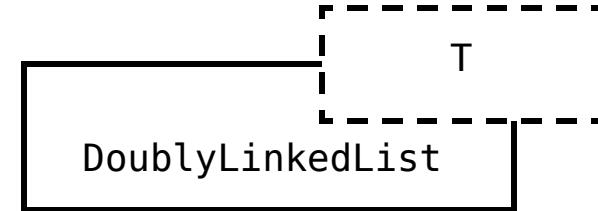
(Fortsetzung)

Beispiel für die Parametrisierung:

```
public class DoublyLinkedList<T> { ... }
```

Deklaration der Klasse  
mit Typparameter T

generische Klasse



**Beispiel für eine generische Klasse: DoublyLinkedList<T>**

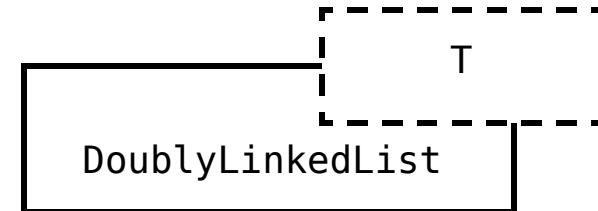
(Fortsetzung)

Beispiel für die Parametrisierung:

```
public class DoublyLinkedList<T> { ... }
```

Deklaration der Klasse  
mit Typparameter T

generische Klasse

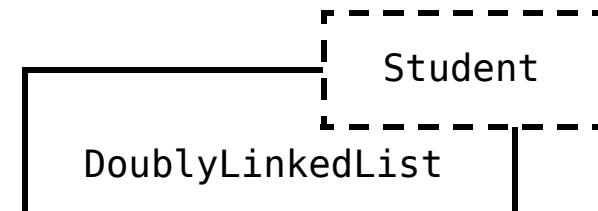


```
DoublyLinkedList<Student> students = new DoublyLinkedList<Student>();
```

Deklaration der Referenz  
mit Typargument

Aufruf des Konstruktors  
mit Typargument

parametisierte Klasse



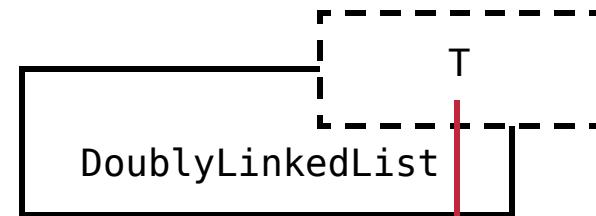
Student

**Beispiel für eine generische Klasse: DoublyLinkedList<T>**

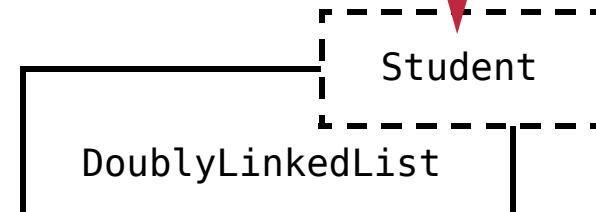
(Fortsetzung)

Beispiel für die Parametrisierung:

```
public class DoublyLinkedList<T> { ... }
```

**generische Klasse****Typargument: Student**

```
DoublyLinkedList<Student> students = new DoublyLinkedList<Student>();
```

**parametisierte Klasse**

Student

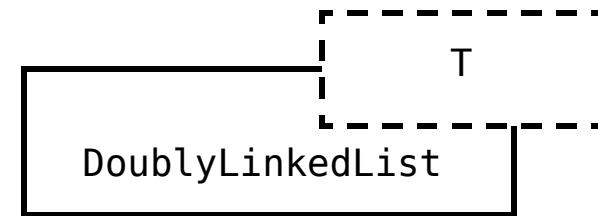
## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Beispiel für die Parametrisierung:

```
public class DoublyLinkedList<T> { ... }
```

generische Klasse



```
DoublyLinkedList<Student> students = new DoublyLinkedList<Student>();
```

Typargument

Typargument

- ❑ Das Erzeugen einer parametrisierten Klasse aus einer generischen Klasse erfolgt durch die Übergabe von Typargumenten.
- ❑ Die Übergabe von Typargumenten erfolgt ebenfalls in <...>.

**Beispiel für eine generische Klasse: DoublyLinkedList<T>**

(Fortsetzung)

Beispiel für die Parametrisierung:

```
DoublyLinkedList<Student> students = new DoublyLinkedList<Student>();
```

Deklaration der Referenz  
mit Typargument

Aufruf des Konstruktors  
mit Typargument

- ❑ DoublyLinkedList<Student> leitet aus der *generischen Klasse* DoublyLinkedList<T> die *parametisierte Klasse* ab, bei der der Typparameter T durch die als Typargument übergebene Klasse Student ersetzt wird.
- ❑ students ist also eine Referenz, die auf Objekte der parametrisierten Klasse verweisen kann.
- ❑ DoublyLinkedList<Student>() ist der Aufruf des Konstruktors der generischen Klasse DoublyLinkedList<T>, bei der der Typparameter T durch die als Typargument übergebene Klasse Student ersetzt wird, so dass das *erzeugte Listen-Objekt* ausschließlich mit Inhalten der Klasse Student umgehen kann.
- ❑ Die Liste students ist dadurch *typsicher*:
  - Es können nur Student-Objekte als Inhalte eingefügt werden.
  - Es kann sicher davon ausgegangen werden, dass Student-Objekte zurückgegeben werden.

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

```
public class DoublyLinkedList<T>
{
    private Element first, last;
    private int size;

    ... // Methoden der Liste

    private class Element
    {
        private T content;
        private Element pred, succ;

        ... // Methoden des Elements
    }
}
```



Typparameter T  
Deklaration einer  
generischen Klasse

- T ist Platzhalter für *einen*, immer gleichen konkreten Typ.

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

```
public class DoublyLinkedList<T> {  
    private Element first, last;  
    private int size;  
  
    ... // Methoden der Liste  
  
    private class Element {  
        private T content;  
        private Element pred, succ;  
  
        ... // Methoden des Elements  
    }  
}
```

- ❑ T ist Platzhalter für *einen*, immer gleichen konkreten Typ.
- ❑ Element ist eine innere (Instanz-)Klasse, der Parameter T ist dort sichtbar.

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Methoden der Klasse Element:

```
public Element( T c )
{
    content = c;
    pred = succ = null;
}

public T getContent()
{
    return content;
}

public void setContent( T c )
{
    content = c;
}
```

Parameter des Typs T

- Nutzung des Typparameters  $T$  in der inneren Klasse `Element`:  
 $T$  kann im Programmtext die Angabe einer konkreten Klasse ersetzen.

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Methoden der Klasse Element:

```
public Element( T c )
{
    content = c;
    pred = succ = null;
}

public T getContent()
{
    return content;
}

public void setContent( T c )
{
    content = c;
}
```

Rückgabe des Typs T

- Nutzung des Typparameters  $T$  in der inneren Klasse `Element`:  
 $T$  kann im Programmtext die Angabe einer konkreten Klasse ersetzen.

**Beispiel für eine generische Klasse: DoublyLinkedList<T>**

(Fortsetzung)

Methoden der Klasse DoublyLinkedList:

```
public void add( T content ) ← Parameter des Typs T
{
    Element e = new Element( content );
    if ( isEmpty() )
    {
        first = last = e;
    } else {
        last.connectAsSucc( e );
        last = e;
    }
    size++;
}

public T getFirst() ← T ist Rückgabetyp
{
    if ( !isEmpty() )
    {
        return first.getContent(); ← getContent()
    } else {                      liefert Ergebnis vom Typ T
        throw new IllegalStateException();
    }
}
```

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Methoden der Klasse DoublyLinkedList:

```
public class DoublyLinkedList<T>
{
    private Element first, last;
    private int size;

    public DoublyLinkedList() {  Deklaration des Konstruktor
        first = last = null;
        size = 0;
    }

    ...
}
```

- Der Aufruf des Konstruktors erfolgt aber mit der Angabe eines Typarguments:  
Es wird ein Objekt erzeugt, bei dem `T` durch den als Argument übergebenen Typ ersetzt wird.

`... = new DoublyLinkedList<Student>();`

**Beispiel für eine generische Klasse: DoublyLinkedList<T>**

(Fortsetzung)

Anmerkungen zu Methoden der Klasse Element:

- Eine Deklaration der Form `public void setContent( Object c )` erlaubt zur Laufzeit die Übergabe eines beliebigen Objekts als Argument.
- Eine Deklaration der Form `public void setContent( T c )` erlaubt zur Laufzeit nur die Übergabe eines Objekts eines bestimmten Typs als Argument. Dieser Typ wird während der Übersetzung durch die Übergabe eines Typarguments an `T` festgelegt und vom Compiler überprüft.
  
- Eine Deklaration der Form `public Object getContent()` erlaubt zur Laufzeit die Übergabe eines beliebigen Objekts als Rückgabewert.
- Eine Deklaration der Form `public T getContent()` erlaubt zur Laufzeit nur die Rückgabe eines Objekts eines bestimmten Typs. Dieser Typ wird während der Übersetzung durch die Übergabe eines Typarguments an `T` festgelegt und vom Compiler überprüft.

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Die Deklaration der generischen Klasse `Iterator<X>` erfolgt *außerhalb* der Klasse `DoublyLinkedList<T>`, da Iteratoren ein Konzept sind, das nicht auf Listen beschränkt ist.

```
public abstract class Iterator<X>
{
    public abstract boolean hasNext();
    public abstract X next();
```

← Typparameter X

← Typ der Rückgabe ist beschränkt  
auf das an X übergebene Argument

- Die Deklaration von `next()` wird angepasst, da die Rückgabe vom Typ der Elemente abhängt, über die iteriert wird.
- Die Deklaration von `hasNext()` ist nicht betroffen, da die Rückgabe nicht vom Typ der Elemente abhängt, über die iteriert wird.

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Die generische Fassung der Methode `Iterator<T> iterator()` in `DoublyLinkedList<T>`:

```

public Iterator<T> iterator()           ← Der Typparameter T von DoublyLinkedList
{                                         wird als Typargument an X übergeben
    return new ForwardIterator();
}

private abstract class ListIterator extends Iterator<T>
{
    Element current;

    public T next() {                   ← T ist Typargument
        if ( hasNext() ) {           ← für externe Klasse
            T content = current.getContent();
            current = step();
            return content;
        } else {
            throw new IllegalStateException();
        }
    }
    ...
}

```

Annotations:

- Der Typparameter `T` von `DoublyLinkedList` wird als Typargument an `X` übergeben
- `T` ist Typargument für externe Klasse
- `ListIterator` ohne Typparameter, da `T` in (Instanz-)Klasse sichtbar

**Beispiel für eine generische Klasse:** DoublyLinkedList<T>

(Fortsetzung)

Die generische Fassung der Klasse `Iterator<T> iterator()` *in* DoublyLinkedList<T>:

```
private class ForwardIterator extends ListIterator {
    public ForwardIterator() {
        current = first; ← first ist in innerer (Instanz-)Klasse bekannt
    }

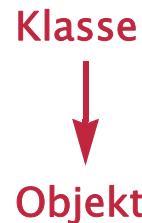
    Element step() {
        return current.getsucc();
    }
}

private class ReverseIterator extends ListIterator {
    public ReverseIterator() {
        current = last; ← last ist in innerer (Instanz-)Klasse bekannt
    }

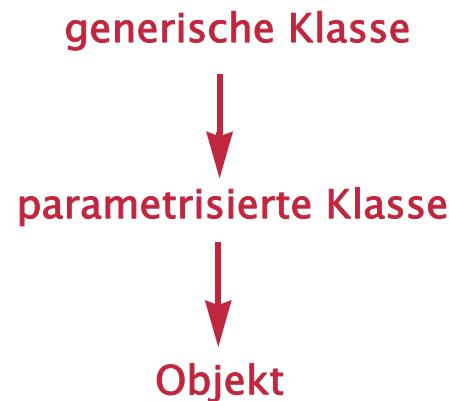
    Element step() {
        return current.getpred();
    }
}
```

## Erinnerung: Grundlegende Idee generischer Klassen

bisher:



jetzt:



Spezialisierung ist möglich in den Formen

- ❑ generische Klasse erbt von generischer Klasse:

```
class DoublyLinkedList<T> { ... }
```

```
class SpecialList<E> extends DoublyLinkedList<E> { ... }
```

- ❑ Klasse erbt von parametrisierter Klasse:

```
class IntegerList extends DoublyLinkedList<Integer> { ... }
```

Typargumente

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Parametrisierung und Aufruf:

```
DoublyLinkedList<Double> doubles = new DoublyLinkedList<Double>();
doubles.add( 14.1 ); doubles.add( 7.4 ); doubles.add( 2.3 ); doubles.add( 5.0 );
Iterator<Double> it = doubles.iterator();
double sum = 0.0;
while ( it.hasNext() )
{
    sum += it.next();
}
System.out.println( "sum: " + sum );
```

Deklaration mit Wrapper-Klasse  
als Typargument

Konstruktor mit  
Typargument

passende Deklaration:  
Iterator liefert hier Double-Werte

kein Type Cast nötig:  
next() liefert hier Double

- Die Liste `doubles` ist jetzt *typsicher*:
  - Es können nur Double-Objekte als Inhalte eingefügt werden.
  - Es kann sicher davon ausgegangen werden, dass Double-Objekte zurückgegeben werden.

## Generische Klassen – Anmerkungen

- *Nicht* möglich ist:

```
DoublyLinkedList<Double> doubles = new DoublyLinkedList<Double>();  
doubles.add( 14 );
```

- `DoublyLinkedList<Double>` erlaubt nur Objekte in der Liste, die kompatibel zur Klasse `Double` sind.
- `14` kann zwar durch Boxing in ein `Integer`-Objekt überführt werden, aber `Integer` ist nicht kompatibel zu `Double`.
- Möglich wäre:

```
DoublyLinkedList<Number> doubles = new DoublyLinkedList<Number>();  
doubles.add( 14 ); doubles.add( 7.4 );
```

- `Number` ist die Oberklasse von `Double` und `Integer`.  
Aber ein Iterator würde jetzt für `Number` deklariert werden *müssen*:  
`Iterator<Number> it = doubles.iterator();`

## Generische Klassen – Anmerkungen

(Fortsetzung)

- Generische Klassen sind erst in einer späteren Java-Version eingeführt worden.  
Bis dahin konnten nur Klassen mit Referenzen auf `Object` allgemein genutzt werden.
- Um bei der Einführung von generischen Klassen keine neue Ausführungsumgebung (Java Virtual Machine) einführen zu müssen und Kompatibilität zu vorhandenen JVM-Implementierungen zu bewahren, wurde folgende Umsetzung durch den Compiler festgelegt:
  - Es erfolgt *keine* spezifische Implementierung für das Typargument, es wird immer eine allgemeine Implementierung mit Referenzen auf `Object` benutzt.
  - Der Compiler ergänzt nur passende *Type Cast*-Operatoren.
  - Die geforderte Typsicherheit wird bereits durch die semantischen Überprüfungen des Compilers sichergestellt.
- Konsequenz:  
Innerhalb des Quellcodes der generischen Klasse ist `T` ein Platzhalter, zur Laufzeit ist aber das Typargument in der Klasse nicht bekannt.  
⇒ Es können beispielsweise keine Objekte oder Felder von `T` angelegt werden.

## Generische Klassen – Anmerkungen

(Fortsetzung)

- Generische Klassen sind erst in einer späteren Java-Version eingeführt worden.  
Bis dahin konnten nur Klassen mit Referenzen auf `Object` allgemein genutzt werden.
- Um für bestehende Klassen eine Überführung in eine generische Version zu ermöglichen und gleichzeitig den Aufruf der generischen Versionen in vorhandenem Programmtext zu ermöglichen, wurde die folgende Sprachsyntax zusätzlich aufgenommen:

```
public class DoublyLinkedList<T> { ... }
```

Eine Deklaration einer Referenz oder der Aufruf eines Konstruktors sind auch *ohne* Typparameter möglich:

```
DoublyLinkedList doubles = new DoublyLinkedList();
```

Es wird dann auf die Generierung von *Type Cast*-Operatoren verzichtet.

- Dieser Typ wird als *Raw Type* der generischen Klasse bezeichnet.
- Da der *Raw Type* der internen Darstellung entspricht, ist er kompatibel zu allen durch Parametrisierung festgelegten Typen. Die Benutzung führt in der Regel zu Warnungen, da typunsichere Zuweisungen erfolgen müssen.
- **Die Benutzung eines Raw Type sollte immer vermieden werden.**

## Generische Klassen – Anmerkungen

(Fortsetzung)

- ❑ Die Klasse `Number` ist die Oberklasse der Klasse `Integer`.
- ❑ `DoublyLinkedList<Number>` ist aber *nicht* die Oberklasse von `DoublyLinkedList<Integer>`.  
Beide Klassen sind vielmehr parametrisierte Versionen der gleichen generischen Klasse.
- ❑ Überlegung (Nachweis einer sinnvollen Regel durch Widerspruch)  
Falls `DoublyLinkedList<Number>` eine Oberklasse von `DoublyLinkedList<Integer>` wäre, würde folgender Programmtext erlaubt sein:

```
DoublyLinkedList<Number> x;  
x = new DoublyLinkedList<Integer>();  
x.add( 3.8 );
```

 Fehlermeldung

Der `double`-Wert `3.8` wäre zwar kompatibel zu der mit `Number` parametrisierten Deklaration der Methode `add`, nicht aber zu dem mit `Integer` parametrisierten Listen-Objekt.

## Generische Klassen – Anmerkungen

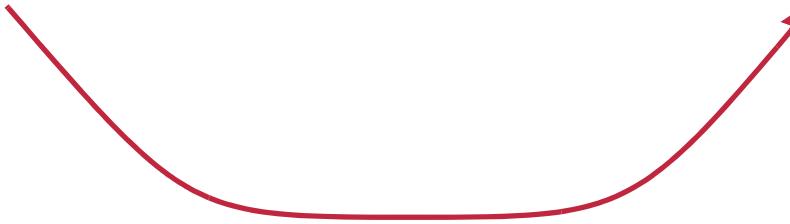
(Fortsetzung)

- ❑ Verkürzende Schreibweise:

<>-Operator (*diamond operator*) bestimmt – soweit möglich – den benötigten Typ beim *Aufruf eines Konstruktors* aus der Deklaration der Referenz, zu der eine Zuweisung erfolgt.

- ❑ Beispiel:

```
DoublyLinkedList<Double> doubles = new DoublyLinkedList<>();
```



leitet den Typ Double ab und ergänzt ihn implizit

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Generische statische öffentliche innere Klasse SubstitutionStrategy<E>  
für den Einsatz des Strategiemusters für die Klasse DoublyLinkedList<T>:

- ❑ Die Methode `substituteAll` arbeitet mit einem Objekt von `SubstitutionStrategy`. Daher gehört die Klasse `SubstitutionStrategy` zu der Klasse `DoublyLinkedList<T>`.
- ❑ Unterklassen zu `SubstitutionStrategy` sollen unabhängig von einer bestimmten Liste deklariert werden können. Daher wird `SubstitutionStrategy` als statische Klasse deklariert.
- ❑ Um Typsicherheit zu gewährleisten, muss `SubstitutionStrategy` als generische Klasse deklariert werden.

```
public static abstract class SubstitutionStrategy<E>
{
    public abstract E substitute( E ref );
}

public void substituteAll( SubstitutionStrategy<T> s )
{
    Element current = first;
    while ( current != null ) {
        current.setContent( s.substitute( current.getContent() ) );
        current = current.getSucc();
    }
}
```

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Generische statische öffentliche innere Klasse SubstitutionStrategy<E>  
für den Einsatz des Strategiemusters für die Klasse DoublyLinkedList<T>:

- ❑ Die Methode substituteAll arbeitet mit einem Objekt von SubstitutionStrategy. Daher gehört die Klasse SubstitutionStrategy zu der Klasse DoublyLinkedList<T>.
- ❑ Unterklassen zu SubstitutionStrategy sollen unabhängig von einer bestimmten Liste deklariert werden können. Daher wird SubstitutionStrategy als statische Klasse deklariert.
- ❑ Um Typsicherheit zu gewährleisten, muss SubstitutionStrategy dann aber als generische Klasse deklariert werden.

```
public static abstract class SubstitutionStrategy<E>
{
    public abstract E substitute( E ref );
}

public void substituteAll( SubstitutionStrategy<T> s )
{
    Element current = first;
    while ( current != null ) {
        current.setContent( s.substitute( current.getContent() ) );
        current = current.getSucc();
    }
}
```



## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Generische statische öffentliche innere Klasse SubstitutionStrategy<E>  
für den Einsatz des Strategiemusters für die Klasse DoublyLinkedList<T>:

- ❑ Die Methode `substituteAll` arbeitet mit einem Objekt von `SubstitutionStrategy`. Daher gehört die Klasse `SubstitutionStrategy` zu der Klasse `DoublyLinkedList<T>`.
- ❑ Unterklassen zu `SubstitutionStrategy` sollen unabhängig von einer bestimmten Liste deklariert werden können. Daher wird `SubstitutionStrategy` als statische Klasse deklariert.
- ❑ Um Typsicherheit zu gewährleisten, muss `SubstitutionStrategy` dann aber als generische Klasse deklariert werden.

```
public static abstract class SubstitutionStrategy<E>
{
    public abstract E substitute( E ref );
}

public void substituteAll( SubstitutionStrategy<T> s )
{
    Element current = first;
    while ( current != null ) {
        current.setContent( s.substitute( current.getContent() ) );
        current = current.getSucc();
    }
}
```



Typargument T garantiert Kompatibilität zum ausführenden Objekt

## Beispiel für eine generische Klasse: DoublyLinkedList<T>

(Fortsetzung)

Generische statische öffentliche innere Klasse SubstitutionStrategy<E>  
für den Einsatz des Strategiemusters für die Klasse DoublyLinkedList<T>

- Deklaration einer Strategie:

```
public class DoubleIntegersStrategy
extends DoublyLinkedList.SubstitutionStrategy<Integer>
{
    public Integer substitute( Integer ref ) {
        return 2 * ref;
    }
}
```

- Benutzung dieser Strategie:

```
DoublyLinkedList<Integer> ints = new DoublyLinkedList<Integer>();
...
DoubleIntegersStrategy manip = new DoubleIntegersStrategy();
ints.substituteAll( manip );
```

Typargumente müssen  
kompatibel sein

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 12.2. Generische Klassen – Binärer Suchbaum

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-879

## Lernziele des Kapitels

### Generische Gestaltung der Datenstruktur Binärer Suchbaum

Nach Durcharbeiten des Kapitels Generische Gestaltung der Datenstruktur Binärer Suchbaum sollen die teilnehmenden Studierenden

- die Vorteile generischer Klassen an einem weiteren Beispiel kennen,
- Möglichkeiten zur Einschränkung von Typparametern kennen,
- das Zusammenwirken verschiedener generischer Klassen kennen,
- das Prinzip der Delegation zur Wiederverwendung vorhandener Klassen kennen.

## Rückblick: Binärer Suchbaum

(siehe Folie 555)

### Ein *binärer Suchbaum*

ist ein binärer Baum, für den gilt, dass

- die Information in der Wurzel größer ist als die Information in ihrem linken Kind und
  - die Information in der Wurzel kleiner ist als die Information in ihrem rechten Kind und
  - der linke Teilbaum auch ein binärer Suchbaum ist und
  - der rechte Teilbaum auch ein binärer Suchbaum ist.
- 
- Der leere Baum ist auch ein binärer Suchbaum.

## Rückblick: Binärer Suchbaum

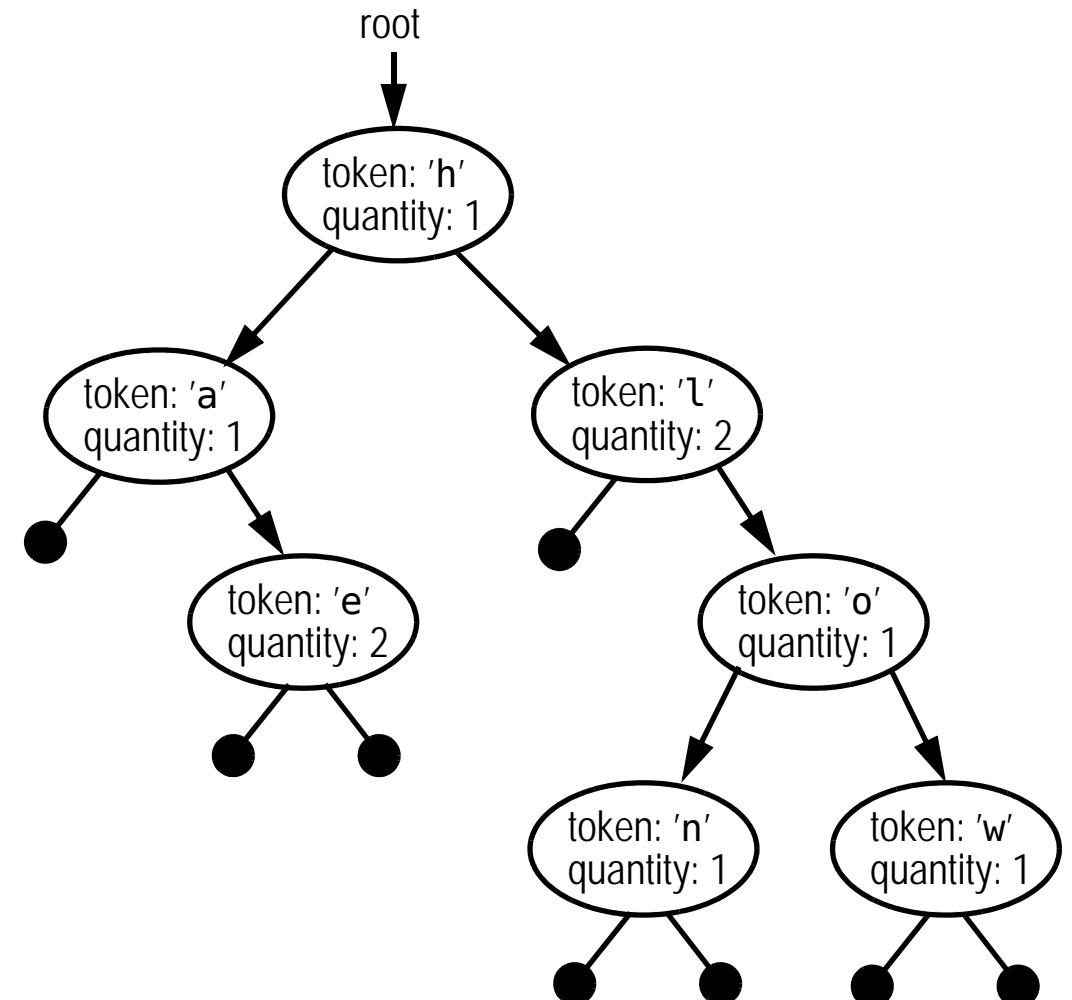
(siehe Folie 565)

(Fortsetzung)

### Suchvorgang:

In jedem Knoten wird überprüft,  
ob das zu suchende Zeichen

kleiner: dann weiter nach links – oder  
größer: dann weiter nach rechts – oder  
gleich: gefunden!  
ist.



## Rückblick: Nutzung von Vererbung

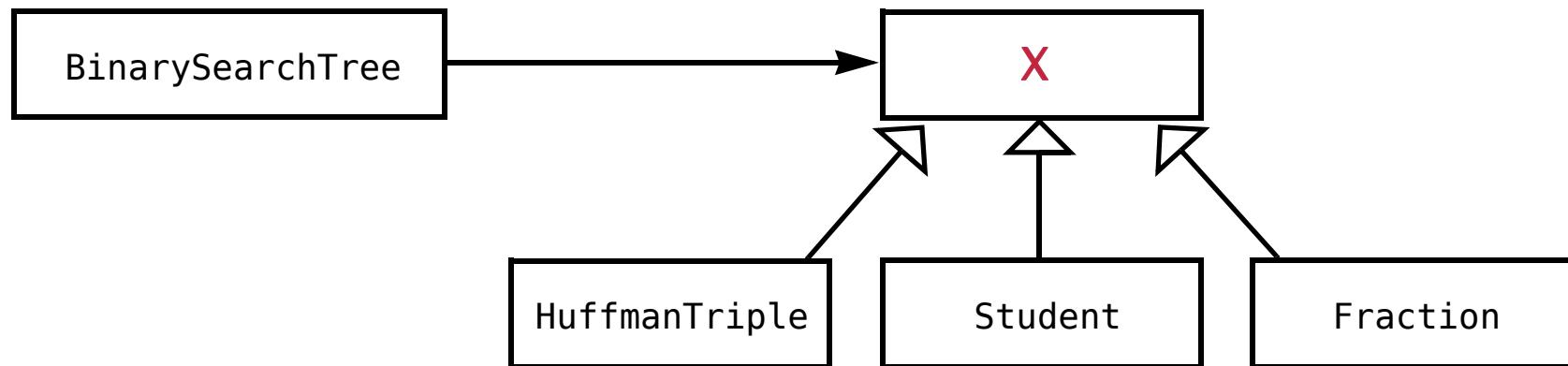
(siehe Folie 714)

neuer Lösungsansatz:

nur eine Sorte von Knoten, die flexibel zur Datenhaltung benutzt werden können

```
public class BinarySearchTree
{
    private X content;
    private BinarySearchTree leftChild, rightChild;
```

← Die Klasse X muss eine  
Ordnungsrelation garantieren



## Generische Klasse BinarySearchTree<T ... >

```
public class BinarySearchTree<T ... >
{
    private T content;
    private BinarySearchTree<T> leftChild, rightChild;

    public BinarySearchTree()
    {
        content = null;
        leftChild = null;
        rightChild = null;
    }

    ...
    // Methoden des Suchbaums
}
```

Typparameter T  
Deklaration der generischen Klasse

- ❑ Der Typparameter T ist Platzhalter für *einen*, immer gleichen konkreten Typ.
- ❑ aber:
  - Die Werte eines als Typargument übergebenen Typs müssen eine *Ordnungsrelation* besitzen.
  - Die Ordnung muss innerhalb der Klasse BinarySearchTree bestimmt werden können:  
Ein als Typargument übergebener Typ muss *geeignete Methoden* besitzen.

## Generische Klasse Comparable<T>

```
public abstract class Comparable<T>
{
    public abstract int compareTo( T t );
}
```

Typparameter T

- Die Klasse Comparable ist *abstrakt* und gibt die Implementierung genau einer Methode vor.
- Die Klasse Comparable ist *generisch* und besitzt einen Typparameter, der den Typ des Parameters der Methode compareTo bestimmt.
- Die Methode compareTo gibt einen *int*-Wert zurück, der für x und y eines Typs T folgendermaßen bestimmt werden soll:
  - falls x *größer als* y ist, dann soll gelten:  $x.compareTo( y ) > 0$
  - falls y *größer als* x ist, dann soll gelten:  $x.compareTo( y ) < 0$
  - falls x *gleich* y ist, dann soll gelten:  $x.compareTo( y ) = 0$
- Anmerkungen
  - Es muss immer gelten:  $x.compareTo( y ) == -(y.compareTo( x ))$
  - Es sollte immer gelten:  $x.compareTo( y ) == 0 \Rightarrow x.equals( y )$

## Generische Klasse Comparable<T>

(Fortsetzung)

```
public abstract class Comparable<T>
{
    public abstract int compareTo( T t );
}
```

- Jede Unterklasse der Klasse Comparable ermöglicht es, zwei beliebige Objekte eines Typs T miteinander zu vergleichen.
- Die Objekte von Unterklassen der Klasse Comparable sind also geeignete Inhalte für binäre Suchbäume, da die geforderte Ordnung hergestellt werden kann.
- Die Deklaration der Klasse BinarySearchTree<T ... > muss also sicherstellen, dass als Typargumente für T nur Unterklassen der Klasse Comparable verwendet werden dürfen.

## Generische Klasse BinarySearchTree<T extends Comparable<T>>

```
public class BinarySearchTree<T extends Comparable<T>>
{
    private T content;
    private BinarySearchTree<T> leftChild, rightChild;

    public BinarySearchTree()
    {
        content = null;
        leftChild = null;
        rightChild = null;
    }

    ...
    // Methoden des Suchbaums
}
```

Typparameter T wird eingeschränkt

- Als Argument für den Typparameter T ist nun nur noch ein solcher Typ zulässig, der die Klasse Comparable<T> spezialisiert, also eine compareTo-Methode besitzt.
- Da der Typparameter T immer den gleichen Typ bezeichnet, stellt die Einschränkung zugleich sicher, dass die compareTo-Methode ein Argument des Typs T erwartet.

**Generische Klasse BinarySearchTree<T extends Comparable<T>>**

(Fortsetzung)

```
public T getContent()
{
    return content;
}

public boolean isEmpty()
{
    return content == null;
}

public boolean isLeaf()
{
    return !isEmpty() && leftChild.isEmpty() && rightChild.isEmpty();
}

public int size()
{
    if ( isEmpty() )
    {
        return 0;
    } else {
        return 1 + leftChild.size() + rightChild.size();
    }
}
```

**Methoden in  
der Klasse BinarySearchTree**

**Generische Klasse BinarySearchTree<T extends Comparable<T>>**

(Fortsetzung)

```
public boolean contains( T t )
{
    if ( isEmpty() )
    {
        return false;
    }
    else
    {
        if ( content.compareTo( t ) > 0 )
        {
            return leftChild.contains( t );
        }
        else if ( content.compareTo( t ) < 0 )
        {
            return rightChild.contains( t );
        }
        return true;
    }
}
```

weitere Methode in  
der Klasse BinarySearchTree

content ist Referenz auf T,  
T muss Comparable  
spezialisieren, besitzt also  
eine Methode compareTo



## Generische Klasse BinarySearchTree<T extends Comparable<T>>

(Fortsetzung)

```
public void add( T t )
{
    if ( isEmpty() )
    {
        content = t;
        leftChild = new BinarySearchTree<T>();
        rightChild = new BinarySearchTree<T>();
    }
    else
    {
        if ( content.compareTo( t ) > 0 )
        {
            leftChild.add( t );
        }
        else if ( content.compareTo( t ) < 0 )
        {
            rightChild.add( t );
        }
    }
}
```

weitere Methode in  
der Klasse BinarySearchTree

- ❑ Der hier deklarierte Suchbaum für *allgemeine Inhalte* weist *keine* spezifische Funktionalität auf, wie es bei der Klasse CharacterSearchTree der Fall war: quantity++

## Generische Klasse `BinarySearchTree<T extends Comparable<T>>`

(Fortsetzung)

```
private void toList( DoublyLinkedList<T> list )
{
    if ( !isEmpty() )
    {
        leftChild.toList( list );
        list.add( content );
        rightChild.toList( list );
    }
}
```

weitere Methode in  
der Klasse `BinarySearchTree`

- ❑ Die Methode `toList` besitzt einen Parameter der generischen Klasse `DoublyLinkedList`.
- ❑ Durch Übergabe von `T` als Typargument werden nur Listen-Objekte als Argumente von `toList` akzeptiert, bei denen der Typ ihrer Inhalte zum Typ der im Baum abgelegten Inhalte passt.
- ❑ Die Methode `toList` durchläuft den Baum in *InOrder*-Reihenfolge.
- ❑ Die Methode `toList` wird jetzt genutzt, um einen einfachen *Iterator* über den Baum bereitzustellen.  
Erinnerung: Ein Iterator ist ein Objekt, das mit den beiden Methoden `hasNext` und `next` einen einmaligen Durchlauf durch eine Datenstruktur ermöglicht.

## Rückblick: Generische Klasse Iterator<T>

(siehe Folie 865)

```
public abstract class Iterator<T>
{
    public abstract boolean hasNext();
    public abstract T next();
}
```

- Die Klasse Iterator besitzt jetzt einen Typparameter T.
- Ein Iterator, der die Klasse Iterator spezialisiert, liefert also durch seinen next-Methode immer zu T kompatible Objekte.

**Generische Klasse BinarySearchTree<T extends Comparable<T>>**

(Fortsetzung)

```
public Iterator<T> iterator()
{
    DoublyLinkedList<T> list = new DoublyLinkedList<T>();
    toList( list );
    return list.iterator();
}
```

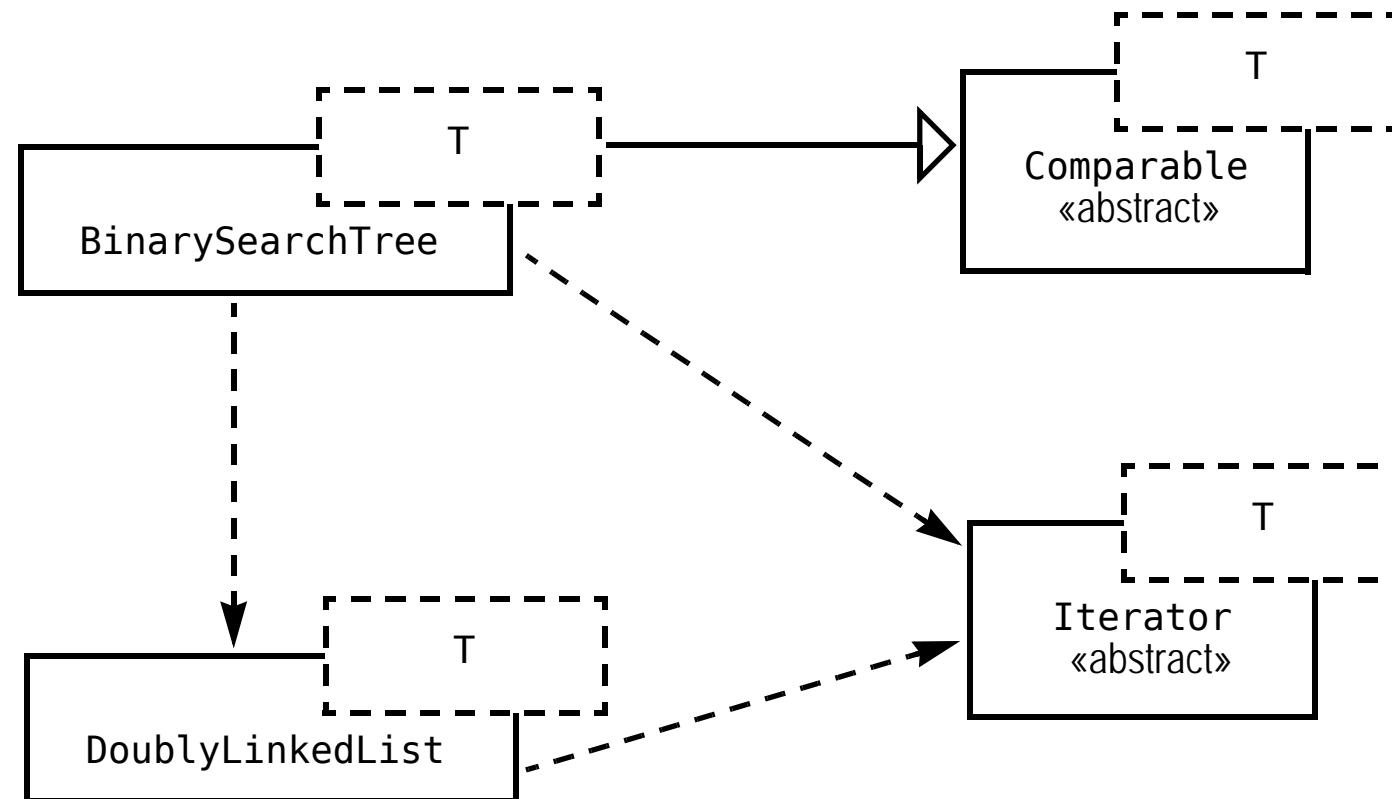
**weitere Methode  
in der Klasse  
BinarySearchTree**

- ❑ Die Methode `iterator()` liefert einen zum Inhaltstyp `T` passenden Iterator.
- ❑ `new DoublyLinkedList<T>()` liefert ein Listen-Objekt, das in seinen Elementen Inhalte des Typs `T` verwalten kann.
- ❑ Der Aufruf `toList( list )` trägt alle Inhalte des Baums in die Liste `list` ein.
- ❑ Da `toList` einen *InOrder*-Durchlauf verwendet, sind die Inhalte der Liste aufsteigend sortiert.
- ❑ Die Referenz `list` verweist auf ein `DoublyLinkedList`-Objekt.  
Jedes `DoublyLinkedList`-Objekt kann `Iterator`-Objekte bereitstellen.
- ❑ `return list.iterator()` stellt das von dem Listen-Objekt `list` bereitgestellte `Iterator`-Objekt als Iterator für den Baum zur Verfügung, da ja alle Inhalte des Baums auch über die Liste erreicht werden können.

## Generische Klasse `BinarySearchTree<T extends Comparable<T>>`

(Fortsetzung)

Visualisierung der Klassenstruktur



## Iterator für BinarySearchTree

Implementierte Lösung:

- ❑ Es wird eine Liste erstellt, die die Inhalte aller Knoten des Baums enthält:  
Das ist mit einem *InOrder*-Durchlauf einfach zu implementieren.
- ❑ Der Iterator nutzt dann diese Liste für seinen Durchlauf:  
Der ist schon implementiert in der Klasse `ForwardIterator` der Liste.
- ❑ Die Methode `iterator()` gibt also einfach den (passenden) Iterator der Liste zurück.



## Iterator für BinarySearchTree

(Fortsetzung)

Implementierte Lösung:

- ❑ Es wird eine Liste erstellt, die die Inhalte aller Knoten des Baums enthält:  
Das ist mit einem *InOrder*-Durchlauf einfach zu implementieren.
- ❑ Der Iterator nutzt dann diese Liste für seinen Durchlauf:  
Der ist schon implementiert in der Klasse `ForwardIterator` der Liste.
- ❑ Die Methode `iterator()` gibt also einfach den (passenden) Iterator der Liste zurück.



Eine solche Vorgehensweise ist typisch für objektorientierte Software:

Eine Klasse (bzw. jedes Objekt dieser Klasse) lässt eine seiner Aufgaben von einem Objekt einer anderen Klasse erledigen.

Diese Art des einfachen Übertragens einer Aufgabe wird als *Delegation* bezeichnet.

## Iterator für BinarySearchTree

(Fortsetzung)

- Der durch Delegation bereitgestellte Iterator ist ein einfacher Lösungsansatz, dessen Konzeption und Realisierung ohne viel Aufwand vorgenommen werden konnten.
- Die Benutzung des Iterators ist identisch zu der von Iteratoren auf der Liste – es wird ja letztlich der gleiche Iterator benutzt.

Nachteile:

- Da die Inhalte des Baums in eine Liste ausgelagert werden, über die iteriert wird, arbeitet diese Lösung beim Ändern des Baums (Löschen/Hinzufügen von Inhalten) fehlerfrei – allerdings auf der Basis möglicherweise veralteter Inhalte.
- Der Aufwand während der Ausführung ist groß:
  - Zunächst muss immer ein vollständiger Durchlauf durch den Baum erfolgen – auch dann, wenn später gar nicht über alle Inhalte iteriert wird.
  - Bei großen Bäumen besitzt die aufgebaute Liste selbst einen entsprechend großen, eigenen Speicherbedarf.

## Iterator für BinarySearchTree

(Fortsetzung)

Implementierung eines Iterators ohne die beiden Nachteile:

- Voraussetzungen:
  - Ein Iterieren über einen binären Baum erfolgt immer durch ein Hinab- und Hinaufsteigen durch die Äste des Baums.
  - Dabei kann aber durch die nur in eine Richtung vorgenommene Referenzierung nicht einfach nach oben navigiert werden.
- Lösungsansatz:
  - Für die Fortsetzung eines bereits begonnenen Durchlaufs durch den Baum werden nur die Knoten benötigt, die auf dem Ast zwischen der Wurzel und dem aktuell betrachteten Knoten liegen.
  - Der Iterator muss also eine geeignete Datenstruktur besitzen, in der er vorübergehend die Vorgänger des aktuell betrachteten Knotens aufbewahrt.

Diese Lösung wird in der privaten Klasse `StackBasedIterator` umgesetzt.

## Iterator für BinarySearchTree

### Klasse StackBasedIterator

(Fortsetzung)

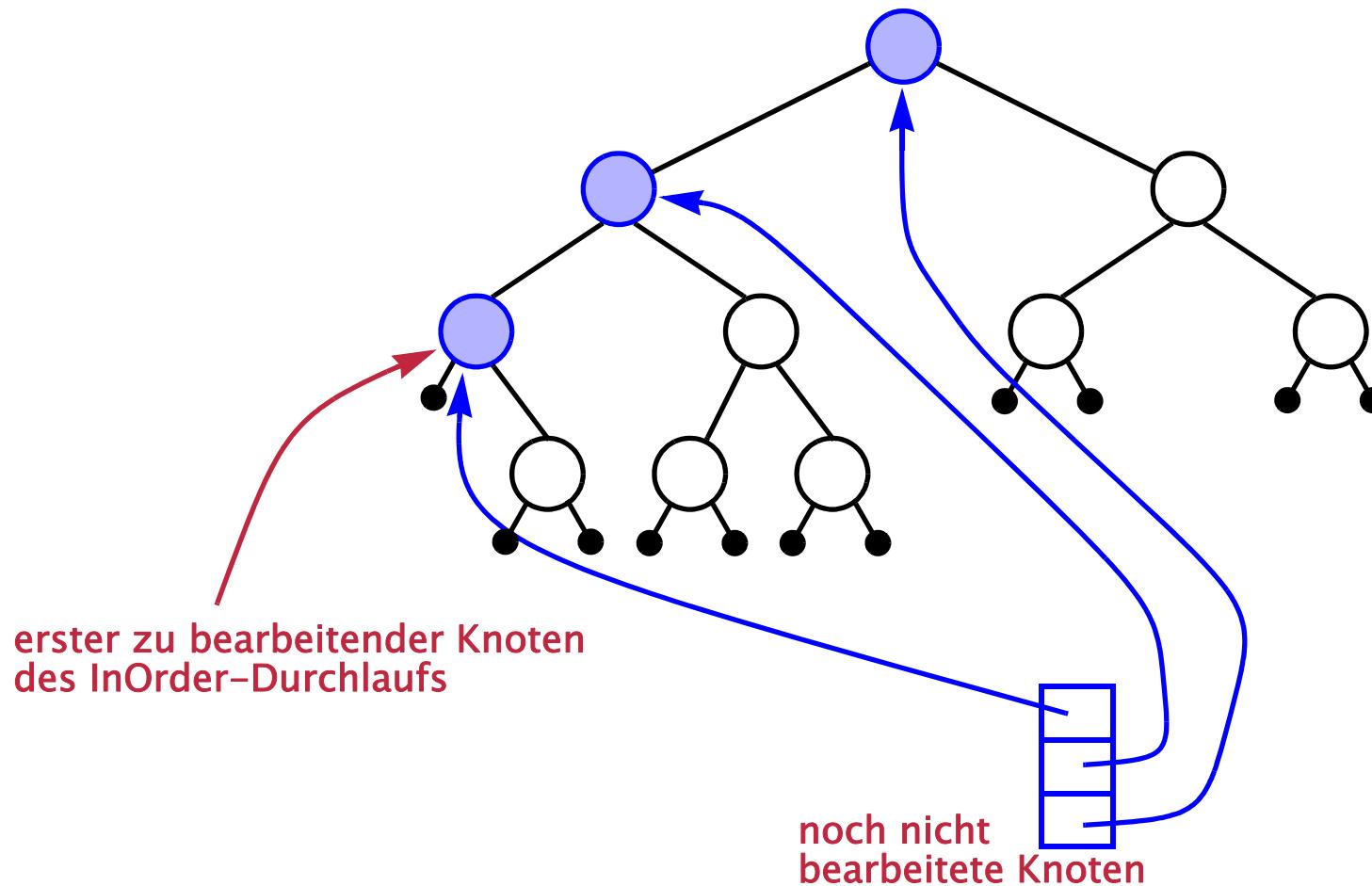
Algorithmus:

- ❑ Es werden immer für den aktuell vom Iterator angebotenen Knoten alle *noch nicht bearbeiteten* Knoten abgespeichert, die auf dem Pfad liegen, der von dem aktuellen Knoten zur Wurzel führt.
- ❑ Wird ein Knoten durch den Iterator abgearbeitet, so wird sein *InOrder*-Nachfolger als nächster Knoten ausgewählt. Das ist der äußerste linke Knoten in seinem rechten Teilbaum.
- ❑ Gibt es keinen rechten Teilbaum, also keinen rechten Nachfolger, wird mit dem *noch nicht bearbeiteten* Vorgänger des abgearbeiteten Knotens fortgefahrene.

