LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung

Wintersemester 2018/19

Kapitel 1: Einführung und Grundbegriffe

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 23. Oktober 2018

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Übersicht: Inhalt und Ziele

- Begriffsbildungen: Informatik, Algorithmus, Programm, Programmiersprache, Compiler, ...
- Einordnung der Programmiersprache Java
- Ein erstes Java-Programm erstellen, übersetzen und ausführen
- Dokumentation und Kommentierung von (Java-)Programmen

Was ist Informatik?

- **Informatik** ist ein Kunstwort, entstanden in 1960er Jahren in Frankreich (Informatique)
- Verschmelzung von **Information** und **Mathematik**
- Im Englischen „Computer Science“, neuerdings auch „Informatics“

Informatik

Wissenschaft von der systematischen **Verarbeitung von Informationen**, insbesondere mithilfe von Computern

Was ist Informatik?

- **Informatik** ist ein Kunstwort, entstanden in 1960er Jahren in Frankreich (Informatique)
- Verschmelzung von **Information** und **Mathematik**
- Im Englischen „Computer Science“, neuerdings auch „Informatics“

Informatik

Wissenschaft von der systematischen **Verarbeitung von Informationen**, insbesondere mithilfe von Computern

Gründung der Informatik in Deutschland:

- WS 1969/70: Universität Karlsruhe bietet als erste deutsche Hochschule den Studiengang Diplom-Informatik an.
- September 1969: Gründung der „Gesellschaft für Informatik e.V.“

Teilgebiete der Informatik

Praktische Informatik

Programmierung und -sprachen,
Softwareentwicklung und -technik,
Datenbanksysteme,
Betriebssysteme,
Verteilte Systeme

Technische Informatik

Hardware,
Digitale und Analoge Schaltungen,
Rechnerarchitekur,
Rechnernetze,
Eingebettete Systeme

Theoretische Informatik

Automatentheorie, Formale Sprachen,
Berechenbarkeitstheorie,
Algorithmen und Komplexität,
Theorie der Programmiersprachen,
Logik

Angewandte Informatik

Anwenden der Informatik in anderen Wissenschaften:
Ingenieurinformatik, Medieninformatik,
Wirtschaftsinformatik, Geoinformatik,
Computerlinguistik, ...

Informatik und Gesellschaft

Auswirkungen der Informatik auf die Gesellschaft: Datensicherheit,
Allgegenwärtigkeit von Rechnern, soziale Netzwerke, ...

Information

- Begriff **Information** für die Informatik von zentraler Bedeutung
- stammt vom lateinischen „informare“ (Gestalt geben)
- Information: Wissen über Sachverhalte und Vorgänge der realen Welt

Information

- Begriff **Information** für die Informatik von zentraler Bedeutung
- stammt vom lateinischen „informare“ (Gestalt geben)
- Information: Wissen über Sachverhalte und Vorgänge der realen Welt
- Für die Informatik wichtig:

  Trennung von Repräsentation (Syntax) und Bedeutung (Semantik)

  Beispiel: Verschiedene Repräsentationen der Zahl 12:
  - 12 (Dezimalsystem mit arabischen Ziffern)
  - XII (Römische Zahlen)
  -  (Strichliste, Unärsystem)
  - 1100 (Binärsystem)

  Beachte: Ob Operationen auf Zahlen (Addition, Multiplikation, . . .) „schwierig“ sind, hängt auch von der Repräsentation ab.

Information und Computer

Computer verwenden das **Binärsystem** zur Darstellung von Information, z.B.

- Bitfolgen repräsentieren Daten
  Z.B. Texte: Jeder Buchstabe wird durch eine Bitfolge repräsentiert, die wiederum zu Folgen zusammengehängt werden und Texte repräsentieren
- Bitfolgen repräsentieren Programme
  Einzelne Prozessorbefehle sind durch Bitfolgen repräsentiert, die wiederum zu Befehlsfolgen zusammengehängt werden

Binärsystem

Dezimalsystem

- Ziffern: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Ziffern einer Zahl stellen **Koeffizienten von Zehnerpotenzen** dar

  Dezimalwert von 2198 im Dezimalsystem:
  $2 \cdot 10^3 + 1 \cdot 10^2 + 9 \cdot 10^1 + 8 \cdot 10^0 = 2000 + 100 + 90 + 8$
- Allgemein: $d_n\, d_{n-1} \ldots d_0$ repräsentiert den Dezimalwert $\sum_{i=0}^{n} d_i \cdot 10^i$

Binärsystem

Dezimalsystem

- Ziffern: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Ziffern einer Zahl stellen **Koeffizienten von Zehnerpotenzen** dar

  Dezimalwert von 2198 im Dezimalsystem:
  $2 \cdot 10^3 + 1 \cdot 10^2 + 9 \cdot 10^1 + 8 \cdot 10^0 = 2000 + 100 + 90 + 8$
- Allgemein: $d_n\, d_{n-1} \ldots d_0$ repräsentiert den Dezimalwert $\sum_{i=0}^{n} d_i \cdot 10^i$

Binärsystem

- Ziffern: $\{0, 1\}$
- Ziffern einer Zahl stellen **Koeffizienten von Zweierpotenzen** dar

  Dezimalwert von 1001110 im Binärsystem:
  $1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 64 + 8 + 4 + 2 = 78$
- Allgemein: $b_n\, b_{n-1} \ldots b_0$ repräsentiert den Dezimalwert $\sum_{i=0}^{n} b_i \cdot 2^i$

Binärsystem: Umrechnen vom und ins Dezimalsystem

binär $\rightarrow$ **dezimal**: Das haben wir eben gesehen:

$b_n\, b_{n-1} \ldots b_0$ repräsentiert den Dezimalwert $\sum_{i=0}^{n} b_i \cdot 2^i$

Binärsystem: Umrechnen vom und ins Dezimalsystem

binär $\rightarrow$ dezimal: Das haben wir eben gesehen:

$b_n\, b_{n-1} \ldots b_0$ repräsentiert den Dezimalwert $\sum_{i=0}^{n} b_i \cdot 2^i$

dezimal $\rightarrow$ binär: Sei D eine Dezimalzahl.

- Teile wiederholt durch 2 mit ganzahliger Division mit Rest, bis der ganzzahlige Anteil zu 0 geworden ist.
- Seien $b_0, b_1, \ldots, b_n$ die Reste dieser Divisionen.
- Dann ist $b_n\, b_{n-1} \ldots b_0$ die Binärdarstellung von D.

Binärsystem: Umrechnen vom und ins Dezimalsystem

binär $\rightarrow$ dezimal: Das haben wir eben gesehen:

$b_n\, b_{n-1} \ldots b_0$ repräsentiert den Dezimalwert $\sum_{i=0}^{n} b_i \cdot 2^i$

dezimal $\rightarrow$ binär: Sei D eine Dezimalzahl.