## Iterator für BinarySearchTree

Klasse StackBasedIterator

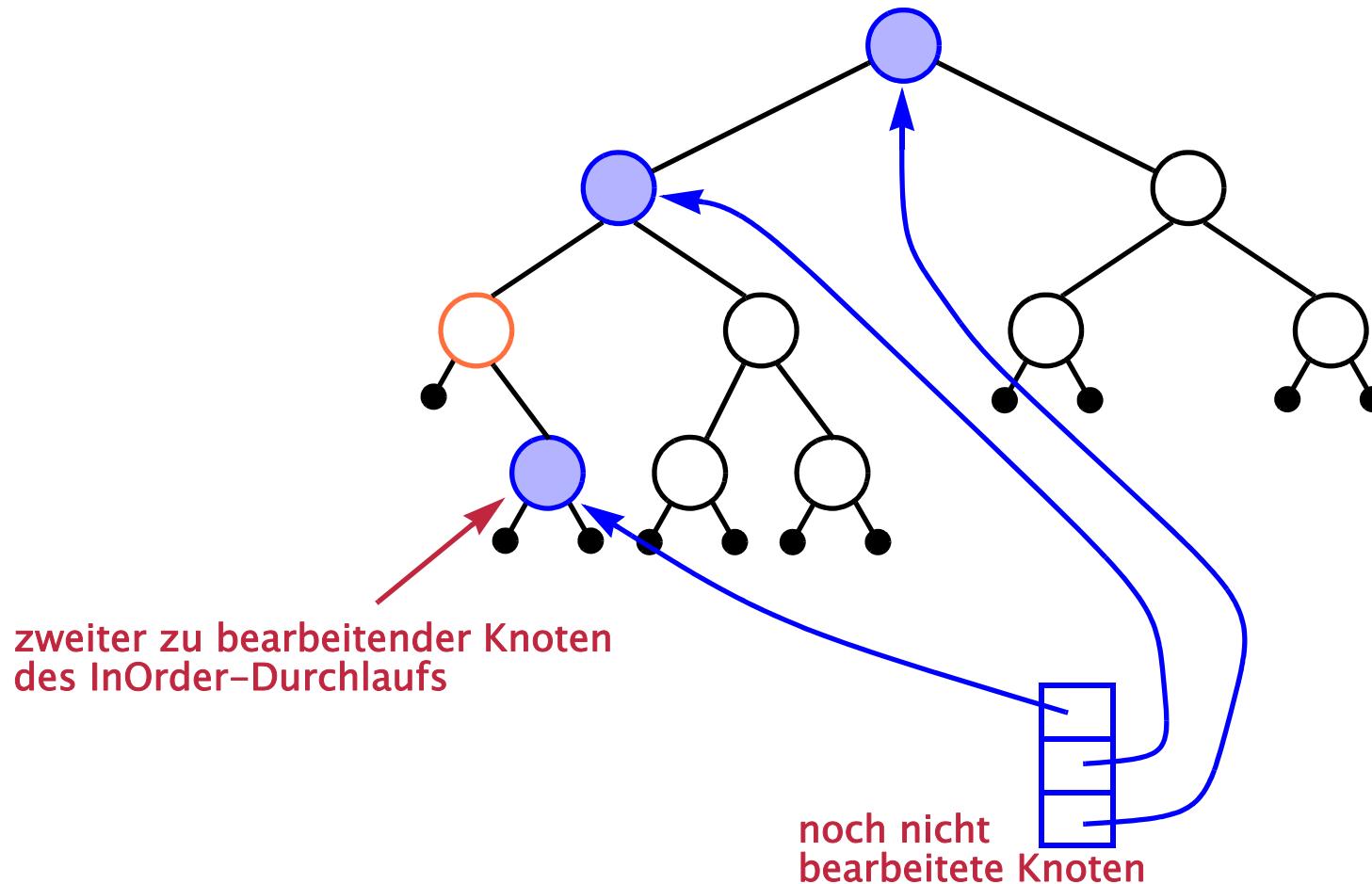
(Fortsetzung)



## Iterator für BinarySearchTree

Klasse StackBasedIterator

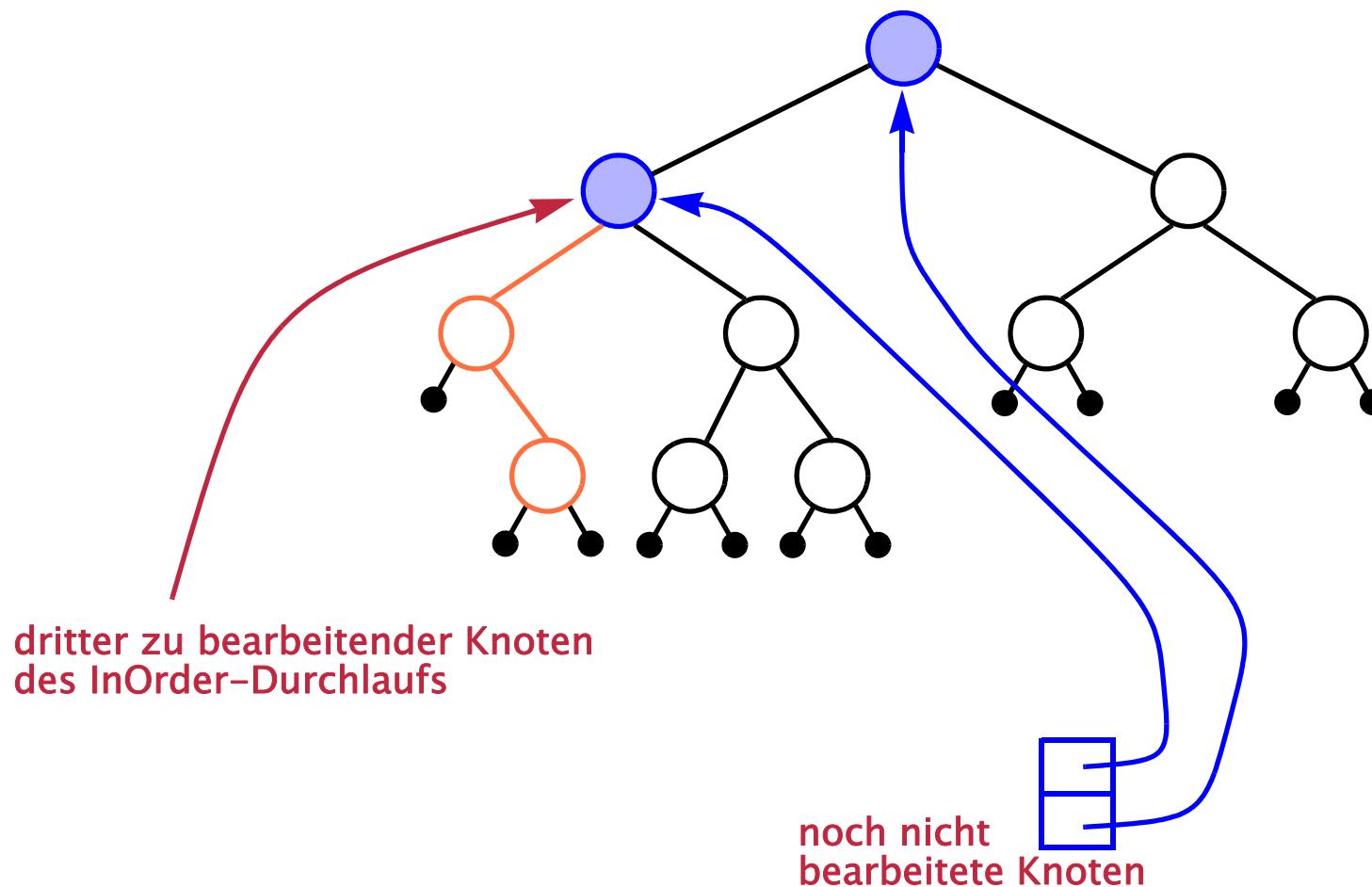
(Fortsetzung)



## Iterator für BinarySearchTree

Klasse StackBasedIterator

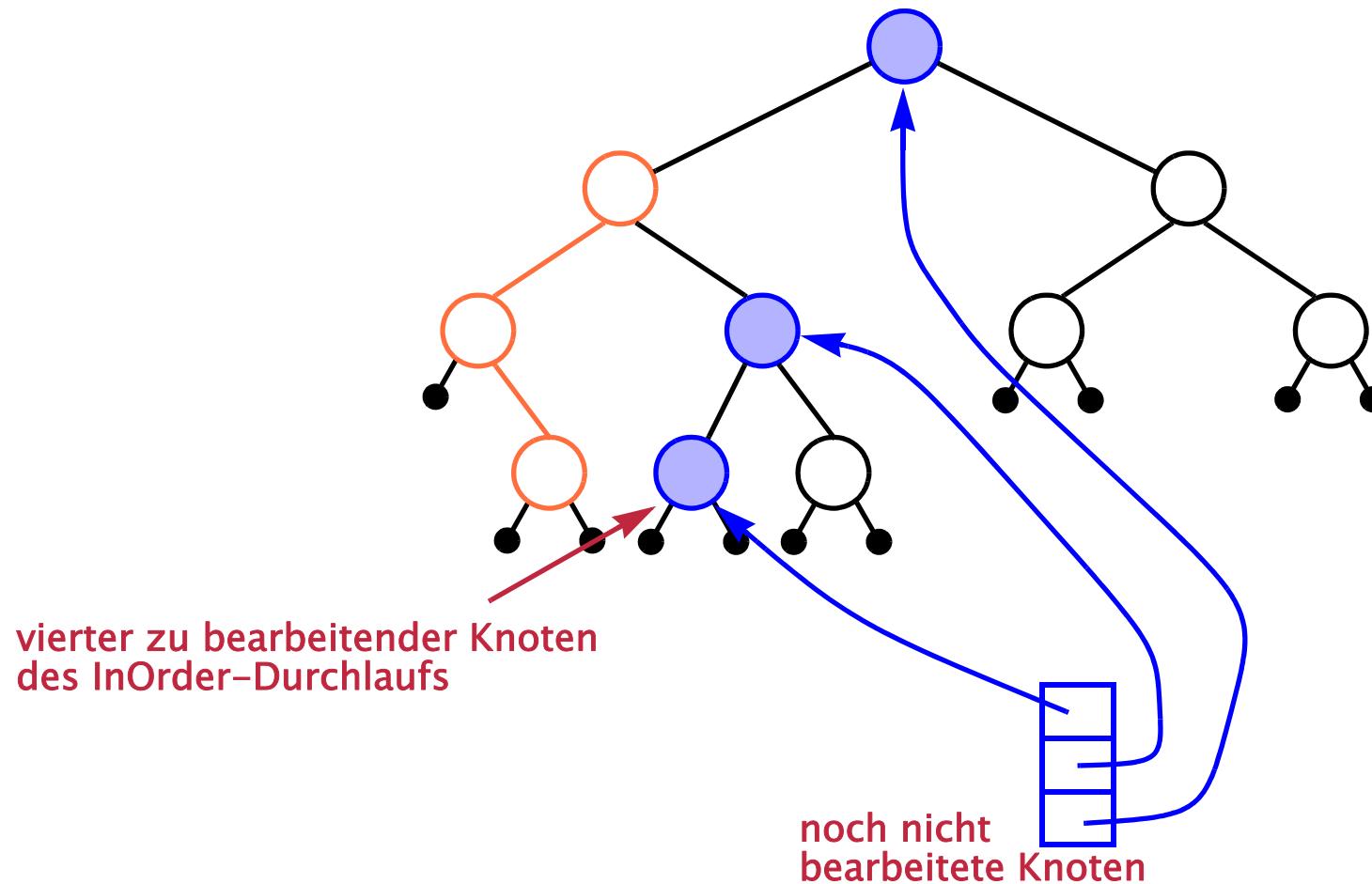
(Fortsetzung)



## Iterator für BinarySearchTree

Klasse StackBasedIterator

(Fortsetzung)



## Iterator für BinarySearchTree

Klasse StackBasedTreeIterator

(Fortsetzung)

Algorithmus:

- ❑ Es werden immer für den aktuell vom Iterator angebotenen Knoten alle *noch nicht bearbeiteten* Knoten abgespeichert, die auf dem Pfad liegen, der von dem aktuellen Knoten zur Wurzel führt.
- ❑ Wird ein Knoten durch den Iterator abgearbeitet, so wird sein *InOrder*-Nachfolger als nächster Knoten ausgewählt. Das ist der äußerste linke Knoten in seinem rechten Teilbaum.
- ❑ Gibt es keinen rechten Teilbaum, also keinen rechten Nachfolger, wird mit dem *noch nicht bearbeiteten* Vorgänger des abgearbeiteten Knotens fortgefahrene.

- ❑ Für die Implementierung dieses Algorithmus wird eine Datenstruktur benötigt, die es erlaubt, den *zuletzt* abgelegten Vorgänger als *ersten* weiterzubearbeiten.
- ❑ Eine solche Datenstruktur wird auch *Stapel (Stack)* oder *Kellerspeicher* genannt.  
Das Verhalten tritt beispielsweise bei einem Stapel von Büchern auf:  
Das zuletzt aufgelegte Buch wird als erstes wieder entfernt.
- ❑ Eine Klasse *Stack* lässt sich leicht durch die geeignete Nutzung der Klasse *DoublyLinkedList* implementieren. Jedes Objekt der Klasse *Stack* *delegiert* dabei die Speicherung von Objekten an ein Objekt der Klasse *DoublyLinkedList*.

## Stapel

Die Datenstruktur *Stapel* benötigt nur wenige Operationen:

- Ein Hinzunehmen von Inhalten auf den Stapel:

Methode `push`

- Ein Entfernen von Inhalten vom Stapel:

Methode `pop`

- Ein Inspizieren des obersten Inhalts des Stapels:

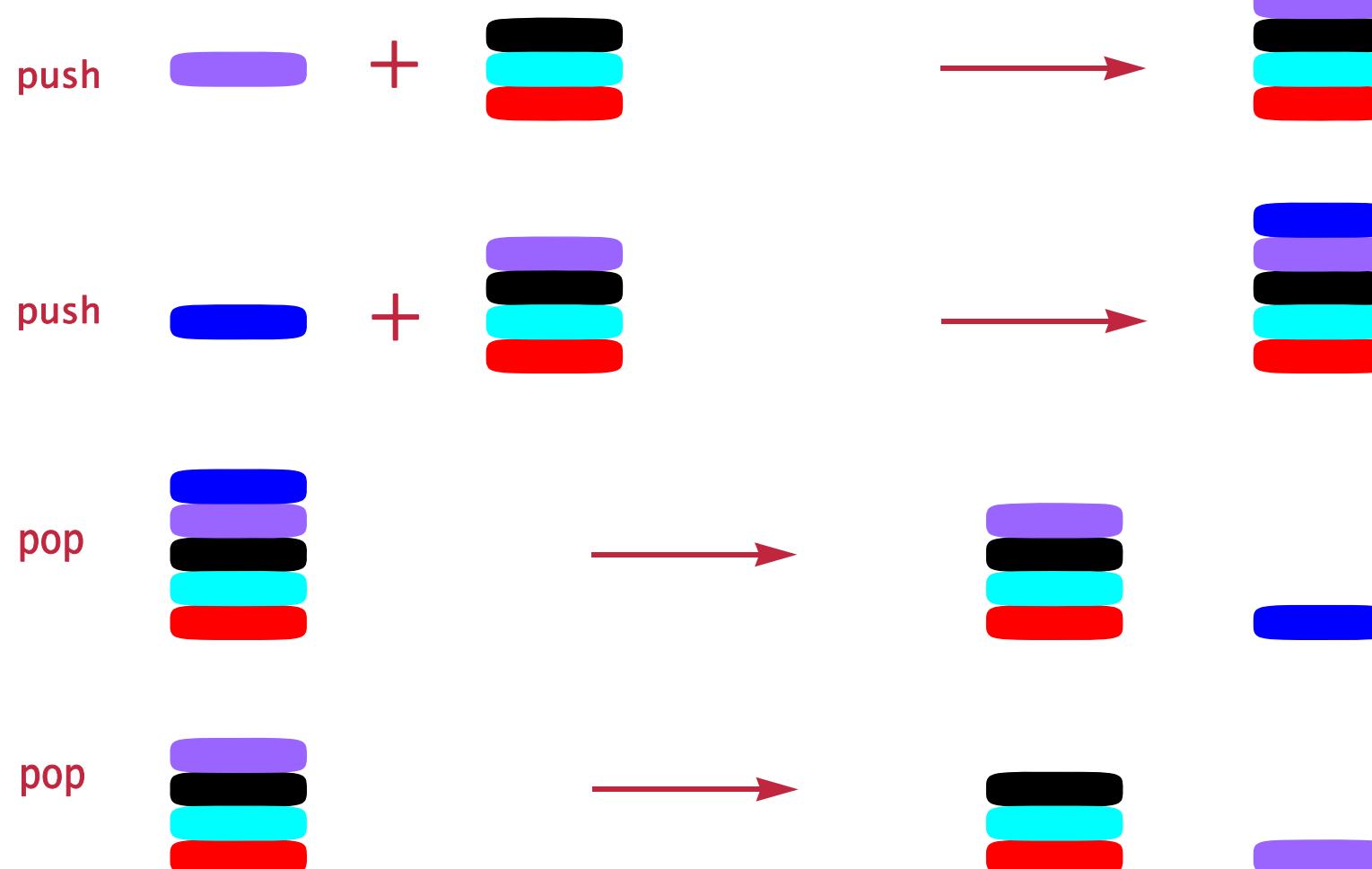
Methode `peek`

- Ein Prüfen auf Elemente im Stapel:

Methode `isEmpty`

# Stapel

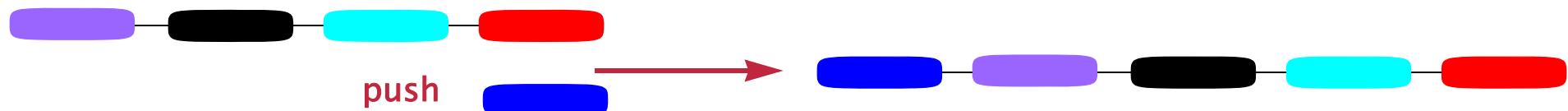
(Fortsetzung)



# Stack

Darstellung als Liste

(Fortsetzung)



## Stapel

(Fortsetzung)

```

public class Stack<T>
{
    private DoublyLinkedList<T> elements;

    public Stack() {
        elements = new DoublyLinkedList<T>();
    }

    public void push( T o ) {
        elements.addFirst( o );
    }

    public T pop() {
        return elements.removeFirst();
    }

    public T peek() {
        return elements.getFirst();
    }

    public boolean isEmpty() {
        return elements.isEmpty();
    }
}

```

The code is annotated with red arrows pointing from the right side to specific parts of the code:

- An arrow points to the line `private DoublyLinkedList<T> elements;` with the text "Liste als Attribut".
- An arrow points to the line `elements = new DoublyLinkedList<T>();` with the text "Liste wird erzeugt".
- An arrow points to the line `elements.addFirst( o );` with the text "push = vorne einfügen".
- An arrow points to the line `elements.removeFirst();` with the text "pop = vorne entnehmen".
- An arrow points to the line `elements.getFirst();` with the text "peek = vorne inspizieren".
- An arrow points to the line `elements.isEmpty();` with the text "Vorhandensein von Inhalt".

## Stapel

(Fortsetzung)

Warum ist die Klasse `Stack` *keine* Spezialisierung der Klasse `DoublyLinkedList`?

- ❑ Bei der Spezialisierung werden alle Methoden der Oberklasse in die Unterklasse übernommen.  
Bei der Spezialisierung ist ein Verzichten auf Methoden nicht möglich.
- ❑ Daher würde bei einer Spezialisierung die Klasse `Stack` auch Methoden wie `get` (siehe Folie 750) besitzen, die zur Verwaltung eines Stapels *unerwünscht* sind.
- ❑ Die Klasse `Stack` soll aufgrund ihrer andersartigen Verwaltung der gespeicherten Objekte auch gar nicht kompatibel zur Klasse `DoublyLinkedList` sein.
- ❑ Benutzung und Delegation sind also ein Weg, um
  - eine bereits vorhandene Klasse effizient zu nutzen,
  - Methoden der vorhandenen Klasse zu verbergen,
  - *keine* zwangsläufige Klassenhierarchie erstellen zu müssen.
- ❑ Soll eine vorhandene Implementierung wiederverwendet werden,  
*ohne* dass zugleich eine Hierarchie von Typen entsteht,  
ist *Delegation* häufig eine bessere Lösung als Spezialisierung (Vererbung).

## Iterator für BinarySearchTree

### Klasse StackBasedIterator

(Fortsetzung)

Implementierung der Klasse StackBasedIterator:

- Konstruktor:
  - legt einen leeren Stack an
  - sucht den äußersten linken Knoten des Baums – hier startet die Iteration –  
und legt alle auf dem Weg besuchten Knoten auf den Stack
- Methode `next`:
  - gibt den obersten Knoten vom Stack zurück
  - sucht den äußersten linken Knoten des rechten Teilbaums – hier geht die Iteration weiter –  
und legt alle auf dem Weg besuchten Knoten auf den Stack
- Methode `hasNext`:
  - überprüft, ob noch ein Knoten auf dem Stack liegt
  
- Methode `descendLeftAndPush`:
  - sucht den äußersten linken Knoten des (Teil-)Baums  
und legt alle auf dem Weg besuchten Knoten auf den Stack

## Iterator für BinarySearchTree

(Fortsetzung)

Klasse StackBasedIterator

```
private class StackBasedIterator extends Iterator<T>
{
    private Stack<BinarySearchTree<T>> nodes;
    public StackBasedIterator()
    {
        nodes = new Stack<BinarySearchTree<T>>();
        descendLeftAndPush( BinarySearchTree.this );
    }
    ...
}
```



- StackBasedIterator ist Iterator über dem Typ T.  
der Typparameter T ist in der inneren Klasse sichtbar.
- Auf dem Stapel sollen die Knoten eines binären Suchbaums abgelegt werden.  
Daher muss als Typargument BinarySearchTree<T> angegeben werden.

## Iterator für BinarySearchTree

Klasse StackBasedIterator

(Fortsetzung)

```
private class StackBasedIterator extends Iterator<T>
{
    private Stack<BinarySearchTree<T>> nodes;

    public StackBasedIterator()
    {
        nodes = new Stack<BinarySearchTree<T>>();
        descendLeftAndPush( BinarySearchTree.this );
    }
    ...
}
```

Zugriff auf  
umgebendes Objekt

- ❑ Das Argument für den ersten Aufruf der Methode `descendLeftAndPush` muss der ausführende Knoten der Klasse `BinarySearchTree` sein.
- ❑ Innerhalb der inneren Klasse `StackBasedIterator` bezeichnet `this` das den Zugriff umgebende Objekt der Klasse `StackBasedIterator`.
- ❑ Die Konstruktion `Klassename.this` ermöglicht den Zugriff von einem Objekt einer inneren Klasse aus zu dem zugehörigen Objekt der umgebenden Klasse.

## Iterator für BinarySearchTree

Klasse StackBasedIterator

(Fortsetzung)

```

...
private void descendLeftAndPush( BinarySearchTree<T> root )
{
    BinarySearchTree<T> current = root;
    while ( !current.isEmpty() )           ← Abbruch bei leerem Baum
    {
        nodes.push( current );
        current = current.leftChild;       ← Ablegen auf Stack
    }
}
...

```

- Die Methode `descendLeftAndPush` betrachtet alle Knoten, die auf dem äußersten linken Pfad liegen und legt diese auf dem Stack `nodes` ab.  
Dadurch können sie leicht in umgekehrter Reihenfolge zur Bearbeitung abgerufen werden.

## Iterator für BinarySearchTree

### Klasse StackBasedIterator

(Fortsetzung)

```
...
public T next()
{
    if ( hasNext() ) {
        T content = nodes.peek().getContent();
        descendLeftAndPush( nodes.pop().rightChild );
        return content;
    } else {
        throw new IllegalStateException();
    }
}
...
```

**Knoten vom Stack wählen und Inhalt sichern**

**äußerst linken Knoten im rechten Teilbaum suchen**

**Aufruf von next nicht zulässig, da alle Inhalte geliefert wurden**

## Iterator für BinarySearchTree

### Klasse StackBasedIterator

(Fortsetzung)

```
...
public boolean hasNext()
{
    return !nodes.isEmpty();
}
```

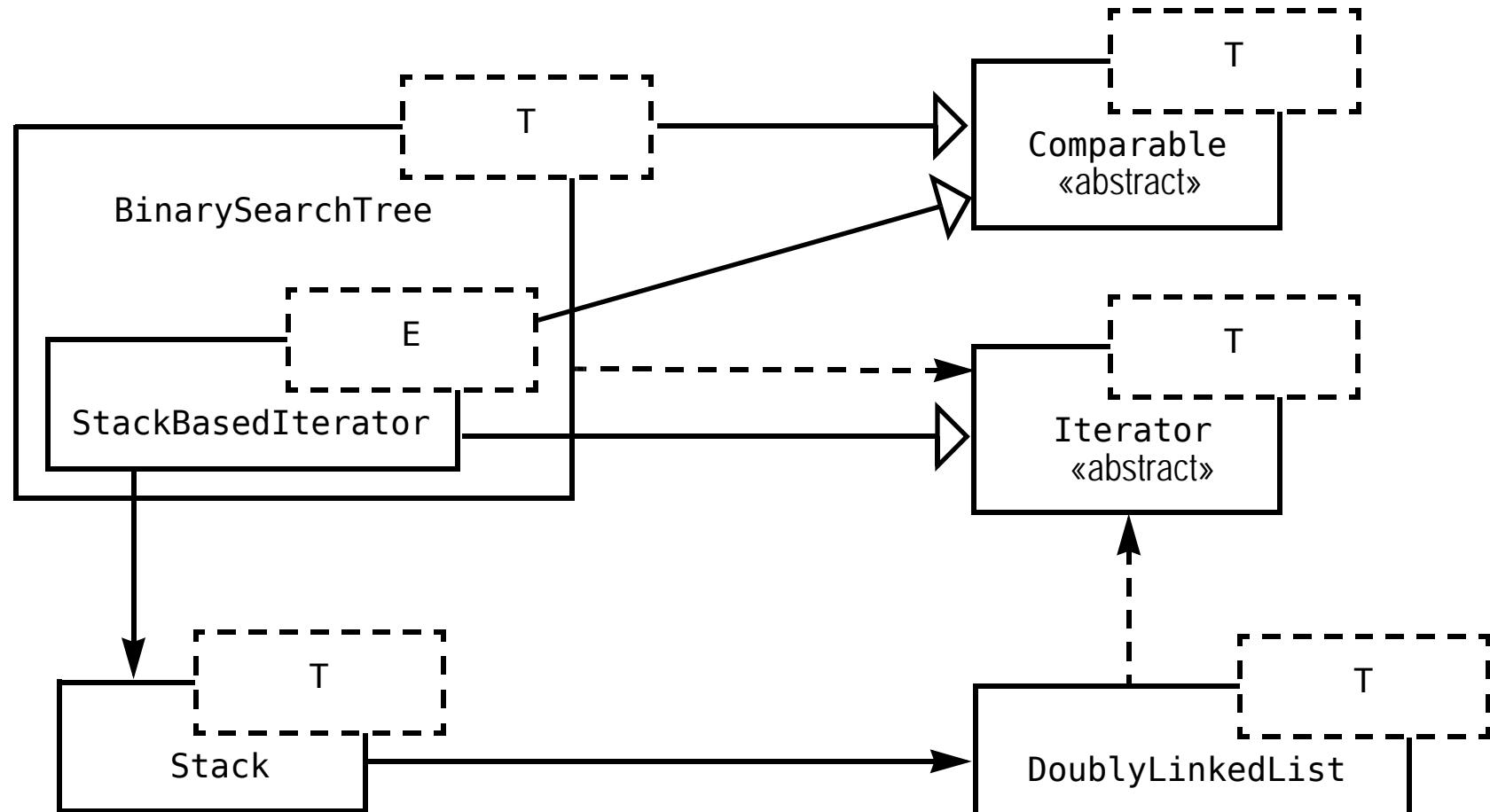
solange Knoten auf dem Stack liegen,  
kann weiter iteriert werden

- Mit der Klasse `StackBasedIterator` steht ein Iterator zur Verfügung, der mit geringem zusätzlichen Speicherbedarf einen über `next`-Aufrufe gesteuerten *InOrder*-Durchlauf ermöglicht.
- Dieser Iterator liefert auch solche Inhalte, die während des Iterierens eingefügt werden und einen größeren Inhalt als der aktuelle Knoten besitzen – also im InOrder-Durchlauf später angezeigt werden.
- Dieser Iterator berücksichtigt jedoch nicht das Löschen von Knoten aus dem Baum.
- Der Iterator arbeitet analog zum Umgang des Java-Laufzeitsystems mit Methoden-Aufrufen:  
dort werden die lokalen Variablen der unterbrochenen Instanzen der Methoden auf einem Stack abgelegt

## Generische Klasse `BinarySearchTree<T extends Comparable<T>>`

(Fortsetzung)

Visualisierung der Klassenstruktur



# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 13. Interface

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-917

## Lernziele des Kapitels 13. Interface

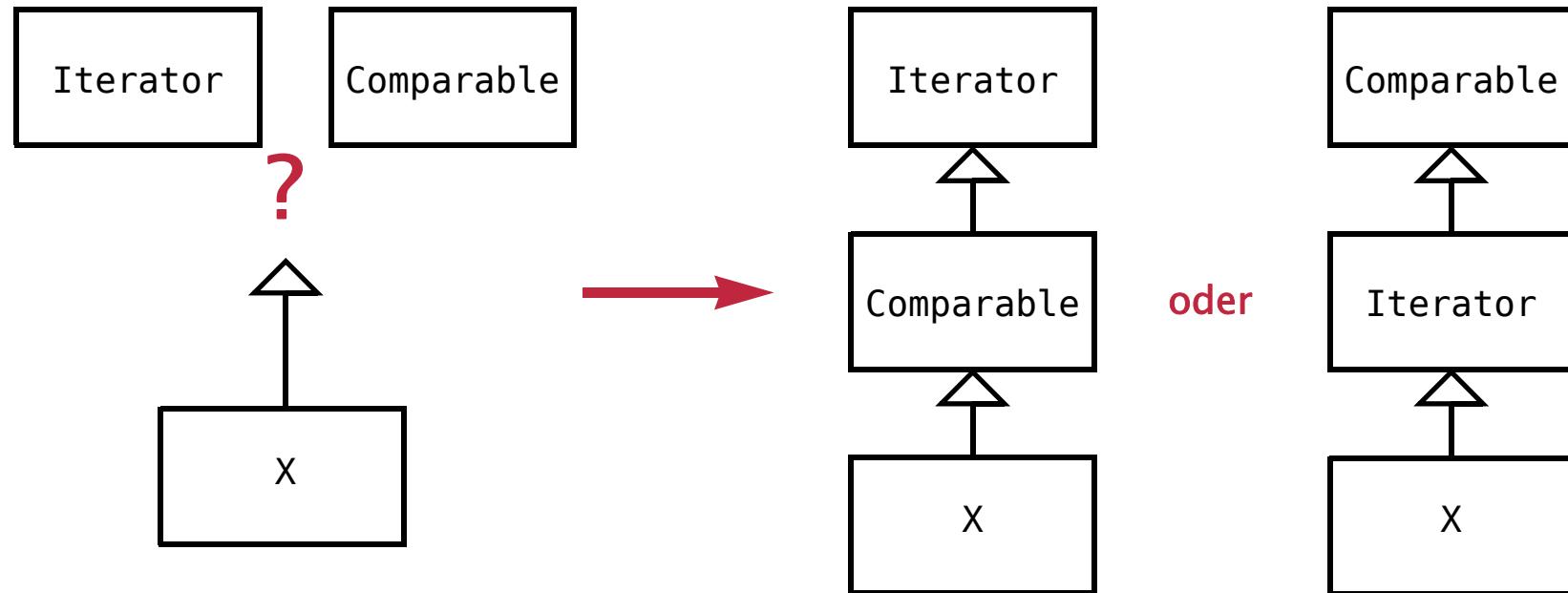
Nach Durcharbeiten des Kapitels Interface sollen die teilnehmenden Studierenden

- das Konstrukt des Interfaces kennen,
- Interfaces und Hierarchien von Interfaces konzipieren und programmieren können,
- Interfaces durch Klassen implementieren können,
- Interfaces und (abstrakte) Klasse gegeneinander abgrenzen können,
- generische Methoden deklarieren und einsetzen können.

# Typkompatibilität und Klassenhierarchie

Problem:

In Java kann jede Klasse nur genau eine Oberklasse besitzen.  
Soll eine Klassenhierarchie eine Klasse enthalten,  
die zu mehreren Klassen kompatibel sein soll,  
so müssen diese selbst wiederum in einer Spezialisierungsbeziehung stehen.

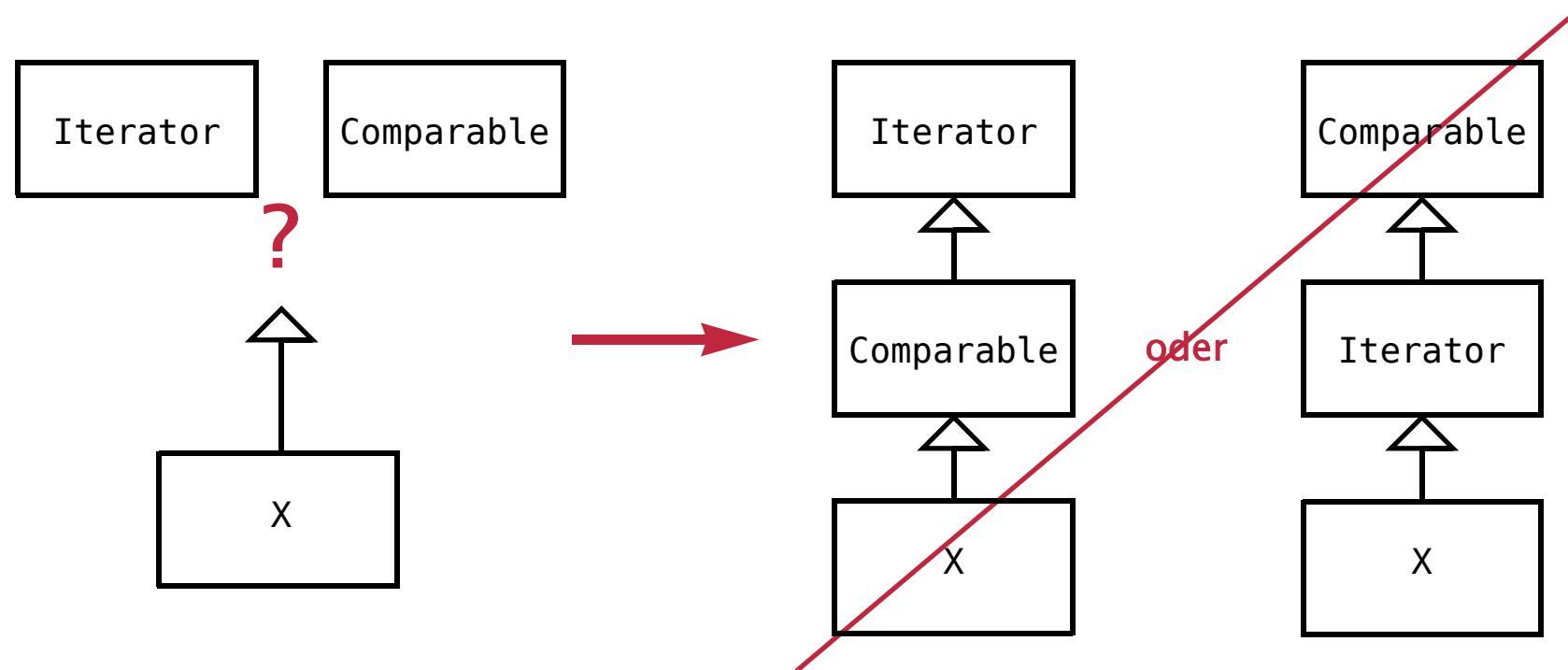


## Typkompatibilität und Klassenhierarchie

(Fortsetzung)

Diese Lösung ist aber *untauglich*,

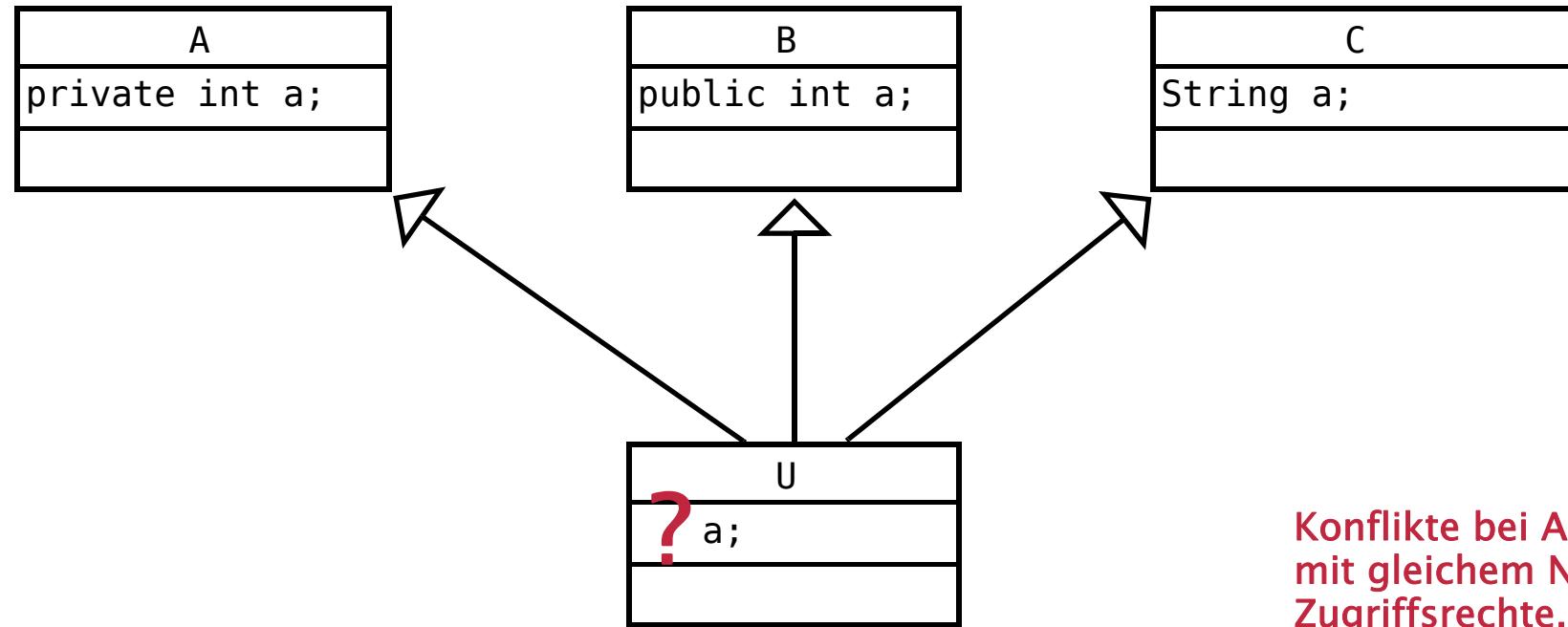
- ❑ wenn es viele Kombinationen von Typkompatibilitäten gibt oder
- ❑ wenn die Klassenhierarchie nicht angepasst werden kann, da die Implementierungen der betroffenen Klassen nicht verfügbar sind.



## Spezialisierung in Java

Warum erlaubt Java für jede Klasse nur *genau* eine Oberklasse?

Es werden so Probleme vermieden,  
die mehrfaches Erben (*multiple inheritance*) mit sich bringen würde:

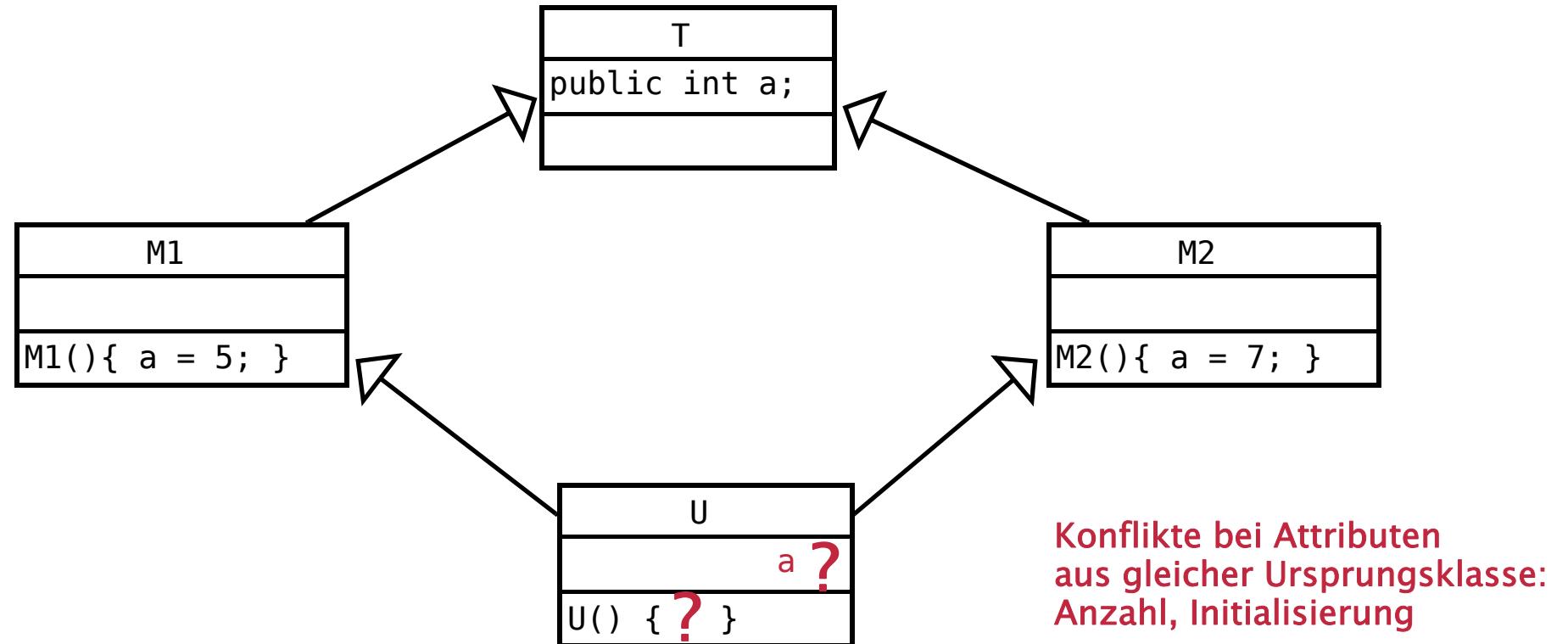


## Spezialisierung in Java

(Fortsetzung)

Warum erlaubt Java für jede Klasse nur *genau* eine Oberklasse?

Es werden so Probleme vermieden,  
die mehrfaches Erben (*multiple inheritance*) mit sich bringen würde:

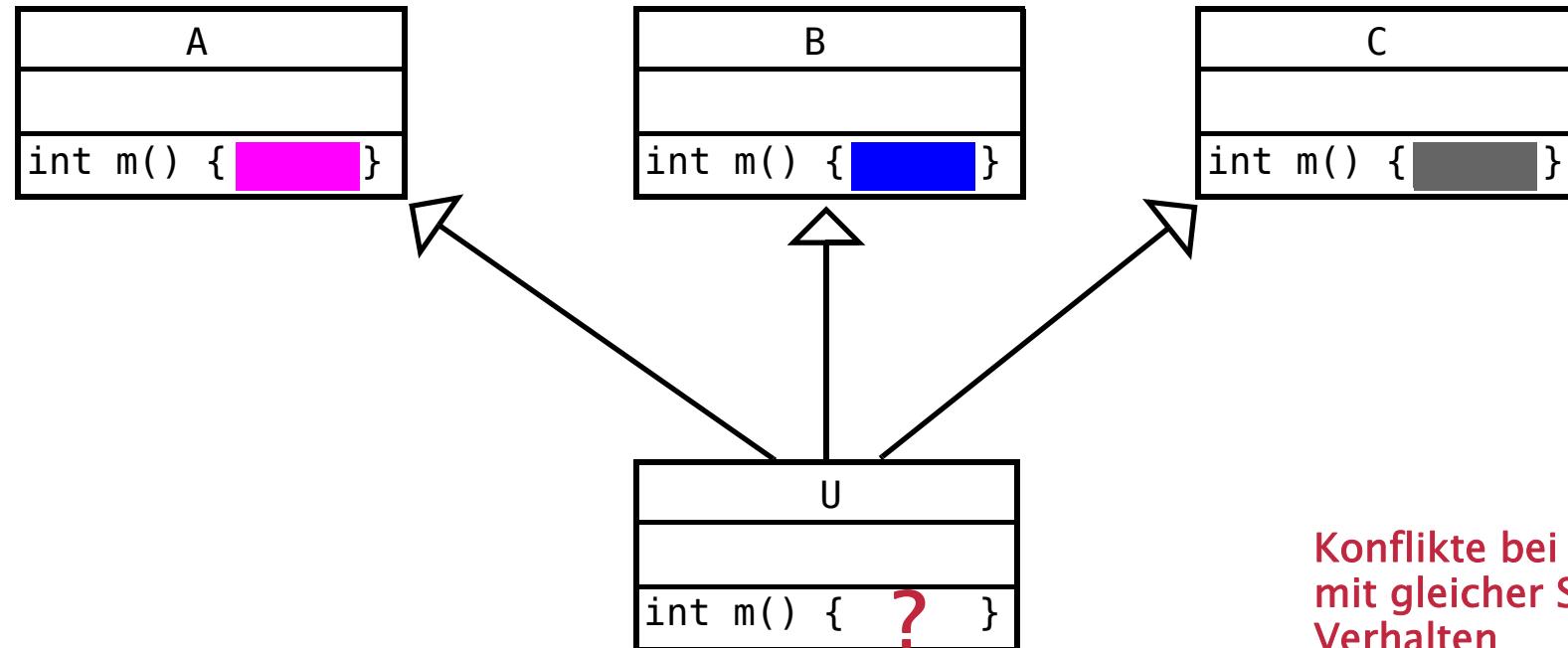


## Spezialisierung in Java

(Fortsetzung)

Warum erlaubt Java für jede Klasse nur *genau* eine Oberklasse?

Es werden so Probleme vermieden,  
die mehrfaches Erben (*multiple inheritance*) mit sich bringen würde:



## Spezialisierung in Java

Warum erlaubt Java für jede Klasse nur *genau* eine Oberklasse?

Es werden so Probleme vermieden,  
die mehrfaches Erben (*multiple inheritance*) mit sich bringen würde:

- Es könnten Attribute gleichen Namens mit unterschiedlichen Typen und Zugriffsrechten aus den Oberklassen geerbt werden.
- Es könnten Attribute von mehreren Oberklassen geerbt, die einen gemeinsamen Ursprung in einer gemeinsamen Oberklasse haben.
- Alle Oberklassen würden Konstruktoren liefern, die im Konstruktor der UnterkLASSE ausgeführt werden müssen.
- Es könnten Methoden mit gleichen Signaturen und unterschiedlichen Implementierungen aus den Oberklassen geerbt werden.

## Rückblick: Klassen Iterator, Comparable und SubstitutionStrategy

```
public abstract class Iterator<T> {  
    public abstract boolean hasNext();  
    public abstract T next();  
}  
  
public abstract class Comparable<T> {  
    public abstract int compareTo( T t );  
}  
  
public static abstract class SubstitutionStrategy<E> {  
    public abstract E substitute( E ref );  
}
```

- ❑ Alle drei Klassen sind abstrakt und enthalten ausschließlich abstrakte Methoden.
- ❑ Durch die Spezialisierung dieser Klassen sollen Klassenhierarchien geschaffen werden, in denen für die Unterklassen bestimmte *Eigenschaften* (in Form von Methoden) garantiert werden können.
- ❑ Offensichtlich ist es nützlich für die Gestaltung von vielseitig verwendbaren Klassen, wenn Klassen wie `Iterator`, `Comparable` oder `SubstitutionStrategy` aber keine weiteren Vorgaben zur Umsetzung dieser Eigenschaften machen.

## Spezialisierung in Java

(Fortsetzung)

Einsatzziele des Spezialisierungskonzepts

	Typkompatibilität	Wiederverwendung
<input type="checkbox"/> normale Klasse (alle Methoden implementiert)	X	X
<input type="checkbox"/> abstrakte Klasse (einige Methoden implementiert)	X	X
<input type="checkbox"/> <i>ausschließlich</i> abstrakte Klasse ( <i>keine</i> Methode implementiert)	X	

## Spezialisierung in Java

(Fortsetzung)

Einsatzziele des Spezialisierungskonzepts

	Typkompatibilität	Wiederverwendung
<input type="checkbox"/> normale Klasse (alle Methoden implementiert)	X	<input type="checkbox"/>
<input type="checkbox"/> abstrakte Klasse (einige Methoden implementiert)	X	<input type="checkbox"/> <i>Probleme bei mehrfachem Erben</i>
<input type="checkbox"/> <i>ausschließlich</i> abstrakte Klasse ( <i>keine</i> Methode implementiert)	X	<input type="checkbox"/>

## Spezialisierung in Java

(Fortsetzung)

Einsatzziele des Spezialisierungskonzepts

	Typkompatibilität	Wiederverwendung
<input type="checkbox"/> normale Klasse (alle Methoden implementiert)	X	X
<input type="checkbox"/> abstrakte Klasse (einige Methoden implementiert)	X	X
<input type="checkbox"/> <i>ausschließlich</i> abstrakte Klasse ( <i>keine</i> Methode implementiert)	X	

Java bietet ein anderes Konzept an, um

- die Vorteile des mehrfachen Erbens beim Herstellen von Typkompatibilitäten zu ermöglichen und
- die Nachteile des mehrfachen Erbens von *konkreten* Eigenschaften weitgehend zu vermeiden.

## Interface

Java bietet ein anderes Konzept an: *Interface*

- Ein *Interface* entspricht *ungefähr* einer Klasse, die keine Attribute enthält. \*)
- Ein *Interface* enthält also ausschließlich Methoden.
- Methoden müssen einen der Modifizierer **abstract**, **default** oder **static** besitzen.  
Der Modifizierer **abstract** kann bei der Deklaration entfallen.
- Deklarationen von Methoden müssen öffentlich sein, so dass jede in einem Interface deklarierte Methode immer das Zugriffsrecht **public** besitzen muss.  
Der Modifizierer **public** kann bei der Deklaration entfallen.
- Aus Interfaces können eigene Spezialisierungshierarchien gebildet werden.
- Ein Interface kann mehrere Interfaces spezialisieren – also «mehrfach erben».
- Eine Klasse kann mehrere Interfaces spezialisieren – also «mehrfach erben».
- Das Spezialisieren eines Interfaces durch eine Klasse wird als *Implementierung* bezeichnet.

\*) Das stimmt nicht ganz: Ein Interface kann auch Deklarationen von konstanten Attributen enthalten.  
Dieser Aspekt wird später kurz betrachtet.

## Interface

(Fortsetzung)

Warum erlaubt Java für Interfaces mehrfaches Spezialisieren/Erben?

Die bei Klassen für mehrfaches Erben genannten Probleme treten im Zusammenhang mit Interfaces weitgehend *nicht* auf:

- Es können Attribute gleichen Namens mit unterschiedlichen Typen und Zugriffsrechten aus den Oberklassen geerbt werden.  
**Ein Interface besitzt keine Attribute.**
- Es können Attribute von mehreren Oberklassen geerbt, die einen gemeinsamen Ursprung in einer gemeinsamen Oberklasse haben.  
**Ein Interface besitzt keine Attribute.**
- Alle Oberklassen liefern Konstruktoren, die im Konstruktor der Unterkasse ausgeführt werden müssen.  
**Zu einem Interface können keine Objekte erzeugt werden; es besitzt keinen Konstruktor.**
- Es können Methoden mit gleichen Signaturen und unterschiedlichen Implementierungen aus den Oberklassen geerbt werden.  
**Abstrakte Methoden besitzen keine Implementierung.**  
**Für default- und static-Methoden gelten besondere Regeln.**

## Interface

(Fortsetzung)

Beispiele für die Syntax mit abstrakten Methoden

```
public interface I1
{
    void m();
}
```

```
public interface I3
{
    int g( int i );
    void m();
}
```

```
public interface I2
{
    int f();
}
```

Schlüsselwort: **interface**

- Die Deklaration der Methoden kann *ohne* Modifizierer erfolgen:

**void m();** steht in einem Interface für **public abstract void m();**

## Interface

(Fortsetzung)

Beispiele für die Syntax mit abstrakten Methoden

```
public interface I1
{
    void m();
}
```

```
public interface I2
{
    int f();
}
```

```
public interface I3
{
    int g( int i );
    void m();
}
```

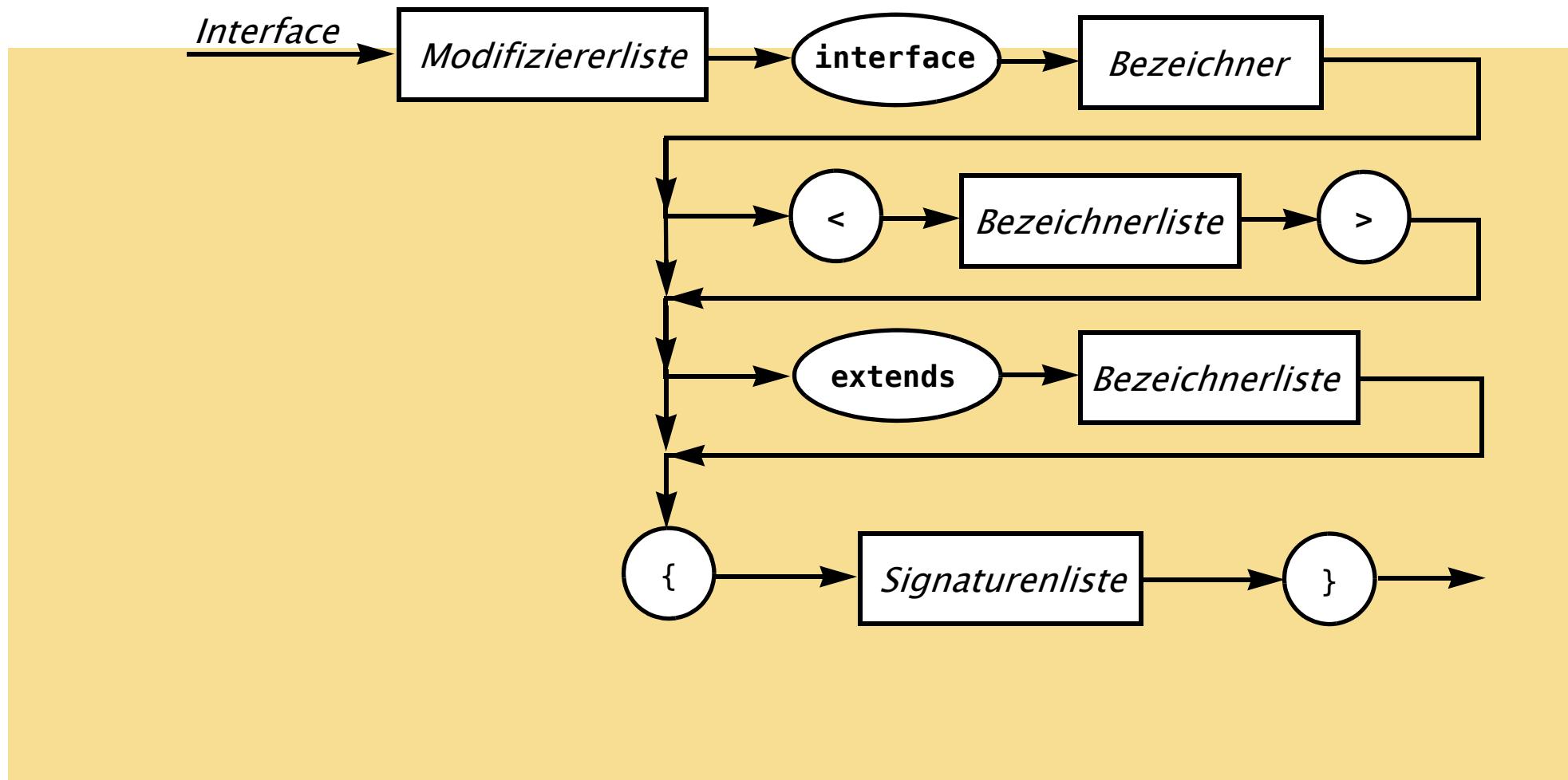
```
public interface Both extends I1, I2
{
    double h( double p );
}
```



Spezialisierung von  
Interfaces: **extends**

- Das Interface Both erweitert die *beiden* Interfaces I1 und I2:  
Klassen, die Both implementieren, müssen also drei Methoden anbieten:
  - **void m()** Deklaration in Interface I1
  - **int f()** Deklaration in Interface I2
  - **double h( double p )** Deklaration in Interface Both

## Syntaxdiagramm zu *Interface*



## Interface

(Fortsetzung)

Beispiele für die Syntax mit abstrakten Methoden

```
public interface I1
{
    void m();
}
```

```
public interface I2
{
    int f();
}
```

```
public interface I3
{
    int g( int i );
    void m();
}
```

```
public interface Both extends I1, I2
{
    double h( double p );
}
```

```
public class C implements Both, I3
{
    public void m() { ... }
    public int f() { ... }
    public int g( int i ) { ... }
    public double h( double p ) { ... }
}
```

Realisierung von Interfaces durch eine Klasse:  
**implements**

Die Klasse muss alle vier Methoden aus I1, I2,  
I3 und Both implementieren.  
m() aus I1 und m() aus I3 können dabei  
nicht unterschieden werden

## Interface

(Fortsetzung)

Beispiele für die Syntax mit abstrakten Methoden

```
public interface I1
{
    void m();
}
```

```
public interface I2
{
    int f();
}
```

```
public interface I3
{
    int g( int i );
    void m();
}
```

```
public interface Both extends I1, I2
{
    double h( double p );
}
```

```
public class C implements Both, I3
{
    public void m() { ... }
    public int f() { ... }
    public int g( int i ) { ... }
    public double h( double p ) { ... }
}
```

Modifizierer in Klasse notwendig: **public**

## Interface

(Fortsetzung)

Beispiele für die Syntax mit abstrakten Methoden

```
public interface I1
{
    void m();
}
```

```
public interface I2
{
    int f();
}
```

```
public interface I3
{
    int g( int i );
    void m();
}
```

```
public interface Both extends I1, I2
{
    double h( double p );
}
```

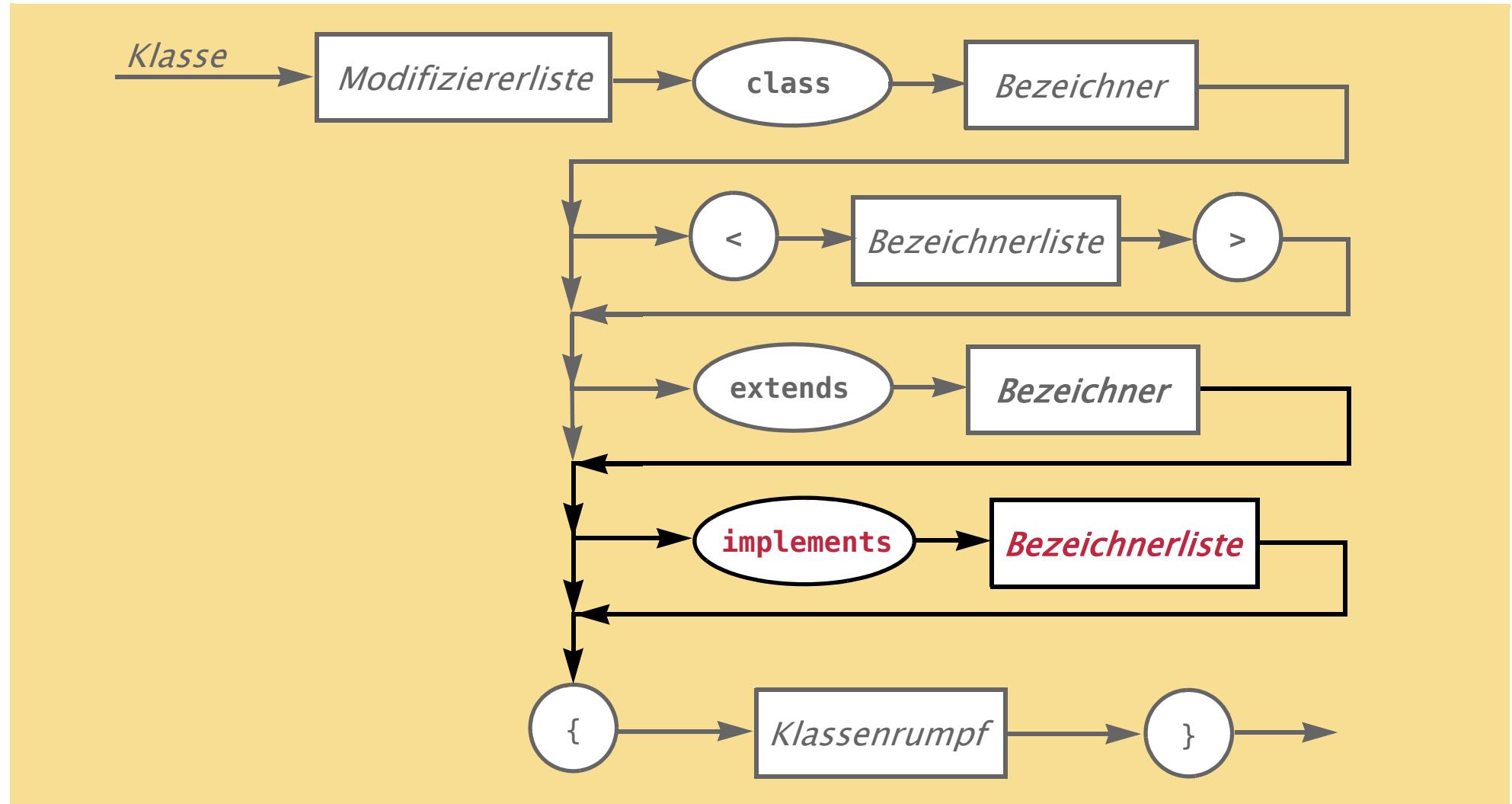
```
public abstract class A
implements Both, I3
{
    public void m() { ... }
    public int f() { ... }
}
```

alternativ:

Eine abstrakte Klasse kann von Interface vorgeschriebene Implementierung auf eigene Unterklassen verschieben.

Die Implementierungen von g und h fehlen noch.

## Syntaxdiagramm zu *Klasse*



## Einsatz des Konzepts Interface am Beispiel der Klasse DoublyLinkedList<T>

Die bei der Deklaration von DoublyLinkedList verwendeten abstrakten Klassen werden durch passende Interfaces ersetzt.

```
public class DoublyLinkedList<T>
implements Iterable<T> { ←
    ...
    public Iterator<T> iterator() { ... }

    private abstract class ListIterator
    implements Iterator<T> { ... }

    private class ForwardIterator extends ListIterator { ... }

    public static interface SubstitutionStrategy<T> { ... }
    public static interface InspectionStrategy<T> { ... }
    public static interface DeletionStrategy<T> { ... }

    private class Element { ... }
}

public interface Iterator<E> { ... }

public interface Iterable<E> { ... } ← wird neu ergänzt
```

## Beispiel: Interfaces für Iteratoren

- Die Syntax von generischen Interfaces entspricht der von generischen Klassen.  
Das Interface wird analog zu der bekannten Klasse deklariert:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
}
```

- Die Umsetzung eines Interfaces durch eine Klasse wird erzwungen durch die Angabe von **implements**:

```
private abstract class ListIterator
implements Iterator<T>
{
    public boolean hasNext() { ... }
    public T next() { ... }

    ...
}
```

## Beispiel: Interfaces für Iteratoren

(Fortsetzung)

Deklaration eines weiteren Interfaces, das Klassen charakterisiert und so universell nutzbar macht, die einen Iterator anbieten:

```
public interface Iterable<E> {  
    Iterator<E> iterator();  
}
```

Klassen, die einen Iterator anbieten, können unmittelbar erkannt werden.

```
public class DoublyLinkedList<T>  
implements Iterable<T> {  
    ...  
    public Iterator<T> iterator()  
    {  
        return new ForwardIterator();  
    }  
    ...  
}
```

Der Iterator wird immer über die gleiche Methode bereitgestellt, die Nutzung ist uniform.

```
public class BinarySearchTree<T extends Comparable<T>>  
implements Iterable<T> {  
    ...  
}
```

## Generische Methoden

Nun können auch Methoden deklariert werden, die auf allen Datenstrukturen arbeiten, die das Interface `Iterable` umsetzen:

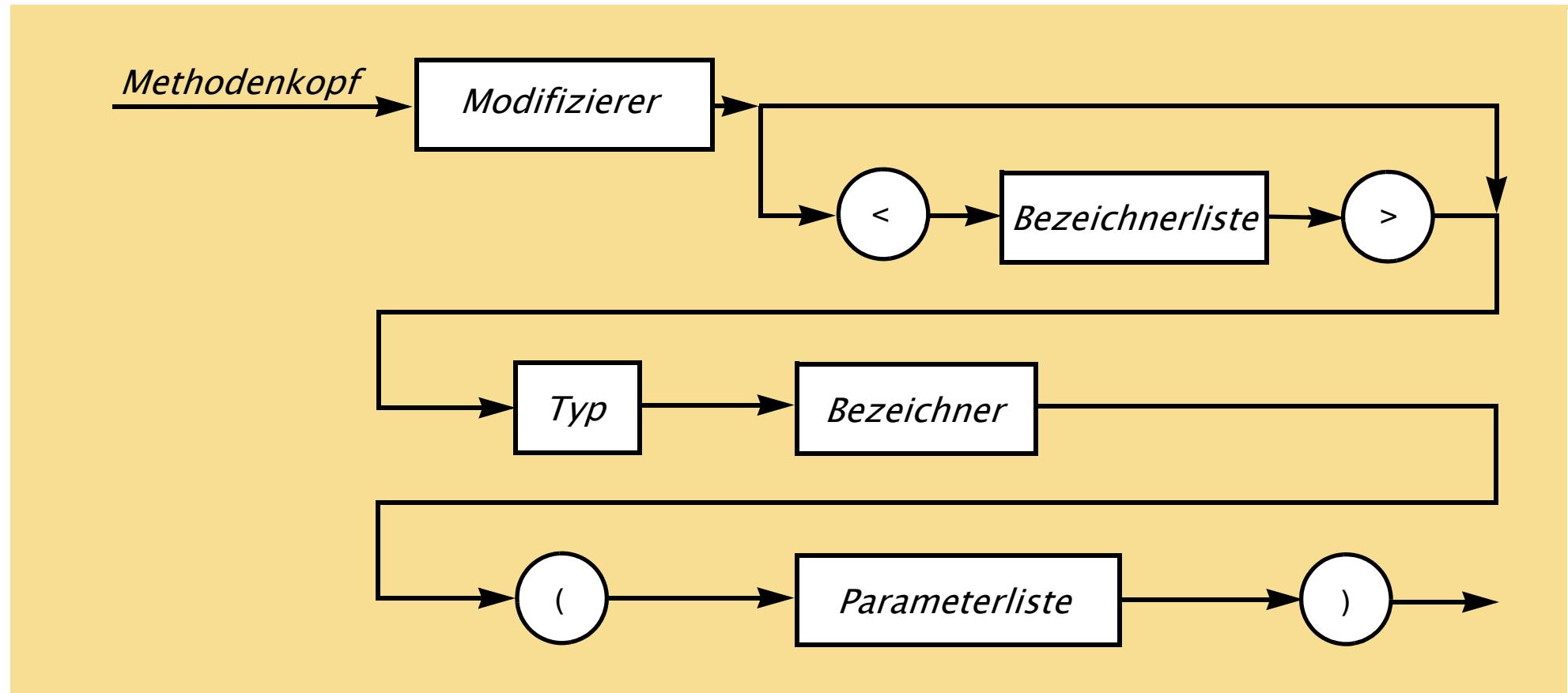
```
public static <F> void printData( Iterable<F> toIter )
{
    Iterator<F> it = toIter.iterator();
    while ( it.hasNext() )
    {
        System.out.println( it.next() );
    }
}
```

Typparameter  
für `printData`

- ❑ `printData` ist eine *generische Methode*, die ihren *eigenen* Typparameter `F` besitzt. \*)
- ❑ Der Typparameter wird zwischen den Modifizierern und dem Rückgabetyp angeführt.
- ❑ `printData` gibt den Inhalt jeder Datenstruktur aus, die das Interface `Iterable` implementiert.
- ❑ `Iterable` zeigt an, dass die Datenstruktur eine Methode `iterator()` besitzt, die ein `Iterator<F>`-Objekt liefert.

\*) Die Methode `printData` muss nicht innerhalb einer generischen Klasse deklariert werden.

## Syntaxdiagramm zu *Methodenkopf*



## Generische Methoden

(Fortsetzung)

```
DoublyLinkedList<Integer> ints = new DoublyLinkedList<>();
ints.add( 15 );
...
printData( ints );

BinarySearchTree<Student> students = new BinarySearchTree<>();
students.add( new Student( "B", "Inf", 18 ) );
...
printData( students );
```

- ❑ Die Methode `printData` ist außerhalb der Klassen `DoublyLinkedList` und `BinarySearchTree` deklariert.
- ❑ Die Methode `printData` kann mit Argumenten von allen Klassen arbeiten, die das Interface `Iterable` implementieren.
- ❑ Das Typargument für den Typparameter `F` wird von Java *implizit* aus dem der Methode `printData` übergebenen Argument bestimmt:
  - aus `printData( ints )` wird für `F` das Typargument `Integer` bestimmt,
  - aus `printData( students )` wird für `F` das Typargument `Student` bestimmt.

## Interface - **default**-Methoden \*)

Das Interface `Iterable` wird von der Java-Klassenbibliothek bereitgestellt und genutzt. \*\*)

- ❑ Viele (fertige) Klassen implementieren das Interface `Iterable`.
- ❑ Ein Hinzunehmen von weiteren (abstrakten) Methoden würde alle (fertigen) implementierenden Klassen in abstrakte Klassen verwandeln, da sie keine Realisierungen für die neu hinzugekommenen (abstrakten) Methoden besitzen.

\*) `default`-Methoden sind ein Sprachkonzept seit Java 8.

\*\*) siehe: <https://docs.oracle.com/en/java/javase/>

## Interface - **default**-Methoden \*)

Das Interface `Iterable` wird von der Java-Klassenbibliothek bereitgestellt und genutzt. \*\*)

- ❑ Viele (fertige) Klassen implementieren das Interface `Iterable`.
- ❑ Ein Hinzunehmen von weiteren (abstrakten) Methoden würde alle (fertigen) implementierenden Klassen in abstrakte Klassen verwandeln, da sie keine Realisierungen für die neu hinzugekommenen (abstrakten) Methoden besitzen.

Lösung:

Ein Interface darf **default**-Implementierungen von Methoden besitzen, so dass in den implementierenden Klassen keine Realisierung erzwungen wird.

\*) `default`-Methoden sind ein Sprachkonzept seit Java 8.

\*\*) siehe: <https://docs.oracle.com/en/java/javase/>

## Interface - **default**-Methoden \*)

Das Interface `Iterable` wird von der Java-Klassenbibliothek bereitgestellt und genutzt. \*\*)

- ❑ Viele (fertige) Klassen implementieren das Interface `Iterable`.
- ❑ Ein Hinzunehmen von weiteren (abstrakten) Methoden würde alle (fertigen) implementierenden Klassen in abstrakte Klassen verwandeln, da sie keine Realisierungen für die neu hinzugekommenen (abstrakten) Methoden besitzen.

Lösung:

Ein Interface darf **default**-Implementierungen von Methoden besitzen, so dass in den implementierenden Klassen keine Realisierung erzwungen wird.

Probleme:

Es können jetzt Konflikte auftreten, wenn Methoden mit gleicher Signatur und unterschiedlicher **default**-Implementierung aus verschiedenen Interfaces geerbt werden.

\*) **default**-Methoden sind ein Sprachkonzept seit Java 8.

\*\*) siehe: <https://docs.oracle.com/en/java/javase/>

## Interface - default-Methoden

(Fortsetzung)

Geänderte Deklaration des Interfaces Iterable (seit Java 8):

```
public interface Iterable<T>
{
    Iterator<T> iterator();

    default void forEach( ... )
    { ... }

    default Spliterator<T> spliterator()
    { ... }
}
```

bereits bekannte Methode:  
erzeugt ein Iterator-Objekt

forEach führt eine Aktion für  
jedes Element der iterierbaren  
Struktur aus (vgl. Strategiemuster)

spliterator erzeugt ein  
spliterator-Objekt  
(``Iterator`` für parallele Verarbeitung)

- ❑ Da ein Interface keine Attribute besitzen kann, müssen auch die Implementierungen der beiden Methoden `forEach` und `spliterator` ohne Attribute auskommen.

## Interface - **default**-Methoden

(Fortsetzung)

Wie werden die entstehenden Konflikte gelöst, wenn Methoden mit gleicher Signatur und unterschiedlicher **default**-Implementierung aus verschiedenen Interfaces geerbt werden?

- Eine Klasse implementiert ein Interface und überschreibt dabei eine geerbte **default**-Methode:  
Es wird die in der Klasse deklarierte Methode ausgeführt.
- Ein Interface erweitert ein Interface und überschreibt dabei eine geerbte **default**-Methode:  
Es wird die in dem erweiternden Interface deklarierte Methode in der Vererbungs-/Implementierungshierarchie weitergegeben.

**Verhalten bei einfachen Erbungsstrukturen:  
konform zu den bisher bekannten Regeln**

## Interface - **default**-Methoden

(Fortsetzung)

Wie werden die entstehenden Konflikte gelöst, wenn Methoden mit gleicher Signatur und unterschiedlicher **default**-Implementierung aus verschiedenen Interfaces geerbt werden?

- Eine Klasse implementiert ein Interface und überschreibt dabei eine geerbte **default**-Methode:  
Es wird die in der Klasse deklarierte Methode ausgeführt.
- Ein Interface erweitert ein Interface und überschreibt dabei eine geerbte **default**-Methode:  
Es wird die in dem erweiternden Interface deklarierte Methode in der Vererbungs-/Implementierungshierarchie weitergegeben.
- Eine Klasse implementiert ein Interface und erweitert eine Klasse und erbt dabei von beiden Vorgängertypen eine **default**-Methode mit gleicher Signatur:  
Es wird die in der Oberklasse deklarierte Methode ausgeführt.

**Verhalten bei *gemischter* Erbungsstruktur:**  
**Implementierung der (einzigsten) Oberklasse wird ausgewählt.**

## Interface - **default**-Methoden

(Fortsetzung)

Wie werden die entstehenden Konflikte gelöst, wenn Methoden mit gleicher Signatur und unterschiedlicher **default**-Implementierung aus verschiedenen Interfaces geerbt werden?

- Eine Klasse implementiert ein Interface und überschreibt dabei eine geerbte **default**-Methode:  
Es wird die in der Klasse deklarierte Methode ausgeführt.
- Ein Interface erweitert ein Interface und überschreibt dabei eine geerbte **default**-Methode:  
Es wird die in dem erweiternden Interface deklarierte Methode in der Vererbungs-/Implementierungshierarchie weitergegeben.
- Eine Klasse implementiert ein Interface und erweitert eine Klasse und erbt dabei von beiden Vorgängertypen eine **default**-Methode mit gleicher Signatur:  
Es wird die in der Oberklasse deklarierte Methode ausgeführt.
- Ein Interface erweitert mehrere Interfaces und erbt dabei mehrfach eine **default**-Methode mit gleicher Signatur:  
Es muss *explizit* eine *neue default*-Deklaration angelegt werden.  
Auf die geerbten Methoden kann Bezug genommen werden mit:

*InterfaceName.super.MethodName*

## Interface - statische Methoden

Interfaces können auch zur Deklaration von statischen Methoden benutzt werden.

Eine Methode wie `calculateGcd` (siehe Folie 233) kann in einem Interface deklariert werden, um unabhängig von einer Klasse oder einem Objekt aufrufbar zu sein.

```
public interface Utilities
{
    static int calculateGcd( int v1, int v2 ) { ... }
}
```

- ❑ Eine in einem Interface deklarierte statische Methode wird *nicht* vererbt.  
Daher kann in Spezialisierungshierarchien und bei der Implementierung durch Klassen kein Konflikt auftreten.
- ❑ Der Aufruf kann *ausschließlich* über den Namen des deklarierenden Interfaces erfolgen:  
`Utilities.calculateGcd( 27, 18 )`

## Unveränderbare Deklarationen: **final**

Das Schlüsselwort **final** bezeichnet in Java eine unveränderbare Deklaration:

- ❑ Eine mit **final** gekennzeichnete Klasse darf *nicht* geerbt werden.
- ❑ Eine mit **final** gekennzeichnete Methode darf in Unterklassen *nicht* redefiniert werden.
- ❑ Mit **final** gekennzeichneten Attributen, Variablen und Parametern darf nur *einmal* ein Wert zugewiesen werden.
- ❑ Mit **final** gekennzeichnete Attribute oder Variablen repräsentieren also konstante Werte und werden kurz auch als *Konstante* bezeichnet.

Konstante erhalten in Java üblicherweise Bezeichner aus Großbuchstaben:

```
final int EXCELLENT = 1;
```

## Interface - Konstante

In einem Interface können auch *öffentliche statische konstante Attribute* deklariert werden:

```
public interface Grades
{
    int EXCELLENT = 1;
    int GOOD = 2;
}
```

- ❑ Die Modifizierer **public static final** können alle oder einzeln entfallen.
- ❑ Konstante werden von Interfaces an Interfaces oder Klassen vererbt.
- ❑ Namenskonflikte werden vom Compiler nur gemeldet, wenn diese aufgelöst werden müssten. Dann muss im erweiternden Interface oder der implementierenden Klasse eine Redefinition erfolgen.

## Interface – Zusammenfassung

- ❑ Das Konzept *Interface* ermöglicht die Deklaration von Verhalten durch das Festlegen einer Menge von Methodensignaturen.
- ❑ Interfaces können dazu benutzt werden, die Methoden zusammenzufassen.
- ❑ Interfaces können dazu benutzt werden, den Zugang zu funktionalen Konzepten wie Iteratoren zu vereinheitlichen.
- ❑ Interfaces können sehr flexibel kombiniert werden, da in einer Hierarchie von Interfaces mehrfaches Erben möglich ist.
- ❑ Über Interfaces kann eine Hierarchie kompatibler Typen angelegt werden.
- ❑ Eine Referenz auf ein Interface kann wie eine Referenz auf eine Klasse genutzt werden. Eine Referenz auf ein Interface kann auf die Objekte der Klassen verweisen, die das Interface implementieren.
- ❑ Interfaces verbessern die Flexibilität der Typstruktur, da eine Klasse *mehrere* Interfaces implementieren und *zusätzlich* noch von *genau einer* Oberklasse erben kann.
- ❑ Interfaces können Methoden mit **default**-Implementierungen deklarieren. \*)
- ❑ Interfaces können zusätzlich statische Methoden und Konstante deklarieren. \*)

\*) Diese Möglichkeiten werden im Rahmen dieser Vorlesung nicht genutzt.

## Interface – Zusammenfassung

(Fortsetzung)

### Übersicht

	<i>Klasse</i>	<i>Interface</i>
kann erben von	Klasse/Interface	Interface
kann vererben an	Klasse	Klasse/Interface
Attribute	+	–
(statische) Konstante	+	+
Konstruktor	+	–
abstrakte Methode	+	+
(Instanz-)Methode	+	–
<b>default</b> -Methode	( + )	
statische Methode	+	+ (keine Vererbung)

## Interface – Zusammenfassung

(Fortsetzung)

Die Java-Entwicklungsumgebung (JDK) stellt eine Vielzahl von (fertigen) Klassen und Interfaces in verschiedenen Paketen zur Verfügung.

Hierzu zählen insbesondere auch die folgenden drei Interfaces, die in ihren Deklarationen den in der Vorlesung vorgestellten Fassungen entsprechen:

- ❑ **public interface Comparable<T>**  
im Paket `java.lang`
- ❑ **public interface Iterable<T>**  
im Paket `java.lang`
- ❑ **public interface Iterator<E>**  
im Paket `java.util`

In den in der weiteren Vorlesung folgenden Implementierungen wird nun immer auf diese Interfaces zurückgegriffen.

\*) Das Paket `java.lang` ist immer auch ohne expliziten Import verfügbar.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 14. Anonyme Klassen und Lambda-Ausdrücke

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-957

## Lernziele des Kapitels 14. Anonyme Klassen und Lambda-Ausdrücke

Nach Durcharbeiten des Kapitels Anonyme Klassen und Lambda-Ausdrücke sollen die teilnehmenden Studierenden

- die Möglichkeiten des Einsatzes anonymer Klassen kennen,
- die Bedeutung funktionaler Interfaces kennen,
- einige Möglichkeiten des Einsatzes von Lambda-Ausdrücken kennen.

## Rückblick: Einsatz des Entwurfsmusters Strategie

Deklaration einer Schnittstelle für Strategien:

```
public static interface SubstitutionStrategy<T>
{
    T substitute( T ref );
}
```

Deklaration einer passenden Strategie:

```
public class DoubleIntegersStrategy
implements DoublyLinkedList.SubstitutionStrategy<Integer>
{
    public Integer substitute( Integer ref )
    {
        return 2 * ref;
    }
}
```

## Rückblick: Einsatz des Entwurfsmusters Strategie

(Fortsetzung)

### Anlegen von Strategie-Objekten

Analyse:

- ❑ Die Methode `substituteAll` der Klasse `DoublyLinkedList` sorgt während der Ausführung dafür, dass die `substitute`-Methode des als Argument übergebenen Strategie-Objekts auf alle Inhalte in der Liste angewandt wird.
  - ❑ Dafür muss eine geeignete Implementierung des Interfaces `SubstitutionStrategy` realisiert werden. Eine geeignete Klasse muss spezifisch deklariert werden.
  - ❑ Es werden also Klassen benötigt, von denen *möglicherweise nur genau ein* Objekt erzeugt werden soll. Beim häufigen Einsatz des Entwurfsmusters Strategie müssen sehr viele von solchen Klassen implementiert werden: Die Programmstruktur wird unübersichtlich.
- ❑ Lösungsansatz:  
**anonyme Klasse**

## Anonyme Klasse

- ❑ Idee:
  - Deklaration der Klasse und
  - Erzeugen des (einzig möglichen) Objekts fallen zusammen.
- ❑ Konsequenzen:
  - Die Klasse erhält keinen eigenen Namen. Sie bleibt *anonym*.
  - Die Nutzung ist nur möglich, wenn die Klasse einen bereits bekannten Typ, also eine Klasse oder ein Interface, erweitert bzw. implementiert.
- ❑ Dann wird der Zugriff auf die Methode des so erzeugten Objekt möglich über die Deklarationen der öffentlichen Methoden in der erweiterten Klasse oder implementierten Schnittstelle.

## Anonyme Klasse

(Fortsetzung)

```
...
DoublyLinkedList<Integer> ints = new DoublyLinkedList<>();
ints.add( 5 );
...
DoublyLinkedList.SubstitutionStrategy<Integer> s =
    new DoublyLinkedList.SubstitutionStrategy<Integer>()
{
    public Integer substitute( Integer ref )
    {
        return 2 * ref;
    }
};
ints.substituteAll( s );
...

```

ähnlich zu Konstruktoraufruf

Rumpf der Klasse

- Für `s` wird ein Objekt einer anonymen Klasse erzeugt, die das Interface `SubstitutionStrategy` implementiert.
- Syntaktisch wird das durch eine konstruktor-analoge Erzeugung mit nachfolgender Deklaration des Klassenrumpfs erreicht.

## Anonyme Klasse

(Fortsetzung)

mögliche Fassung ohne Nutzung einer zusätzlichen Referenzvariablen:

```
...
DoublyLinkedList<Integer> ints = new DoublyLinkedList<>();
ints.add( 5 );
...
ints.substituteAll(
    new DoublyLinkedList.SubstitutionStrategy<Integer>()
{
    public Integer substitute( Integer ref )
    {
        return 2 * ref;
    }
);
...

```

**Objekt wird direkt als Argument übergeben**

**Rumpf der Klasse**

## Anonyme Klasse

(Fortsetzung)

```
...
DoublyLinkedList<Integer> ints = new DoublyLinkedList<>();
ints.add( 5 );
...
int factor = 2;
ints.substituteAll(
    new DoublyLinkedList.SubstitutionStrategy<Integer>()
{
    public Integer substitute( Integer ref )
    {
        return factor * ref;
    }
);
...

```

effektiv finale Variable factor

- ❑ Die anonyme Klasse wird *innerhalb* ihrer Umgebung vereinbart und hat Zugriff auf die dort deklarierten Attribute und auf solche Variable, die *effektiv final* oder **final** sind.
- ❑ *Effektiv final* sind Variable, die nicht als **final** deklariert sind, denen aber trotzdem nur einmal ein Wert zugewiesen wird, die also im Programm wie Konstante behandelt werden.

## Anonyme Klasse

(Fortsetzung)

- ❑ verbleibendes Problem bei komplexeren anonymen Klassen:

Ein Konstruktor kann nicht deklariert werden, da die Klasse keinen Namen – und somit auch keinen Namen für den Konstruktor – hat.

## Anonyme Klasse

(Fortsetzung)

- verbleibendes Problem bei komplexeren anonymen Klassen:

Ein Konstruktor kann nicht deklariert werden, da die Klasse keinen Namen – und somit auch keinen Namen für den Konstruktor – hat.

- Lösung:

### *Exemplarinitialisierer*

Einfache Blöcke, die im Rumpf der anonymen Klasse stehen und nacheinander in der Reihenfolge ihrer Deklaration in der Klasse ausgeführt werden.

## Exkurs: Initialisierung von Attributen

Attribute können an verschiedenen Stellen einer Klasse initialisiert werden.

- ❑ Zunächst wird jedes Attribut bei der Reservierung des Speicherplatzes mit 0-Bits belegt. Jedes Attribut ist daher mit einem Wert vorbelegt, auf Attribute kann auch ohne weitere Zuweisung fehlerfrei lesend zugegriffen werden.
- ❑ Ein Attribut kann unmittelbar bei der Deklaration initialisiert werden:

**private int** value = 27;

Solche Initialisierungen sind auf die Angabe eines Ausdrucks reduziert. Komplexere Abläufe und insbesondere das voneinander abhängige Initialisieren mehrerer Attribute kann so nicht vorgenommen werden.

Die Initialisierung bei der Deklaration wird immer vor der Ausführung des Rumpfes eines Konstruktors ausgeführt.

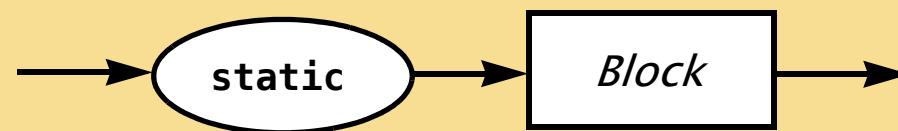
- ❑ Darüber hinaus kann eine komfortable Belegung mit Werten in einem Konstruktor erfolgen. Hier sind Berechnungen auf der Basis von Anweisungen und Fehlerbehandlungen möglich.
- ❑ *Exemplarinitialisierer* sind speziell für die Initialisierung von Objekten anonymer Klassen notwendig, können aber in allen Klassen eingesetzt werden.  
Exemplarinitialisierer werden vor dem Konstruktor ausgeführt.
- ❑ Bei der Erzeugung eines Objekts der Klasse C werden alle Exemplarinitialisierer und der Konstruktor der Oberklasse vor allen Initialisierungen der Klasse C ausgeführt (Deklaration, Exemplarinitialisierer, Konstruktor).

## Exkurs: Initialisierung von Attributen

(Fortsetzung)

- ❑ Konstruktoren können auf statische Attribute zugreifen.
- ❑ Da ein statisches Attribut auch ohne das Vorliegen eines Objekts und somit vor dem ersten Aufruf eines Konstruktors verfügbar ist, kann es mit einem Konstruktor *nicht* sicher initialisiert werden.
- ❑ Statische Attribute können daher über einen *Klasseninitialisierer* mit Werten belegt werden.
- ❑ *Klasseninitialisierer* können mehrfach in einer Klasse auftreten. Sie werden in der Reihenfolge ausgeführt, in der sie in der Klasse auftreten. Die Ausführung erfolgt nach der Initialisierung bei der Deklaration von Klassenvariablen dann, wenn die Klasse in die virtuelle Maschine JVM geladen wird.

Syntax (innerhalb des Rumpfes einer Klasse):



- ❑ Initialisierung bei der Deklaration, durch Initialisierer und durch Konstruktoren sollten vorsichtig (und sinnvoll) miteinander kombiniert werden, da die Abhängigkeiten schnell zu unübersichtlichen Abläufen führen. Insbesondere werden Initialisierer in langen Klassen leicht übersehen.

## Funktionales Interface (seit Java 8)

- ❑ Ein *funktionales Interface* ist ein Interface, das *genau eine* abstrakte Methode vereinbart.
- ❑ Funktionale Interfaces werden normalerweise passend annotiert mit @FunctionalInterface

- ❑ Beispiel:

```
@FunctionalInterface
public static interface SubstitutionStrategy<T>
{
    T substitute( T ref );
}
```

- ❑ Eigenschaften:
  - Implementierungen eines funktionalen Interfaces unterscheiden sich bezüglich des über den Typ des Interfaces erreichbaren Verhaltens nur durch die Implementierungen der einzigen (abstrakten) Methode.
  - Die Implementierung kann daher auf die Deklaration der einzigen Methode reduziert werden:  
*Lambda-Ausdruck*

## Lambda-Ausdruck (seit Java 8)

- Ein Lambda-Ausdruck ist die Deklaration einer *anonymen Methode*.
- Beispiel:

```
( int a, int b ) -> { return a * b; }
```

Parameterliste              Methodenrumpf

Der Typ des Rückgabewertes wird nicht angegeben, sondern vom Compiler durch geeignetes Schließen implizit bestimmt (*Typinferenz*).

Das Beispiel entspricht der Deklaration einer Methode der Form:

```
int prod( int a, int b )
{
    return a * b;
}
```

## Lambda-Ausdruck

(Fortsetzung)

- Ein Lambda-Ausdruck ist die Deklaration einer *anonymen Methode*.
- Beispiel:

```
( int a, int b ) -> { return a * b; }
```

- Kann vom Compiler aus der Verwendung der Methode auf den Typ der Parameter geschlossen werden, so werden auch diese durch *Typinferenz* bestimmt.  
Dann können die Typangaben entfallen:

```
( a, b ) -> { return a * b; }
```

## Lambda-Ausdruck

(Fortsetzung)

- ❑ Ein Lambda-Ausdruck ist die Deklaration einer *anonymen Methode*.
- ❑ Beispiel:

```
( int a, int b ) -> { return a * b; }
```

```
( a, b ) -> { return a * b; }
```

- ❑ Enthält der Rumpf der Methode nur eine **return**-Anweisung, so können der Block – und damit die Klammern – und das Schlüsselwort **return** entfallen:

```
( a, b ) -> a * b
```

## Lambda-Ausdruck

(Fortsetzung)

- ❑ Ein Lambda-Ausdruck ist die Deklaration einer *anonymen Methode*.
- ❑ Beispiel:

```
( int a, int b ) -> { return a * b; }
```

```
( a, b ) -> { return a * b; }
```

```
( a, b ) -> a * b
```

- ❑ Hat die Methode nur einen Parameter, so kann die Klammerung des Parameters entfallen:

```
a -> a * a
```

## Lambda-Ausdruck

(Fortsetzung)

- ❑ Ein Lambda-Ausdruck ist die Deklaration einer *anonymen Methode*.
- ❑ Beispiel:

```
( int a, int b ) -> { return a * b; }
```

```
( a, b ) -> { return a * b; }
```

```
( a, b ) -> a * b
```

```
a -> a * a
```

- ❑ Hat die Methode keinen Parameter, so muss eine leere Klammerung angegeben werden:

```
() -> 2
```

```
() -> System.out.println( "lambda" )
```

## Lambda-Ausdruck

(Fortsetzung)

### Einsatz von Lambda-Ausdrücken

- ❑ Lambda-Ausdrücke können Objekte von funktionalen Interfaces ersetzen:  
Ein funktionales Interface erfordert für seine Implementierung die Realisierung genau einer abstrakten Methode,  
ein Lambda-Ausdruck liefert die Implementierung genau einer einzelnen Methode.
- ❑ Der Compiler stellt zwischen beiden Konzepten eine implizite Beziehung her:  
Wird eine Instanz eines funktionalen Interfaces erwartet und wird stattdessen ein Lambda-Ausdruck angegeben,  
so wird die durch diesen gegebene Methode als Umsetzung der einzigen abstrakten Methode des Interfaces interpretiert und verwendet.
- ❑ Da das funktionale Interface die vollständige Signatur seiner abstrakten Methode deklariert,  
ist die implizite Bestimmung der Typen der Parameter und des Rückgabewertes in der Regel einfach.

## Lambda-Ausdrücke zur Verhaltensbeschreibung

Substitution-Strategie – Lösung mit anonymer Klasse (siehe Folie 963):

```
ints.substituteAll(  
    new DoublyLinkedList.SubstitutionStrategy<Integer>()  
    {  
        public Integer substitute( Integer ref )  
        {  
            return 2 * ref;  
        }  
    }  
);
```

Substitution-Strategie – gleichwertige Lösung mit Lambda-Ausdruck:

```
ints.substituteAll( ref -> 2 * ref );
```

- ❑ Der Lambda-Ausdruck liefert eine Methode, die einen Parameter besitzt.
- ❑ Die Methode `substituteAll` erwartet als Argument eine Implementierung des Interfaces `SubstitutionStrategy<Integer>`, das die Realisierung der einen Methode `substitute` mit einem Parameter (des Typs `Integer`) und einer `Integer`-Rückgabe erfordert.  
Genau das leistet der Lambda-Ausdruck `ref -> 2 * ref`.

## Lambda-Ausdrücke zur Verhaltensbeschreibung

(Fortsetzung)

Inspection-Strategie – Lösung mit Lambda-Ausdruck:

:

```
ints.inspectAll( x -> System.out.print( x + ", " ) );
```

- ❑ Die Methode `inspectAll` erwartet als Argument eine Implementierung des Interfaces `InspectionStrategy<Integer>`, das die Realisierung der einen Methode `inspect` mit einem Parameter (des Typs `Integer`) und ohne Rückgabe erfordert.  
Genau das leistet der Lambda-Ausdruck `x -> System.out.print( x + ", " )`.
- ❑ Das Typargument `Integer` der Erzeugung des der Referenzvariablen `ints` zugewiesenen Objekts bildet zugleich das Typargument für die innere Klasse `InspectionStrategy`.

Deletion-Strategie – Lösung mit Lambda-Ausdruck:

:

```
ints.deleteSelected( x -> x % 2 == 0 );
```

- ❑ Alle Elemente mit geraden Inhalten werden aus der Liste entfernt.

## Lambda-Ausdrücke zur Verhaltensbeschreibung

(Fortsetzung)

Substitution-Strategie – weitere Lösung mit Lambda-Ausdruck

```
int factor = 2;  
ints.substituteAll( x -> factor * x );
```

effektiv finale Variable factor

- Der Lambda-Ausdruck wird *innerhalb* seiner lokalen Umgebung vereinbart und kann die dort deklarierten Variablen benutzen («*einfangen*»), sofern diese *effektiv final* oder **final** sind.
- *Hinweis:*
  - Die Schlüsselwörter **this** und **super** beziehen sich in Lambda-Ausdrücken auf die sie umgebende Klasse.
  - In anonymen Klassen beziehen sich die Schlüsselwörter **this** und **super** *ebenfalls* auf die sie umgebende Klasse – das ist aber in diesem Fall die anonyme Klasse selbst!

## Funktionstypen

Das Paket `java.util.function` enthält zahlreiche vordefinierte funktionale Interfaces:

- ❑ `public interface Function<T,R> { R apply( T t ); ... }`
- ❑ `public interface BiFunction<T,U,R> { R apply( T t, U u ); ... }`
- ❑ `public interface Predicate<T> { boolean test( T t ); ... }`
- ❑ `public interface Consumer<T> { void accept( T t ); ... }`
- ❑ `public interface Supplier<T> { T get(); }`
- ❑ ...

## Funktionstypen

(Fortsetzung)

Objekte der implizit erzeugten Klassen können Referenzen auf die Interfaces zugewiesen werden:

- **public interface Function<T,R>** { R apply( T t ); ... }

```
Function<Integer, Integer> add5 = x -> x + 5;
int j = add5.apply( i );
```

auch möglich ist:

```
( (Function<Integer, Integer>) (Integer x) -> x + 7 ).apply( 5 )
```

Type-Cast ermöglicht Aufruf

Lambda-Ausdruck

- **public interface Predicate<T>** { boolean test( T t ); ... }

```
Predicate<Integer> isPositive = x -> x > 0;
boolean b = isPositive.test( i );
```

- **public interface Consumer<T>** { void accept( T t ); ... }

```
Consumer<String> out = s -> System.out.println( s );
out.accept( "hello" );
```

## Funktionstypen

(Fortsetzung)

Die Methoden von implizit erzeugten Klassen können auf bestehenden Methoden aufgebaut werden:

```
public class BooleanUtilities
{
    public static boolean positiveNumber( int i )
    {
        return i > 0;
    }
}
```

dann ist möglich:

```
...
Predicate<Integer> gt0 = BooleanUtilities::positiveNumber;
boolean b = gt0.test( i );
...
```

Der `::-`-Operator kürzt ab:

statt	<code>Predicate&lt;Integer&gt; gt0 = x -&gt; BooleanUtilities.positiveNumber( x )</code>
ist möglich	<code>Predicate&lt;Integer&gt; gt0 = BooleanUtilities::positiveNumber</code>

## weitere Methoden in vordefinierten funktionalen Interfaces

Beispiel: **interface** java.util.function.Function<T,R>

```
@FunctionalInterface
public interface Function<T,R>
{
    R apply(T t); ← die abstrakte Methode

    static <T> Function<T,T> identity()
    {
        return t -> t;
    }

    default <V> Function<T,V> andThen( Function<? super R,> V > after )
    {
        return t -> after.apply( apply( t ) );
    }

    default <V> Function<V,R> compose( Function<? super V,> T > before )
    {
        return t -> apply( before.apply( t ) );
    }
}
```

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: **interface** java.util.function.Function<T,R>

```

@FunctionalInterface
public interface Function<T,R>
{
    R apply(T t);

    static <T> Function<T,T> identity()           ← erzeugt Function-Objekt,
    {                                              dessen apply Identität ist
        return t -> t;
    }

    default <V> Function<T,V> andThen( Function<? super R,>? extends V> after )
    {
        return t -> after.apply( apply( t ) );
    }

    default <V> Function<V,R> compose( Function<? super V,>? extends T> before )
    {
        return t -> apply( before.apply( t ) );
    }
}

```

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: `interface java.util.function.Function<T,R>`

Deklaration der Methode `andThen`

`<V> Function<T,V> andThen( Function<? super R,? extends V> after )`

Typ der Rückgabe

Typ des Parameters

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: `interface java.util.function.Function<T,R>`

Deklaration der Methode `andThen`

```
<V> Function<T,V> andThen( Function<? super R,? extends V> after )
```

- `V` ist ein nur für diese Methode deklarierter Typparameter, der die Beziehung zwischen dem Typ der Rückgabe und dem Typ des Parameters herstellt.

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: `interface java.util.function.Function<T,R>`

Deklaration der Methode `andThen`

`<V> Function<T,V> andThen( Function<? super R,? extends V> after )`

- `V` ist ein nur für diese Methode deklarierter Typparameter, der die Beziehung zwischen dem Typ der Rückgabe und dem Typ des Parameters herstellt.
- Die zulässigen Typargumente werden eingeschränkt durch: `? super R` und `? extends V`  
Erlaubt sind nur Oberklassen von `R` oder Interfaces, die von `R` implementiert werden.  
Erlaubt sind nur Unterklassen von `V` oder Klassen, die `V` implementieren.

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: `interface java.util.function.Function<T,R>`

Deklaration der Methode `andThen`

```
<V> Function<T,V> andThen( Function<? super R,? extends V> after )
```

- ❑ `V` ist ein nur für diese Methode deklarierter Typparameter, der die Beziehung zwischen dem Typ der Rückgabe und dem Typ des Parameters herstellt.
- ❑ Die zulässigen Typargumente werden eingeschränkt durch: `? super R` und `? extends V`  
Erlaubt sind nur Oberklassen von `R` oder Interfaces, die von `R` implementiert werden.  
Erlaubt sind nur Unterklassen von `V` oder Klassen, die `V` implementieren.
- ❑ Da der Typ der Unterkasse selbst in der Deklaration der Signatur nicht benötigt wird,  
wird die *Wildcard* `?` als Platzhalter verwendet.
- ❑ Für Typeinschränkungen gilt, dass `T` selbst die Bedingungen erfüllt.  
Es gilt immer: `T super T` und `T extends T`

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: **interface** java.util.function.Function<T,R>

```
public interface Function<T,R>
{
    R apply(T t);

    default <V> Function<T,V> andThen( Function<? super R,? extends V> after )
    {
        return t -> after.apply( apply( t ) );
    }

    ...
}
```

- Function enthält eine Methode apply, die ein T-Objekt in ein R-Objekt überführt.

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: `interface java.util.function.Function<T,R>`

```
public interface Function<T,R>
{
    R apply(T t);

    default <V> Function<T,V> andThen( Function<? super R,? extends V> after )
    {
        return t -> after.apply( apply( t ) );
    }

    ...
}
```

- ❑ `Function` enthält eine Methode `apply`, die ein `T`-Objekt in ein `R`-Objekt überführt.
- ❑ `andThen` wendet die `apply`-Methode ihres Arguments auf das Ergebnis des eigenen `apply` an:

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: `interface java.util.function.Function<T,R>`

```
public interface Function<T,R>
{
    R apply(T t);

    default <V> Function<T,V> andThen( Function<? super R,? extends V> after )
    {
        return t -> after.apply( apply( t ) );
    }

    ...
}
```

- Function enthält eine Methode apply, die ein T-Objekt in ein R-Objekt überführt.
- andThen wendet die apply-Methode ihres Arguments auf das Ergebnis des eigenen apply an:
  - Der Typ V der Rückgabe von andThen hängt nur von after und nicht von T und R ab.
  - Die Bedingungen des Typs V werden auch von allen Unterklassen von V eingehalten, wobei der genaue Typ der Rückgabe von after nicht interessiert: ? extends V
  - Damit after.apply auf das Ergebnis von apply angewandt werden kann, muss after so definiert sein, dass Objekte von R verarbeitet werden können, wobei der genaue Typ nicht interessiert: ? super R

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: `interface java.util.function.Function<T,R>`

```
public interface Function<T,R>
{
    R apply(T t);

    default <V> Function<T,V> andThen( Function<? super R,? extends V> after )
    {
        return t -> after.apply( apply( t ) );
    }
    ...
}
```



- Der Lambda-Ausdruck `t -> ...` erzeugt ein zu `Function` kompatibles Objekt, dessen `apply`-Methode die folgende Anweisung implementiert:

`after.apply(apply( t ))`

- Dabei wird der effektiv finale Parameter `after` «eingefangen».
- Das durch den Lambda-Ausdruck erzeugte Objekt wird durch die Methode `andThen` zurückgegeben.

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: **interface Function<T,R>**

```
Function<Integer, Integer> f = x -> x * x; erzeugt Function-Objekt, das  $f(x)=x^2$  definiert  
Function<Integer, Integer> g = x -> 3 * x; erzeugt Function-Objekt, das  $g(x)=3x$  definiert
```

- ❑ `f.compose( Function.identity() ).apply( 5 )` erzeugt Function-Objekt, das  $f(id(x))$  definiert und für das durch den `apply`-Aufruf  $f(id(5)) = 25$  berechnet wird.

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: **interface Function<T,R>**

```
Function<Integer, Integer> f = x -> x * x;  
Function<Integer, Integer> g = x -> 3 * x;
```

- ❑ `f.compose( Function.identity() ).apply( 5 )` erzeugt Function-Objekt, das  $f(id(x))$  definiert und für das durch den apply-Aufruf  $f(id(5)) = 25$  berechnet wird.
- ❑ `f.compose( g ).apply( 5 )` erzeugt Function-Objekt, das  $f(g(x))$  definiert und für das durch den apply-Aufruf  $f(g(5)) = 225$  berechnet wird.

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: **interface Function<T,R>**

```
Function<Integer, Integer> f = x -> x * x;  
Function<Integer, Integer> g = x -> 3 * x;
```

- `f.compose( Function.identity() ).apply( 5 )` erzeugt Function-Objekt, das  $f(id(x))$  definiert und für das durch den apply-Aufruf  $f(id(5)) = 25$  berechnet wird.
- `f.compose( g ).apply( 5 )` erzeugt Function-Objekt, das  $f(g(x))$  definiert und für das durch den apply-Aufruf  $f(g(5)) = 225$  berechnet wird.
- `f.andThen( g ).apply( 5 )` erzeugt Function-Objekt, das  $g(f(x))$  definiert und für das durch den apply-Aufruf  $g(f(5)) = 75$  berechnet wird.

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: **interface Function<T,R>**

```
Function<Integer, Integer> f = x -> x * x;  
Function<Integer, Integer> g = x -> 3 * x;
```

- `f.compose( Function.identity() ).apply( 5 )` erzeugt Function-Objekt, das  $f(id(x))$  definiert und für das durch den apply-Aufruf  $f(id(5)) = 25$  berechnet wird.
- `f.compose( g ).apply( 5 )` erzeugt Function-Objekt, das  $f(g(x))$  definiert und für das durch den apply-Aufruf  $f(g(5)) = 225$  berechnet wird.
- `f.andThen( g ).apply( 5 )` erzeugt Function-Objekt, das  $g(f(x))$  definiert und für das durch den apply-Aufruf  $g(f(5)) = 75$  berechnet wird.
- `f.andThen( g ).compose( x -> x + 7 ).apply( 5 )` keine Typbestimmung möglich, da der Parametertyp von `compose` zu wenig Vorgaben enthält.

## weitere Methoden in vordefinierten funktionalen Interfaces

(Fortsetzung)

Beispiel: **interface Function<T,R>**

```
Function<Integer, Integer> f = x -> x * x;  
Function<Integer, Integer> g = x -> 3 * x;
```

- `f.compose( Function.identity() ).apply( 5 )` erzeugt Function-Objekt, das  $f(id(x))$  definiert und für das durch den apply-Aufruf  $f(id(5)) = 25$  berechnet wird.
- `f.compose( g ).apply( 5 )` erzeugt Function-Objekt, das  $f(g(x))$  definiert und für das durch den apply-Aufruf  $f(g(5)) = 225$  berechnet wird.
- `f.andThen( g ).apply( 5 )` erzeugt Function-Objekt, das  $g(f(x))$  definiert und für das durch den apply-Aufruf  $g(f(5)) = 75$  berechnet wird.
- `f.andThen( g ).compose( x -> x + 7 ).apply( 5 )` keine Typbestimmung möglich, da der Parametertyp von compose zu wenig Vorgaben enthält.
- `f.andThen( g ).compose( Integer x -> x + 7 ).apply( 5 )` erzeugt Function-Objekt, das  $g(f(x+7))$  definiert und für das durch den apply-Aufruf  $g(f(5+7)) = 432$  berechnet wird.

## vordefinierte Umsetzungen des Entwurfsmusters Strategie

Vordefinierte Klassen setzen das Entwurfsmuster Strategie ein, um über Lambda-Ausdrücke interne Abläufe zu steuern.

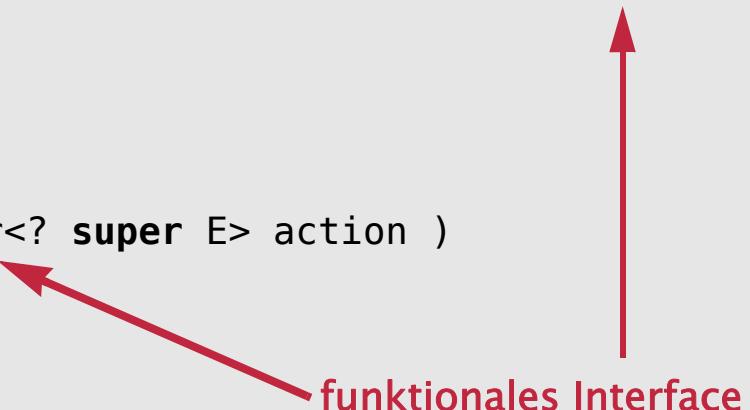
Beispiel:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();

    default void remove()
    { /* optional operation */ }

    default void forEachRemaining( Consumer<? super E> action )
    {
        while ( hasNext() )
        {
            action.accept( next() );
        }
    }
}
```

```
public interface Consumer<T> {
    void accept( T t );
}
```



## vordefinierte Umsetzungen des Entwurfsmusters Strategie

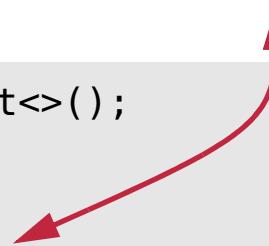
(Fortsetzung)

Beispiel: **interface** java.util.Iterator<E>

funktionales Interface: **interface** Consumer<T> { **void** accept( T t ); ... }

```
DoublyLinkedList<Integer> numbers = new DoublyLinkedList<>();
numbers.add( 2 ); numbers.add( 4 ); numbers.add( 11 );
numbers.add( 5 ); numbers.add( 6 ); numbers.add( 7 );

numbers.iterator().forEachRemaining( x -> System.out.print( x + " " ) );
```



erzeugt als Ausgabe:

2 4 11 5 6 7

```
Iterator<Integer> it = numbers.iterator();
it.next();
it.forEachRemaining( x -> System.out.print( x + " " ) );
```

erzeugt als Ausgabe:

4 11 5 6 7

## vordefinierte Umsetzungen des Entwurfsmusters Strategie

(Fortsetzung)

Beispiel: **interface** java.util.List<E>

Interface, das die üblichen Methoden  
für Listen vorschreibt

```
public interface List<E>
{
    default void replaceAll( UnaryOperator<E> operator )
    {
        final ListIterator<E> li = listIterator();
        while (li.hasNext()) {
            li.set( operator.apply( li.next() ) );
        }
    }

    default void sort( Comparator<? super E> c )
    { ... }

    ...
}
```

## vordefinierte Umsetzungen des Entwurfsmusters Strategie

(Fortsetzung)

Beispiel: `interface java.util.List<E>`

```
public interface List<E>
{
    default void replaceAll( UnaryOperator<E> operator )
    {
        final ListIterator<E> li = listIterator();
        while ( li.hasNext() ) {
            li.set( operator.apply( li.next() ) );
        }
    }

    default void sort( Comparator<? super E> c )
    { ... }

    ...
}
```

funktionales Interface:

```
public interface UnaryOperator<T>
extends Function<T,T> ...
```

funktionales Interface:

```
public interface Comparator<T>
{
    int compare( T o1, T o2 );
    ...
}
```

- ❑ Die Methode `sort` stellt eine gemäß des `Comparator`-Objekts *aufsteigende* Sortierung her.

## vordefinierte Umsetzungen des Entwurfsmusters Strategie

(Fortsetzung)

Beispiel: **interface** java.util.List<E>

funktionales Interface: **interface** UnaryOperator<T> **extends** Function<T,T> { ... }

```
numbers.replaceAll( x -> x + 3 );
```

**Wirkung:**      2 4 11 5 6 7      **wird zu**      5 7 14 8 9 10

## vordefinierte Umsetzungen des Entwurfsmusters Strategie

(Fortsetzung)

Beispiel: `interface java.util.List<E>`

funktionales Interface: `interface Comparator<T> { int compare( T o1, T o2 ); ... }`

```
numbers.sort( (x,y) -> y - x );
```

**Wirkung:**    2 4 11 5 6 7      **wird zu**    11 7 6 5 4 2      **absteigend sortiert,  
da sort aufsteigend sortiert**

```
numbers.sort( (x,y) -> x % 2 - y % 2 );
```

**Wirkung:**    2 4 11 5 6 7      **wird zu**    2 4 6 11 5 7      **ungerade größer als gerade**

```
numbers.sort( (x,y) -> y % 2 - x % 2 );
```

**Wirkung:**    2 4 11 5 6 7      **wird zu**    11 5 7 2 4 6      **gerade größer als ungerade**

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 15. Mustererkennung - Einführung

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-1003

## Lernziele des Kapitels 15. Mustererkennung

Nach Durcharbeiten des Kapitels Mustererkennung sollen die teilnehmenden Studierenden

- den Algorithmus zur Erkennung von Mustern in Texten kennen und erklären können,
- die Implementierung des Algorithmus zur Erkennung von Mustern in Texten erklären können,
- den Prozess der Umsetzung des Algorithmus in ein Java-Programm kennen,
- den Gebrauch der Bibliotheksinterfaces `java.util.Set`, `java.util.List` und `java.util.Map` kennen,
- den Gebrauch der Bibliotheksklassen `java.util.HashSet`, `java.util.LinkedList` und `java.util.HashMap` kennen,
- generische Klassen nutzen können.

## Einsatz generischer Klassen der Java Platform

- ❑ Die auf den folgenden Folien beschriebene Problemstellung soll durch den Einsatz von bereits erstellten und zusammen mit dem Java-Compiler bereitgestellten Klassen gelöst werden.
- ❑ Dabei soll demonstriert werden, wie sich eine durchaus komplexe Aufgabenstellung schnell und mit relativ wenig selbst geschriebenem Programmtext realisieren lässt.

## Problemstellung Mustererkennung

### Aufgabe:

Stelle fest ob – und wie häufig – ein (Text-)Muster  $p$  in einem Text  $t$  vorkommt.

### Beispiel:

$t$ : aabaababbaaabbbbababaabaabaaaaba

$p$ : abaa

## Problemstellung Mustererkennung

(Fortsetzung)

### Aufgabe:

Stelle fest ob – und wie häufig – ein (Text-)Muster  $p$  in einem Text  $t$  vorkommt.

### Beispiel:

$t$ : aabaababbaaabbbbababaabaabaaaaba  
 $p$ : abaa

### Lösung:

Vorkommen von  $p$  in  $t$ : 4

$t$ : a $\textcolor{red}{aba}$ babbaaabbbab $\textcolor{red}{aba}$ abaabaaabaa

1

2 3 4

## Problemstellung Mustererkennung

(Fortsetzung)

- ❑ Lohnt sich die Auseinandersetzung mit diesem einfachen Problem überhaupt?  
Ja, insbesondere dann, wenn lange Muster in langen Texten gesucht werden.
- ❑ Ein trivialer (und einfach zu realisierender) Algorithmus lässt sich sofort angeben:

Vergleiche einen Ausschnitt von  $t$  mit  $p$ :

Falls keine Übereinstimmung vorliegt, verschiebe den Ausschnitt von  $t$  um *ein* Zeichen.

	$t:$	aabaababbaaabbbbababaabaabaaaaba	
1.	$p:$	abaa	Mißerfolg
2.	$p:$	abaa	Treffer
3.	$p:$	abaa	Mißerfolg
4.	$p:$	abaa	Mißerfolg
5.	$p:$	abaa	Mißerfolg
6.	$p:$	abaa	Mißerfolg
7.	$p:$	abaa	Mißerfolg
8.	$p:$	abaa ...	Mißerfolg

maximal möglicher Aufwand bei Mißerfolg:  $\text{length}(t) \cdot \text{length}(p)$  Vergleiche

## Problemstellung Mustererkennung

(Fortsetzung)

- Lässt sich die Suche beschleunigen?

Ja, da für den Test alle Zeichen von  $t$

- bis zur Position des ersten Unterschieds zu  $p$  oder
  - bis zum Ende von  $p$  (bei einem Treffer)
- betrachtet worden sind.

## Problemstellung Mustererkennung

(Fortsetzung)

- Lässt sich die Suche beschleunigen?  
Ja, da für den Test alle Zeichen von  $t$ 
  - bis zur Position des ersten Unterschieds zu  $p$  oder
  - bis zum Ende von  $p$  (bei einem Treffer)  
betrachtet worden sind.
- Wird das Ergebnis der Untersuchung für jedes Zeichens von  $t$  gemerkt,  
so muss jedes Zeichen nur einmal betrachtet werden.  
Der maximale Aufwand beträgt dann nur:  $\text{length}(t)$  Vergleiche  
Die Suche kann also um den Faktor  $\text{length}(p)$  beschleunigt werden.

## Problemstellung Mustererkennung

(Fortsetzung)

- Lässt sich die Suche beschleunigen?  
Ja, da für den Test alle Zeichen von  $t$ 
  - bis zur Position des ersten Unterschieds zu  $p$  oder
  - bis zum Ende von  $p$  (bei einem Treffer)  
betrachtet worden sind.
- Wird das Ergebnis der Untersuchung für jedes Zeichens von  $t$  gemerkt,  
so muss jedes Zeichen nur einmal betrachtet werden.  
Der maximale Aufwand beträgt dann nur:  $\text{length}(t)$  Vergleiche  
Die Suche kann also um den Faktor  $\text{length}(p)$  beschleunigt werden.
- Allerdings sind Vorarbeiten notwendig.  
Für jedes gelesene Zeichen gibt es zwei Entscheidungsmöglichkeiten:
  - Das Zeichen aus  $t$  passt zu dem von  $p$  als nächstes geforderten Zeichen.  
Dann kann mit dem aktuellen Vergleich fortgefahrene werden.
  - Das Zeichen aus  $t$  passt nicht zu dem von  $p$  als nächstes geforderten Zeichen.  
Dann muss der Vergleich neu begonnen werden.

## Problemstellung Mustererkennung

(Fortsetzung)

Beispiel:

Falls bei der Suche nach `abaa` die ersten drei Zeichen eine Übereinstimmung ergeben haben und beim Vergleich des vierten Zeichen in  $t$  ein `b` auftritt, dann ist folgende Sequenz in  $t$  zuletzt betrachtet worden:

$t$ :     `...abab????...`

Und es gilt:

- $\text{abab}$  → Es ist keine erfolgversprechende Fortsetzung möglich.
- $\text{abab}$  → Es ist keine erfolgversprechende Fortsetzung möglich.
- $\text{abab}$  → Es ist *eine* erfolgversprechende Fortsetzung möglich,  
falls `aa` folgt.

Der Vergleich wird also fortgesetzt  
mit dem nächsten Zeichen aus  $t$ : `?`  
und der Information,  
dass zwei Zeichen bereits als richtig erkannt sind: `ab`

## Problemstellung Mustererkennung

(Fortsetzung)

- Benötigt werden also Regeln, die angeben, wie bei der Suche fortgefahrene wird.

Diese Regeln haben die folgende Struktur:

Falls die ersten  $n$  Zeichen von  $p$  vorliegen und nun das Zeichen  $c$  gelesen wird:  
dann werden anschließend  $m$  Zeichen als erkannt angenommen mit  $m \in \{0, \dots, n+1\}$ .

- Diese Regeln sind spezifisch für das zu suchende Muster  $p$ .
- Das Muster  $p$  ist immer dann gefunden, wenn  $m = \text{length}(p)$  gilt.

## Problemstellung Mustererkennung

(Fortsetzung)

- Benötigt werden also Regeln, die angeben, wie bei der Suche fortgefahrene wird.

Diese Regeln haben die folgende Struktur:

Falls die ersten  $n$  Zeichen von  $p$  vorliegen und nun das Zeichen  $c$  gelesen wird:  
dann werden anschließend  $m$  Zeichen als erkannt angenommen mit  $m \in \{0, \dots, n+1\}$ .

- Diese Regeln sind spezifisch für das zu suchende Muster  $p$ .
- Das Muster  $p$  ist immer dann gefunden, wenn  $m = \text{length}(p)$  gilt.
- Eine Möglichkeit, solche Regeln auszudrücken, bietet der – einigen Studierenden aus der Veranstaltung *Rechnerstrukturen* bekannte – *Moore-Automat*. \*)

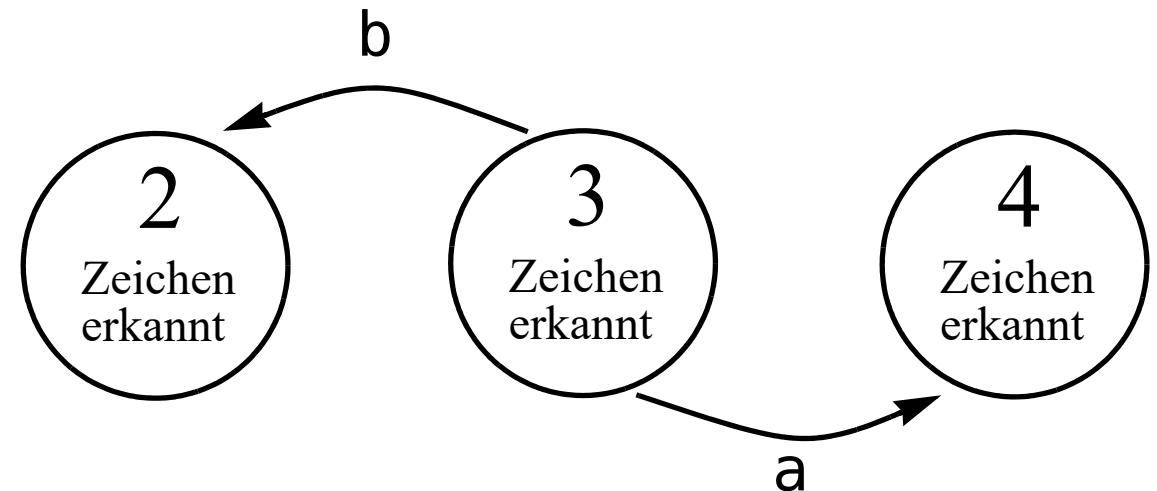
\*) Rechnerstrukturen: Foliensatz Synchrone Schaltwerke

## Moore-Automat

(Fortsetzung)

für die Suche nach dem Muster abaa

bekanntes Beispiel: drei Zeichen aba sind erkannt



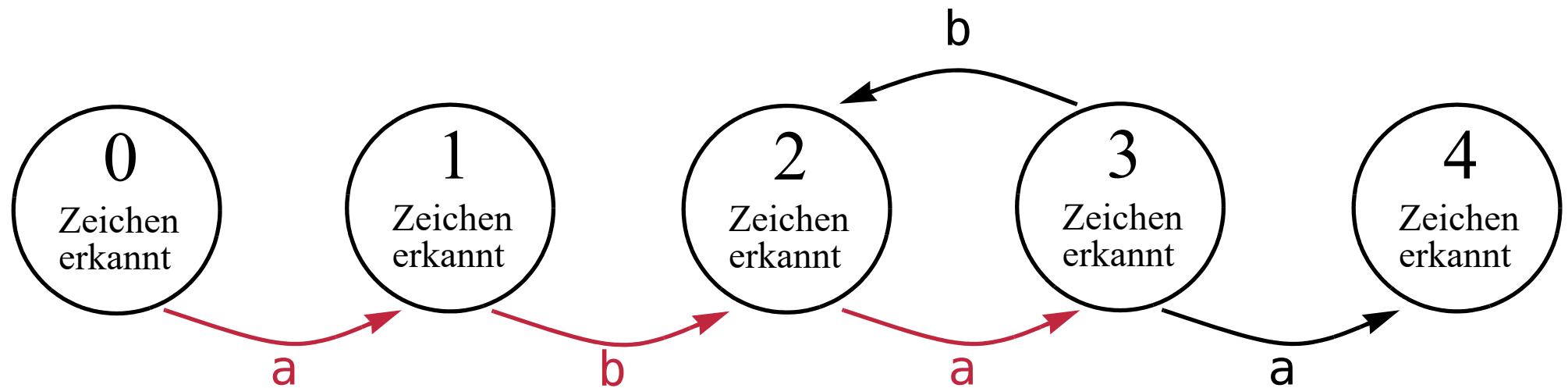
○ ist ein Zustand

→ ist ein Zustandsübergang bei Eingabe von x

## Moore-Automat

(Fortsetzung)

für die Suche nach dem Muster abaa

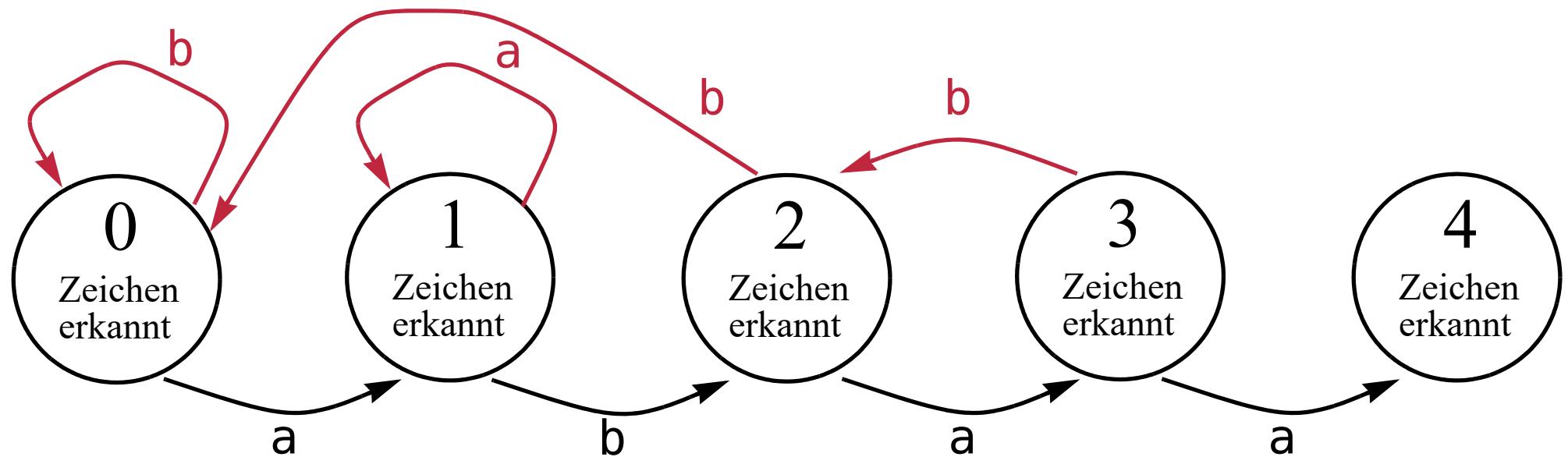


Übergänge, wenn die gesuchten Zeichen auftreten

## Moore-Automat

(Fortsetzung)

für die Suche nach dem Muster abaa

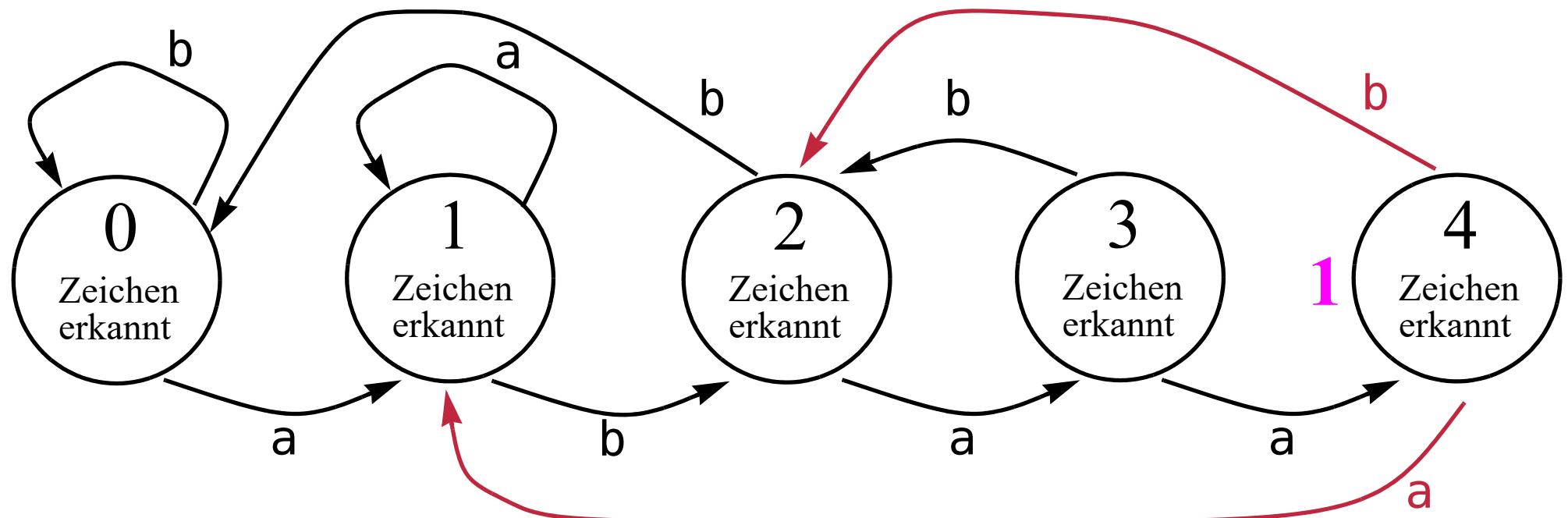


Übergänge, wenn die gesuchten Zeichen nicht auftreten

## Moore-Automat

(Fortsetzung)

für die Suche nach dem Muster abaa



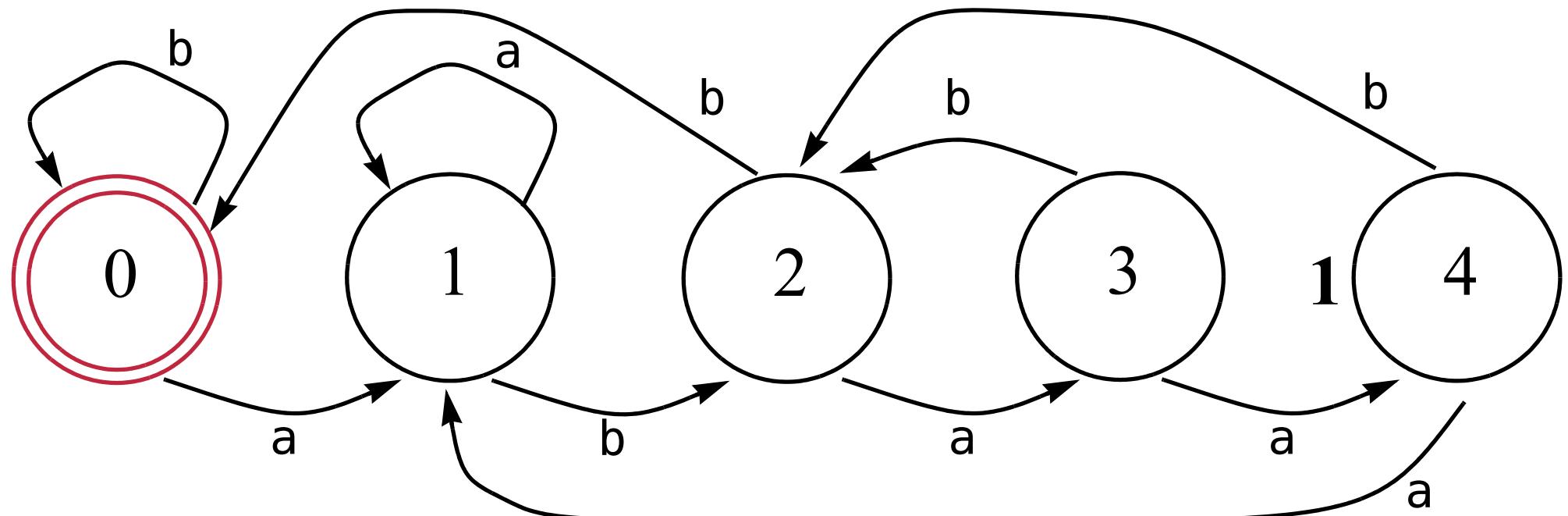
Übergänge und Ausgabe, wenn das gesamte Muster erkannt wurde

Ausgaben erfolgen im Moore-Automat beim Betreten eines Zustands

## Moore-Automat

(Fortsetzung)

für die Suche nach dem Muster abaa

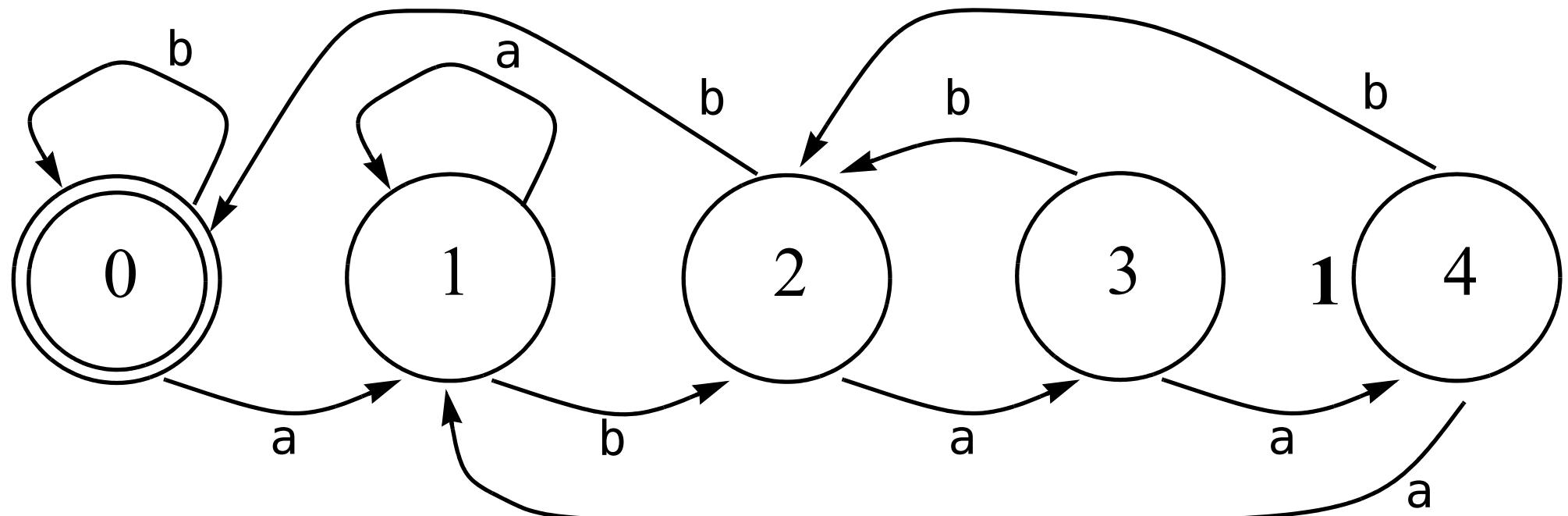


Startzustand festlegen

## Moore-Automat

(Fortsetzung)

für die Suche nach dem Muster abaa



*nicht eingezeichnet:*

alle nicht im Muster  $p$  vorkommenden Zeichen führen immer zum Startzustand zurück

## Moore-Automat

(Fortsetzung)

für die Suche nach dem Muster abaa:

$$M_{\text{abaa}} = ( Q, 0, \Sigma, \Delta, \delta, \lambda )$$

Zustandmenge

$$Q = \{ 0, 1, 2, 3, 4 \}$$

Startzustand

$$0 \in Q$$

Eingabealphabet

$$\Sigma = \{ a, b \} \cup \Sigma_t \quad \text{mit } \Sigma_t \text{ Alphabet des Textes } t$$

Ausgabealphabet

$$\Delta = \{ 1 \} \cup \{ \varepsilon \} \quad \text{mit } \varepsilon = \text{leeres Wort}$$

Ausgabefunktion

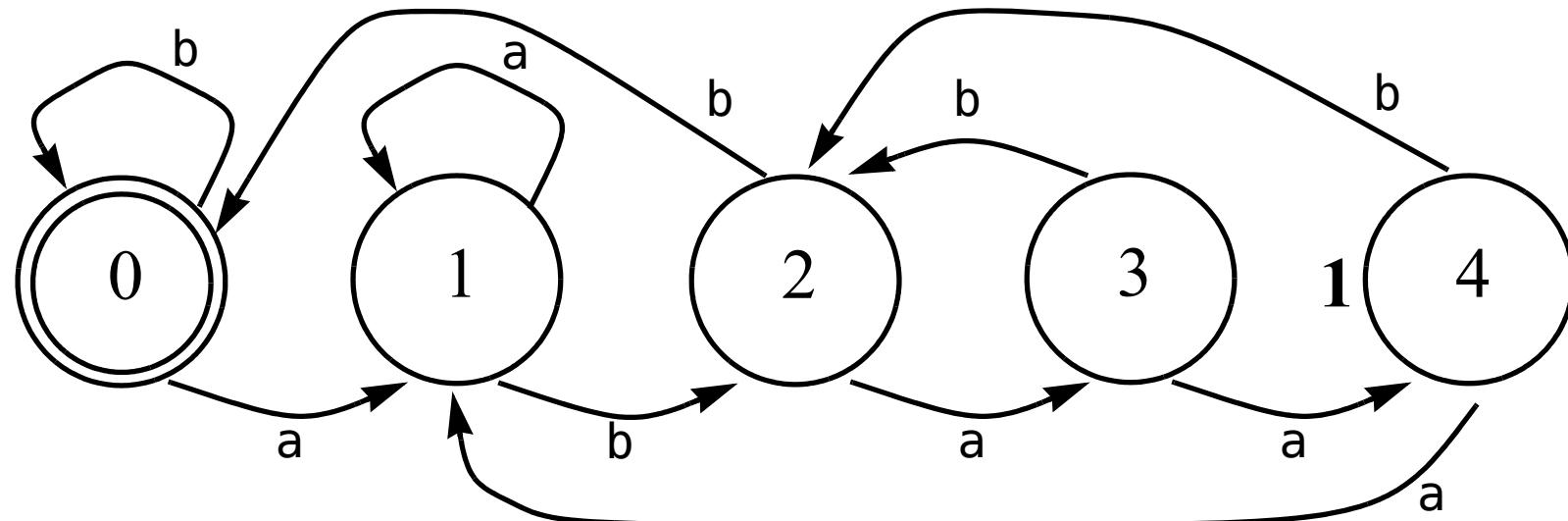
$$\lambda: Q \rightarrow \Delta \quad \text{mit } \lambda(4) = 1 \text{ und } \lambda(q) = \varepsilon \text{ für alle } q \in Q \setminus \{ 4 \}$$

Zustandsüberführungsfunktion

$$\delta: Q \times \Sigma \rightarrow Q$$

## Moore-Automat

(Fortsetzung)



Zustandsübergangsfunktion

$$\delta: Q \times \Sigma \rightarrow Q$$

$\delta$ :

$Q \times \Sigma$	0	1	2	3	4
a	1	1	3	4	1
b	0	2	0	2	2
$x \in \Sigma_t \setminus \{a, b\}$	0	0	0	0	0

## Moore-Automat

(Fortsetzung)

für die Suche nach dem Muster abaa:

$$M_{\text{abaa}} = (Q, 0, \Sigma, \Delta, \delta, \lambda)$$

Zustandmenge

$$Q = \{0, 1, 2, 3, 4\}$$

Startzustand

$$0 \in Q$$

Eingabealphabet

$$\Sigma = \{a, b\} \cup \Sigma_t \quad \text{mit } \Sigma_t \text{ Alphabet des Textes } t$$

Ausgabealphabet

$$\Delta = \{1\} \cup \{\varepsilon\} \quad \text{mit } \varepsilon = \text{leeres Wort}$$

Ausgabefunktion

$$\lambda: Q \rightarrow \Delta \quad \text{mit } \lambda(4) = 1 \text{ und } \lambda(q) = \varepsilon \text{ für alle } q \in Q \setminus \{4\}$$

Zustandsüberführungsfunktion

$$\delta: Q \times \Sigma \rightarrow Q$$

$Q \times \Sigma$	0	1	2	3	4
a	1	1	3	4	1
b	0	2	0	2	2
$x \in \Sigma_t \setminus \{a, b\}$	0	0	0	0	0

## Moore-Automat

(Fortsetzung)

allgemeine Gestaltung von  $M_p = (Q, 0, \Sigma, \Delta, \delta, \lambda)$

Zustandmenge

$$Q = \{ 0, \dots, \text{length}(p) \}$$

ergibt sich direkt aus der Länge von  $p$

Startzustand

$$\text{ist immer } 0 \in Q$$

kein Zeichen erkannt

Eingabealphabet

$$\Sigma = \Sigma_p \cup \Sigma_t$$

Ausgabealphabet

$$\text{immer } \Delta = \{ 1 \} \cup \{ \epsilon \}$$

Ausgabefunktion

$$\lambda: Q \rightarrow \Delta \quad \text{immer mit}$$

$$\lambda(\text{length}(p)) = 1 \text{ und } \lambda(q) = \epsilon \text{ für alle } q \in Q \setminus \{ \text{length}(p) \}$$

Zustandsüberführungsfunktion

...

## Moore-Automat

(Fortsetzung)

allgemeine Gestaltung von  $M_p = (Q, 0, \Sigma, \Delta, \delta, \lambda)$

Zustandmenge

$$Q = \{ 0, \dots, \text{length}(p) \}$$

ergibt sich direkt aus der Länge von  $p$

Startzustand

$$\text{ist immer } 0 \in Q$$

kein Zeichen erkannt

Eingabealphabet

$$\Sigma = \Sigma_p \cup \Sigma_t$$

Ausgabealphabet

$$\text{immer } \Delta = \{ 1 \} \cup \{ \epsilon \}$$

Ausgabefunktion

$$\lambda: Q \rightarrow \Delta \quad \text{immer mit}$$

$$\lambda(\text{length}(p)) = 1 \text{ und } \lambda(q) = \epsilon \text{ für alle } q \in Q \setminus \{ \text{length}(p) \}$$

Zustandsüberführungsfunktion

$w_n$  sei der Anfang – die ersten  $n$  Zeichen – eines Wortes  $w$   
 $\text{prefixes}(w)$  sei die Menge aller Anfangsstücke eines Textes  $w$ ,  
 also  $\text{prefixes}(w) = \{ w_0, w_1, w_2, \dots, w_{n-1}, w \}$

Beispiel: Für  $w = \text{abaa}$  sind

$$w_0 = \epsilon, w_1 = a, w_2 = ab, w_3 = aba, w_4 = \text{abaa}$$

Also gilt:

$$\text{prefixes}(\text{abaa}) = \{ \epsilon, a, ab, aba, \text{abaa} \}$$

## Moore-Automat

(Fortsetzung)

allgemeine Gestaltung von  $M_p = (Q, 0, \Sigma, \Delta, \delta, \lambda)$

Zustandmenge

$$Q = \{ 0, \dots, \text{length}(p) \}$$

ergibt sich direkt aus der Länge von  $p$

Startzustand

$$\text{ist immer } 0 \in Q$$

kein Zeichen erkannt

Eingabealphabet

$$\Sigma = \Sigma_p \cup \Sigma_t$$

Ausgabealphabet

$$\text{immer } \Delta = \{ 1 \} \cup \{ \epsilon \}$$

Ausgabefunktion

$$\lambda: Q \rightarrow \Delta \quad \text{immer mit}$$

$$\lambda(\text{length}(p)) = 1 \text{ und } \lambda(q) = \epsilon \text{ für alle } q \in Q \setminus \{ \text{length}(p) \}$$

Zustandsüberführungsfunktion

$w_n$  sei der Anfang – die ersten  $n$  Zeichen – eines Wortes  $w$

$\text{prefixes}(w)$  sei die Menge aller Anfangsstücke eines Textes  $w$ ,

$$\text{also } \text{prefixes}(w) = \{ w_0, w_1, w_2, \dots, w_{n-1}, w \}$$

$\text{suffixes}(w)$  sei (analog) die Menge aller Endstücke eines Textes  $w$

Beispiel: Für  $w = \text{abaa}$  gilt dann:

$$\text{suffixes}(\text{abaa}) = \{ \epsilon, a, aa, baa, abaa \}$$

## Moore-Automat

(Fortsetzung)

allgemeine Gestaltung von  $M_p = (Q, 0, \Sigma, \Delta, \delta, \lambda)$

Zustandmenge

$$Q = \{ 0, \dots, \text{length}(p) \}$$

ergibt sich direkt aus der Länge von  $p$

Startzustand

$$\text{ist immer } 0 \in Q$$

kein Zeichen erkannt

Eingabealphabet

$$\Sigma = \Sigma_p \cup \Sigma_t$$

Ausgabealphabet

$$\text{immer } \Delta = \{ 1 \} \cup \{ \epsilon \}$$

Ausgabefunktion

$$\lambda: Q \rightarrow \Delta \quad \text{immer mit}$$

$$\lambda(\text{length}(p)) = 1 \text{ und } \lambda(q) = \epsilon \text{ für alle } q \in Q \setminus \{ \text{length}(p) \}$$

Zustandsüberführungsfunktion

$w_n$  sei der Anfang – die ersten  $n$  Zeichen – eines Wortes  $w$

$\text{prefixes}(w)$  sei die Menge aller Anfangsstücke eines Textes  $w$ ,

$$\text{also } \text{prefixes}(w) = \{ w_0, w_1, w_2, \dots, w_{n-1}, w \}$$

$\text{suffixes}(w)$  sei (analog) die Menge aller Endstücke eines Textes  $w$

Dann gilt  $\delta: Q \times \Sigma \rightarrow Q$

...

## Moore-Automat

(Fortsetzung)

allgemeine Gestaltung von  $M_p = (Q, 0, \Sigma, \Delta, \delta, \lambda)$

Zustandmenge

$$Q = \{ 0, \dots, \text{length}(p) \}$$

ergibt sich direkt aus der Länge von  $p$

Startzustand

$$\text{ist immer } 0 \in Q$$

kein Zeichen erkannt

Eingabealphabet

$$\Sigma = \Sigma_p \cup \Sigma_t$$

Ausgabealphabet

$$\text{immer } \Delta = \{ 1 \} \cup \{ \epsilon \}$$

Ausgabefunktion

$$\lambda: Q \rightarrow \Delta \quad \text{immer mit}$$

$$\lambda(\text{length}(p)) = 1 \text{ und } \lambda(q) = \epsilon \text{ für alle } q \in Q \setminus \{ \text{length}(p) \}$$

Zustandsüberführungsfunktion

$w_n$  sei der Anfang – die ersten  $n$  Zeichen – eines Wortes  $w$

$\text{prefixes}(w)$  sei die Menge aller Anfangsstücke eines Textes  $w$ ,

$$\text{also } \text{prefixes}(w) = \{ \epsilon, w_1, w_2, \dots, w_{n-1}, w \}$$

$\text{suffixes}(w)$  sei (analog) die Menge aller Endstücke eines Textes  $w$

Dann gilt

$$\delta: Q \times \Sigma \rightarrow Q$$

$$\delta(n, c) = n' \text{ mit } n' = \text{length}(w_{\max})$$

und  $w_{\max}$  ist längstes Wort in  $\text{suffixes}(p_n c) \cap \text{prefixes}(p)$

Insbesondere gilt daher immer:  $\delta(n, x) = 0$  für alle  $x \in \Sigma_t \setminus \Sigma_p$ ,  
da  $\text{suffixes}(p_n x) \cap \text{prefixes}(p) = \{\epsilon\}$  mit  $\text{length}(\epsilon) = 0$  ist.

## Moore-Automat

(Fortsetzung)

Zustandsüberführungsfunktion ...

Dann gilt  $\delta: Q \times \Sigma \rightarrow Q$   
 $\delta(n, c) = n'$  mit  $n' = \text{length}(w_{\max})$   
 und  $w_{\max}$  ist längstes Wort in  $\text{suffixes}(p_n c) \cap \text{prefixes}(p)$

Beispiel: Für  $w = abaa$  gilt:

$$\text{prefixes}(abaa) = \{ \epsilon, a, \mathbf{ab}, \mathbf{aba}, \mathbf{abaa} \}$$

$$\text{suffixes}(abab) = \{ \epsilon, b, \mathbf{ab}, \mathbf{bab}, \mathbf{abab} \}$$

$$\text{length}(\mathbf{ab}) = 2$$

bisher wurde erkannt: aba

$Q \times \Sigma$	0	1	2	3	4
a	1	1	3	4	1
b	0	2	0	2	2
$x \in \Sigma_t \setminus \{a, b\}$	0	0	0	0	0

## Implementierung der Mustererkennung

- ❑ Als Vorbereitung für die Mustererkennung muss also zunächst der passende Moore-Automat  $M_p$  erzeugt werden.
- ❑ Diese Erzeugung von  $M_p$  ist abhängig von dem zu suchenden Muster  $p$ , aber unabhängig von dem zu durchsuchenden Text  $t$ .

$$p \longrightarrow M_p$$

- ❑ Der Text  $t$  wird anschließend als Eingabe für Moore-Automat  $M_p$  benutzt.
  - einfache Suche: erfolgreich, wenn der Automat die erste Ausgabe von 1 erzeugt.
  - Zählen der Vorkommen von  $p$  in  $t$ : Summe aller Ausgaben bilden

## Überlegungen zur Implementierung der Erzeugung des Automaten $M_p$ (Fortsetzung)

- Zustandmenge:  $Q = \{ 0, \dots, \text{length}(p) \}$

Da die Zustände fortlaufend durch natürliche Zahlen bezeichnet werden, wird keine besondere Datenstruktur zur Verwaltung benötigt. Variablen des Typs `int` sind ausreichend für die Ablage von Zuständen.

- Startzustand ist immer der Zustand 0, also ein eindeutiger `int`-Wert.

- Eingabealphabet:  $\Sigma = \Sigma_p \cup \Sigma_t$

Für alle Elemente von  $\Sigma_p$  müssen Zustandsübergänge bestimmt werden, so dass  $\Sigma_p$  explizit ermittelt und gespeichert werden muss. Das kann einfach geschehen, indem jedes Zeichen aus  $p$  zu einer Menge hinzugefügt wird:

Set<Character> generateTokens(String s)

- Ausgabealphabet:  $\Delta = \{ 1 \} \cup \{ \epsilon \}$

Das einfache Ausgabealphabet muss nicht explizit implementiert werden.

- Ausgabefunktion:  $\lambda: Q \rightarrow \Delta$

Die Ausgabefunktion muss nicht explizit implementiert werden, da die einzige Ausgabe immer für genau den einen Zustand  $\text{length}(p)$  erfolgt.

## Überlegungen zur Implementierung der Erzeugung des Automaten $M_p$ (Fortsetzung)

- Zustandsüberführungsfunktion:  $\delta: Q \times \Sigma \rightarrow Q$

Für die Umsetzung des Algorithmus wird die Prefix-Menge zu  $p$  benötigt. Diese kann einfach durch wiederholtes Aufrufen der Methode `substring` der Klasse `String` bestimmt werden:

`Set<String> generatePrefixes( String s )`

## Überlegungen zur Implementierung der Erzeugung des Automaten $M_p$ (Fortsetzung)

- Zustandsüberführungsfunktion:  $\delta: Q \times \Sigma \rightarrow Q$

Für die Umsetzung des Algorithmus wird die Prefix-Menge zu  $p$  benötigt. Diese kann einfach durch wiederholtes Aufrufen der Methode `substring` der Klasse `String` bestimmt werden:

`Set<String> generatePrefixes( String s )`

- Das Interface `Set` gibt Methoden für eine *Menge* vor.
- `Set` ist ein Interface aus dem Paket `java.util`.
- `Set<String>` beschreibt eine Menge von Texten, die durch Objekte des Typs `String` repräsentiert werden.

## Interface `java.util.Set<E> extends Collection<E>`

Eine Auswahl aus den durch das Interface `Collection` vorgegebenen Methoden:

- ❑ `boolean add(E e)` – fügt `e` zu Menge hinzu, falls `e` noch nicht enthalten ist
- ❑ `boolean addAll(Collection<? extends E> c)` – fügt alle Inhalte von `c` zu der Menge hinzu, die noch nicht enthalten sind (*Vereinigung*)
- ❑ `boolean contains(Object o)` – gibt `true` zurück, falls `o` enthalten ist
- ❑ `boolean isEmpty()` – gibt `true` zurück, falls die Menge leer ist
- ❑ `boolean remove(Object o)` – löscht `o` aus der Menge, falls `o` enthalten ist; gibt `true` zurück, falls gelöscht wurde
- ❑ `boolean removeAll(Collection<?> c)` – löscht alle Inhalte von `c` aus der Menge, die in dieser enthalten sind
- ❑ `boolean retainAll(Collection<?> c)` – belässt alle Elemente in der Menge, die auch in `c` enthalten sind (*Durchschnitt*)
- ❑ `int size()` – gibt die Anzahl der Elemente in der Menge zurück
- ❑ `Iterator<E> iterator()` – gibt einen Iterator zurück, mit dem über die Elemente der Menge iteriert werden kann

**Interface `java.util.Set<E>` extends `Collection<E>`**

(Fortsetzung)

Deklaration des Rückgabetyps von Methoden:

```
boolean addAll( Collection<? extends E> c )
```

- ❑ Die zulässigen Typargumente werden eingeschränkt durch: ? **extends** E  
Erlaubt sind nur Typen, die E erweitern oder implementieren.
- ❑ Aufgrund der durch Vererbung/Implementierung gegebenen Typkompatibilität reicht es aus, wenn die von der Oberklasse von E sichergestellten Methoden verfügbar sind.
- ❑ Da der Typ der Unterklasse selbst in der Deklaration der Signatur nicht benötigt wird, wird die *Wildcard* ? als Platzhalter verwendet.
- ❑ Erinnerung: Für Typeinschränkungen gilt, dass T selbst die Bedingung **extends** T erfüllt.

## Überlegungen zur Implementierung der Erzeugung des Automaten $M_p$ (Fortsetzung)

- Zustandsüberführungsfunktion:  $\delta: Q \times \Sigma \rightarrow Q$

Für die Umsetzung des Algorithmus wird die Prefix-Menge zu  $p$  benötigt. Diese kann einfach durch wiederholtes Aufrufen der Methode `substring` der Klasse `String` bestimmt werden:

```
Set<String> generatePrefixes( String s )
```

Weiterhin werden Suffix-Mengen zu wechselnden Texten benötigt. Diese können ebenfalls durch wiederholtes Aufrufen der Methode `substring` bestimmt werden. Allerdings muss das längste Endstück von  $p_n c$  bestimmt werden, das in `prefixes(p)` ist. Daher werden die Anfangsstücke nach ihrer Länge sortiert in einer Liste abgelegt, so dass eine Überprüfung direkt mit den längeren Endstücken beginnen kann:

```
List<String> generateSuffixes( String s )
```

## Überlegungen zur Implementierung der Erzeugung des Automaten $M_p$ (Fortsetzung)

- Zustandsüberführungsfunktion:  $\delta: Q \times \Sigma \rightarrow Q$

Für die Umsetzung des Algorithmus wird die Prefix-Menge zu  $p$  benötigt. Diese kann einfach durch wiederholtes Aufrufen der Methode `substring` der Klasse `String` bestimmt werden:

```
Set<String> generatePrefixes( String s )
```

Weiterhin werden Suffix-Mengen zu wechselnden Texten benötigt. Diese können ebenfalls durch wiederholtes Aufrufen der Methode `substring` bestimmt werden. Allerdings muss das längste Endstück von  $p_n c$  bestimmt werden, das in `prefixes(p)` ist. Daher werden die Anfangsstücke nach ihrer Länge sortiert in einer Liste abgelegt, so dass eine Überprüfung direkt mit den längeren Endstücken beginnen kann:

```
List<String> generateSuffixes( String s )
```

- Das Interface `List` gibt Methoden für eine *Liste* vor.
  - `List` ist ein Interface aus dem Paket `java.util`.
  - `List<String>` beschreibt eine Liste von Texten, die durch Objekte des Typs `String` repräsentiert werden.

## Interface `java.util.List<E>` extends `Collection<E>`

Das Interface gibt neben den für das Interface `Collection` vorgegebenen Methoden zusätzlich solche Operationen vor, die sich an der Listenstruktur orientieren.

- ❑ `E get(int index)` – gibt den Inhalt der Liste an der Position `index` zurück
- ❑ `int indexOf(Object o)` – gibt die Position des ersten Vorkommens von `o` in der Liste zurück; falls `o` nicht enthalten ist, wird `-1` zurückgegeben
- ❑ `int lastIndexOf(Object o)` – gibt die Position des letzten Vorkommens von `o` in der Liste zurück; falls `o` nicht enthalten ist, wird `-1` zurückgegeben
- ❑ `E set(int index, E element)` – ersetzt das Element an der Position `index` durch `e`
- ❑ `List<E> subList(int fromIndex, int toIndex)` – gibt eine Teilliste zurück

Anmerkung:

Die Semantik der aus dem Interface `Collection` übernommenen Methoden ist der sequentiellen Datenstruktur der Liste angepasst, beispielsweise gilt:

`boolean add(E e)` – fügt `e` **am Ende der Liste an**

## Überlegungen zur Implementierung der Erzeugung des Automaten $M_p$ (Fortsetzung)

- Zustandsüberführungsfunktion:  $\delta: Q \times \Sigma \rightarrow Q$

Für die Umsetzung des Algorithmus wird die Prefix-Menge zu  $p$  benötigt. Diese kann einfach durch wiederholtes Aufrufen der Methode `substring` der Klasse `String` bestimmt werden:

```
Set<String> generatePrefixes( String s )
```

Weiterhin werden Suffix-Mengen zu wechselnden Texten benötigt. Diese können ebenfalls durch wiederholtes Aufrufen der Methode `substring` bestimmt werden. Allerdings muss das längste Endstück von  $p_n c$  bestimmt werden, das in `prefixes(p)` ist. Daher werden die Anfangsstücke nach ihrer Länge sortiert in einer Liste abgelegt, so dass eine Überprüfung direkt mit den längeren Endstücken beginnen kann:

```
List<String> generateSuffixes( String s )
```

Die Überführungsfunktion ist eine Abbildung, die jedem Zustand für jedes Zeichen einen Folgezustand zuordnet. Für die Implementierung bietet sich daher eine Klasse an, die das **Map**-Interface implementiert, das solche Abbildungen beschreibt. Die Realisierung erfolgt durch zwei Abbildungen `Map<Integer, Map<Character, Integer>>`.

Das «äußere» Abbildung (`Map`) ordnet jedem Zustand (`Integer`) eine Abbildung (`Map`) zu, die jedem Eingabezeichen (`Character`) den entsprechenden Nachfolgezustand (`Integer`) zuordnet.

## Überlegungen zur Implementierung der Erzeugung des Automaten $M_p$ (Fortsetzung)

- Zustandsüberführungsfunktion:  $\delta: Q \times \Sigma \rightarrow Q$

Für die Umsetzung des Algorithmus wird die Prefix-Menge zu  $p$  benötigt. Diese kann einfach durch wiederholtes Aufrufen der Methode `substring` der Klasse `String` bestimmt werden:

```
Set<String> generatePrefixes( String s )
```

Weiterhin werden Suffix-Mengen zu wechselnden Texten benötigt. Diese können ebenfalls durch wiederholtes Aufrufen der Methode `substring` bestimmt werden. Allerdings muss das längste Endstück von  $p_n c$  bestimmt werden, das in `prefixes(p)` ist. Daher werden die Anfangsstücke nach ihrer Länge sortiert in einer Liste abgelegt, so dass eine Überprüfung direkt mit den längeren Endstücken beginnen kann:

```
List<String> generateSuffixes( String s )
```

Die Überführungsfunktion ist eine Abbildung, die jedem Zustand für jedes Zeichen einen Folgezustand zuordnet. Für die Implementierung bietet sich daher eine Klasse an, die das Map-Interface implementiert, das solche Abbildungen beschreibt. Die Realisierung erfolgt durch zwei Abbildungen `Map<Integer, Map<Character, Integer>>`.

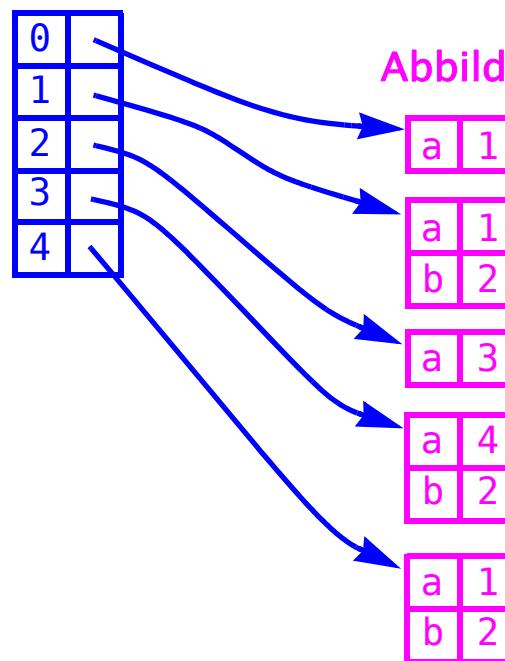
Das «äußere» Abbildung (`Map`) ordnet jedem Zustand (`Integer`) eine Abbildung (`Map`) zu, die jedem Eingabezeichen (`Character`) den entsprechenden Nachfolgezustand (`Integer`) zuordnet.

Überführungen in den Zustand  $\emptyset$ , die u.a. für alle  $\Sigma_t \setminus \Sigma_p$  vorgenommen werden müssen, werden nicht in die Abbildung aufgenommen.

# Überlegungen zur Implementierung der Erzeugung des Automaten $M_p$ (Fortsetzung)

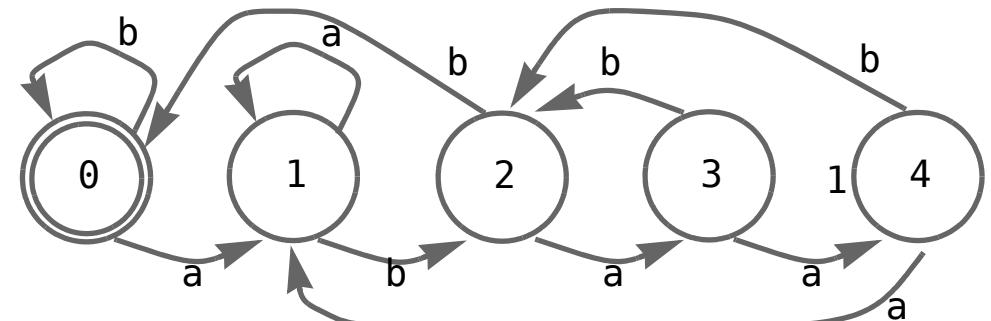
- Zustandsüberführungsfunktion:  $\delta: Q \times \Sigma \rightarrow Q$   
Implementierung als `Map<Integer, Map<Character, Integer>>`

Abbildung `Integer → Map<Character, Integer>`



Abbildungen `Character → Integer`

Beispiel:



## Interface `java.util.Map<K,V>`

Das Interface gibt Methoden vor, die den Umgang mit einer partiellen Abbildung  $f: K \rightarrow V$  ermöglichen, beispielsweise:

- ❑ `boolean containsKey(Object key)` – gibt `true` zurück, falls `key` ein Wert aus der Zielmenge `V` zugeordnet wurde
- ❑ `boolean containsValue(Object value)` – gibt `true` zurück, falls `value` in der Zielmenge der Abbildung enthalten ist
- ❑ `V get(Object key)` – gibt den Wert der Zielmenge zurück, der `key` zugeordnet ist; falls `key` kein Wert zugeordnet ist, wird `null` zurückgegeben
- ❑ `Set<K> keySet()` – gibt alle Werte des Definitionsbereichs zurück
- ❑ `V put(K key, V value)` – ordnet `key` den Wert `value` zu
- ❑ `V remove(Object key)` – entfernt das `key` zugeordnete Element der Zielmenge, falls ein solches existiert
- ❑ `boolean remove(Object key, Object value)` – entfernt eine Zuordnung aus der Abbildung
- ❑ `V replace(K key, V value)` – ersetzt die Zuordnung zu `key` durch `value` nur dann, wenn `key` bereits ein Element der Zielmenge zugeordnet ist

## Implementierung der Erzeugung des Automaten $M_p$

```
public static Set<Character> generateTokens( String s )
{
    Set<Character> result = new HashSet<>();
    for ( int i = 0; i < s.length(); i++ )
    {
        result.add( s.charAt( i ) );
    }
    return result;
}
```

- ❑ Set ist das bereits bekannte Interface aus dem Paket `java.util`. `Set<Character>` beschreibt eine Menge von Zeichen, die durch Objekte des Typs `Character` repräsentiert werden.
- ❑ `java.util.HashSet` ist eine Klasse, die das Interface `Set` implementiert.  
Die Implementierung basiert auf einer *Hashtable* (und wird später genauer betrachtet).
- ❑ `new HashSet<>()` erzeugt ein Objekt dieser Klasse, das als Elemente Objekte erwartet, die kompatibel zu `Character` sind.
- ❑ Die Methode `charAt` bestimmt das Zeichen an der als Argument übergebenen Position des `String`-Objekts.
- ❑ Da `result` eine Menge realisiert, werden auch mehrfach hinzugefügte Zeichen nur einmal gespeichert.

## Implementierung der Erzeugung des Automaten $M_p$

(Fortsetzung)

```
private static Set<String> generatePrefixes( String s )
{
    Set<String> result = new HashSet<String>();
    for( int i = 0; i <= s.length(); i++ )
    {
        result.add( s.substring( 0, i ) );
    }
    return result;
}
```

- ❑ Die Methode `substring( int from, int to )` liefert den Ausschnitt des Textes im Bereich (einschließlich) `from` bis (ausschließlich) `to`.
- ❑ Da immer größere Teilstexte gebildet werden, können hier keine doppelten Einträge in die Menge `result` erfolgen.

## Implementierung der Erzeugung des Automaten $M_p$

(Fortsetzung)

```
private static List<String> generateSuffixes( String s )
{
    List<String> result = new LinkedList<String>();
    for( int i = 0; i < s.length(); i++ )
    {
        result.add( s.substring( i, s.length() ) );
    }
    return result;
}
```

- ❑ `List` ist das bereits bekannte Interface aus dem Paket `java.util`.
- ❑ `java.util.LinkedList` ist eine Klasse aus dem Paket , die das Interface `Set` implementiert.
- ❑ Es werden die Endstücke des Textes `s` so gebildet, dass zunächst die längeren und anschließend kürzer werdende Texte erzeugt werden.
- ❑ Durch das Einsortieren in eine Liste liefert die Iteration über die Liste ebenfalls zunächst die längeren und anschließend die kürzer werdenden Endstücke.

## Implementierung der Erzeugung des Automaten $M_p$

(Fortsetzung)

```

private static Map<Integer,Map<Character,Integer>> generatetransitionTable( String p )
{
    Map<Integer,Map<Character,Integer>> transitionTable = new HashMap<>();
    Set<Character> tokens = generateTokens( p );
    Set<String> prefixes = generatePrefixes( p ); Initialisierungen
    for ( String prefix : prefixes ) {
        for ( char c : tokens ) {
            String continuation = prefix + c;
            List<String> suffixes = generateSuffixes( continuation );
            for ( String suffix : suffixes ) {
                if ( prefixes.contains( suffix ) ) {
                    if ( ! transitionTable.containsKey( prefix.length() ) ) {
                        transitionTable.put(
                            prefix.length(), new HashMap<Character,Integer>());
                    }
                    transitionTable.get( prefix.length() ).put( c, suffix.length() );
                    break; ← bricht Schleife ab
                }
            }
        }
    }
    return transitionTable;
}

```

## Kontrolle von Schleifen mit break

- ❑ Die **break**-Anweisung ermöglicht das Abbrechen der Ausführung von Schleifen.
- ❑ Beispiel **break**:

```
while ( true )
{
    account = requestAccountFromUser();
    password = requestPasswordFromUser();
    if ( loginOk( account, password ) )
    {
        break;
    }
}
```

- ❑ Funktionsweise **break**:
  - Die Ausführung der **break**-Anweisung unterbricht die Ausführung der (innersten) umgebenden Schleife .
  - Die Programmausführung wird mit der *ersten Anweisung hinter* der Schleife fortgesetzt.

## Implementierung der Erzeugung des Automaten $M_p$

(Fortsetzung)

```

private static Map<Integer,Map<Character,Integer>> generatetransitionTable( String p )
{
    Map<Integer,Map<Character,Integer>> transitionTable = new HashMap<>();
    Set<Character> tokens = generateTokens( p );
    Set<String> prefixes = generatePrefixes( p );
    for ( String prefix : prefixes ) {
        for ( char c : tokens ) {
            String continuation = prefix + c;
            List<String> suffixes = generateSuffixes( continuation );
            for ( String suffix : suffixes ) {
                if ( prefixes.contains( suffix ) ) {
                    if ( ! transitionTable.containsKey( prefix.length() ) ) {
                        transitionTable.put(
                            prefix.length(), new HashMap<Character,Integer>());
                    }
                    transitionTable.get( prefix.length() ).put( c, suffix.length() );
                    break;
                }
            }
        }
    }
    return transitionTable;
}

```

- ❑ **new** **HashMap**<>() erzeugt ein Objekt der Klasse, das als Elemente Objekte erwartet, die kompatibel zu Integer und Map<Character, Integer> sind.

## Implementierung der Erzeugung des Automaten $M_p$

(Fortsetzung)

```

private static Map<Integer,Map<Character,Integer>> generatetransitionTable( String p )
{
    Map<Integer,Map<Character,Integer>> transitionTable = new HashMap<>();
    Set<Character> tokens = generateTokens( p );
    Set<String> prefixes = generatePrefixes( p );
    for ( String prefix : prefixes ) {
        for ( char c : tokens ) {
            String continuation = prefix + c;
            List<String> suffixes = generateSuffixes( continuation );
            for ( String suffix : suffixes ) {
                if ( prefixes.contains( suffix ) ) {
                    if ( ! transitionTable.containsKey( prefix.length() ) ) {
                        transitionTable.put(
                            prefix.length(), new HashMap<Character,Integer>());
                    }
                    transitionTable.get( prefix.length() ).put( c, suffix.length() );
                    break;
                }
            }
        }
    }
    return transitionTable;
}

```

- ❑ Set erweitert das Interface `java.util.Iterable`.  
Jede Menge lässt sich also mit einem Iterator durchlaufen.
- ❑ Die `for-each`-Schleife nutzt – wenn möglich – *implizit* einen Iterator, wenn die durchlaufene Datenstruktur das Interface `java.util.Iterable` erweitert.

## Implementierung der Erzeugung des Automaten $M_p$

(Fortsetzung)

```

private static Map<Integer,Map<Character,Integer>> generatetransitionTable( String p )
{
    Map<Integer,Map<Character,Integer>> transitionTable = new HashMap<>();
    Set<Character> tokens = generateTokens( p );
    Set<String> prefixes = generatePrefixes( p );
    for ( String prefix : prefixes ) {
        for ( char c : tokens ) {
            String continuation = prefix + c;
geordnete  
suffix-Menge erzeugen
            List<String> suffixes = generateSuffixes( continuation );
            for ( String suffix : suffixes ) {
                if ( prefixes.contains( suffix ) ) {
                    if ( ! transitionTable.containsKey( prefix.length() ) ) {
                        transitionTable.put(
                            prefix.length(), new HashMap<Character,Integer>());
                    }
                    transitionTable.get( prefix.length() ).put( c, suffix.length() );
das zuerst gefundene passende Endstück  
ist aufgrund der Konstruktion auch immer  
das längste
                    break;
                }
            }
        }
    }
    return transitionTable;
}

```

## Implementierung der Erzeugung des Automaten $M_p$

(Fortsetzung)

```

private static Map<Integer,Map<Character,Integer>> generatetransitionTable( String p )
{
    Map<Integer,Map<Character,Integer>> transitionTable = new HashMap<>();
    Set<Character> tokens = generateTokens( p );
    Set<String> prefixes = generatePrefixes( p );
    for ( String prefix : prefixes ) {
        for ( char c : tokens ) {
            String continuation = prefix + c;
            List<String> suffixes = generateSuffixes( continuation );
            for ( String suffix : suffixes ) {
                if ( prefixes.contains( suffix ) ) {
                    if ( ! transitionTable.containsKey( prefix.length() ) ) {
                        transitionTable.put(
                            prefix.length(), new HashMap<Character,Integer>());
                    }
                    transitionTable.get( prefix.length() ).put( c, suffix.length() );
                    break;
                }
            }
        }
    }
    return transitionTable;
}

```

- da **prefix** das bereits gefundene Anfangsstück ist,  
 befindet sich der Automat im Zustand `prefix.length()`  
 - da **suffix** das gefundene Anfangsstück für die  
**Fortsetzung** ist, wechselt der Automat in den Zustand  
`suffix.length()`

## Überlegungen zur Implementierung der Mustererkennung mit dem Automaten $M_p$

- ❑ Ziel: Überprüfung des Inhalts eines Textes
- ❑ Suche nach dem Muster beginnt im Startzustand  $\emptyset$  mit dem ersten Zeichen aus des Textes.
- ❑ Der Folgezustand wird aus der mit `generateTransitionTable` angelegten Tabelle ermittelt:
  - Ist in der Tabelle ein Folgezustand verfügbar, so wird mit diesem weitergearbeitet.
  - Ist in der Tabelle kein Folgezustand verfügbar, so wird mit dem Zustand  $\emptyset$  weitergearbeitet.
  - Wird der Endzustand  $length(p)$  erreicht, so wurde das Muster  $p$  gefunden.
- ❑ Verschiedene Varianten:
  - Feststellen, an welcher Stelle des Textes das Muster  $p$  erstmals vorkommt:  
`int containsAt( String pattern, String text )`
  - Prüfen, ob das Muster  $p$  mindestens einmal im Text vorkommt:  
`boolean contains( String pattern, String text )`
  - Zählen, wie häufig das Muster  $p$  im Text vorkommt:  
`int count( String pattern, String text )`

## Implementierung der Mustererkennung mit dem Automaten $M_p$

```

public static int containsAt( String p, String text )
{
    Map<Integer,Map<Character,Integer>> transitionTable = generateTransitionTable( p );
    int state = 0;
    final int FINALSTATE = p.length();
    for ( int position = 0; position < text.length(); position++ ) {
        if ( transitionTable.containsKey( state )
            && transitionTable.get( state ).containsKey( text.charAt( position ) ) ) {
            state = transitionTable.get( state ).get( text.charAt( position ) );
            if ( state == FINALSTATE )
            {
                return position - p.length() + 1;          Korrektur auf Anfangsposition
            }                                              des gefundenen Musters
        } else {
            state = 0;
        }
    }
    return -1;           ← Muster nicht gefunden
}

```

## Implementierung der Mustererkennung mit dem Automaten $M_p$

(Fortsetzung)

```
public static boolean contains( String p, String text )
{
    return containsAt( p, text ) != -1;
}
```

## Implementierung der Mustererkennung mit dem Automaten $M_p$

(Fortsetzung)

```

public static int count( String p, String text )
{
  Map<Integer,Map<Character,Integer>> transitionTable = generateTransitionTable( p );
  int quantity = 0;
  int state = 0;
  final int FINALSTATE = p.length();
  for ( int position = 0; position < text.length(); position++ ) {
    if ( transitionTable.containsKey( state )
        && transitionTable.get( state ).containsKey( text.charAt( position ) ) ) {
      state = transitionTable.get( state ).get( text.charAt( position ) );
      if ( state == FINALSTATE ) {
        quantity++;
      }
    } else {
      state = 0;
    }
  }
  return quantity;
}

```

**Suche läuft weiter, kein return**

## Zusammenfassung

Das Beispiel Mustererkennung zeigt,

- ❑ dass auch die Lösung einer trivial erscheinenden Aufgabe eventuell noch optimiert werden kann,
- ❑ dass eine formale Beschreibung als Moore-Automat eine präzise Spezifikation des Verhaltens einer Methode darstellt,
- ❑ dass die einfache Umsetzung des Automaten weitgehend auf der Basis vorhandener Klassen erfolgen kann,
- ❑ dass trotzdem der implementierte Algorithmus zum Aufbau der Überführungsfunktion des Automaten von der formalen Beschreibung abweichen kann, um Laufzeit und Speicherbedarf zu optimieren:
  - Endstücke werden in einer (geordneten) Liste abgelegt,
  - Übergänge in den Zustand  $\emptyset$  werden nicht explizit in die Abbildung aufgenommen.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 16. Ausnahmebehandlung/Zugriff auf Dateien

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-1057

## Lernziele des Kapitels Ausnahmebehandlung

Nach Durcharbeiten des Kapitels Ausnahmebehandlung sollen die teilnehmenden Studierenden

- die Vorteile des Konzepts der Ausnahmebehandlung kennen und erklären können,
- Ausnahmeklassen anlegen können,
- Ausnahme-Objekte erzeugen und werfen können,
- Ausnahme-Objekte fangen und verarbeiten können,
- Checked Exceptions* und *Unchecked Exceptions* unterscheiden und einsetzen können,
- Grundlagen des Umgangs mit Dateien in Java kennen.

## Klassenbibliotheken der Java Platform

- ❑ Java stellt zusammen mit dem Compiler bereits viele Hundert Klassen bereit.
- ❑ Einige sind im Rahmen dieser Vorlesung schon vorgestellt worden:  
`String, Math, Object, HashSet, HashMap, LinkedList`
- ❑ Insbesondere stehen auch geeignete Datenstrukturen bereit.  
Eine Liste wie `DoublyLinkedList` muss also nicht von jedem Entwickler selbst neu gestaltet werden, sondern es wird in der Regel die bereits getestete und erprobte Implementierung der Java Platform benutzt.
- ❑ Teilweise stehen auch verschiedene Implementierungen der gleichen Datenstruktur zur Verfügung, die sich dann in einzelnen Eigenschaften – beispielsweise bezüglich der Laufzeit von bestimmten Operationen – unterscheiden:
  - `LinkedList<T>` Die Methode `T get(int Index)` durchläuft die Liste sequentiell vom näheren Ende aus bis zum gewünschten Index.
  - `ArrayList<T>` Die Methode `T get(int Index)` greift in konstanter Zeit auf den gewünschten Index zu, dafür benötigt die Methode `add` einen längeren Zeitraum.
- ❑ Die Dokumentation findet sich unter:  
<https://docs.oracle.com/en/java/javase/>
- ❑ Die gängigen Datenstrukturen finden sich im Paket `java.util`.

## Beispiel - Lesen von Werten aus einer Datei

- ❑ Paket: `java.io`
- ❑ Klasse `File`
  - verwaltet Metainformationen zu Dateien:  
Name, Pfad, Zugriffsrechte, Größe, Änderungsdatum, ...
  - ermöglicht das Ändern einiger dieser Eigenschaften
  - ein `File`-Objekt wird zum Zugriff auf den Inhalt einer Datei *nicht unbedingt* benötigt
- ❑ `Stream`
  - ist ein Datenstrom, also eine Folge von Zeichen
  - muss nicht unbedingt eine Datei sein
- ❑ Klasse `InputStreamReader`  
verwaltet einen nur lesbaren Stream von Zeichen
- ❑ Klasse `FileReader` **extends** `InputStreamReader`
  - liest die Zeichen aus einer Datei, die beim Anlegen eines `FileReader`-Objekts angegeben werden muss
  - ermöglicht den Zugang zu den Inhalten einer Datei

## Java-Programme und ihre Umgebung

- bisher betrachtet:

Programme, die während der Ausführung Ergebnisse erzeugen,  
die nach der Ausführung verloren sind.

Das ist *nicht* der Normalfall.

- übliche Anwendungen:

Ergebnisse einer Ausführung bleiben erhalten und  
können bei späteren Ausführungen wiederverwendet werden:

Die Daten werden dauerhaft (*persistent*) gespeichert,  
beispielsweise als Dateien.

- aber:

Dateien werden durch das Betriebssystem verwaltet!

- daher:

Bei der Ausführung besitzt das Programm nicht die alleinige Kontrolle.

## Java-Programme und ihre Umgebung

(Fortsetzung)

❑ Folge:

zuvor geprüfte Randbedingungen bleiben *nicht* sicher erhalten:

```
if ( Datei ist vorhanden )
{
    while ( ! Ende von Datei )
    {
        lese und verarbeite Daten
    }
}
```

← kann in jedem Durchlauf scheitern

❑ Gründe für ein Scheitern des Lesens:

- Strom der Festplatte fällt aus.
- Netzwerkverbindung zur Festplatte wird getrennt.
- Lesefehler der Festplatte.

❑ Konsequenz:

Es wird ein Mechanismus benötigt, der die Weiterarbeit des Programms ermöglicht, obwohl ein Problem aufgetreten ist, dessen Ursache außerhalb des Programms liegt.

## Java-Klassen und ihre Benutzung

□ bisher betrachtet:

In DAP 1 wurden bisher Klassen erstellt, die anschließend selbst genutzt wurden. Das ist *nicht* der Normalfall.

□ üblicher Einsatz:

Der Entwickler einer Klasse ist an deren Benutzung nicht beteiligt.

□ daher:

Der Entwickler einer Klasse kann die zukünftigen Einsatzbereiche der von ihm erstellten Klasse nicht vorhersehen.

Er plant ein aus seiner Sicht vernünftiges Verhalten, kann aber nie sicher sein,

- dass Argumente die vorgesehenen Werte enthalten und
- dass von ihm als fehlerhaft betrachtete Situationen in bestimmten Nutzungsszenarien doch möglich sein sollen.

## Java-Klassen und ihre Benutzung

(Fortsetzung)

- Folge:  
Der Entwickler kann nicht abschätzen, ob *ungewöhnliche* Situationen beim Aufruf einer Methode gewünscht oder fehlerhaft sind.
  
- ungewöhnliche Aufrufe für eine Liste könnten zum Beispiel sein:
  - `removeFirst()` für eine leere Liste
  - `add( null )`
  - `get( -1 )`
- Konsequenz:  
Es wird ein Mechanismus benötigt, der
  - einerseits eine solche ungewöhnliche Aufrufsituation signalisiert und
  - andererseits trotz einer solchen Situation auch die Weiterarbeit des Programms ermöglicht.

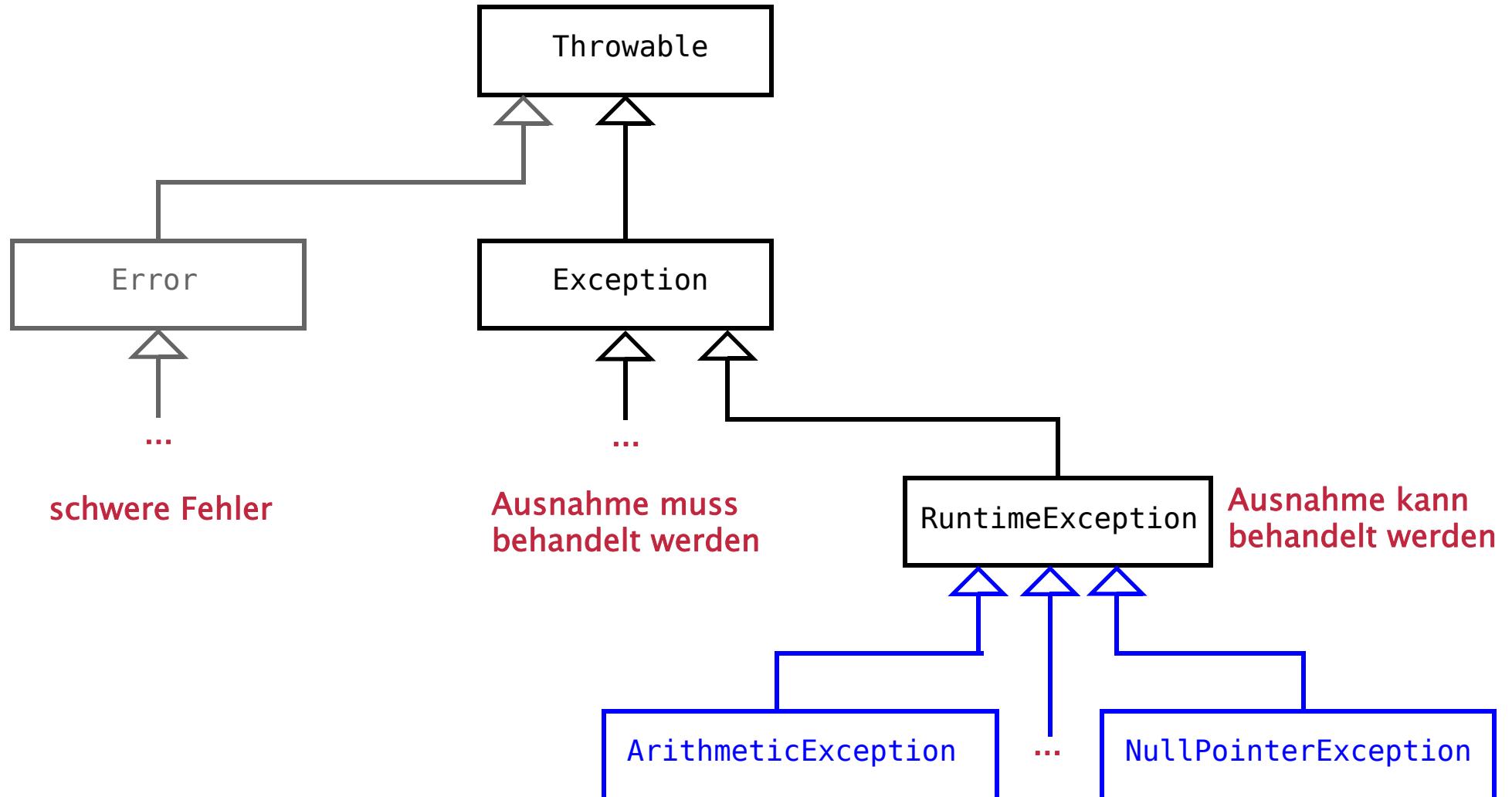
## Ausnahmebehandlung

- ❑ Ausnahmen sind ein Konzept, mit dem Methoden während der Laufzeit ungewöhnliche Situationen (eben: *Ausnahmesituationen*) anzeigen können.
- ❑ Die Behandlung einer angezeigten Ausnahme kann dann in der *aufrufenden Methode* erfolgen.
- ❑ Probleme und damit Ausnahmen treten in verschiedenen Varianten auf:
  - als Ausnahmen, die von der aufrufenden Methode behandelt werden *können*,
  - als Ausnahmen, die von der aufrufenden Methode behandelt werden *müssen*,
  - als schwere Fehler, die zu immer einem Abbruch des Programms führen.

## Ausnahmeklassen

- ❑ Eine *Ausnahmeklasse* ist eine Klasse, die von der Klasse `Throwable` erbt.
  - ❑ Ausnahmeklassen bilden eine Klassenhierarchie,  
für die die üblichen Vererbungs- und Kompatibilitätsregeln gelten.
  - ❑ Ein Objekt einer Ausnahmeklasse kann *alternativ* zu dem im Methodenkopf deklarierten und durch `return` bestimmten Rückgabewert zurückgegeben werden.
  - ❑ Die Möglichkeit der Rückgabe eines Objekts einer Ausnahmeklasse *kann zusätzlich* im Methodenkopf deklariert werden.
  - ❑ Die Rückgabe eines Objekts einer Ausnahmeklasse wird im Methodenrumpf durch eine spezielle Anweisung ausgelöst.
- 
- ❑ Terminologie:
    - Das Zurückgeben einer Ausnahme wird als *Werfen* bezeichnet.
    - Sowohl die Ausnahmeklasse als auch das Objekt einer solchen Klasse werden (ungenau) als *Ausnahme* bezeichnet.

## Hierarchie der Ausnahmeklassen



## Anlegen einer Ausnahmeklasse

```
public class NoElementException
extends Exception
{}

public class WrongIndexException
extends RuntimeException
{}
```

- ❑ Die Klasse `NoElementException` erbt von der Klasse `Exception` und kann daher benutzt werden, um ein Ausnahme-Objekt zu erzeugen.  
Die Klasse `WrongIndexException` erbt von der Klasse `RuntimeException` und kann daher benutzt werden, um ein Ausnahme-Objekt zu erzeugen.
- ❑ In vielen Fällen werden im Rumpf einer Ausnahmeklasse keine Attribute oder Methoden deklariert, da das Vorliegen eines Objekts der Klasse selbst ausreichende Informationen für die Behandlung der Ausnahme bietet.
- ❑ Die Klasse `Exception` stellt einige Konstruktoren bereit, u.a.:

```
public Exception( String message )
```

Jedes Ausnahme-Objekt bietet also die Möglichkeit, zusätzliche Informationen in Form eines Textes abzulegen, indem ein geeigneter eigener Konstruktor erstellt wird.

## Checked Exceptions

```
public class NoElementException
extends Exception
{}
```

- ❑ Ausnahmeklassen, die `Exception` – aber nicht `RuntimeException` – erweitern, führen zu *Checked Exceptions*.
- ❑ Auf Checked Exceptions muss im Programmtext «reagiert» werden.  
Der Compiler erzwingt entsprechende Anweisungen.
- ❑ Kann eine Methode eine Checked Exception werfen, so muss sie dieses in ihrem Kopf durch eine `throws`-Angabe anzeigen.

## Checked Exceptions

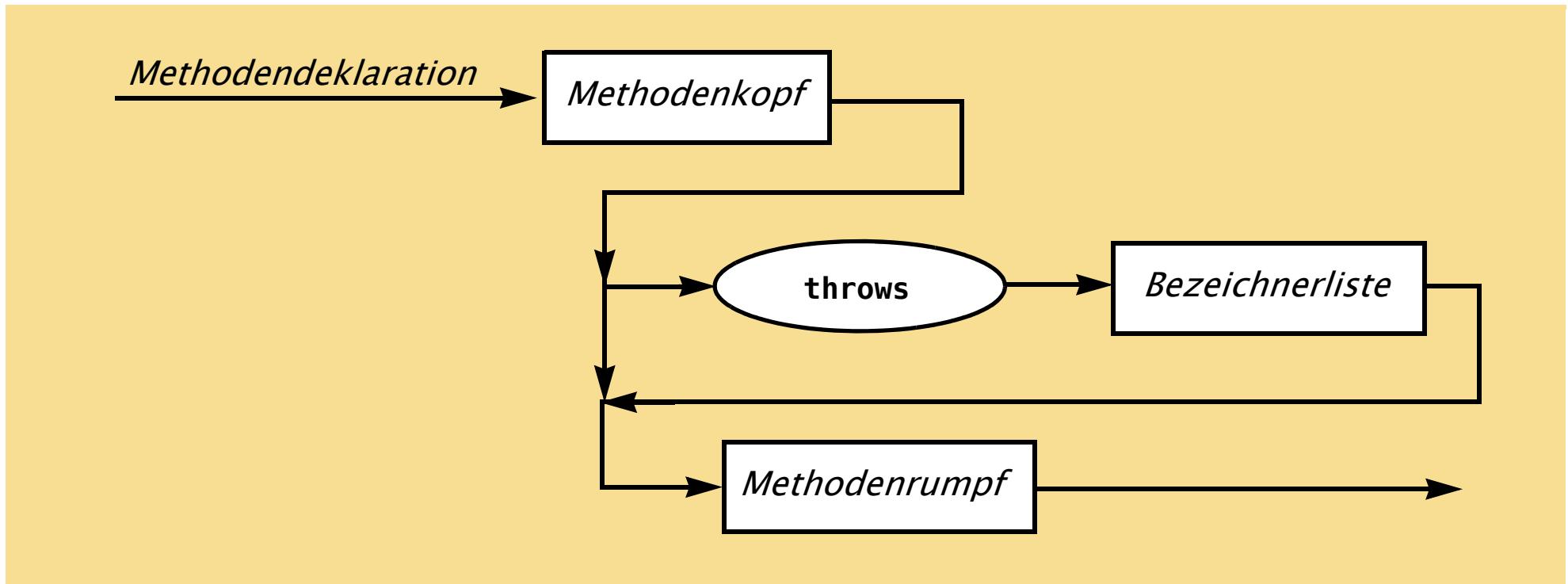
(Fortsetzung)

In der Klasse `DoublyLinkedList<T>`:

```
public T removeFirst()
throws NoSuchElementException
{
    ...
}
```

- ❑ Die Methode `removeFirst` gibt im normalen Ablauf den Inhalt des gelöschten Elements vom Typ `T` zurück, kann aber alternativ auch eine Checked Exception der Klasse `NoElementException` werfen.
- ❑ Die Typen der möglicherweise geworfenen Checked Exceptions werden als Liste von Klassennamen hinter der Parameterliste hinter dem Schlüsselwort `throws` und vor dem Rumpf der Methode aufgelistet.  
*Erinnerung:* Die angegebenen Klassen müssen alle von `Throwable` erben.
- ❑ Aufgabe der Ausnahme für die Methode `removeFirst`:  
Zeigt an, dass kein erstes Element vorhanden war und daher weder ein Element gelöscht werden noch sein Inhalt zurückgegeben werden konnte.

## Syntaxdiagramm zu *Methode*



- ❑ Die Bezeichnerliste hinter dem Schlüsselwort **throws** darf ausschließlich Namen von Klassen enthalten, die (direkte oder indirekte) Unterklassen von **Throwable** sind.

## Werfen einer Ausnahme

Implementierung der Methode `removeFirst`:

```
public T removeFirst() throws NoElementException
{
    if ( !isEmpty() )
    {
        T result = first.getContent();
        if ( first.hasSucc() )
        {
            first = first.disconnectSucc();
        } else {
            first = last = null;
        }
        size--;
        return result;
    } else {
        throw new NoElementException();
    }
}
```

`throws` zeigt an, welche Typen ein geworfenes Objekt besitzen kann

`throw` wirft ein Objekt, das kompatibel mit den Angaben im Kopf der Methode sein muss

- ❑ Die Anweisung `throw` unterbricht die Ausführung der Methode.
- ❑ Der Ablauf kehrt zur aufrufenden Methode zurück und liefert dieser das geworfene Ausnahme-Objekt der Klasse `NoElementException`.

## Einsatz von Ausnahmen

- ❑ Ausnahmeklassen sind Klassen, die von der Klasse `Throwable` erben.
- ❑ Methoden zeigen eventuell durch die `throws`-Klausel im Kopf der Deklaration an, Objekte welcher Ausnahmen sie werfen können.
- ❑ Eine Methode liefert also immer nur *entweder* einen Wert/ein Objekt des in der Deklaration angegebenen Rückgabetyps *oder* ein Objekt von einer der in der Deklaration angegebenen Ausnahmeklassen.
- ❑ Methoden, die Ausnahmen werfen, können daher sehr unterschiedliche Ergebnisse – reguläre Rückgabe oder Ausnahme – an die aufrufende Methode liefern.
- ❑ Idee des Konzepts:
  - Kann eine Methode so ausgeführt werden, dass sie die Aufgabe erledigt, die bei ihrer Entwicklung vorgesehen war, so liefert sie ihren Rückgabewert oder auch gar kein Ergebnis.
  - Tritt eine Situation auf, für die der Entwickler der Methode ein sinnvolles Weiterarbeiten nicht für möglich hält oder in der ein aus seiner Sicht vermutlich nicht brauchbares Ergebnis entsteht, wirft die Methode ein Ausnahme-Objekt.
  - Die Situationen, in denen eine Methode Ausnahme-Objekte wirft, muss der Entwickler der Methode also bereits bei deren Implementierung erkennen und einarbeiten.
  - Handelt es sich um eine Checked Exception, muss auch die aufrufende Methode die Rückgabe eines Ausnahme-Objekts vorsehen.

## Werfen einer Ausnahme

auch möglich, aber unüblich und eher verwirrend:

```
public T removeFirst() throws Exception
{
    if ( !isEmpty() )
    {
        T result = first.getContent();
        if ( first.hasSucc() )
        {
            first = first.disconnectSucc();
        } else {
            first = last = null;
        }
        size--;
        return result;
    } else {
        throw new NoElementException();
    }
}
```

*signalisiert:  
irgendeine Ausnahme wird geworfen*

*throw wirft ein Objekt einer  
Unterklasse von Exception*

- ❑ Auch für Ausnahmen gilt: Objekte der Unterklasse sind kompatibel zur Oberklasse.
- ❑ Daher können Objekte von Unterklassen der hinter **throws** angegebenen Klassen geworfen werden.

## Unchecked Exceptions

```
public class WrongIndexException  
extends RuntimeException  
{}
```

- Ausnahmeklassen, die `RuntimeException` erweitern, führen zu *Unchecked Exceptions*.
- Unchecked Exceptions müssen im Programmtext nicht beachtet werden.  
Allerdings können sie bei Bedarf genau so wie Checked Exceptions behandelt werden.
- Kann eine Methode eine Unchecked Exception werfen, so muss sie dieses nicht anzeigen.

## Unchecked Exceptions

(Fortsetzung)

Weitere Beispiele für die Klasse DoublyLinkedList<T>:

```
public T get( int index ) {  
    ...  
}
```

← keine throws -Angabe

- Methode get:
  - Für einen unzulässigen Index kann get kein Ergebnis liefern.
  - Da **null** ein gültiger Inhalt eines Elements der Liste sein kann, kann der Misserfolg des Aufrufs nicht über die Rückgabe von **null** angezeigt werden.

Daher soll in diesem Fall eine Unchecked Exception der Klasse WrongIndexException geworfen werden.

## Unchecked Exceptions

(Fortsetzung)

Implementierung der Methode `get`:

```
public T get( int index )
{
    if ( index >= 0 && index < size )
    {
        Element<T> current = first;
        for ( int i = 0; i < index; i++ )
        {
            current = current.getsucc();
        }
        return current.getContent();
    } else {
        throw new WrongIndexException();
    }
}
```

return- oder throw-Anweisung  
beenden die Methode

- Die Ausnahme kann geworfen werden, obwohl keine `throws`-Angabe vorliegt.

## Fangen von Ausnahmen

Konzept:

- ❑ Die Situationen, in denen Ausnahmen eingesetzt werden, sollen – im umgangssprachlichen Sinne – Ausnahmen und damit *selten* bleiben.
- ❑ Das Auftreten von solchen Situationen hat in der Regel zur Folge, dass größere Abschnitte des Programmtextes nicht sinnvoll weitergeführt werden können:
  - Wenn das Lesen einer Datei unerwartet nicht mehr möglich ist, macht der Versuch einer weiteren Verarbeitung der *nicht* gelesenen Daten keinen Sinn.
  - Wenn aus einer Liste kein Wert abgefragt werden kann, muss die weitere Bearbeitung ohne diesen Wert erfolgen
- ❑ Das Berücksichtigen von Ausnahmen, die von aufgerufenen Methoden geworfen werden, muss daher syntaktisch so erfolgen, dass
  - der Programmtext für die wesentlichen Abläufe lesbar und verständlich bleibt und
  - die vom Auftreten einer Ausnahme betroffenen Abläufe zusammenhängend und gemeinsam behandelt werden können.
- ❑ In Java erfolgt die Behandlung von Ausnahmen in der aufrufenden Methode innerhalb einer **try-catch**-Struktur.
- ❑ Terminologie:  
Das Verarbeiten eines Ausnahme-Objekts wird als *Fangen* bezeichnet.

## Fangen von Ausnahmen

(Fortsetzung)

Beispiel:

```
try
{
    double result = compute( list.get( i ) );
    if ( result > 0 )
    {
        ...
    }
}
catch( WrongIndexException e )
{
    System.out.println( "wrong index i: " + i );
}
```

hier kann eine Ausnahme auftreten

- ❑ Innerhalb des **try**-Blocks wird *versucht*, alle Anweisungen auszuführen.
- ❑ Wirft ein Methodenaufruf innerhalb eines **try**-Blocks ein Ausnahme-Objekt, wird die Ausführung des **try**-Blocks abgebrochen.
- ❑ Die Ausführung wird *hinter* dem **try**-Block fortgesetzt.
- ❑ Der **try**-Block definiert also einen Bereich, dessen Ausführung beim Auftreten einer Ausnahme nicht weiter sinnvoll ausgeführt werden kann.

## Fangen von Ausnahmen

(Fortsetzung)

Beispiel:

```
try
{
    double result = compute( list.get( i ) );
    if ( result > 0 )
    {
        ...
    }
    catch( WrongIndexException e )
    {
        System.out.println( "wrong index i: " + i );
    }
}
```

hier kann eine Ausnahme auftreten

hier wird festgelegt, welche Ausnahmen an dieser Position gefangen werden sollen

- ❑ An einen **try**-Block können sich (mehrere) **catch**-Blöcke anschließen.
- ❑ Die **catch**-Anweisung bestimmt, welcher Typ von Ausnahmen gefangen werden soll.
- ❑ Ist im **try**-Block eine Ausnahme aufgetreten, so wird der Block hinter der *ersten* **catch**-Anweisung ausgeführt, die das geworfene Ausnahme-Objekt fangen kann.
- ❑ Da sich das Fangen am Typ des geworfenen Objekts orientiert, benötigen Ausnahmeklassen meist keine zusätzlichen Attribute, um die Problemsituation zu beschreiben.

## Fangen von Ausnahmen

(Fortsetzung)

Beispiel:

```
try
{
    double result = compute( list.get( i ) );
    if ( result > 0 )
    {
        ...
    }
}
catch( WrongIndexException e )
{
    System.out.println( "wrong index i: " + i );
}
```

- ❑ Wird in einem **try**-Block *keine* Ausnahme geworfen, so werden die sich anschließenden **catch**-Blöcke bei der Ausführung *übersprungen*.
- ❑ Der Programmtext für die *wesentlichen* Abläufe befindet sich daher ausschließlich *innerhalb* des **try**-Blocks.

## Fangen von Ausnahmen

(Fortsetzung)

Beispiel:

```
try
{
    double result = compute( list.get( i ) );
    if ( result > 0 )
    {
        ...
    }
}
catch( WrongIndexException e )
{}
```



- ❑ Auch möglich sind *leere catch*-Blöcke:  
Beim Auftreten einer Ausnahme wird der **try**-Block beendet,  
die Ausnahme wird durch eine passende **catch**-Anweisung gefangen  
und ist damit bearbeitet, da der folgende Block leer ist.
- ❑ Es geht dann nur darum, den Programmtext zwischen dem Auftreten der Ausnahme und dem  
Ende des **try**-Blocks zu überspringen.

## Fangen von Ausnahmen

(Fortsetzung)

Beispiel:

```
try
{
    list.removeFirst();
    double result = compute( list.get( i ) );
    list.addFirst( result + eps );
}
catch( WrongIndexException e ) { ... }
catch( NoElementException e ) { ... }
```

- Die beiden Methoden `removeFirst` und `get` können jede eine Ausnahme einer anderen Klasse werfen.
- Eine der beiden `catch`-Anweisungen fängt eines der möglicherweise geworfenen Objekte einer dieser beiden Klassen und behandelt dann im folgenden Block die Ausnahmesituation.
- `catch`-Blöcke, deren `catch`-Anweisung *kein* Ausnahme-Objekt fängt, werden übersprungen.
- Sobald eine `catch`-Anweisung das Ausnahme-Objekt gefangen hat, wird der zugehörige Block ausgeführt. Damit ist diese Ausnahme *bearbeitet*.  
Nachfolgende `catch`-Anweisungen und -Blöcke werden daher übersprungen, da ja immer nur genau ein Ausnahme-Objekt vorliegt und dieses bearbeitet ist.

## Fangen von Ausnahmen

(Fortsetzung)

Beispiel:

```
try
{
    list.removeFirst();
    double result = compute( list.get( i ) );
    list.addFirst( result + eps );
}
catch( WrongIndexException | NoElementException e )
{ ... }
```



- ❑ Ausnahmen der Klassen `WrongIndexException` und `NoElementException` werden mit der gleichen `catch`-Anweisung gefangen.
- ❑ Das Symbol `|` kann als (umgangssprachliches) *ODER* gelesen werden.
- ❑ Die Zahl der `catch`-Anweisungen und -Blöcke kann mit dieser Schreibweise verkürzt werden.

## Fangen von Ausnahmen

(Fortsetzung)

Beispiel:

```
try
{
    list.removeFirst();
    double result = compute( list.get( i ) );
    list.addFirst( result + eps );
}
catch( WrongIndexException e ) { ... }
catch( Exception e ) { ... }
```

← fängt alle noch nicht  
gefangenen Ausnahmen

- Da `Exception` die Oberklasse aller Ausnahmeklassen ist, fängt die zweite `catch`-Anweisung alle möglicherweise geworfenen Ausnahme-Objekte, die nicht vom Typ `WrongIndexException` sind.  
(Diese werden durch die vorangehende `catch`-Anweisung bereits gefangen.)
- Die Oberklasse steht stellvertretend für sich und ihre Unterklassen.  
Die Zahl der benötigten `catch`-Blöcke kann so reduziert werden.

## Fangen von Ausnahmen

(Fortsetzung)

Beispiel:

```
try
{
    list.removeFirst();
    double result = compute( list.get( i ) );
    list.addFirst( result + eps );
}
catch( Exception e ) { ... }
catch( WrongIndexException e ) { ... }
```



- Eine Folge von **catch**-Anweisungen, in der zunächst ein Fangen der Objekte der Oberklasse und dann das Fangen von Objekten von Unterklassen erfolgt, macht *niemals* einen Sinn, da die nachfolgenden **catch**-Anweisungen nie erreicht werden können.

## Zusammenfassung: Grundlagen des Fangens von Ausnahmen

- ❑ Der Bereich, in dem Ausnahmen gefangen werden sollen, wird durch einen **try**-Block festgelegt.
- ❑ Das Fangen einer Ausnahme erfolgt durch eine **catch**-Anweisung.
- ❑ Eine gefangene Ausnahme wird in dem auf die fangende **catch**-Anweisung folgenden Block bearbeitet.
- ❑ Das Fangen von Ausnahmen kann zusammengefasst werden durch
  - die Kombination von zu selektierenden Klassen durch den Einsatz des **|**-Operators oder
  - die Wahl einer Oberklasse als zu selektierende Klasse in einer **catch**-Anweisung.

## Zusammenhang von Werfen und Fangen

Beispiel:

Kann eine Methode eine Checked Exception werfen und liegt also eine **throws**-Angabe vor

```
public T removeFirst() throws NoSuchElementException { ... }
```

so **muss** eine aufrufende Methode dieses Werfen berücksichtigen.

Variante mit *Fangen und Bearbeiten*:

```
public void call( int i )
{
    try
    {
        double result = compute( list.removeFirst() );
        if ( result > 0 )
        {
            ...
        }
    }
    catch( NoSuchElementException e ) { ... }
}
```

## Zusammenhang von Werfen und Fangen

(Fortsetzung)

Beispiel:

Kann eine Methode eine Checked Exception werfen und liegt also eine **throws**-Angabe vor

```
public T removeFirst() throws NoSuchElementException { ... }
```

so **muss** eine aufrufende Methode dieses Werfen berücksichtigen.

Variante mit *Weiterwerfen*

```
public void call( int i ) throws NoSuchElementException
{
    double result = compute( list.removeFirst() );
    if ( result > 0 )
    {
        ...
    }
}
```

- ❑ Da die Methode `call` die möglicherweise von `get` geworfene Ausnahme nicht fängt, aber ihrerseits ein Werfen dieser Ausnahme deklariert, *fliegt* die Ausnahme *weiter* als Rückgabe der Methode `call`. In diesem Fall wird kein `try`-Block benötigt.

## Zusammenhang von Werfen und Fangen

(Fortsetzung)

Beispiel:

auch möglich: Variante mit *Fangen oder Weiterwerfen*

```
public void call( int i ) throws NoElementException
{
    try
    {
        double result = compute( list.removeFirst() );
        check( list.get( i ) );
        list.addFirst( result + eps );
        if ( result > 0 )
        {
            ...
        }
    }
    catch( WrongIndexException e ) { ... }
}
```

The diagram illustrates the flow of an exception. A red double-headed vertical arrow connects the `throws NoElementException` declaration at the top to the `catch( WrongIndexException e ) { ... }` block at the bottom. A red curved arrow originates from the `list.removeFirst()` call in the `try` block and points to the `WrongIndexException` catch block, indicating that the exception is caught after it is thrown.

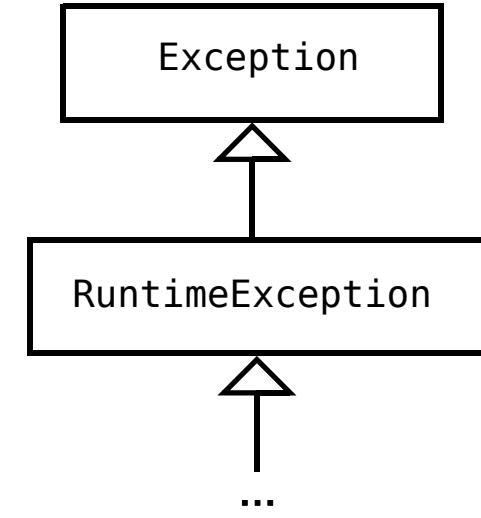
## Zusammenhang von Werfen und Fangen

(Fortsetzung)

- ❑ Eine aufrufende Methode **muss** Checked Exceptions fangen oder weiterwerfen.
- ❑ Ist im Kopf einer aufgerufenen Methode eine Checked Exception einer Oberklasse angegeben (siehe Folie 1070), so muss die aufrufende Methode sich um Ausnahmen dieser Oberklasse kümmern
  - auch dann, wenn tatsächlich nur das Objekte von Unterklassen geworfen werden.
- ❑ Der Compiler überprüft, dass deklarierte Ausnahmen in den aufrufenden Methoden behandelt werden. Erfolgt das notwendige Fangen oder Weiterwerfen nicht, meldet der Compiler einen Fehler.

## Klasse RuntimeException \*)

- ❑ Unchecked Exceptions, die die Klasse RuntimeException spezialisieren, können auch dann geworfen werden, wenn sie *nicht* im Kopf der Methode deklariert sind.
- ❑ Ausnahmen, die die Klasse RuntimeException spezialisieren, müssen daher *nicht* von aufrufenden Methoden behandelt werden.
- ❑ Aufrufende Methoden können entsprechend diese unerwartet aufgetretenen Ausnahmen ihrerseits *ohne* Deklaration weiterwerfen.
- ❑ RuntimeException-Objekte dienen in der Regel dazu, Programmierfehler anzuzeigen.
- ❑ Einige der bereits bekannten Ausnahmen zählen dazu:
  - `ArrayIndexOutOfBoundsException`
  - `ArithmetricException`
  - `NullPointerException`

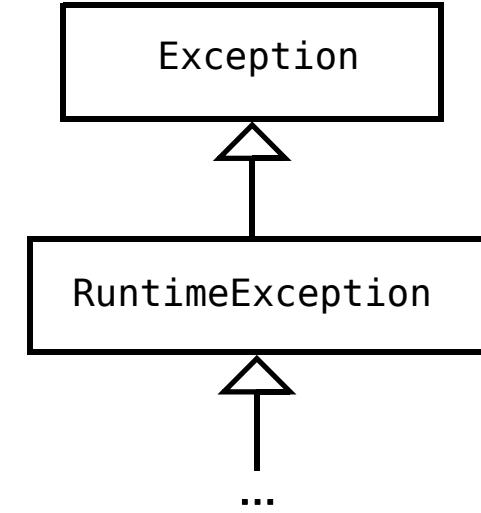


\*) Der Name `RuntimeException` ist etwas verwirrend, da alle Ausnahmen während der Laufzeit auftreten.

## Klasse RuntimeException

(Fortsetzung)

- ❑ Die Möglichkeit, Unchecked Exceptions ohne Angabe im Kopf der Deklaration zu werfen, wurde in verschiedenen bisher vorgestellten Methoden genau dazu ausgenutzt, um eine nicht zu behandelnde Ausnahmesituation anzuzeigen:  
`throw new IllegalStateException();`
- ❑ Da das Konzept des Fangens nicht bekannt war, erfolgte nie eine weitere Bearbeitung.
- ❑ Wird ein RuntimeException-Objekt nie gefangen, so löst es beim Weiterleiten durch die `main`-Methode einen Programmabbruch aus.



Es wird dann die *StackTrace* ausgegeben.

## Wiederholung - Lesen von Werten aus einer Datei

(siehe Folie 1060)

- ❑ Paket: `java.io`
- ❑ Klasse `File`
  - verwaltet Metainformationen zu Dateien:  
Name, Pfad, Zugriffsrechte, Größe, Änderungsdatum, ...
  - ermöglicht das Ändern einiger dieser Eigenschaften
  - ein `File`-Objekt wird zum Zugriff auf den Inhalt einer Datei nicht unbedingt benötigt
- ❑ *Stream*
  - ist ein Datenstrom, also eine Folge von Zeichen
  - muss nicht unbedingt eine Datei sein
- ❑ Klasse `InputStreamReader`  
verwaltet einen nur lesbaren Stream von Zeichen
- ❑ Klasse `FileReader` **extends** `InputStreamReader`
  - liest die Zeichen aus einer Datei, die beim Anlegen eines `FileReader`-Objekts angegeben werden muss
  - ermöglicht den Zugang zu den Inhalten einer Datei

## Beispiel - Lesen von Werten aus einer Datei

(Fortsetzung)

wichtige Methoden der Klasse `FileReader`

□ Konstruktoren:

```
public FileReader( String fileName ) throws FileNotFoundException *)
```

```
public FileReader( File file ) throws FileNotFoundException
```

*öffnen* die als Argument übergebene Datei

□ Leseoperation:

```
public int read() throws IOException
```

liest ein Zeichen aus der Datei und gibt es (als `int`-Wert) zurück  
oder gibt `-1` zurück, wenn das Ende der Datei erreicht ist

□ Schließen der Datei:

```
public void close() throws IOException
```

gibt die zugehörigen Ressourcen im umgebenden Betriebssystem frei  
*(= schließt den Stream)*

\*) `FileNotFoundException` ist eine Unterklasse von `IOException`

## Beispiel - Lesen von Werten aus einer Datei

(Fortsetzung)

Beispiel *nur* mit **catch**-Anweisung:

```
public static void showFile( String filename )
{
    FileReader f = null;           ← Anlegen der Stream-Referenz
    try
    {
        f = new FileReader( filename ); ← Erzeugen des Stream-Objekts
        int c = f.read();             ← Lesen des ersten Zeichens
        while ( c != -1 ) {
            System.out.print( (char)c ); ← Ausgabe: Type-cast notwendig
            c = f.read();             ← zeichenweises Lesen
        }
        f.close();                  ← Schließen des Streams
    }
    catch ( IOException e )
    {
        System.out.println( "Datei nicht lesbar" );
        if ( f != null ) {
            try { f.close(); } catch ( IOException x ) {}
        }
    }
}
```

## Beispiel - Lesen von Werten aus einer Datei

(Fortsetzung)

Beispiel **nur** mit **catch**-Anweisung:

```
public static void showFile( String filename )
{
    FileReader f = null;
    try
    {
        f = new FileReader( filename );
        int c = f.read();
        while ( c != -1 ) {
            System.out.print( (char)c );
            c = f.read();
        }
        f.close();
    }
    catch ( IOException e )           ← Abfangen der Ausnahme
    {
        System.out.println( "Datei nicht lesbar" );
        if ( f != null ) {
            try { f.close(); } catch ( IOException x ) {}
        }
    }
}
```

Schließen des Streams  
close kann Exception  
werfen

## Beispiel - Lesen von Werten aus einer Datei

(Fortsetzung)

Beispiel *nur* mit **catch**-Anweisung:

```
public static void showFile( String filename )
{
    FileReader f = null;
    try
    {
        f = new FileReader( filename );
        int c = f.read();
        while ( c != -1 ) {
            System.out.print( (char)c );
            c = f.read();
        }
        f.close();
    }
    catch ( IOException e )
    {
        System.out.println( "Datei nicht lesbar" );
        if ( f != null ) {
            try { f.close(); } catch ( IOException x ) {}
        }
    }
}
```

close wird immer benötigt

## Beispiel - Lesen von Werten aus einer Datei

(Fortsetzung)

Beispiel mit **finally**-Block:

```
public static void showFile( String filename ) {
    FileReader f = null;
    try
    {
        f = new FileReader( filename );
        int c = f.read();
        while ( c != -1 ) {
            System.out.print( (char)c );
            c = f.read();
        }
    }
    catch ( IOException e )
    {
        System.out.println( "Datei nicht lesbar" );
    }
    finally
    {
        if ( f != null ) {
            try { f.close(); } catch (IOException x) {} }
    }
}
```

**finally**-Block wird  
immer ausgeführt

## Beispiel - Lesen von Werten aus einer Datei

(Fortsetzung)

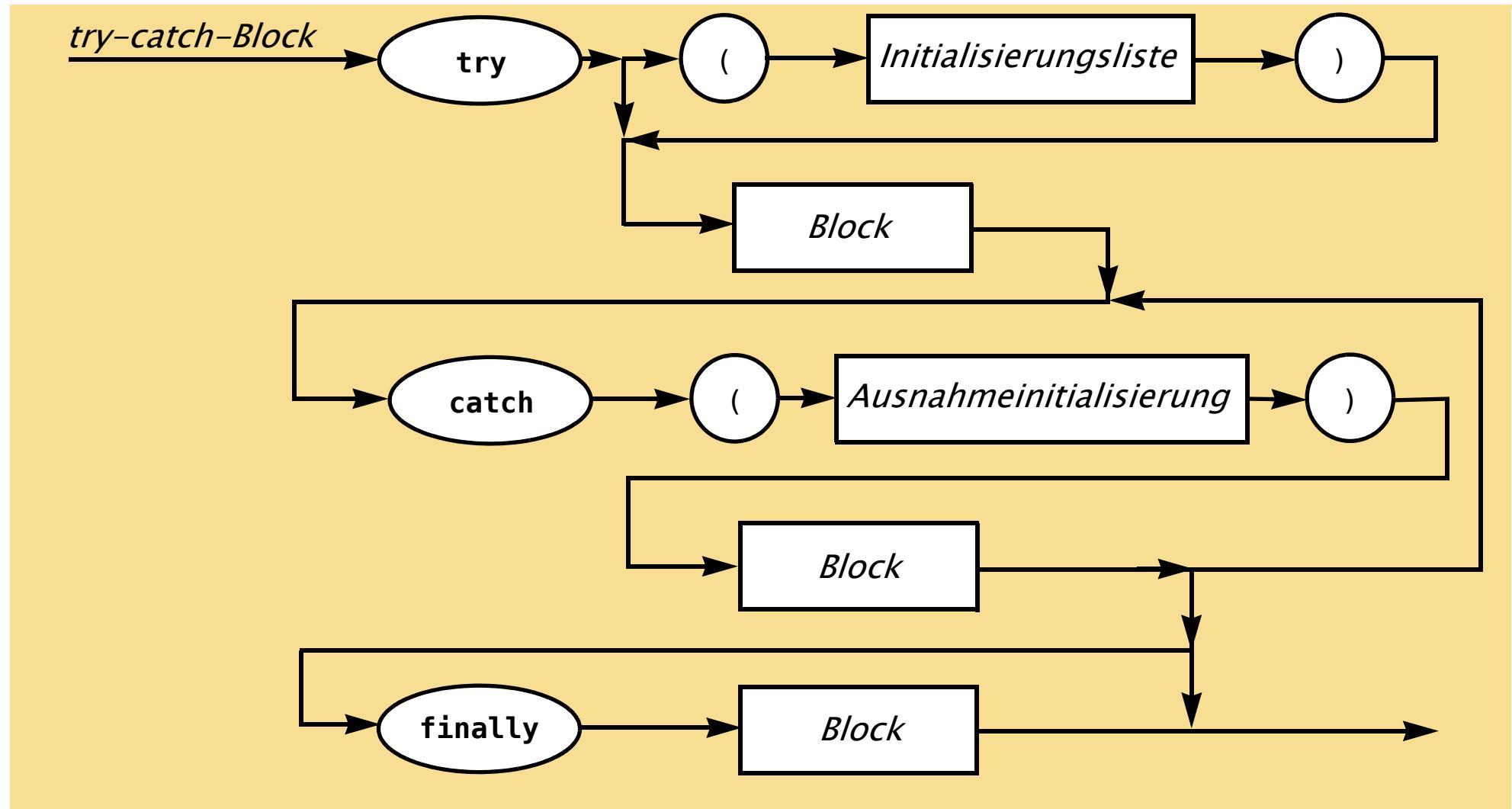
Beispiel mit *Ressourcen-Liste*:

```
public static void showFile( String filename )
{
    try ( FileReader f = new FileReader( filename ) )
    {
        int c;
        while ( ( c = f.read() ) != -1 )
        {
            System.out.print( (char)c );
        }
    }
    catch ( IOException e )
    {
        System.out.println( "Datei nicht lesbar" );
    }
}
```

so geöffnete Streams  
werden immer geschlossen

- ❑ In der *Ressourcen*-Liste hinter `try` können nur solche Objekte deklariert werden, die nach der Abarbeitung des `try`-Blocks wieder *geschlossen* werden müssen.
- ❑ Die Klassen dieser Objekte müssen das Interface `java.io.Closeable` implementieren.

## Syntaxdiagramm zu **try-catch-Block**



## gefangene Ausnahme-Objekte

**catch**-Anweisung:

```
catch ( IOException e ) { ... }
```

- Das gefangene Ausnahme-Objekt kann unter dem Namen `e` in dem folgenden Block angesprochen werden.

Zum Beispiel kann eine Ausgabe der Inhalte des Objekts erfolgen:

```
catch ( IOException e ) { System.out.println( e.toString() ); }
```

- Sind für das Ausnahme-Objekt weitere Methoden deklariert, so können diese eventuell benutzt werden, um eine präzise Beschreibung des Fehlers zu ermöglichen.
- Jede Ausnahmeklasse besitzt eine von `Throwable` geerbte Methode, die die noch geöffneten Instanzen von Methoden zum Zeitpunkt des Werfens der Ausnahme ausgibt:

```
public void printStackTrace()
```

Also ist möglich:

```
catch ( IOException e ) { e.printStackTrace(); }
```

## Mustererkennung mit dem Automaten $M_p$ für eine Datei

(siehe Folie 1052)

- Ziel: Überprüfung des Inhalts einer Datei
- Suche nach dem Muster beginnt *mit dem ersten Zeichen aus der Datei*.
- Der Folgezustand wird aus der mit `generateTransitionTable` angelegten Tabelle ermittelt; die folgenden Zeichen werden *einzelnen aus der Datei gelesen*.
  
- Zwei (weitgehend bekannte) Varianten werden vorgestellt:
  - Feststellen, an welcher Stelle der Datei das Muster  $p$  erstmals in der Datei vorkommt:  
`int containsAt( String pattern, String file )`
  - Zählen, wie häufig das Muster  $p$  in der Datei vorkommt:  
`int count( String pattern, String file )`

## Mustererkennung mit dem Automaten $M_p$ für eine Datei

(Fortsetzung)

```

public static int containsAt( String p, String file )
throws IOException
{
    Map<Integer,Map<Character,Integer>> transitionTable = generateTransitionTable(p);
    int state = 0;
    final int FINALSTATE = p.length();
    int position = 0;
    FileReader f = new FileReader( file );
    int c = f.read();
    while ( c != -1 ) {
        position++;
        if ( transitionTable.containsKey( state ) &&
            transitionTable.get( state ).containsKey( (char)c ) ) {
            state = transitionTable.get( state ).get( (char)c );
            if ( state == FINALSTATE ) {
                return position - p.length() + 1;
            }
        } else {
            state = 0;
        }
        c = f.read();
    }
    return -1; ←
}

```

analog zu `read()`

## Mustererkennung mit dem Automaten $M_p$ für eine Datei

(Fortsetzung)

```
public static int count( String p, String file )
throws IOException
{
    Map<Integer,Map<Character,Integer>> transitionTable = generateTransitionTable(p);
    int quantity = 0;
    int state = 0;
    final int FINALSTATE = p.length();
    FileReader f = new FileReader( file );
    int c = f.read();
    while ( c != -1 ) {
        if ( transitionTable.containsKey( state )
            && transitionTable.get( state ).containsKey( (char)c ) ) {
            state = transitionTable.get( state ).get( (char)c );
            if ( state == FINALSTATE ) {
                quantity++;
            }
        } else {
            state = 0;
        }
        c = f.read();
    }
    return quantity;
}
```

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 17. Mustererkennung - Benutzungsoberfläche

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-1106

## Lernziele des Kapitels 17. Mustererkennung – Benutzeroberfläche

Nach Durcharbeiten des Kapitels Mustererkennung – Benutzeroberfläche sollen die teilnehmenden Studierenden

- die Idee der Gestaltung von grafischen Benutzeroberflächen kennen.

## Rückblick Mustererkennung

(siehe Folie 1105)

Für den Aufruf der Methode

```
public static int count( String pattern, String file )
throws IOException
```

werden benötigt:

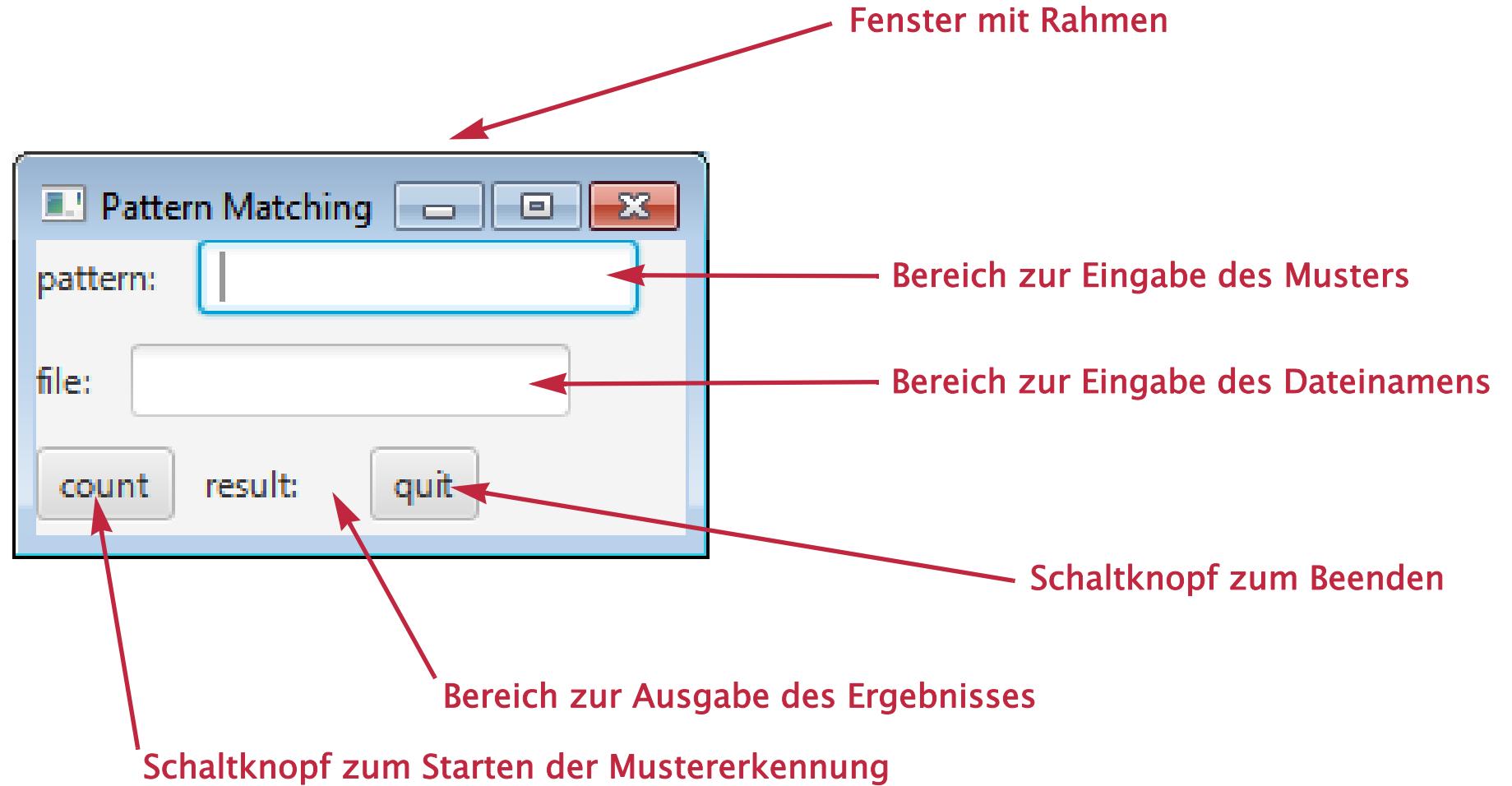
- ein Text: das zu suchende Muster (als Argument für den Parameter `pattern`)
- ein Text: der Name der zu durchsuchenden Datei (als Argument für den Parameter `file`)

### Aufgabe:

Entwicklung einer grafischen Benutzeroberfläche, um die benötigten Texte anzugeben und das Ergebnis der Zählung durch die Methode `count` auszugeben.

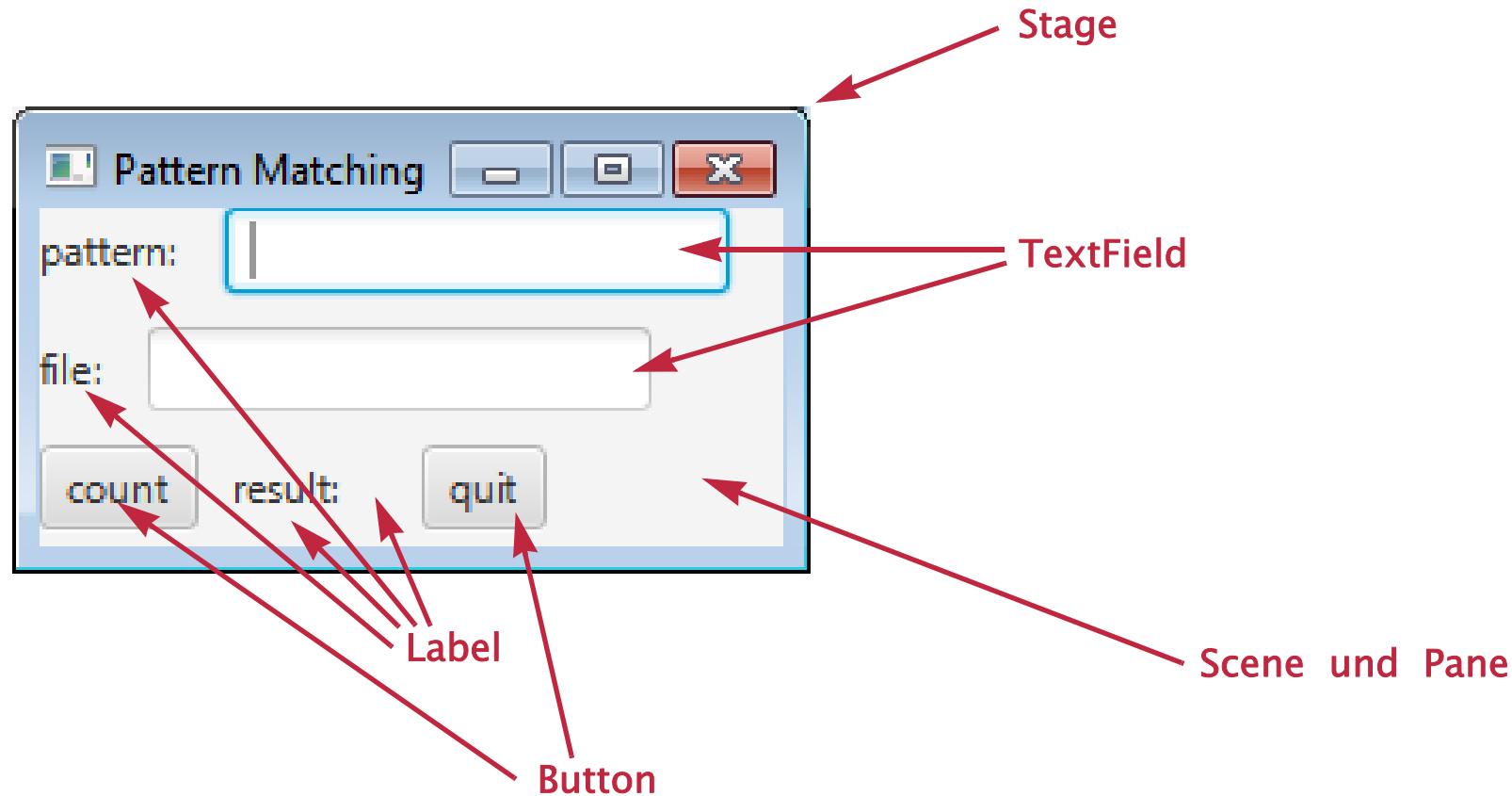
## Bestandteile der grafischen Benutzeroberfläche

### Konzeption



## Bestandteile der grafischen Benutzeroberfläche

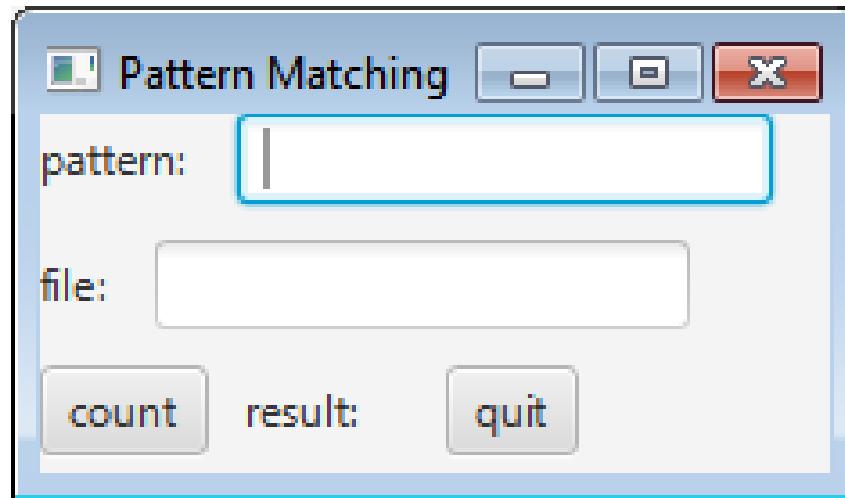
technische Bestandteile aus der Bibliothek JavaFX



## Bestandteile der grafischen Benutzeroberfläche

(Fortsetzung)

technische Bestandteile in JavaFX



Stage – das ganze Fenster

Scene – Fensterinhalt

Pane – (geordnete) Gruppe von Elementen

Label – Element zur Textdarstellung

TextField – Element zur Texteingabe

Button – Element, das Aktion bewirkt

ausführliche Beschreibung unter:

<http://docs.oracle.com/javase/8/javafx/api/>

## Bestandteile der grafischen Benutzungsoberfläche

(Fortsetzung)

```
import javafx.application.Application;  
import javafx.stage.Stage;  
  
...  
  
public class PatternCounting extends Application  
{  
  
    public void start( Stage primaryStage )  
    {  
        ...  
    }  
  
    public static void main( String[] args )  
    {  
        launch( args );  
    }  
}
```

Application stellt einen Ausführungsrahmen bereit



start wird implizit mit einem geeigneten Argument aufgerufen und erzeugt das Fenster

## Bestandteile der grafischen Benutzeroberfläche

(Fortsetzung)

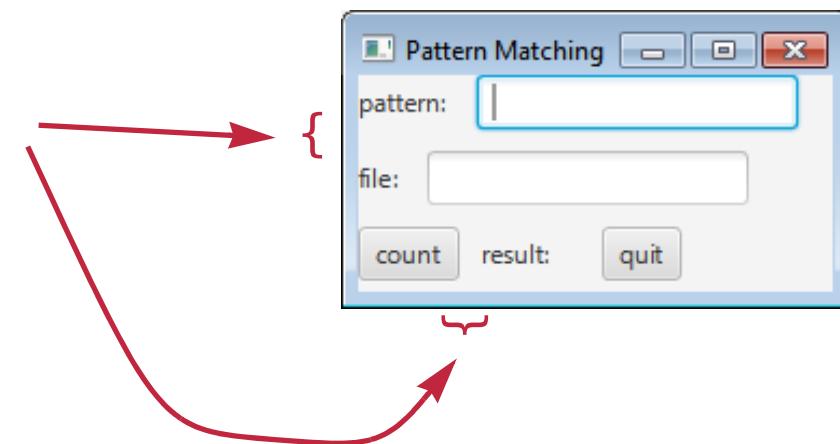
```
...
import javafx.scene.layout.FlowPane;

public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {

        FlowPane pane = new FlowPane();
        pane.setHgap( 10 );
        pane.setVgap( 10 );
        ...
    }
}
```

FlowPane ordnet Elemente  
hintereinander an

definieren den Abstand zwischen  
den angezeigten Elementen



## Bestandteile der grafischen Benutzungsoberfläche

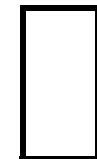
(Fortsetzung)

Layout-Klassen aus dem Paket `javafx.scene.layout`:

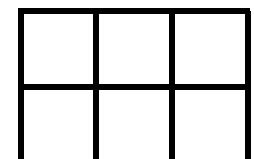
HBox die Elemente werden nebeneinander  
(horizontal) angeordnet



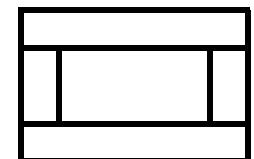
VBox die Elemente werden übereinander  
(vertikal) angeordnet



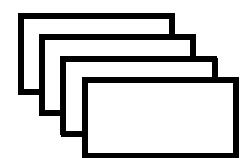
FlowPane die Elemente werden fließend angeordnet



BorderPane die Elemente werden entlang der Ränder angeordnet



StackPane die Elemente werden übereinander gestapelt



## Bestandteile der grafischen Benutzeroberfläche

(Fortsetzung)

```

...
import javafx.scene.control.*;

public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {
        FlowPane pane = new FlowPane();
        pane.setHgap( 10 );
        pane.setVgap( 10 );
        pane.getChildren().add( new Label( "pattern: " ) );           ← fügt Label hinzu
        TextField pattern = new TextField();                         ← legt Label an
        pane.getChildren().add( pattern );                          ← legt TextField an
        ...
    }
}

```

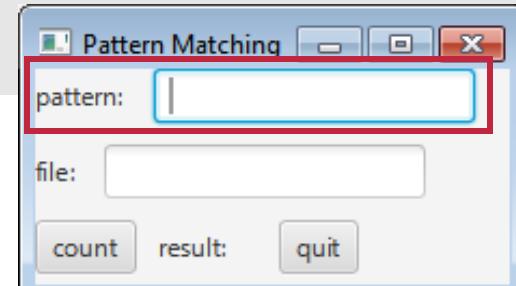
liefert Liste der von pane verwalteten Elemente

fügt TextField hinzu

fügt Label hinzu

legt Label an

legt TextField an



## Bestandteile der grafischen Benutzungsoberfläche

(Fortsetzung)

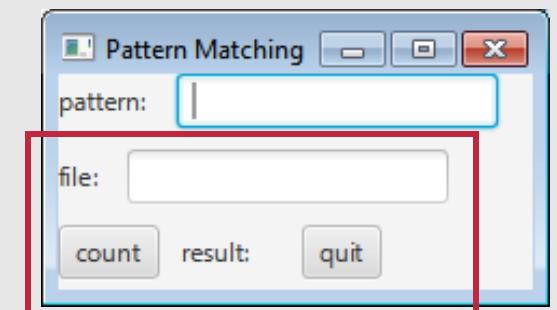
```
...
public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {
        ...
        pane.getChildren().add( new Label( "file: " ) );
        TextField file = new TextField();
        pane.getChildren().add( file );

        Button count = new Button();
        count.setText( "count" );
        pane.getChildren().add( count );

        pane.getChildren().add( new Label( "result: " ) );
        Label result = new Label();
        pane.getChildren().add( result );

        Button quit = new Button();
        quit.setText( "quit" );
        pane.getChildren().add( quit );
        ...
    }
}
```

analog



## Bestandteile der grafischen Benutzungsoberfläche

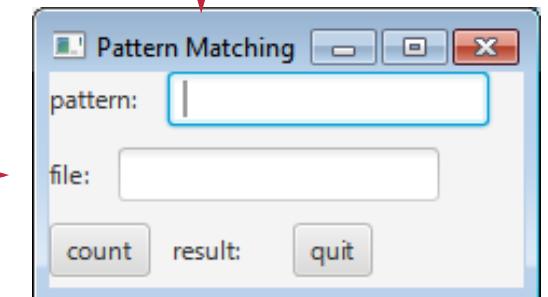
(Fortsetzung)

```
...
public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {
        ...
        primaryStage.setTitle( "Pattern Matching" );
        primaryStage.setScene( new Scene( pane, 220, 100 ) );
        primaryStage.show();
        ...
    }
}
```

Fenster sichtbar machen

Scene-Objekt anlegen und  
FlowPane mit Anfangsgröße zuordnen

Titel des Fensters setzen



## Bestandteile der grafischen Benutzungsoberfläche

(Fortsetzung)

Anmerkungen:

- ❑ Das Layout von graphischen Oberflächen wird in der Regel nicht manuell implementiert.
- ❑ Hierfür existieren spezielle *GUI-Builder*, die ein Gestalten von Oberflächen im WYSIWYG-Stil (*what-you-see-is-what-you-get*) ermöglichen.
- ❑ Beispiele für JavaFx-GUI-Builder:
  - JavaFX Scene Builder (Oracle) 1)
  - JavaFX Composer (NetBeans) 2)
- ❑ Darüber hinaus arbeitet JavaFX mit Cascading Style Sheets (CSS) zusammen. 3)

1) <http://www.oracle.com/technetwork/java/javafx/tools/index.html>

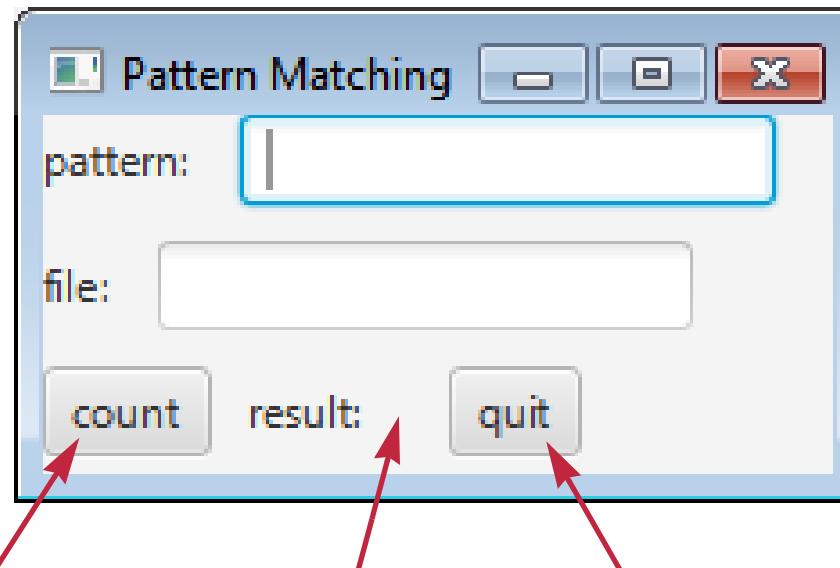
2) <https://netbeans.org/features/javafx/composer.html>

3) <https://openjfx.io/>

## Bestandteile der grafischen Benutzeroberfläche

(Fortsetzung)

Zuordnen von Verhalten zu den Button-Objekten



Aufruf der Methode  
`PatternMatcher.count`

Ergebnis anzeigen

Beenden der Ausführung

## Bestandteile der grafischen Benutzungsoberfläche

(Fortsetzung)

Zuordnen von Verhalten zu den Button-Objekten

- Zuordnen einer `onAction`:

Das ist die Aktion, die ausgeführt wird, wenn ein Button ausgewählt wird.

- Deklaration:

```
ObjectProperty<EventHandler<ActionEvent>> onAction
```

- Methode in der Klasse Button, die `onAction` setzt:

```
public void setOnAction( EventHandler<ActionEvent> value )
```

- Interface `EventHandler`:

```
public interface EventHandler<T extends Event>
extends EventListener
{
    void handle( T event );
}
```

## Bestandteile der grafischen Benutzungsoberfläche

(Fortsetzung)

Zuordnen von Verhalten zu den Button-Objekten

```
...
public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {
        ...
        quit.setOnAction( ... );
        count.setOnAction( ... );
    }
}
```

- ❑ Es wird noch die Zuordnung von passenden EventHandler-Implementierungen durch Aufrufe von `setOnAction` benötigt, damit anschließend das Programm durch Aufruf der `main`-Methode ausgeführt werden kann.

## Bestandteile der grafischen Benutzeroberfläche

(Fortsetzung)

Zuordnen von Verhalten zu den Button-Objekten

```
...
public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {
        ...
        quit.setOnAction( event -> System.exit( 0 ) );      Lambda-Ausdruck
        count.setOnAction( ... );
    }
}
```

- ❑ Der Aufruf von `System.exit( 0 )` beendet die Ausführung der virtuellen Maschine JVM und damit auch des Programms.
- ❑ Das Argument wird an das Betriebssystem übergeben.  
Der Wert `0` zeigt ein normales Beenden an.

## Bestandteile der grafischen Benutzungsoberfläche

(Fortsetzung)

Zuordnen von Verhalten zu den Button-Objekten

```
...
public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {
        ...
        quit.setOnAction( event -> System.exit( 0 ) );
        count.setOnAction( event ->
        {
            try {
                result.setText( "" +
                    PatternMatcher.count( pattern.getText(), file.getText() ) );
            } catch ( IOException e ) { result.setText( "file error" ); }
        });
    }
}
```

**Lambda-Ausdruck**

- ❑ Ausnahmebehandlung ist notwendig, da `count` bei Zugriffsproblemen zu der angegebenen Datei (`file.getText()`) eine Ausnahme werfen kann.

## Bestandteile der grafischen Benutzeroberfläche

(Fortsetzung)

Zuordnen von Verhalten zu den Button-Objekten

```
...
public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {
        ...
        quit.setOnAction( event -> System.exit( 0 ) );
        count.setOnAction( event ->
        {
            try {
                result.setText( "" +
                    PatternMatcher.count( pattern.getText(), file.getText() ) );
            } catch ( IOException e ) { result.setText( "file error" ); }
        });
    }
}
```

eingefangene effektiv finale  
Variablen der Methode start

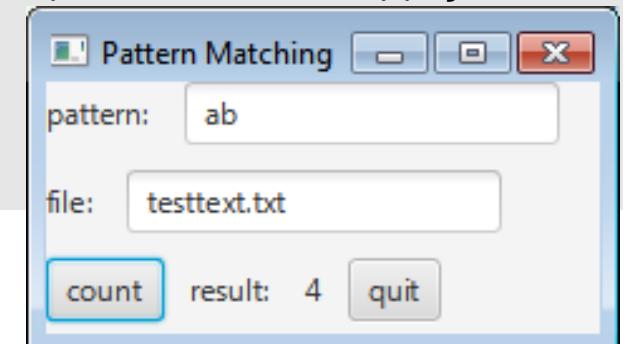
## Bestandteile der grafischen Benutzeroberfläche

(Fortsetzung)

Zuordnen von Verhalten zu den Button-Objekten

```
...
public class PatternCounting extends Application
{
    public void start( Stage primaryStage )
    {
        ...
        quit.setOnAction( event -> System.exit( 0 ) );
        count.setOnAction( event ->
        {
            try {
                result.setText( "" +
                    PatternMatcher.count( pattern.getText(), file.getText() ) );
            } catch ( IOException e ) { result.setText( "file error" ); }
        });
    }
}
```

Nach der Zuordnung der passenden EventHandler-Implementierungen kann das Programm durch Aufruf der main-Methode ausgeführt werden.



# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 18. Datenstruktur Hashtable

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-1126

## Lernziele des Kapitels 18. Datenstruktur Hashtable

Nach Durcharbeiten des Kapitels Hashtable sollen die teilnehmenden Studierenden

- die Arbeitsweise einer Hashtable kennen,
- die Aufgaben einer Hashfunktion kennen,
- die programmtechnische Realisierung einer Hashtable kennen.

## Klassen für Mengen

Bei der Implementierung von Klassen für Mengen müssen folgende Anforderungen berücksichtigt werden:

- ❑ Mengen können beliebige Elemente (eines Grundtyps) enthalten.
- ❑ Einfügen, Entfernen und Prüfen des Enthaltsenseins sollten wenig Aufwand verursachen.

## Klassen für Mengen

(Fortsetzung)

Bei der Implementierung von Klassen für Mengen müssen folgende Anforderungen berücksichtigt werden:

- ❑ Mengen können beliebige Elemente (eines Grundtyps) enthalten.
- ❑ Einfügen, Entfernen und Prüfen des Enthaltsseins sollten wenig Aufwand verursachen.

Java bietet in dem Paket `java.util` auch Klassen an, die Mengen realisieren. Darunter sind:

- ❑ `TreeSet`      Implementierung auf der Basis eines Suchbaums:
  - Einfügen und Suchen (= Prüfen des Enthaltsseins) benötigen daher  $O(\log_2 n)$  Schritte für die Suche bis zur Position im Baum.
  - Diese Form der Implementierung ist bereits bekannt.
  - Voraussetzung ist, dass für die Elemente eine Ordnungsrelation existiert.
  - Durch den Suchbaum ist ein geordneter/sortierter Durchlauf möglich.

## Klassen für Mengen

(Fortsetzung)

Bei der Implementierung von Klassen für Mengen müssen folgende Anforderungen berücksichtigt werden:

- ❑ Mengen können beliebige Elemente (eines Grundtyps) enthalten.
- ❑ Einfügen, Entfernen und Prüfen des Enthaltsenseins sollten wenig Aufwand verursachen.

Java bietet in dem Paket `java.util` auch Klassen an, die Mengen realisieren. Darunter sind:

- ❑ **TreeSet**      Implementierung auf der Basis eines Suchbaums:
  - Einfügen und Suchen (= Prüfen des Enthaltsenseins) benötigen daher  $O(\log_2 n)$  Schritte für die Suche bis zur Position im Baum.
  - Diese Form der Implementierung ist bereits bekannt.
  - Voraussetzung ist, dass für die Elemente eine Ordnungsrelation existiert.
  - Durch den Suchbaum ist ein geordneter/sortierter Durchlauf möglich.
- ❑ **HashSet**      Implementierung auf der Basis einer *Hashtable*:
  - Einfügen benötigt *konstante* Zeit:  $O(1)$
  - Suchen (= Prüfen des Enthaltsenseins) benötigt (fast) *konstante* Zeit.
  - Eine Ordnungsrelation wird nicht vorausgesetzt.
  - Ein geordneter/sortierter Durchlauf ist *nicht* möglich.

## Datenstruktur *Hashtable*

Idee:

- ❑ Gegeben sei eine Menge  $U$  (das *Universum*).
- ❑ Ordne den  $x \in U$  durch eine Funktion  $h(x): U \rightarrow \mathbb{N}$  eine natürliche Zahl (*Hashcode*) zu.
- ❑ Speichere die Elemente einer Menge  $M \subseteq U$  in einer *Tabelle*,  
die durch ein Feld  $t$  realisiert wird.
- ❑ Bestimme den Index, den  $x \in M$  in  $t$  belegen soll,  
direkt aus  $x$  als:

$$t[h(x) \% t.length].$$

Anmerkungen:

- ❑ Die Ablage erfolgt schnell und hängt nur von der Berechnungsdauer des Hashcodes ab.
- ❑ Die Suche erfolgt schnell und hängt nur von der Berechnungsdauer des Hashcodes ab.
- ❑ Die Mächtigkeit von  $U$  kann viel größer als die Länge von  $t$  sein,  
da  $h(x) \% t.length$  dafür sorgt, dass die Größe des Feldes nicht überschritten wird.

## Datenstruktur *Hashtable*

(Fortsetzung)

Beispiel:

$$U = \mathbb{IR}, M = \{ 5.1, 7.3, 8.2, 15.7 \}$$

$$h(x) = (\text{int}) x$$

ergibt für ein Feld `t` der Länge 6:

0	
1	7.3
2	8.2
3	15.7
4	
5	5.1

## Datenstruktur *Hashtable*

(Fortsetzung)

Beispiel:

$$U = \mathbb{R}, M = \{ 5.1, 7.3, 8.2, 15.7 \}$$

$$h(x) = (\text{int}) x$$

ergibt für ein Feld `t` der Länge 6:

0	
1	7.3
2	8.2
3	15.7
4	
5	5.1

Aber nun soll `11.2` aufgenommen werden mit  $h(11.2) = 11$

*Kollision, da gilt:*

`h( 11.2 ) % t.length == 5`

## Datenstruktur *Hashtable*

(Fortsetzung)

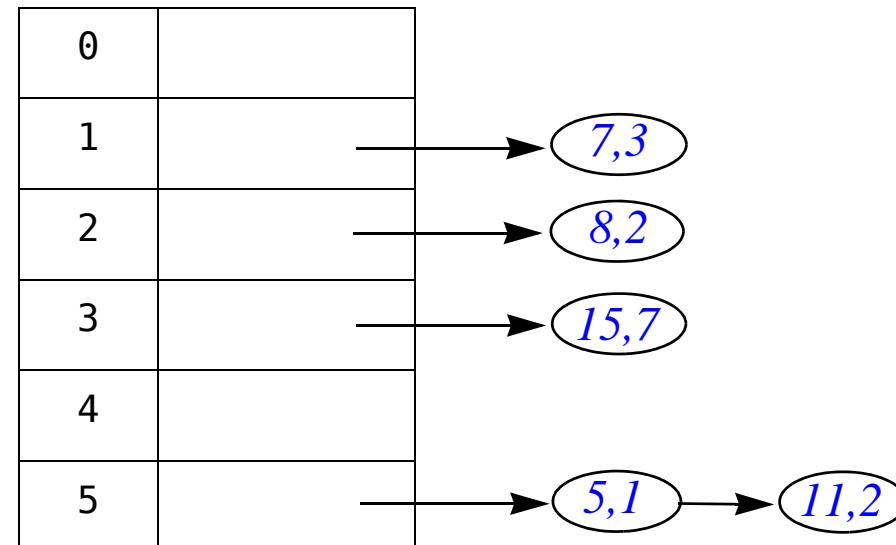
Lösung für Kollisionen von Werten mit gleichem Hashcode: *Liste von Elementen*

Beispiel:

$$U = \mathbb{IR}, M = \{ 5.1, 7.3, 8.2, 15.7, 11.2 \}$$

$$h(x) = (\text{int}) x$$

ergibt für ein Feld `t` der Länge 6:



## Datenstruktur *Hashtable*

(Fortsetzung)

Lösung für Kollisionen von Werten mit gleichem Hashcode: *Liste von Elementen*

Konsequenzen:

- ❑ Die Ablage erfolgt schnell und hängt nur von der Berechnungsdauer des Hashcodes ab.
- ❑ Die Suche erfolgt schnell und hängt von der Berechnungsdauer des Hashcodes  
*und der Länge der Kollisionsliste* an der durch den Hashcode bestimmten Position ab.
- ❑ Je stärker  $h(x)$  «streut», desto seltener treten Kollisionen auf.  
Die geeignete Implementierung von  $h(x)$  liegt in der Verantwortung des Entwicklers.
- ❑ Je mehr Werte in einer Tabelle abgelegt werden, desto größer wird die Wahrscheinlichkeit, dass Kollisionen auftreten.

Um Kollisionen zu vermeiden,  
muss die Datenstruktur Hashtable mit der Anzahl der in ihr abgelegten Werte wachsen.

## Datenstruktur *Hashtable*

(Fortsetzung)

Unterstützung von Hashing in Java:

```
public class Object
{
    ...
    public int hashCode() { ... }
    ...
}
```

- Jedes Objekt in Java besitzt eine Methode `hashCode`, die einen `int`-Wert liefert.
- Für *eine* Ausführung eines Programms liefert `hashCode` für *ein* Objekt stets den gleichen Wert. Bei einer Nutzung einer Datenstruktur, die auf einer *Hashtable* basiert – beispielsweise `HashSet` oder `HashMap`, kann auf die Methode `hashCode` zurückgegriffen werden.
- *Voraussetzung:*

Das Ergebnis von `hashCode` *muss* konsistent mit dem Ergebnis der Methode `equals` sein:  
 $a.equals(b) \Rightarrow a.hashCode() == b.hashCode()$

Falls `equals` überschrieben wird, *muss* auch `hashCode` überschrieben werden.

*Begründung:* Ein Objekt wird in einer `Hashtable` an der durch `hashCode` bestimmten Position des Feldes gesucht. Ohne konsistentes Verhalten von `equals` und `hashCode` wäre diese schnelle Prüfung nicht sicher, da gleiche Objekte auch an anderen Positionen liegen könnten.

## Beispiel-Implementierung *Hashtable*

```
import java.util.LinkedList;

public class Hashtable<T>
{
    List<T>[] table;           ← ein Feld von Listen als Tabelle
    int capacity;              ← Größe der Tabelle
    int size;                  ← Zahl der tatsächlich abgelegten Objekte
```

- Es wird eine von Java bereit gestellte Liste verwendet, um die Kollisionslisten zu verwalten.
- Das Verhältnis von `size` zu `capacity` wird dazu benutzt, um das Feld `table` rechtzeitig zu vergrößern, um so die Zahl der Kollisionen gering zu halten.
  - Die Reorganisation erfolgt im Beispiel dann, wenn gilt: `(double) size / capacity > 0.75`
  - Die Reorganisation wird als *Rehashing* bezeichnet.

## Beispiel-Implementierung *Hashtable*

(Fortsetzung)

```
public Hashtable( int c )           ← Konstruktor
{
    if ( c > 0 )
    {
        capacity = c;
    } else {
        throw new IllegalArgumentException();
    }
    table = new LinkedList[capacity];
    size = 0;
}
```

← hier ist KEINE typsichere Erzeugung möglich

- ❑ Eine getypte Erzeugung von Feldern ist in generischen Klassen nicht möglich.
- ❑ Die ungetypte Erzeugung des Feldes führt zu einer Warnung.
- ❑ Die nicht typsicheren Listen sind technisch unproblematisch, solange innerhalb der Klasse `Hashtable<T>` auch nur Objekte des Typs `T` in den Listen abgelegt werden.

## Beispiel-Implementierung *Hashtable*

(Fortsetzung)

```
@SuppressWarnings({"unchecked"})
public Hashtable( int c )
{
    if ( c > 0 )
    {
        capacity = c;
    } else {
        throw new IllegalArgumentException();
    }
    table = new LinkedList[capacity];
    size = 0;
}
```



Annotation, unterdrückt die  
Warnung



hier ist KEINE typsichere Erzeugung  
möglich

- ❑ Die Annotation (siehe auch Folie 702) unterdrückt die *hier erwartete* Warnung.
- ❑ Um nicht von unerwarteten und damit hilfreichen Warnungen abgelenkt zu werden, können solche erwarteten Warnungen *im Einzelfall* gezielt unterdrückt werden.
- ❑ Der Einsatz von `@SuppressWarnings` sollte zusätzlich durch einen Kommentar erklärt werden.

## Beispiel-Implementierung *Hashtable*

(Fortsetzung)

```
@SuppressWarnings({"unchecked"})
public Hashtable( int c )
{
    if ( c > 0 )
    {
        capacity = c;
    } else {
        throw new IllegalArgumentException();
    }
    table = new LinkedList[capacity];
    size = 0;
    for( int i = 0; i < capacity; i++ )
    {
        table[i] = new LinkedList<>();
    }
}
```

← für jedes Element des Feldes muss  
eine eigene Liste erzeugt werden

## Annotationen (Exkurs)

- ❑ Annotationen sind eine Möglichkeit, Programmtexte mit zusätzlichen Informationen (Metadaten) anzureichern.
- ❑ Annotationen müssen einem fest vorgegebenen syntaktischen Aufbau folgen.
- ❑ Annotationen können während der Übersetzung ausgewertet werden.
- ❑ Annotationen können während der Ausführung ausgewertet werden.
- ❑ Vordefinierte Annotationen:
  - `@Override` die markierte Methode überschreibt eine Methode der Oberklasse
  - `@FunctionalInterface` das markierte Interface enthält genau eine abstrakte Methode
  - `@Deprecated` *veraltete* Methode, sollte nicht genutzt werden
  - `@SuppressWarnings` unterdrückt Warnungen des Compilers
- ❑ Annotationen können selbst definiert werden als Abwandlung einer `interface`-Deklaration:
 

```
@Retention( RetentionPolicy.RUNTIME )
@Target( { ElementType.Method } )
public @interface Info {
    int version();
    String author();
}
```

**Auswertungszeitpunkt festlegen**  
**Wirkungsbereich festlegen**  
**Deklaration einer Annotation mit zwei Parametern**
- ❑ Nutzung:
 

```
@Info( version = 5, author = "Stefan" )
```

## Beispiel-Implementierung *Hashtable*

(Fortsetzung)

```
private int position( T o )
{
    return Math.abs( o.hashCode() % capacity );
}

public boolean contains( T o )
{
    return table[position( o )].contains( o );
}
```

← setzt korrekte Implementierung  
von hashCode voraus

- Die Methode `position` bestimmt aus dem `hashCode` eines Objekts die Position, an der dieses Objekt im Feld abgelegt wird.  
Da alle Methoden diese Position bestimmen müssen, ist die Berechnung in eine private Methode ausgelagert worden.
- Die Methode `contains` der `Hashtable` bestimmt die Position des zu suchenden Objekts im Feld und delegiert dort die Aufgabe an die Methode `contains` des `LinkedList`-Objekts.

## Beispiel-Implementierung *Hashtable*

(Fortsetzung)

```

public void put( T o )
{
    if ( !table[position( o )].contains( o ) ) ← o ist nicht enthalten
    {
        if ( (double)(size + 1) / capacity > 0.75 ) ← Reorganisation notwendig
        {
            rehash();
        }
        table[position( o )].add( o );
        size++;
    }
}

public void remove( T o )
{
    if ( table[position( o )].remove( o ) ) ← remove gibt true zurück,
    {
        size--;
    }
}

```

- ❑ Typischerweise wird die Tabelle bei *Unterschreiten* der 75%-Füllung *nicht* verkleinert.

## Beispiel-Implementierung *Hashtable*

(Fortsetzung)

```
@SuppressWarnings({"unchecked"})
```

```
void rehash()
```

```
{
```

```
    capacity = 2 * capacity;
```

```
    LinkedList<T>[] oldTable = table;
```

```
    table = new LinkedList[capacity];
```

```
    for( int i = 0; i < capacity; i++ )
```

```
{
```

```
    table[i] = new LinkedList<>();
```

```
}
```

```
    size = 0;
```

```
    for ( LinkedList<T> list : oldTable )
```

```
{
```

```
        for ( T elem : list )
```

```
{
```

```
            if ( elem != null )
```

```
{
```

```
                put( elem );
```

```
}
```

```
}
```

```
}
```

← neue Kapazität wird festgelegt

← Feld-Attribut ist eine Referenz

← größeres Feld wird initialisiert

← size wird durch put erhöht

Die Reorganisation muss über put erfolgen, damit die größere Kapazität bei der Verteilung der Objekte auch genutzt wird

## Datenstruktur *Hashtable*

(Fortsetzung)

Zusammenfassung:

- Die *Hashtable* ist eine Datenstruktur, in der eine ungeordnete Menge von Daten schnell zugreifbar abgelegt werden kann.
- Das Auftreten von Kollisionen hängt von der *Qualität* der hashCode-Funktion, der Größe des Feldes und den eingefügten Objekten ab.
- Die hashCode-Funktion ist üblicherweise *nicht injektiv*, zu viele gleiche Funktionswerte erhöhen aber das Auftreten von Kollisionen deutlich.
- Die Normierung mit %capacity führt dazu, dass auch unterschiedliche Funktionswerte der hashCode-Funktion auf gleiche Positionen im Feld abgebildet werden.
- Die hashCode-Funktion sollte schnell berechenbar sein.  
Alternativ möglich ist die *einmalige* Berechnung und Abspeicherung eines hashCode-Wertes durch den Konstruktor der abzulegenden Objekte.  
Die hashCode-Methode gibt dann den einmal berechneten Wert nur noch zurück.
- *Rehashing* ist eine aufwändige Operation, bei der jeweils das sonst schnelle Einfügen eines Objekts deutlich verzögert erfolgt.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## 19. Prioritätswarteschlange (Heap)

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-1146

## Lernziele des Kapitels 19. Prioritätswarteschlange (Heap)

Nach Durcharbeiten des Kapitels Prioritätswarteschlange (Heap) sollen die teilnehmenden Studierenden

- die Datenstruktur *Heap* kennen,
- das Einfügen in einen Heap kennen,
- das Löschen aus einem Heap kennen,
- die Visualisierung eines Heaps als Baum kennen,
- die Abbildung des Baums in ein Feld kennen,
- die Implementierung der Klasse `PriorityQueue` kennen,
- den Algorithmus *HeapSort* und seine Implementierung kennen.

## Datenstruktur *Prioritätswarteschlange*

Eine Prioritätswarteschlange ist eine Datenstruktur,

- zu den Objekten hinzugefügt werden,
- die später wieder entfernt werden.

Die Reihenfolge, in der die Objekte entfernt werden,  
wird anhand einer objektspezifischen Priorität bestimmt.

Beispiele

- Warteschlange beim Bäcker:  
Priorität richtet sich nach der Ankunftszeit.
- Warteschlange beim Flug-Einchecken:  
Priorität richtet sich nach Sitzplatzklasse und Ankunftszeit.
- Warteschlange bei der Notfallambulanz:  
Priorität richtet sich nach Schwere der Erkrankung und Ankunftszeit.
- Warteschlange im Auslieferungslager:  
Priorität richtet sich an der Gesamtsituation aus.

## Datenstruktur *Prioritätswarteschlange*

(Fortsetzung)

Eine Prioritätswarteschlange ist eine Datenstruktur,

- zu den Objekten hinzugefügt werden,
- die später wieder entfernt werden.

Die Reihenfolge, in der die Objekte entfernt werden,  
wird anhand einer objektspezifischen Priorität bestimmt.

Benötigt wird eine Datenstruktur, bei der

- schnell* das Objekt mit der höchsten Priorität bestimmt werden kann,
- dieses Objekt *schnell* entfernt werden kann und
- schnell* ein neues Objekt eingefügt werden kann.

## Datenstruktur *Prioritätswarteschlange*

(Fortsetzung)

Eine Prioritätswarteschlange ist eine Datenstruktur,

- zu den Objekten einzugefügt werden,
- die später wieder entfernt werden.

Die Reihenfolge, in der die Objekte entfernt werden,  
wird anhand einer objektspezifischen Priorität bestimmt.

Benötigt wird eine Datenstruktur, bei der

- schnell* das Objekt mit der höchsten Priorität bestimmt werden kann,
- dieses Objekt *schnell* entfernt werden kann und
- schnell* ein neues Objekt eingefügt werden kann.

Lösungsideen auf der Basis der bekannten Datenstrukturen:

- Objekte an ungeordnete Liste hängen, Objekt mit höchster Priorität wird gesucht
- Objekte in geordneter Liste ablegen, neues Objekt wird eingesortiert
- Objekte in binären Suchbaum eingesortieren

## Datenstruktur *Prioritätswarteschlange*

(Fortsetzung)

Eine Prioritätswarteschlange ist eine Datenstruktur,

- zu den Objekten einzugefügt werden,
- die später wieder entfernt werden.

Die Reihenfolge, in der die Objekte entfernt werden,  
wird anhand einer objektspezifischen Priorität bestimmt.

Benötigt wird eine Datenstruktur, bei der

- schnell* das Objekt mit der höchsten Priorität bestimmt werden kann,
- dieses Objekt *schnell* entfernt werden kann und
- schnell* ein neues Objekt eingefügt werden kann.

Lösungsideen auf der Basis der bekannten Datenstrukturen:

- Objekte an ungeordnete Liste hängen, Objekt mit höchster Priorität wird gesucht
- Objekte in geordneter Liste ablegen, neues Objekt wird eingesortiert
- Objekte in binären Suchbaum eingesortieren
- Objekte in binären Baum eingesortieren, der bei Änderungen Baumstruktur garantiert:

Heap

## Datenstruktur *Heap*

Die Datenstruktur *Heap* (engl. für *Haufen*) nutzt einen *teilweise sortierten binären* Baum, um

- schnell* den größten Knoten bezüglich einer vorgegebenen Ordnung zu finden,
- schnell* den größten Knoten bezüglich einer vorgegebenen Ordnung zu löschen und
- schnell* einen neuen Knoten einzufügen.

Ein (*Max-*)*Heap* ist

- ein *binärer Baum*,
- der *vollständig* ist,
- für dessen Wurzel gilt,
  - dass sie eine *größere* Beschriftung als ihre Kinder besitzt und
  - dass die beiden Teilbäume ihrerseits auch wieder Heaps sind. (*Max-Heap-Bedingung*)

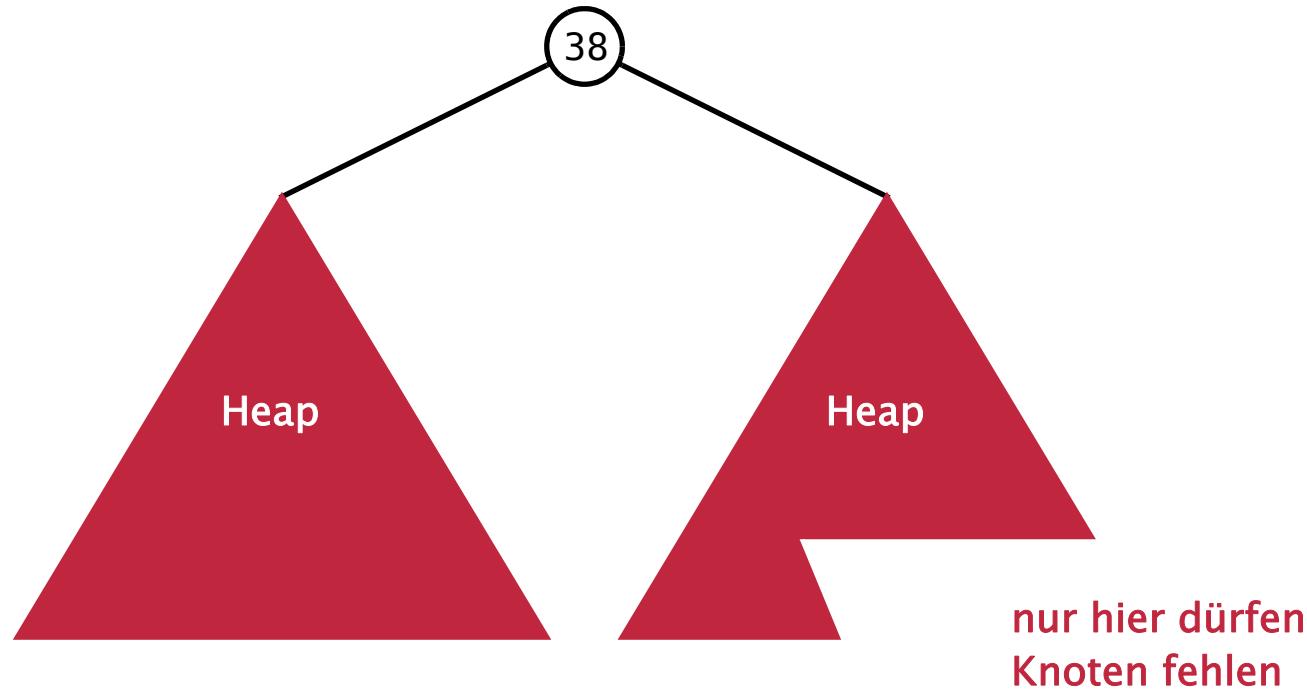
Ein Baum ist *vollständig*,

wenn bis zur vorletzten Ebene jede Ebene ihre maximale Zahl von Knoten enthält und in der letzten Ebene nur von rechts Knoten fehlen.

## Datenstruktur *Heap*

(Fortsetzung)

Beispiel (die Heap-Bedingung orientiert sich hier an der Ordnung der ganzen Zahlen)



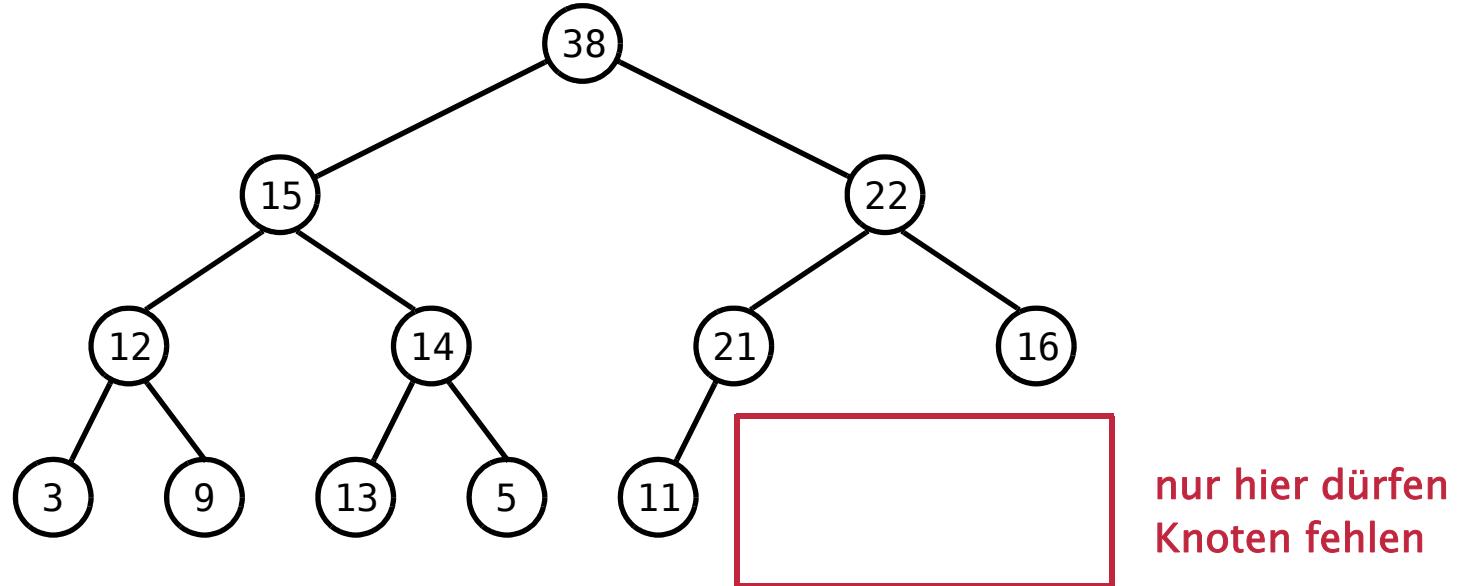
Eigenschaft:

- Der größte Knoten (38) steht in der Wurzel (*Max-Heap*).

## Datenstruktur *Heap*

(Fortsetzung)

Beispiel (die Heap-Bedingung orientiert sich hier an der Ordnung der ganzen Zahlen)



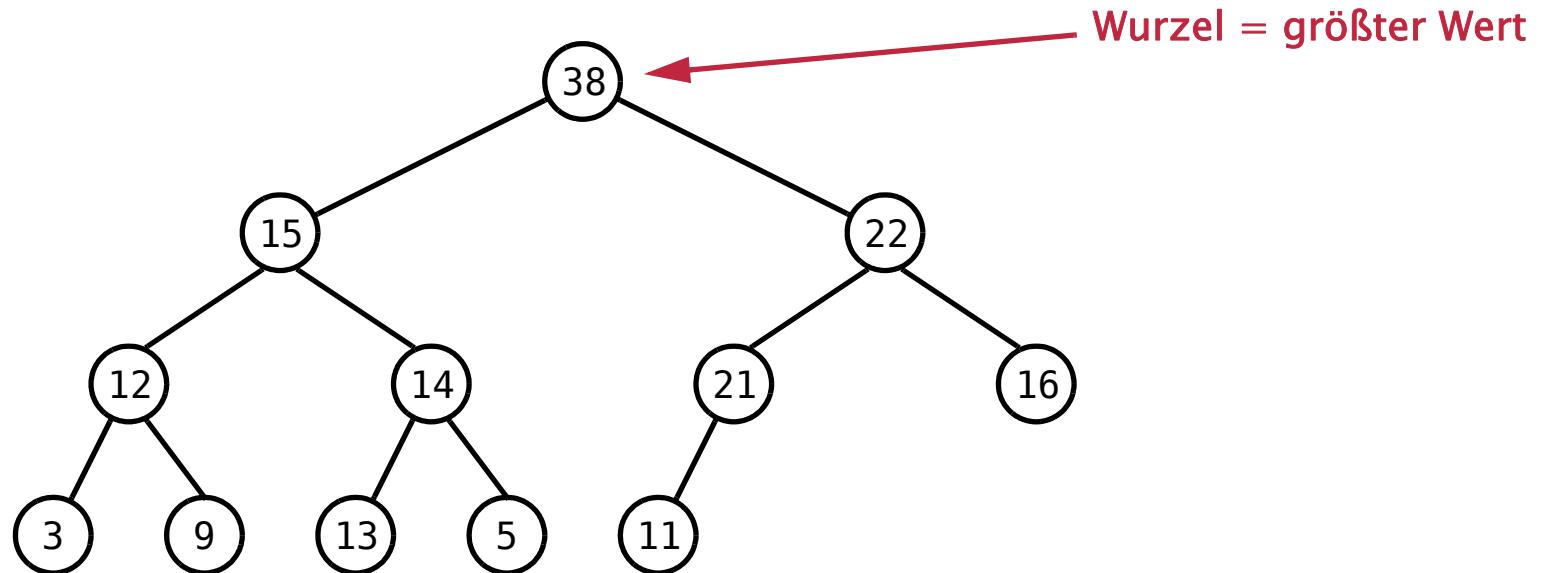
zwei unmittelbar erkennbare weitere Eigenschaften:

- ❑ Der größte Wert (38) steht in der Wurzel, der zweitgrößte (22) in der zweiten Ebene, der drittgrößte (21) höchstens in der dritten Ebene hinter dem zweitgrößten usw.
- ❑ Ein Baum für  $n$  Werte hat höchsten  $\log_2(n)+1$  Ebenen.

## Datenstruktur *Heap*

(Fortsetzung)

Das *Finden* des größten Wertes ist trivial – das ist der Wert in der Wurzel!



## Datenstruktur *Heap* – Entfernen des größten Wertes

- ❑ Ziel:  
Nach dem Entfernen des Wertes der Wurzel muss diese einen neuen Wert erhalten, so dass wieder die Heap-Bedingung im ganzen Baum gilt.
- ❑ Voraussetzung:  
Es liegen zwei Heaps vor, für die eine gemeinsame Wurzel gefunden werden muss.

## Datenstruktur *Heap* – Entfernen des größten Wertes

- Ziel:  
Nach dem Entfernen des Wertes der Wurzel muss diese einen neuen Wert erhalten, so dass wieder die Heap-Bedingung im ganzen Baum gilt.
- Voraussetzung:  
Es liegen zwei Heaps vor, für die eine gemeinsame Wurzel gefunden werden muss.
- Idee:
  - Der neue Wert der Wurzel muss aus den Werten der beiden Teilheaps ausgewählt werden.
  - Die Vollständigkeit muss erhalten bleiben.

## Datenstruktur **Heap** – Entfernen des größten Wertes

□ Ziel:

Nach dem Entfernen des Wertes der Wurzel muss diese einen neuen Wert erhalten, so dass wieder die Heap-Bedingung im ganzen Baum gilt.

□ Voraussetzung:

Es liegen zwei Heaps vor, für die eine gemeinsame Wurzel gefunden werden muss.

□ Idee:

- Der neue Wert der Wurzel muss aus den Werten der beiden Teilheaps ausgewählt werden.
- Die Vollständigkeit muss erhalten bleiben.

□ Algorithmus:

- Lösche den äußersten rechten Knoten der letzten Ebene.

(Die Vollständigkeit bleibt erhalten.)

- Setze den Wert dieses Knotens in die Wurzel ein.
- Vergleiche den Wert der Wurzel mit dem größeren Wert ihrer Kinder:  
Ist der Wert der Wurzel kleiner, vertausche beide Werte und

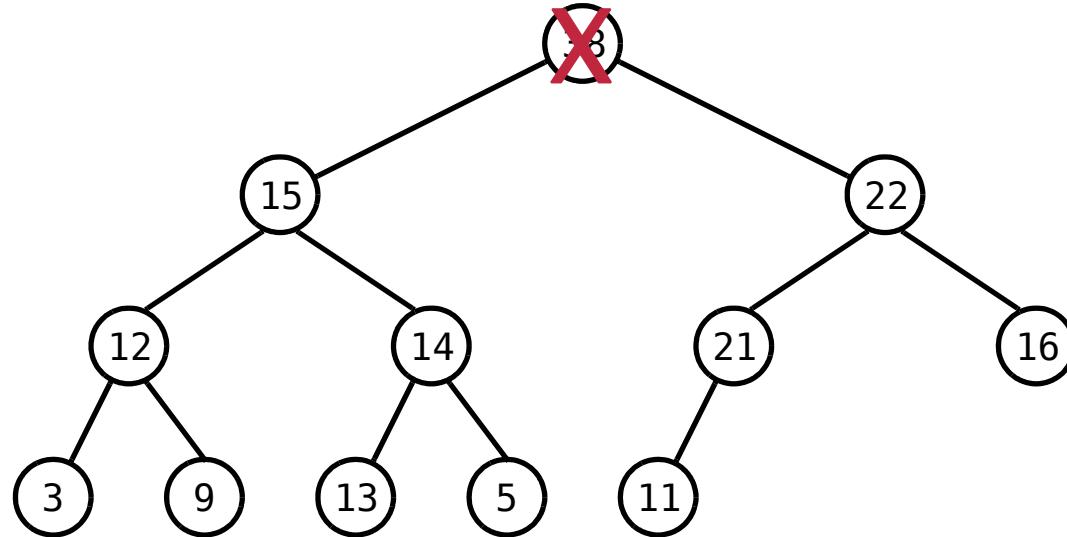
(Die Heap-Bedingung wird hergestellt.)

wiederhole das Vorgehen für den betroffenen Teilbaum.

- Das Verfahren endet, wenn keine Vertauschung vorgenommen wurde oder ein Blatt erreicht wird.

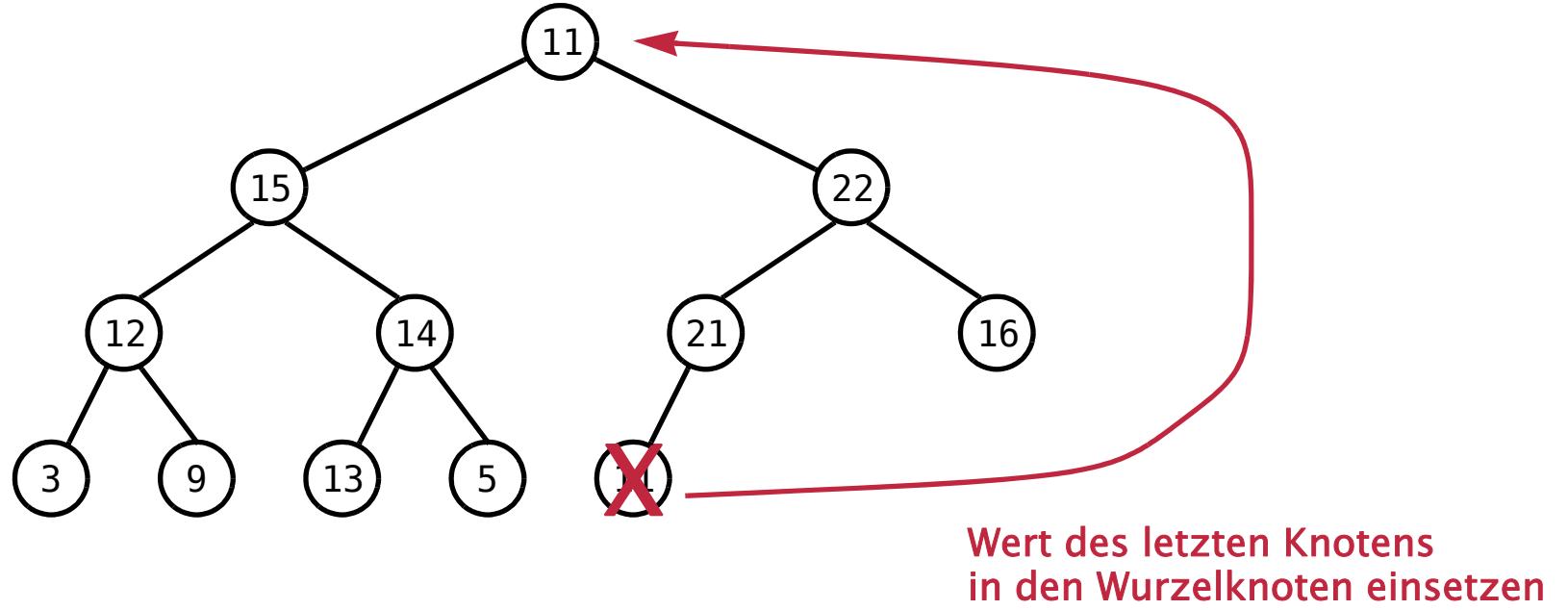
## Datenstruktur *Heap* – Entfernen des größten Wertes

(Fortsetzung)



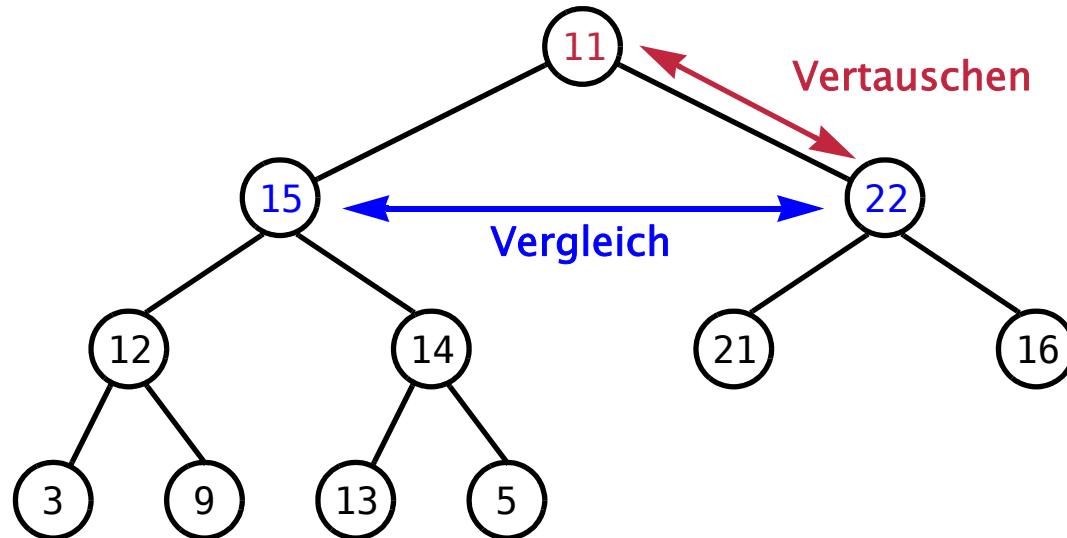
## Datenstruktur *Heap* – Entfernen des größten Wertes

(Fortsetzung)



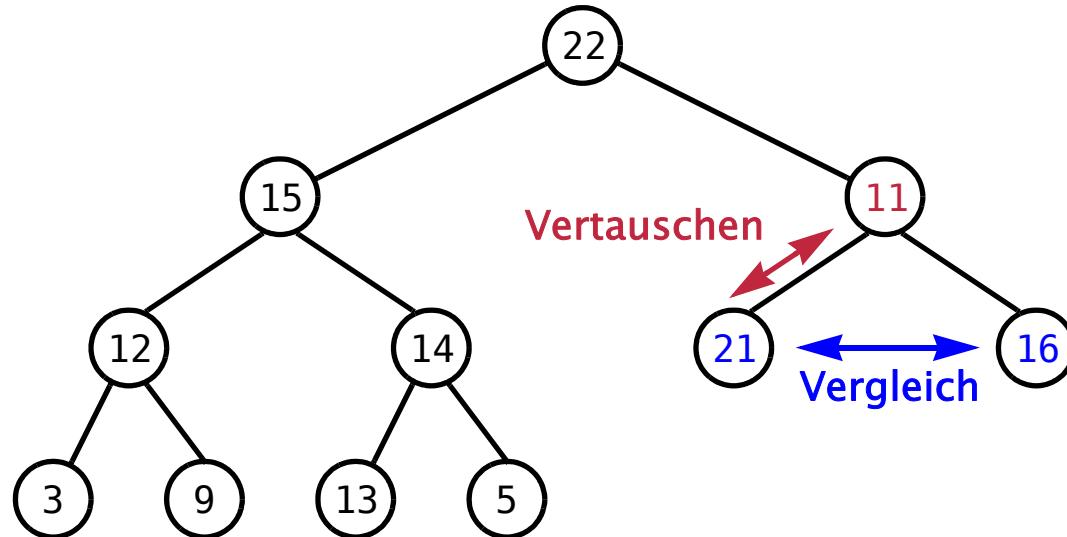
## Datenstruktur *Heap* – Entfernen des größten Wertes

(Fortsetzung)



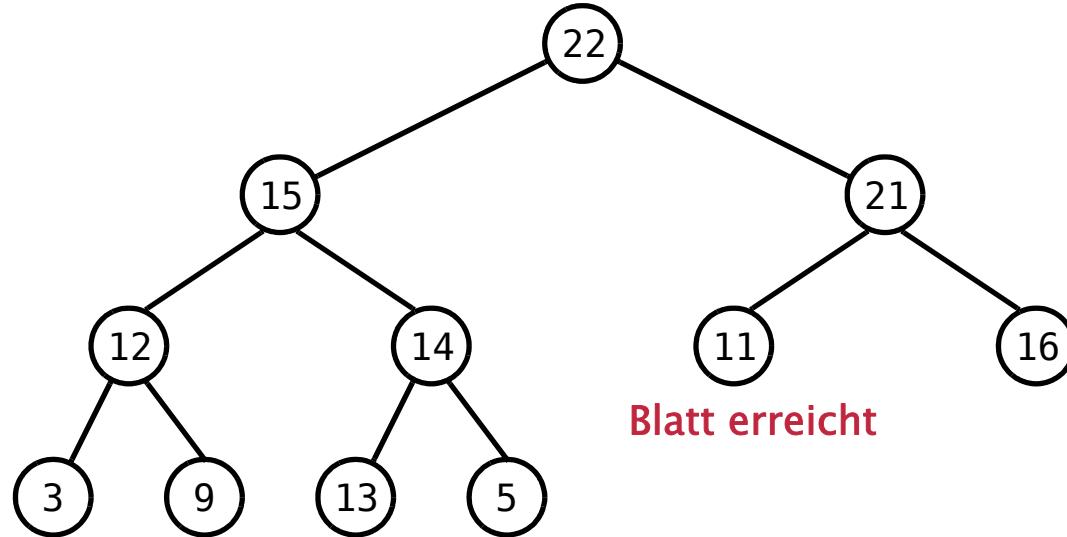
## Datenstruktur *Heap* – Entfernen des größten Wertes

(Fortsetzung)



## Datenstruktur *Heap* – Entfernen des größten Wertes

(Fortsetzung)



- Mit höchstens  $\log_2(n)$  Schritten ist die Heap-Bedingung im gesamten Baum wieder hergestellt.
- Anmerkung:  
Das Erhalten einer vollständigen Ordnung in einer sortierten Liste beim Entfernen des ersten Elements erfordert demgegenüber *gar keinen* Aufwand.

## Datenstruktur *Heap* – Einfügen eines Wertes

- ❑ Ziel:  
Nach dem Einfügen des Wertes muss der Baum die Heap-Bedingung erfüllen.
- ❑ Voraussetzung:  
Es liegt ein Heap vor.

## Datenstruktur *Heap* – Einfügen eines Wertes

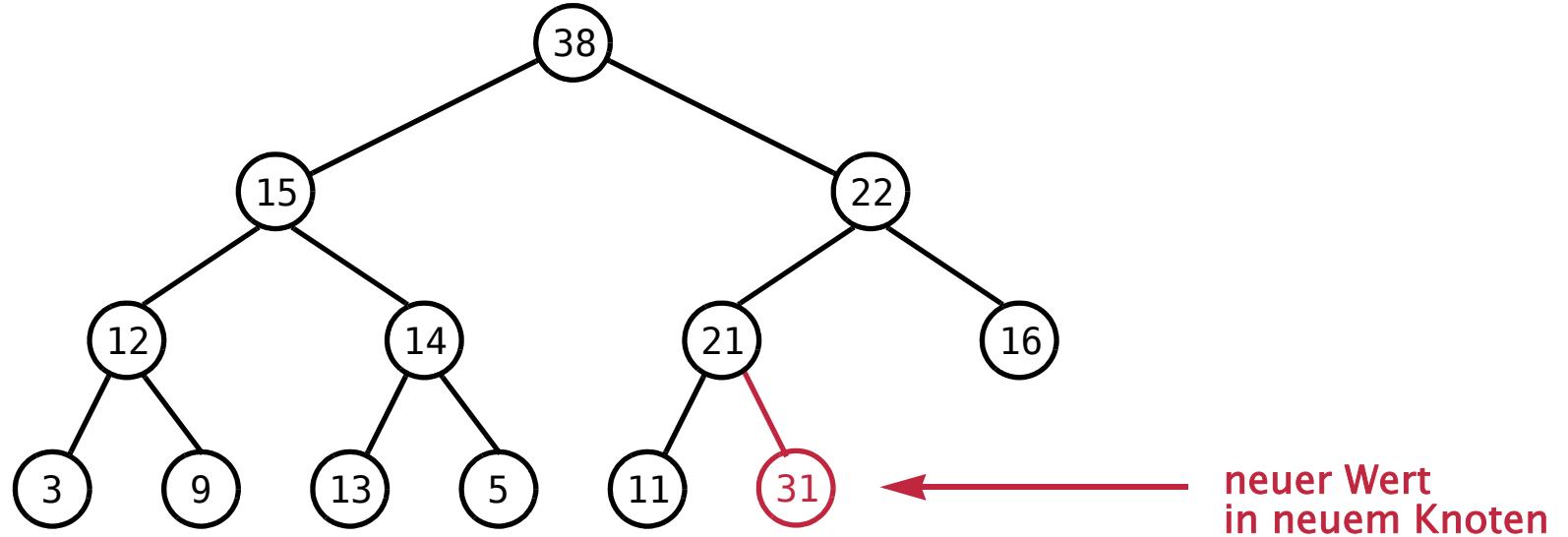
- ❑ Ziel:  
Nach dem Einfügen des Wertes muss der Baum die Heap-Bedingung erfüllen.
- ❑ Voraussetzung:  
Es liegt ein Heap vor.
- ❑ Idee:
  - Der neue Wert muss in einem neuen Knoten ergänzt werden.
  - Die Vollständigkeit muss erhalten bleiben.

## Datenstruktur **Heap** – Einfügen eines Wertes

- Ziel:  
Nach dem Einfügen des Wertes muss der Baum die Heap-Bedingung erfüllen.
- Voraussetzung:  
Es liegt ein Heap vor.
- Idee:
  - Der neue Wert muss in einem neuen Knoten ergänzt werden.
  - Die Vollständigkeit muss erhalten bleiben.
- Algorithmus:
  - Ergänze einen äußersten rechten Knoten in der letzten Ebene.  
**(Die Vollständigkeit bleibt erhalten.)**
  - Setze den neuen Wert in diesen Knoten ein.
  - Vergleiche den Wert dieses Knotens mit dem Wert seines Vorgängers:  
Ist der Wert des Vorgängers kleiner, vertausche beide Werte und  
**(Heap-Bedingung)**  
wiederhole das Vorgehen für den betroffenen Vorgänger.
  - Das Verfahren endet, wenn keine Vertauschung vorgenommen wurde oder die Wurzel erreicht ist.

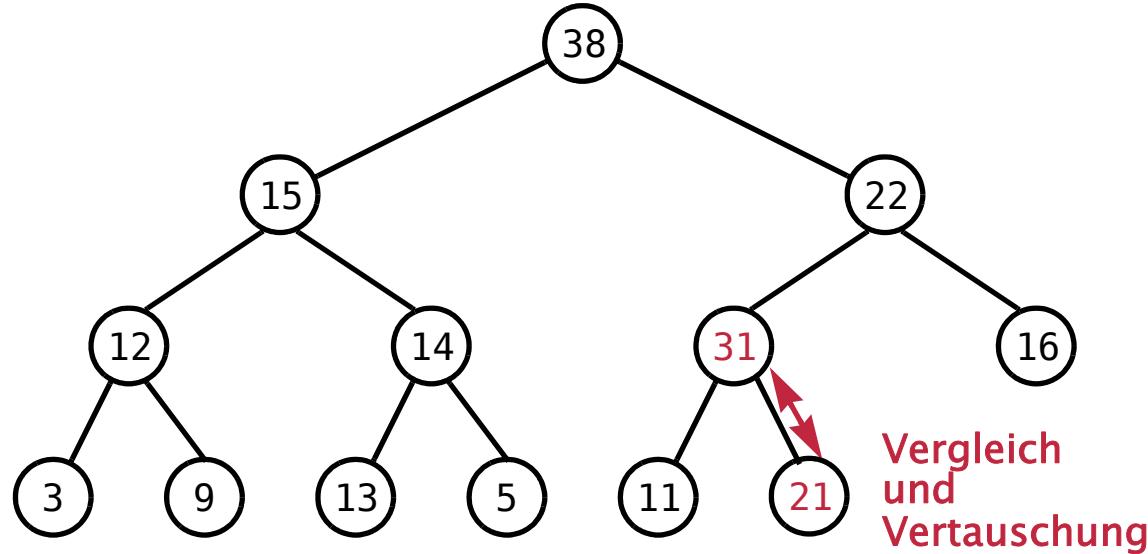
## Datenstruktur *Heap* – Einfügen eines Wertes

(Fortsetzung)



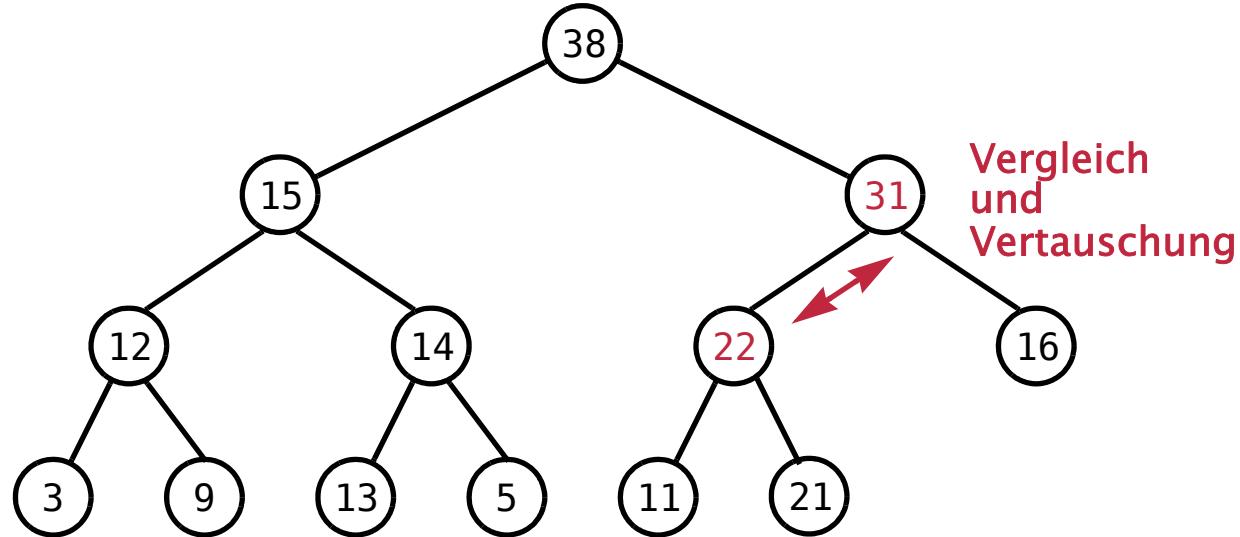
## Datenstruktur *Heap* – Einfügen eines Wertes

(Fortsetzung)

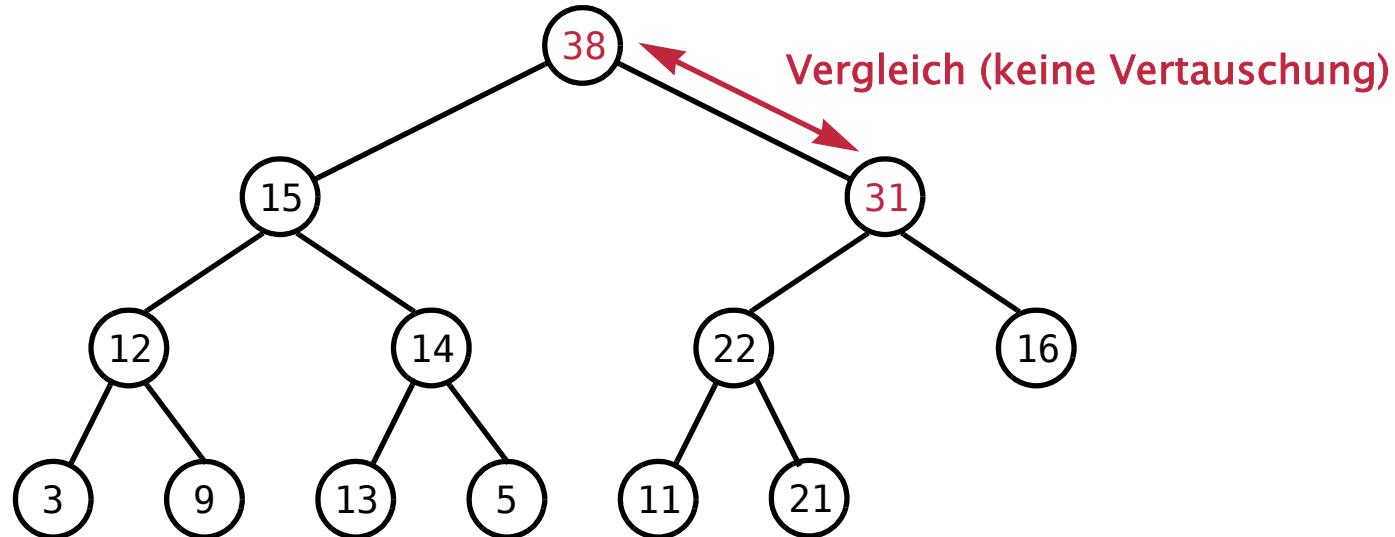


## Datenstruktur *Heap* – Einfügen eines Wertes

(Fortsetzung)



## Datenstruktur *Heap* – Einfügen eines Wertes (5)



- Mit höchstens  $\log_2(n)$  Schritten ist die Heap-Bedingung im gesamten Baum wieder garantiert.
- Das Herstellen einer Sortierung in einer Liste mit  $n$  Werten würde demgegenüber bis zu  $n$  Schritte erfordern.
- In der Summe der notwendigen Schritte für gleichhäufig auftretendes Einfügen und Entfernen schneidet der Heap daher besser als die Lösung mit Sortieren ab.

## Überlegungen zur Implementierung

- Aufgrund der Vollständigkeit kann berechnet werden,  
wie viele Knoten auf jeder Ebene vorhanden sind:

Wurzel	$2^0 = 1$ Knoten
2-te Ebene	$2^1 = 2$ Knoten
3-te Ebene	$2^2 = 4$ Knoten
4-te Ebene	$2^3 = 8$ Knoten
...	
n-te Ebene	$2^{n-1}$ Knoten

## Überlegungen zur Implementierung

- Aufgrund der Vollständigkeit kann berechnet werden,  
wie viele Knoten auf jeder Ebene vorhanden sind:

Wurzel	$2^0 = 1$ Knoten
2-te Ebene	$2^1 = 2$ Knoten
3-te Ebene	$2^2 = 4$ Knoten
4-te Ebene	$2^3 = 8$ Knoten
...	
n-te Ebene	$2^{n-1}$ Knoten

- Ebenso lässt sich berechnen, wo in einer Ebene die Kinder eines Knotens liegen:  
Die Kinder des ersten Knotens der Ebene  $n$  liegen an den Positionen 1 und 2 der folgenden Ebene  $n+1$ ,  
die Kinder des 2-ten Knotens der Ebene  $n$  liegen auf den Positionen 3 und 4 der folgenden Ebene  $n+1$ ,  
usw.

## Überlegungen zur Implementierung

- Aufgrund der Vollständigkeit kann berechnet werden, wie viele Knoten auf jeder Ebene vorhanden sind:

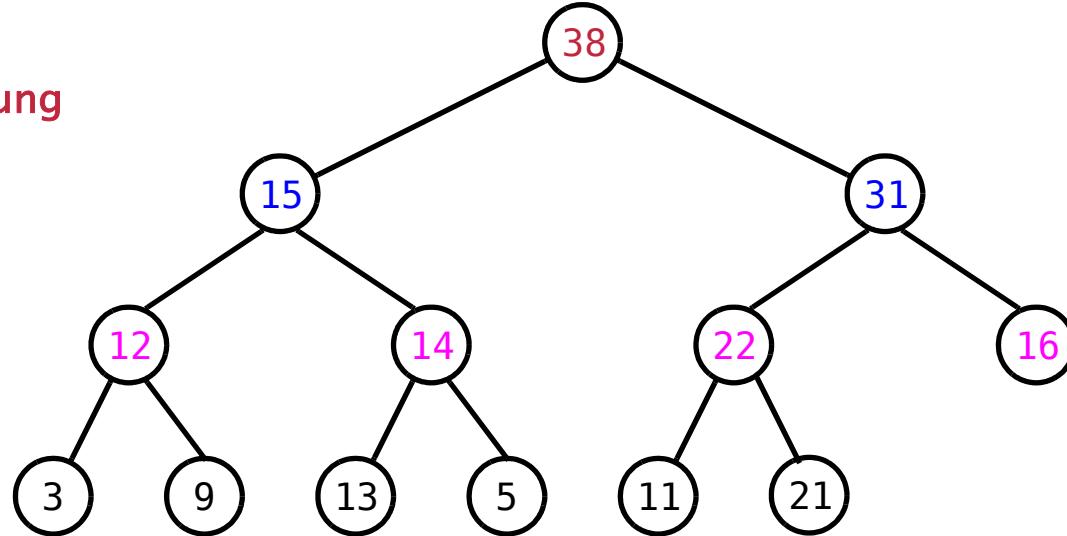
Wurzel	$2^0 = 1$ Knoten
2-te Ebene	$2^1 = 2$ Knoten
3-te Ebene	$2^2 = 4$ Knoten
4-te Ebene	$2^3 = 8$ Knoten
...	
n-te Ebene	$2^{n-1}$ Knoten

- Ebenso lässt sich berechnen, wo in einer Ebene die Kinder eines Knotens liegen:  
Die Kinder des ersten Knotens der Ebene  $n$  liegen an den Positionen 1 und 2 der folgenden Ebene  $n+1$ ,  
die Kinder des 2-ten Knotens der Ebene  $n$  liegen auf den Positionen 3 und 4 der folgenden Ebene  $n+1$ ,  
usw.
- Die Werte der Ebenen können daher *hintereinander* in einem Feld gespeichert und die notwendigen Indexpositionen berechnet werden.  
**(Das lässt sich schnell ausführen!)**
- Eine Implementierung als rekursive Datenstruktur ist nur notwendig, sofern eine dynamische Erweiterbarkeit benötigt wird.

# Überlegungen zur Implementierung

(Fortsetzung)

Baumdarstellung

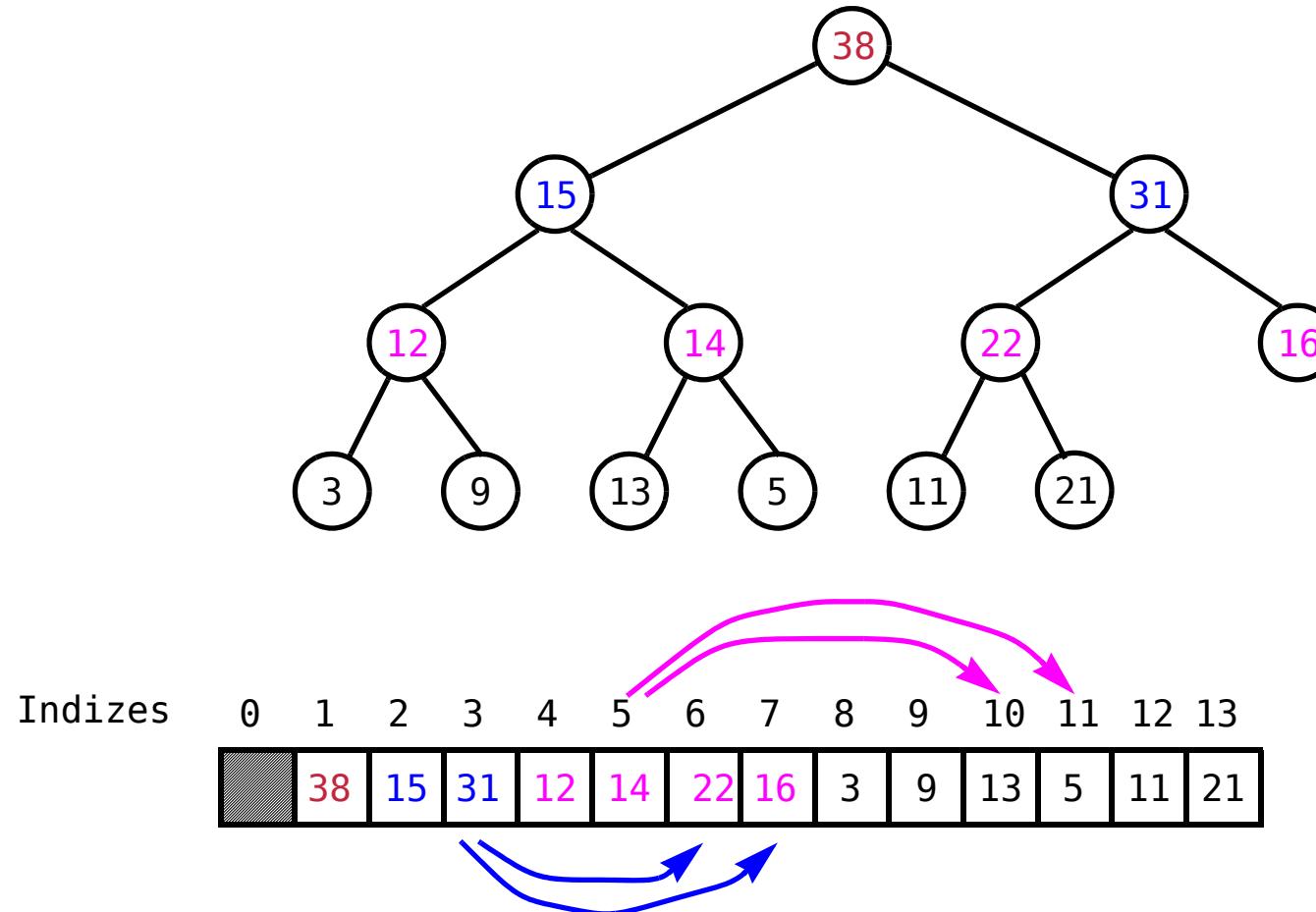


Feld

38	15	31	12	14	22	16	3	9	13	5	11	21
----	----	----	----	----	----	----	---	---	----	---	----	----

# Überlegungen zur Implementierung

(Fortsetzung)



# Überlegungen zur Implementierung

(Fortsetzung)

Planung der Klasse `PriorityQueue<T extends Comparable<? super T>>`

- ❑ Konstruktor, der ein Feld der notwendigen Länge anlegt.
- ❑ Öffentliche Methode `void add( T content )`, die `content` in den Heap einfügt.
- ❑ Öffentliche Methode `T poll()`, die das Objekt mit der höchsten Priorität zurückgibt und aus dem Heap löscht.
- ❑ Öffentliche Methode `int size()`, die die Anzahl der im Feld abgelegten Objekte zurückgibt.
- ❑ Private Methode `void heapify()`, die das Objekt in der Wurzel so tief in einen der beiden Teilheaps einsinken lässt, dass die Heapbedingung wieder erfüllt ist.
- ❑ Private Methode `void swap( int i, int j )`, die Objekte an den Indizes `i` und `j` im Feld vertauscht.
- ❑ Anmerkung:  
Der Vergleich der Prioritäten von `T`-Objekten soll über den bekannten Vergleich mit der Methode `compareTo` erfolgen.

## Implementierung der Klasse **PriorityQueue<T extends Comparable<? super T>>**

```
public class PriorityQueue<T extends Comparable<? super T>>
{
    private Object[] nodes;
    private int size;

    public PriorityQueue( int capacity ) {
        if ( capacity >= 0 ) {
            nodes = new Object[capacity+1];
        } else {
            throw new IllegalStateException();
        }
        size = 0;
    }

    ...
}
```



Das Element am Index 0 bleibt ungenutzt, daher muss ein zusätzliches Element erzeugt werden.

## Implementierung der Klasse `PriorityQueue<T extends Comparable<? super T>>` (Fortsetzung)

```

public void add( T content )
{
    if ( size < nodes.length-1 ) {
        size++;
        nodes[size] = content;           ← neues Objekt am Ende anfügen
        int current = size;
        boolean stop = (current == 1);
        while ( !stop )
        {
            int parent = current/2;      ← Vorgänger bestimmen
            if ( parent>0 && ((T)nodes[current]).compareTo((T)nodes[parent])>0 )
            {
                swap ( current, parent );
                current = parent;
            } else {
                stop = true;
            }
        }
    }
}

```

- Das am Ende angefügte Objekt wird im Heap so lange in Richtung der Wurzel verschoben, bis es auf einen kleineren Vorgänger im Heap trifft oder die Wurzel erreicht ist.

## Implementierung der Klasse `PriorityQueue<T extends Comparable<? super T>>` (Fortsetzung)

```

public T poll()
{
    if ( size >= 1 ) {
        T first = (T)nodes[1];
        nodes[1] = nodes[size];
        nodes[size] = null;
        size--;
        heapify();
        return first;
    } else {
        throw new IllegalStateException();
    }
}

```

The code is annotated with red arrows pointing from the right side to specific parts of the code:

- An arrow points to the condition `size >= 1` with the text "wirft Ausnahme, falls kein Objekt im Heap".
- An arrow points to the assignment `nodes[1] = nodes[size];` with the text "letztes Objekt nach vorne".
- An arrow points to the call `heapify();` with the text "Heap-Bedingung herstellen".
- An arrow points to the `IllegalStateException()` constructor with the text "letztes Objekt nach vorne".

- Die Methode `heapify` macht aus einem Baum mit einem beliebigen Wert in der Wurzel, deren linker und rechter Teilbaum jeweils Heaps sind, einen Heap.

## Implementierung der Klasse `PriorityQueue<T extends Comparable<? super T>>` (Fortsetzung)

```

private void heapify() {
    int current = 1;
    int leftChild, rightChild, largerChild;
    boolean stop = false;
    while ( !stop ) {
        leftChild = 2*current;
        rightChild = leftChild+1;
        if ( leftChild <= size ) {
            if ( rightChild <= size ) {
                if ( ((T)nodes[rightChild]).compareTo( (T)nodes[leftChild] ) > 0 ) {
                    largerChild = rightChild;
                } else {
                    largerChild = leftChild;
                }
            } else {
                largerChild = leftChild;
            }
            if ( ( (T)nodes[largerChild] ).compareTo( (T)nodes[current] ) > 0 ) {
                swap( current, largerChild );
                current = largerChild;
            } else {
                stop = true;
            }
        } else {
            stop = true;
        }
    }
}

```

**Überblick**

Kinder bestimmen

kleineres Kind bestimmen

bei Bedarf tauschen

nächste Folie

## Implementierung der Klasse `PriorityQueue<T extends Comparable<? super T>>` (Fortsetzung)

```

if ( leftChild <= size )
{
    if ( rightChild <= size )
    {
        if ( ((T)nodes[rightChild]).compareTo((T)nodes[leftChild]) > 0 )
        {
            largerChild = rightChild; ← es gibt mindestens 1 Kind
        } else {
            largerChild = leftChild; ← es gibt zwei Kinder
        }
    } else {
        largerChild = leftChild;
    }
    if ( ( (T)nodes[largerChild] ).compareTo( (T)nodes[current] ) > 0 )
    {
        swap( current, largerChild ); ← das größere Kind
        current = largerChild; ← bestimmen
    } else {
        stop = true; ← keine Vertauschung
    }
} else {
    stop = true; ← kein Kind vorhanden
}

```

## Zusammenfassung Datenstruktur Heap

- ❑ Der Zugriff auf das Objekt mit der größten Priorität kann unmittelbar erfolgen.
  - ❑ Das Einfügen eines Objekts in den Heap erfordert *höchstens*  $\log_2(n)$  Schritte.
  - ❑ Das Entfernen des Objekts aus der Wurzel des Heaps erfordert *höchstens*  $\log_2(n)$  Schritte.
- 
- ❑ Der Datenstruktur Heap liegt die Vorstellung eines binären Baums zugrunde.
  - ❑ Der Baum kann aufgrund der gegebenen Randbedingungen sequentialisiert werden:  
Der Implementierung des Heaps liegt ein Feld zugrunde.

## Erweiterung: Sortieren mit einem Heap

Gegeben ist eine Folge von  $n$  Werten, die sortiert werden sollen.

### Vorgehen:

- ❑ Lege einen Heap der Länge  $n$  an.
- ❑ Füge nacheinander alle  $n$  Werte in den Heap ein.  
Das Herstellen der Heap-Bedingung erfordert für jeden einzelnen Wert höchstens  $\log_2(n)$  Schritte, erfolgt also in der Laufzeit  $O(\log_2(n))$ .  
Für alle  $n$  Werte werden daher höchstens  $n \cdot \log_2(n)$  Schritte, Laufzeit ist  $O(n \cdot \log_2(n))$ .
- ❑ Entferne nacheinander jeweils den Knoten mit dem größten Wert (= die Wurzel) aus dem Heap.  
Das Herstellen der Heap-Bedingung erfordert für jedes Entfernen höchstens  $\log_2(n)$  Schritte, also insgesamt für alle  $n$  Werte höchstens  $n \cdot \log_2(n)$  Schritte.  
Auch das «Abbauen» des Heaps erfolgt wieder mit Laufzeit  $O(n \cdot \log_2(n))$ .

## Erweiterung: Sortieren mit einem Heap

(Fortsetzung)

Gegeben ist eine Folge von  $n$  Werten, die sortiert werden sollen.

### Vorgehen:

- Lege einen Heap der Länge  $n$  an.
- Füge nacheinander alle  $n$  Werte in den Heap ein.  
Das Herstellen der Heap-Bedingung erfordert für jeden einzelnen Wert höchstens  $\log_2(n)$  Schritte, erfolgt also in der Laufzeit  $O(\log_2(n))$ .  
Für alle  $n$  Werte werden daher höchstens  $n \cdot \log_2(n)$  Schritte, Laufzeit ist  $O(n \cdot \log_2(n))$ .
- Entferne nacheinander jeweils den Knoten mit dem größten Wert (= die Wurzel) aus dem Heap.  
Das Herstellen der Heap-Bedingung erfordert für jedes Entfernen höchstens  $\log_2(n)$  Schritte, also insgesamt für alle  $n$  Werte höchstens  $n \cdot \log_2(n)$  Schritte.  
Auch das «Abbauen» des Heaps erfolgt wieder mit Laufzeit  $O(n \cdot \log_2(n))$ .
- Diese Art des Sortierens besitzt also eine Laufzeit von  $O(2 \cdot n \cdot \log_2(n)) = O(n \cdot \log_2(n))$ , die auch im schlechtesten Fall garantiert werden kann.

## Erweiterung: Sortieren mit einem Heap

(Fortsetzung)

Gegeben ist eine Folge von  $n$  Werten, die sortiert werden sollen.

### Vorgehen:

- ❑ Lege einen Heap der Länge  $n$  an.
- ❑ Füge nacheinander alle  $n$  Werte in den Heap ein.  
Das Herstellen der Heap-Bedingung erfordert für jeden einzelnen Wert höchstens  $\log_2(n)$  Schritte, erfolgt also in der Laufzeit  $O(\log_2(n))$ .  
Für alle  $n$  Werte werden daher höchstens  $n \cdot \log_2(n)$  Schritte, Laufzeit ist  $O(n \cdot \log_2(n))$ .
- ❑ Entferne nacheinander jeweils den Knoten mit dem größten Wert (= die Wurzel) aus dem Heap.  
Das Herstellen der Heap-Bedingung erfordert für jedes Entfernen höchstens  $\log_2(n)$  Schritte, also insgesamt für alle  $n$  Werte höchstens  $n \cdot \log_2(n)$  Schritte.  
Auch das «Abbauen» des Heaps erfolgt wieder mit Laufzeit  $O(n \cdot \log_2(n))$ .
- ❑ Diese Art des Sortierens besitzt also eine Laufzeit von  $O(2 \cdot n \cdot \log_2(n)) = O(n \cdot \log_2(n))$ , die auch im schlechtesten Fall garantiert werden kann.

Dieses Vorgehen heißt *HeapSort*.

## Erweiterung: Sortieren mit einem Heap

(Fortsetzung)

```
public static <T extends Comparable<? super T>> T[] heapSort( T[] elements )
{
    PriorityQueue<T> pq = new PriorityQueue<>( elements.length );
    for ( T content : elements )
    {
        pq.add( content );
    }
    for ( int i = elements.length-1; i >= 0; i-- )
    {
        elements[i] = pq.poll();
    }
    return elements;
}
```

Die Methode erhält ein Feld von T-Objekten übergeben und

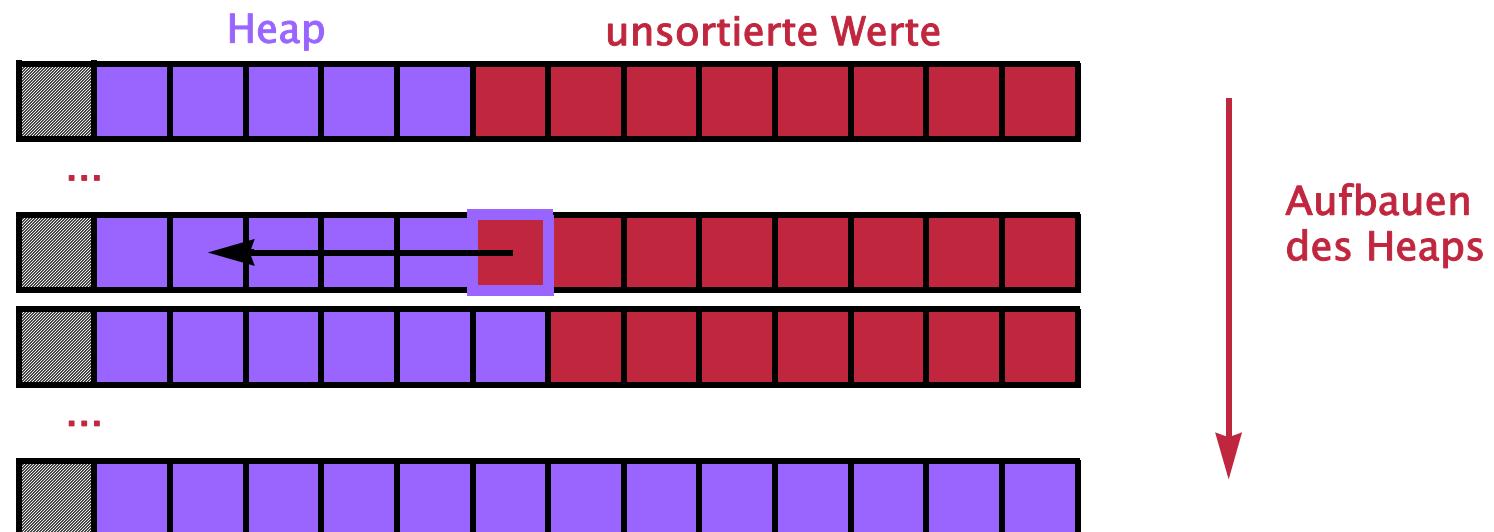
- erzeugt einen Heap,
- fügt alle übergebenen Objekte ein,
- liest nacheinander das jeweils größte Element aus und löscht es aus dem Heap,
- speichert den gelesenen Wert in einem Feld ab.

## Erweiterung: Sortieren mit einem Heap

(Fortsetzung)

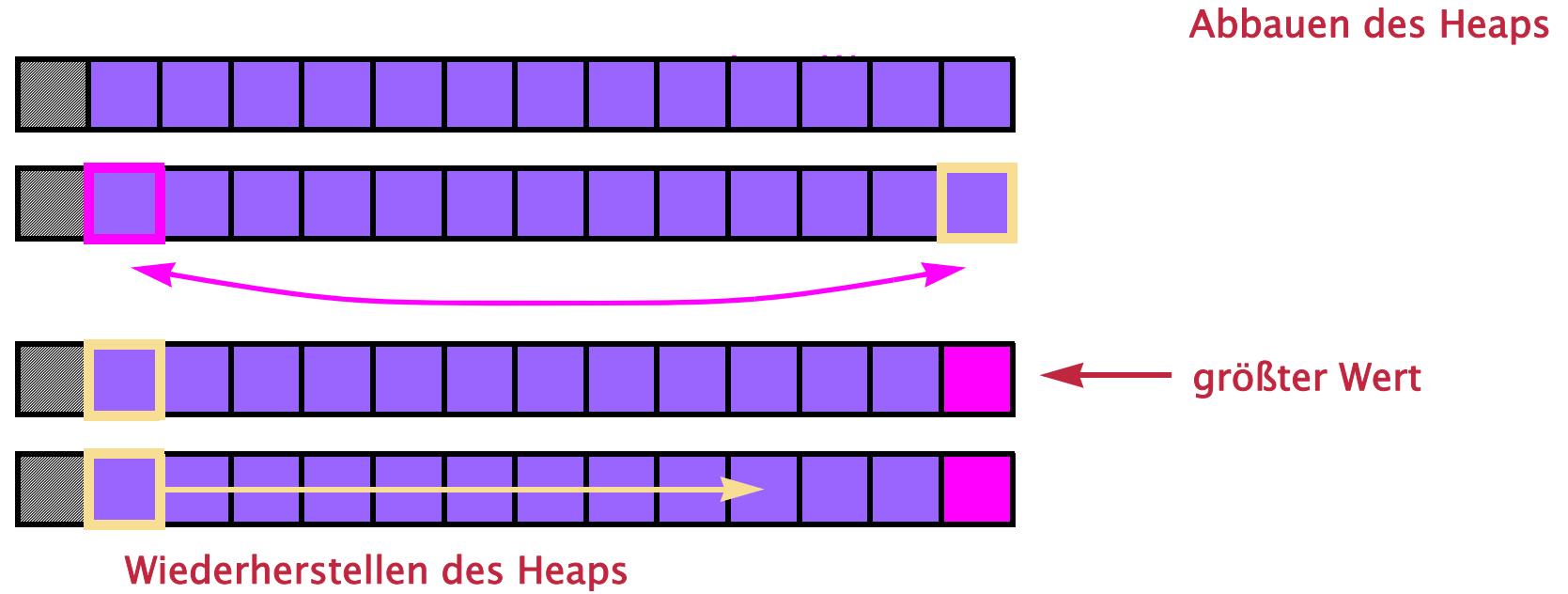
Anmerkung:

*HeapSort* kann auch als *in situ-Algorithmus* implementiert werden, wenn zunächst die Ausgangswerte und der Heap und anschließend der Heap und die sortierte Folge im gleichen Feld abgelegt werden und sich nur die *Grenze* zwischen den beiden Bereichen immer verschiebt.



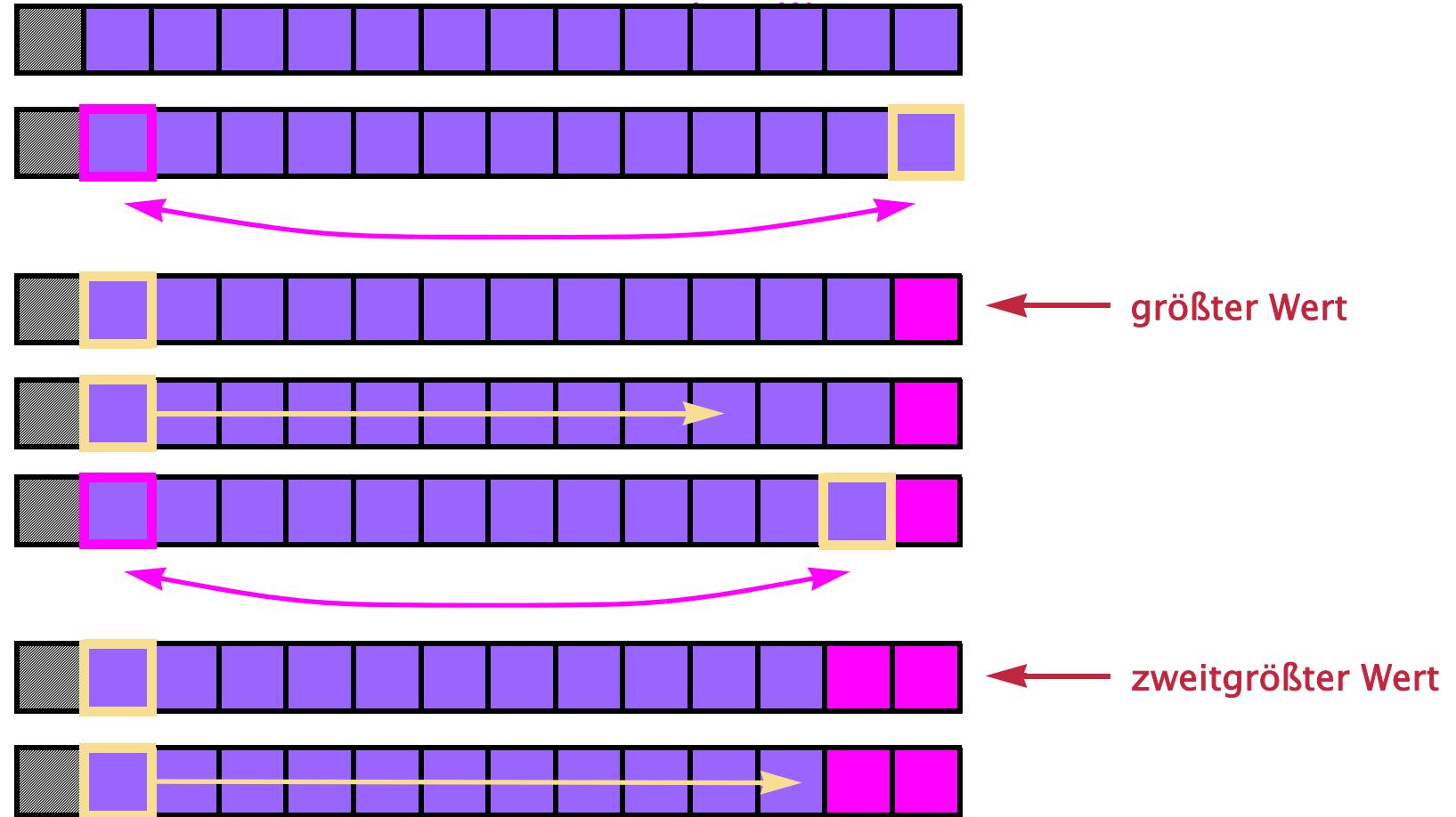
## Erweiterung: Sortieren mit einem Heap

(Fortsetzung)



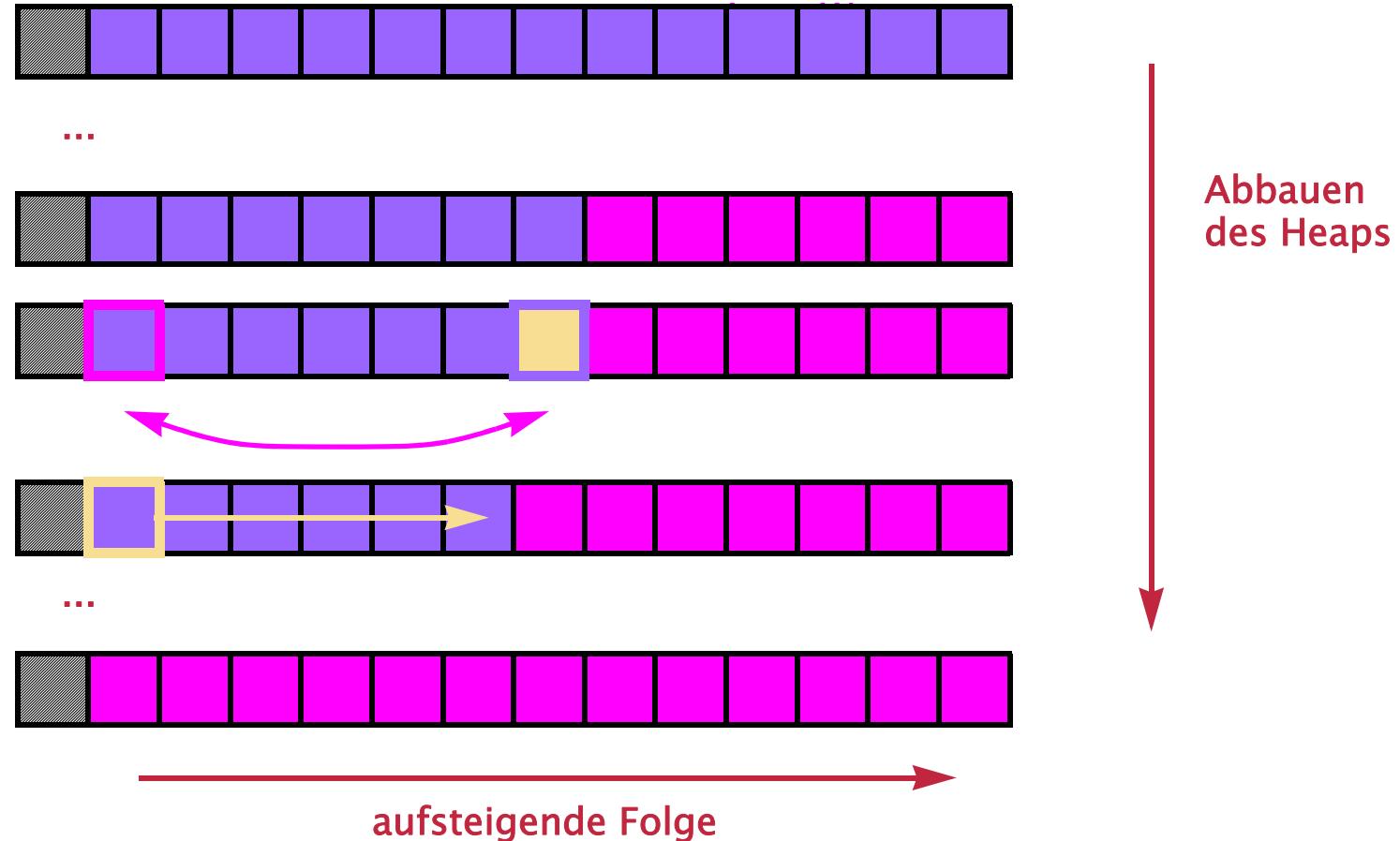
## Erweiterung: Sortieren mit einem Heap

(Fortsetzung)



## Erweiterung: Sortieren mit einem Heap

(Fortsetzung)



## Lernziele des Moduls DAP 1

(siehe Folie 2)

Nach erfolgreichem Abschluss des Moduls DAP 1 sollen die teilnehmenden Studierenden

- Programme in der Programmiersprache Java implementieren können,
- Java-Klassen und -Bibliotheken nutzen können,
- Java-Klassen und -Bibliotheken implementieren können,
- einige wichtige Algorithmen mit ihren Implementierungen kennen,
- einige wichtige Datenstrukturen mit ihren Implementierungen kennen,
- eigene Algorithmen konzipieren und implementieren können,
- eigene Datenstrukturen konzipieren und implementieren können,
- Algorithmen und Datenstrukturen bewerten können,
- objektorientierte Mechanismen anwenden können,
- einige Entwurfsmuster einsetzen können,
- einfache Ideen der Softwaretechnik einsetzen können.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

## Anhang – weitere Anweisungen, Operatoren, Konzepte

Dr. Stefan Dissmann  
Fakultät für Informatik



F-2020-1192

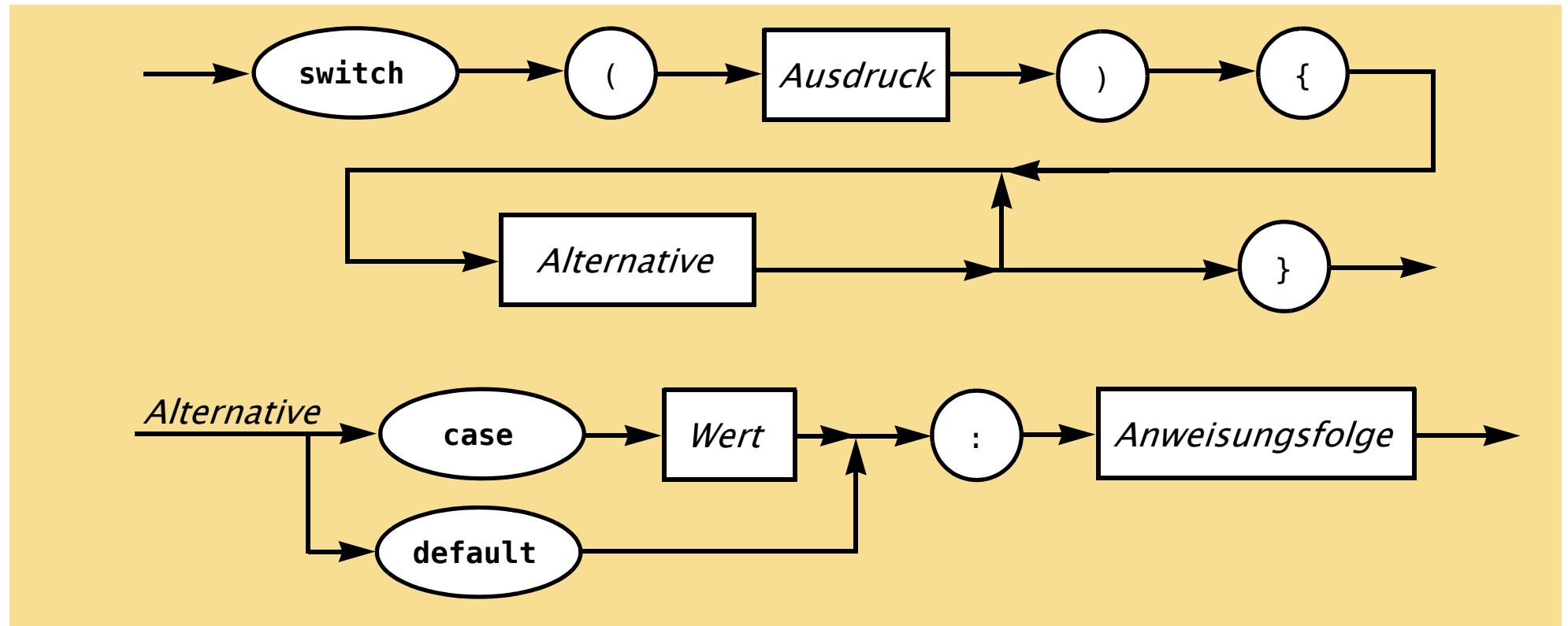
## Vorbemerkungen

Dieser Anhang stellt einige Java-Konzepte vor, die

- ❑ im Rahmen der bisher vorgestellten Beispiele nicht benötigt wurden,
- ❑ im Rahmen der bisher vorgestellten Beispiele *bewusst* nicht verwendet wurden,  
da ihr Einsatz leicht zu einem
  - unübersichtlichen oder
  - schwer verständlichenProgrammierstil führen kann.

## switch-Anweisung

- ❑ Die **switch**-Anweisung ermöglicht das Unterscheiden und kombinieren verschiedener Alternativen.
- ❑ Syntax:



## switch-Anweisung

(Fortsetzung)

### □ Funktionsweise:

Der Ausdruck in (...) wird ausgewertet zu einem Wert  $w$  und es wird mit der Alternative fortgefahrene, bei der der Wert  $w$  angegeben ist.

- Werte können aus den Typen **int** (bzw. **char**, **byte**, **short**) und **String** stammen.
- Eine Anweisungsfolge kann auch leer sein.
- Taucht der Wert  $w$  bei keiner Alternative auf, so wird die Anweisungsfolge hinter **default** angesprungen; fehlt **default**, ist wird die **switch**-Anweisung beendet.
- Es werden **ab**  $w$  **alle** Anweisungen bis zum Ende des Blocks der **switch**-Anweisung ausgeführt, sofern dieser nicht zwangsweise mit einer **break**-Anweisung verlassen wird.

### □ Beispiel:

```
switch (i) {  
    case 0: System.out.println("null"); break;  
    case 1: System.out.println("eins"); break;  
    default: System.out.println("viele");  
}
```

### □ *Risiko:*

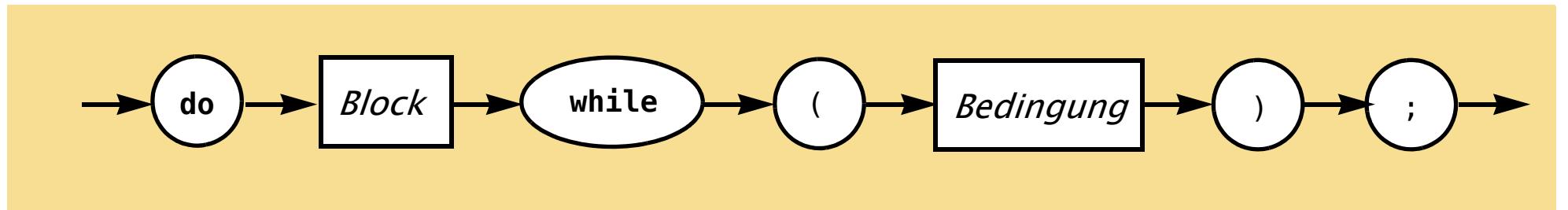
Folgen einem **case** viele Anweisungen und wird der Ablauf nicht konsequent mit **break**-Anweisungen gesteuert, können sehr unübersichtliche Abläufe entstehen.

### □ Vermeidungsstrategien:

- Benutzung von **if-** bzw. **if-else**-Anweisungen
- Polymorphie

## do-while-Schleife

- ❑ Die **do-while**-Schleife ermöglicht Schleifen, deren Rumpf mindestens einmal durchlaufen werden muss, da erst anschließend die Bedingung geprüft wird.
- ❑ Syntax:



- ❑ Funktionsweise:  
Der Block wird ausgeführt, anschließend wird die Bedingung ausgewertet:
  - Ergibt die Bedingung **true**, wird die Ausführung des Blocks wiederholt.
  - Ergibt die Bedingung **false**, ist die Schleife beendet.
  - Nach der ersten Ausführung des Blocks unterscheidet sich diese Schleife in der Ausführung nicht von der bekannten **while**-Schleife.

## do-while-Schleife

(Fortsetzung)

□ Beispiel:

```
do {  
    account = requestAccountFromUser();  
    password = requestPasswordFromUser();  
}  
while ( !loginOk( account, password ) );
```

□ *Risiko*:

Bei einem langen Schleifenrumpf kann erst weit unten im Programmtext das Abbruchkriterium gefunden werden.

□ Vermeidungsstrategie:

Der Rumpf der Schleife wird im Programmtext einmal vor einer **while**-Schleife aufgeführt oder die Bedingung einer **while**-Schleife wird so angepasst, dass diese mindestens einmal betreten wird.

```
boolean ok = false;  
while ( !ok )  
{  
    account = requestAccountFromUser();  
    password = requestPasswordFromUser();  
    ok = loginOk( account, password );  
}
```

## Kontrolle von Schleifen mit `continue`

- ❑ Die `continue`-Anweisung ermöglicht das Abbrechen der Ausführung *eines* Durchlaufs durch eine Schleife.
- ❑ Beispiel:

```
while ( true )
{
    account = requestAccountFromUser();
    if ( accountIsDisabled( account ) )
    {
        continue;
    }
    password = requestPasswordFromUser();
    if ( loginOk( account, password ) )
    {
        break;
    }
}
```

- ❑ Funktionsweise `continue`:
  - Die Ausführung der `continue`-Anweisung unterbricht die Ausführung der (innersten) umgebenden Schleife.
  - Die Programmausführung wird mit dem Prüfen der Bedingung der Schleife fortgesetzt.

## Kontrolle von Schleifen mit `break` und `continue`

### markierte Schleifen:

Um Sprünge aus geschachtelten Schleifen heraus zu ermöglichen, können Schleifen mit Marken versehen werden, die hinter der `break`-/`continue`-Anweisung angegeben werden können. Diese bezieht sich dann auf die entsprechend markierte Schleife.

The diagram shows a snippet of Java code with annotations. A red arrow points from the word 'Marke' to the label 'm:' which is placed above the first brace of the inner loop. Another red arrow points from the word 'break' in the inner loop's body to the text 'break unterbricht die äußere Schleife' located to the right of the code. The code itself is:

```
int i = 0, j = 0;
m: for ( i = 0; i < arr.length; i++ ) {
    for ( j = i+1; j < arr.length; j++ ) {
        if ( arr[i] == arr[j] ) {
            break m;
        }
    }
}
```

**break unterbricht die äußere Schleife**

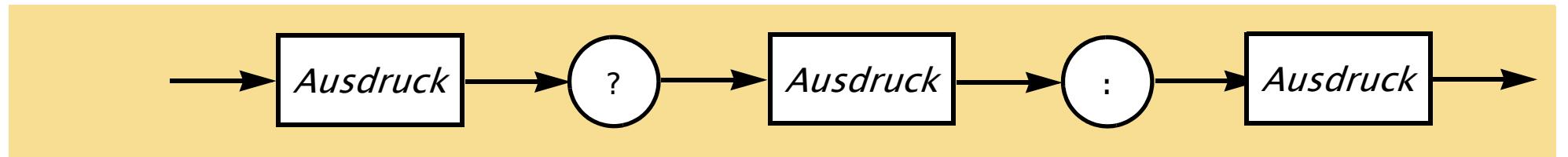
### *Risiko:*

- Struktur der Einrückungen spiegelt nicht mehr den Ablauf des Programms wieder.
- Mehrere miteinander kombinierte markierte Schleifen können unübersichtliche Ausführungsfolgen bewirken.
- Wirkung von `break` und `continue` wird bei Änderungen leicht übersehen.

**Diese Konstruktion sollte daher besser nie eingesetzt werden!**

## Bedingungsoperator ?:

- Der Bedingungsoperator ist der einzige *ternäre* Operator in Java, also ein Operator mit drei Operanden:
  - der erste Operand muss den Typ **boolean** besitzen,
  - die folgenden beiden Operanden müssen Typen besitzen, die beide kompatibel zur weiteren Verarbeitung des Ergebnisses sind,
  - die ersten beiden Operanden werden durch ? getrennt, die hinteren beiden durch :,
- Syntax:



- Funktionsweise:

Falls der erste Operand den Wert **true** ergibt,  
liefert der Bedingungsoperator den Wert des zweiten Operanden,  
falls der erste Operand den Wert **false** ergibt,  
liefert der Bedingungsoperator den Wert des dritten Operanden.

## Bedingungsoperator ?:

(Fortsetzung)

- Beispiele:

```
int abs( int i ) {  
    return i<0 ? -i : i;  
}  
  
int minimum( int i1, int i2 ) {  
    return i1<i2 ? i1 : i2;  
}  
  
boolean exclusiveOr( boolean b1, boolean b2 ) {  
    return b1 ? !b2 : b2;  
}
```

- *Risiko:*

Der Bedingungsoperator führt gerade in komplexen Ausdrücken schnell zu unübersichtlichen Auswertungsreihenfolgen.

- Vermeidungsstrategie:

Es kann eine **if-else**-Anweisung benutzt werden.

- *aber*: sehr praktisch in Lambda-Ausdrücken

## weitere Operatoren

- Typ **boolean**

*Entweder-oder-Operator (**XOR**):*

Es gilt:  $a \wedge b == (a \& !b) | (!a \& b)$

- bitweise wirkende Operatoren, *die als Ergebnis eine Zahl liefern!*

Verschiebeoperatoren:

- $i << n$  verschiebt die Bits von  $i$   $n$ -mal nach links und füllt rechts mit  $0$  auf.
- $i >> n$  verschiebt die Bits von  $i$   $n$ -mal nach rechts und füllt links mit dem Vorzeichen-Bit auf.
- $i >>> n$  verschiebt die Bits von  $i$   $n$ -mal nach rechts und füllt links mit  $0$  auf.

- Bitoperatoren:

Deklariert sind: **int i, j; ...**

- $\sim i$  erzeugt das *bitweise Komplement* für den Wert von  $i$ .
- $i | j$  erzeugt eine *bitweise ODER*-Verknüpfung der Werte von  $i$  und  $j$ .
- $i \& j$  erzeugt eine *bitweise UND*-Verknüpfung der Werte von  $i$  und  $j$ .
- $i ^ j$  erzeugt eine *bitweise XOR*-Verknüpfung der Werte von  $i$  und  $j$ .

**Konsequenz:** Die Symbole der logischen Operatoren lassen sich syntaktisch korrekt auf Zahlen anwenden!

## Operator `instanceof`

- Der binäre Operator `instanceof` vergleicht ein Objekt `o` mit einer Klasse `C`:  
`o instanceof C` gibt `true` zurück, falls `o` ein Objekt der Klasse `C` ist oder `C` eine Oberklasse der Klasse von `o` ist.
- Beispiele:
  - o `instanceof Object` ist immer wahr.

Überschreibungen der Methode `equals` sollten zuerst immer die Typkompatibilität der verglichenen Objekte prüfen:

```
class C
{
    public boolean equals( Object o )
    {
        if ( o instanceof C ) {
            ...
        } else {
            return false;
        }
        ...
    }
}
```

## Operator `instanceof`

(Fortsetzung)

- ❑ Der Operator `instanceof` sollte *ausschließlich dann* benutzt werden, wenn eine Ermittlung des Typs eines Objekt unbedingt notwendig ist (siehe Beispiel `equals`).
- ❑ Der Operator `instanceof` sollte *nicht* benutzt werden, um klassenspezifische, unterschiedliche Abläufe in einer Methode innerhalb einer Vererbungshierarchie zu steuern.  
Diese Aufgabe kann übersichtlicher durch das geeignete Überschreiben von Methoden in den Unterklassen erfolgen.

*Fallunterscheidungen der folgenden Form sind in der Regel fehleranfällig und ein Zeichen mangelnden Verständnisses objektorientierter Denkweisen:*

```
if ( o instanceof C1 ) { ... }
if ( o instanceof C2 ) { ... }
if ( o instanceof C3 ) { ... }
if ( o instanceof C4 ) { ... }
```

# Operatorprioritäten

Die Priorität eines Operators bestimmt den Zeitpunkt seiner Auswertung in einem Ausdruck.

## Priorität Operatoren

1	<code>+, -, ++, --, !, ~, (type)</code>	<i>alle unären Operatoren</i>
2	<code>*, /, %</code>	<i>«Punktrechnung»</i>
3	<code>+, -</code>	<i>«Strichrechnung»</i>
4	<code>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;</code>	<i>Verschiebungsoperatoren</i>
5	<code>&gt;, &gt;=, &lt;, &lt;=, instanceof</code>	<i>Vergleiche</i>
6	<code>==, !=</code>	<i>(Un-)Gleichheit</i>
7	<code>&amp;</code>	<i>UND</i>
8	<code>^</code>	<i>XOR</i>
9	<code> </code>	<i>ODER</i>
10	<code>&amp;&amp;</code>	<i>UND (shortcut)</i>
11	<code>  </code>	<i>ODER (shortcut)</i>
12	<code>?:</code>	<i>Bedingungsoperator</i>
13	<code>=</code>	<i>Zuweisung</i>
14	<code>+=, -=, *=, /=, %=, &amp;=,  =, ^=</code>	<i>verkürzende Zuweisungen</i>

## Zugriffsrechte

Folgende Zugriffsrechte können durch Modifizierer angegeben werden:

- public**  
= in allen Klassen in allen Paketen sichtbar.
  - protected**  
= nur im *eigenen Paket* und in allen erbenden Klassen sichtbar
  - (keine Angabe entspricht dem Zugriffsrecht: *package-private*)  
= nur im eigenen Paket sichtbar
  - private**  
= nur in der eigenen Klasse sichtbar
- 
- Anmerkung:  
Solange Klassen im gleichen Paket liegen,  
zeigt nur **private** eine einschränkende Wirkung.

## Felder

- ❑ Feldvariable und Feldattribute sind *besondere* Referenzen.
- ❑ Zulässig ist daher:

```
int[] arr = null;
```

- ❑ Feldattribute werden immer mit `null` initialisiert.
- ❑ Als Argumente übergebene Felder sollten daher vor ihrer Nutzung auch überprüft werden, ob sie auf ein Feldobjekt verweisen, also ungleich `null` sind.
- ❑ Da die Größe eines Feldobjekts nach seinem Erzeugen mit `new` festgelegt ist und anschließend nicht mehr geändert werden kann, wird diese Länge durch eine öffentliche Konstante bereitgestellt:

```
public final int length
```

## Local-Variable Type Inference `var`

- ❑ Bei der Deklaration von Variablen kann unter bestimmten Voraussetzungen auf die explizite Angabe eines Typs verzichtet werden. Der Compiler bestimmt dann den Typ implizit über *Type Inference*.
- ❑ Wesentliche Voraussetzung für die Nutzung des *Type Inference*-Mechanismus ist eine Deklaration mit einer Initialisierung, aus der sich eindeutig ein Typ bestimmen lässt.
- ❑ Der Compiler ordnet den so bestimmten Typ der Variablen zu. Der so zugeordnete Typ ist dann für die weitere Verwendung der Variablen im Programm fest vorgegeben.
- ❑ Die Nutzung von *Type Inference* wird in der Deklaration durch die Angabe `var` statt einer Typangabe angezeigt.
- ❑ Beispiele:

```
var quantity = 0;  
var average = 0.0;  
var lecture = "dap1";  
var set = new HashSet<Integer>();  
var it = set.iterator();
```

quantity wird der Typ **int** zugeordnet  
avarage wird der Typ **double** zugeordnet  
lecture wird der Typ **String** zugeordnet  
set wird **HashSet<Integer>** zugeordnet  
it wird **Iterator<Integer>** zugeordnet

- ❑ Beispiel für eine nicht eindeutige Initialisierung:

```
var ref = null;
```

## Modifizierer **static**

- ❑ Statische Elemente einer Klasse sind einmal in der Klasse vorhanden und werden nicht für jedes Objekt einzeln erzeugt.
- ❑ Statische Elemente einer Klasse können daher *ohne das Erzeugen eines Objekts* genutzt werden.

Daher beginnt auch die Ausführung eines Java-Programms immer mit einer statischen Methode:

```
public static void main(String[] args)
```

- ❑ Statische Methoden können eingesetzt werden, wenn diese einen funktionalen Charakter besitzen und aus ihren Parametern ein Ergebnis ableiten:  
`int calculateGcd( int v1, int v2 )` (siehe Folie 233)
- ❑ Statische innere Klassen können dann eingesetzt werden, wenn die innere Klasse keinen Bezug zu einem einzelnen Objekt der umgebenden Klasse hat.
- ❑ Statische Attribute sollten vorsichtig eingesetzt werden, insbesondere dann, wenn von einer Klasse auch Objekte erzeugt werden. Ein statisches Attribut kann dann von allen Objekten der Klasse angesprochen und – falls Methoden bereitstehen – geändert werden.

## Variable Parameteranzahl - *Vararg*

- Methoden mit nur *einem* deklarierten Parameter, aber mit der Möglichkeit, eine *variable* Anzahl von *Argumenten* übergeben zu können, können folgendermaßen angelegt werden:

```
public static int maximum( int... values )
{
    int max = 0;
    if ( values.length > 0 )
    {
        max = values[0];
        for ( int candidate : values )
        {
            if ( candidate > max )
            {
                max = candidate;
            }
        }
    }
    return max;
}
```

Auslassungssymbol ...  
Es sind beliebig viele Parameter  
des Typs `int` erlaubt.

intern abgelegt in einem Feld  
von `int`-Werten

## Variable Parameteranzahl - *Vararg*

(Fortsetzung)

erlaubte Aufrufe für `maximum( int... values )`:

```
int m = maximum();  
int m = maximum(17);  
int m = maximum(2,17,6,1,9,8);  
int[] arr = {12,5,6,19}; int m = maximum( arr );
```

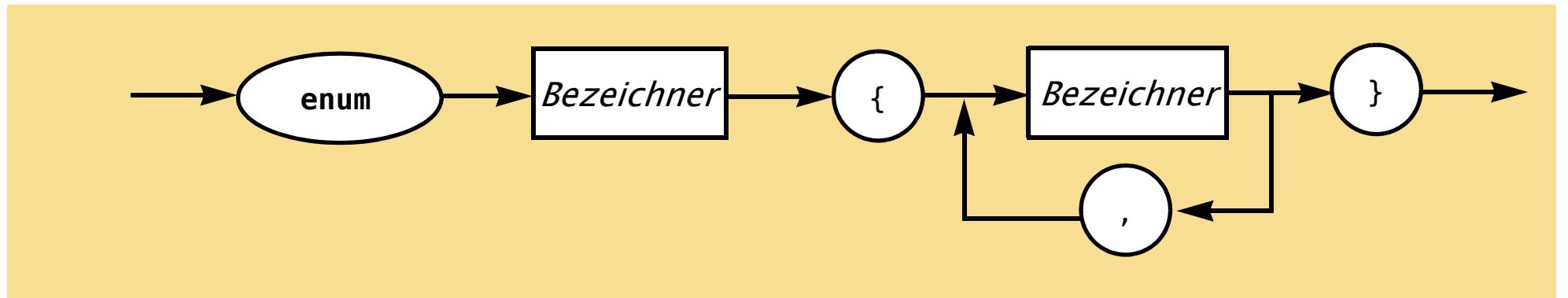
*kein Argument*  
*ein Argument*  
*viele Argumente*  
*ein Feld*

Hinweise:

- ❑ Eine Deklaration *verbraucht* also verschiedene Signaturen.  
Methoden mit diesen Signaturen können nicht noch einmal explizit deklariert werden.
- ❑ In der Parameterliste der Deklaration darf nur höchstens ein *Vararg*-Parameter als letzter Parameter auftreten.
- ❑ Alle möglicherweise vor dem *Vararg*-Parameter stehenden Parameter müssen verpflichtend angegeben werden. `int f( int p, int... values )` erfordert also beim Aufruf immer die Angabe mindestens eines `int`-Wertes, der erste übergeben Wert steht immer in `p` zur Verfügung.
- ❑ *Vararg*-Parameter sehen auf den ersten Blick hilfreich aus, allerdings muss an der Programmposition des Aufrufs die Anzahl der Argumente festgelegt werden (können).

## Aufzählungstyp `enum`

- ❑ Aufzählungen sind Mengen von selbst definierten Konstanten, auf denen Vergleiche möglich sind.
- ❑ Syntax:



- ❑ Beispiel:

```
enum Term
{
    WINTER, SUMMER
}
```

## Aufzählungstyp enum

(Fortsetzung)

- Umsetzung durch Compiler in folgende Klasse:

```
public class Term extends Enum
{
    public static final Term WINTER = new Term( "WINTER", 0 );
    public static final Term SUMMER = new Term( "SUMMER", 1 );

    private Term( String s, int i )
    {
        super( s, i );
    }
    ...
}
```

The code snippet shows the internal representation of an enum. Red annotations explain the behavior of the compiler's implementation:

- A red arrow points from the word "erbt" to the inheritance line "extends Enum".
- A red arrow points from the word "sorgt für Ordnung mit compareTo" to the constructor "Term(String s, int i)".
- A blue arrow points from the word "sorgt für toString()" to the call to "super(s, i);".

- Nutzung:

```
Term actualTerm = Term.WINTER;
```

- Vorteile:

- Der Aufzählungstyp ist abgeschlossen und typsicher.
- Der Aufzählungstyp darf in **switch**-Anweisungen als **case**-Angabe genutzt werden.
- Aufzählungstypen können auch Methoden enthalten und Schnittstellen implementieren.

## Reflexion/Introspektion

Ein Java-Programm kann sich selbst betrachten:

- ❑ Klasse `java.lang.Class<T>`, u.a. mit den Methoden
  - `public String getName()`
  - `public Class<? super T> getSuperclass()`
  - `public Class<?>[] getInterfaces()`
  - `public Method[] getMethods()`
  - `public Method[] getDeclaredMethods()`
  - `public Field[] getFields()`
  - `public Field[] getDeclaredFields()`
  - `public boolean isAnonymousClass()`
  - `...`
- ❑ Klasse `Object`, auch mit der Methode
  - `public final Class<?> getClass()`
- ❑ Beispiel:

```
ints.getClass().getName().equals( "DoublyLinkedList" )
```

  
`...`

## Garbage Collector

- ❑ Das explizite Erzeugen eines Objekts mit `new` reserviert genügend Speicherplatz für die Verwaltung der Informationen dieses Objekts durch das Laufzeitsystem.
- ❑ Die vordefinierte maximale Größe aller reservierte Objekte beträgt für aktuelle Java-Versionen 1/4 des verfügbaren Hauptspeichers bis höchstens 1 GigaByte.
- ❑ Java kennt *keine explizite Freigabe* von nicht mehr benötigtem Speicherplatz.
- ❑ Die Freigabe von nicht mehr benötigtem Speicherplatz erfolgt *implizit* durch den *Garbage Collector (GC)*.
- ❑ Der GC überprüft dabei, ob aus dem laufenden Programm noch ein Zugriff auf ein Objekt möglich ist. Ist das nicht der Fall, wird der Speicherbereich des Objekts freigegeben. Der GC arbeitet auch anforderungsgetrieben, wird also insbesondere dann aktiv, wenn für die Erzeugung eines Objekts nicht mehr genügend Speicherplatz verfügbar ist.
- ❑ Konsequenzen:
  - Der GC kann für ein Objekt feststellen, ob im laufenden Programm eine Referenz existiert, von der aus – eventuell über weitere Referenzen – dieses Objekt erreicht werden kann.
  - Der GC kann bei dynamischen Abläufen nicht feststellen und vorhersagen, ob die noch erreichbaren Objekte später wirklich benötigt werden.
  - Der Entwickler hat also die Aufgabe, nicht mehr benötigte Objekte auch zu nie wieder erreichbaren Objekten zu machen, damit der GC seine Aufgabe erfüllen kann.

# Modul **Datenstrukturen, Algorithmen und Programmierung 1 (DAP 1)**

**Ende**

Dr. Stefan Dissmann  
Fakultät für Informatik



**Vielen Dank  
für Deine Teilnahme!**

F-2020-1216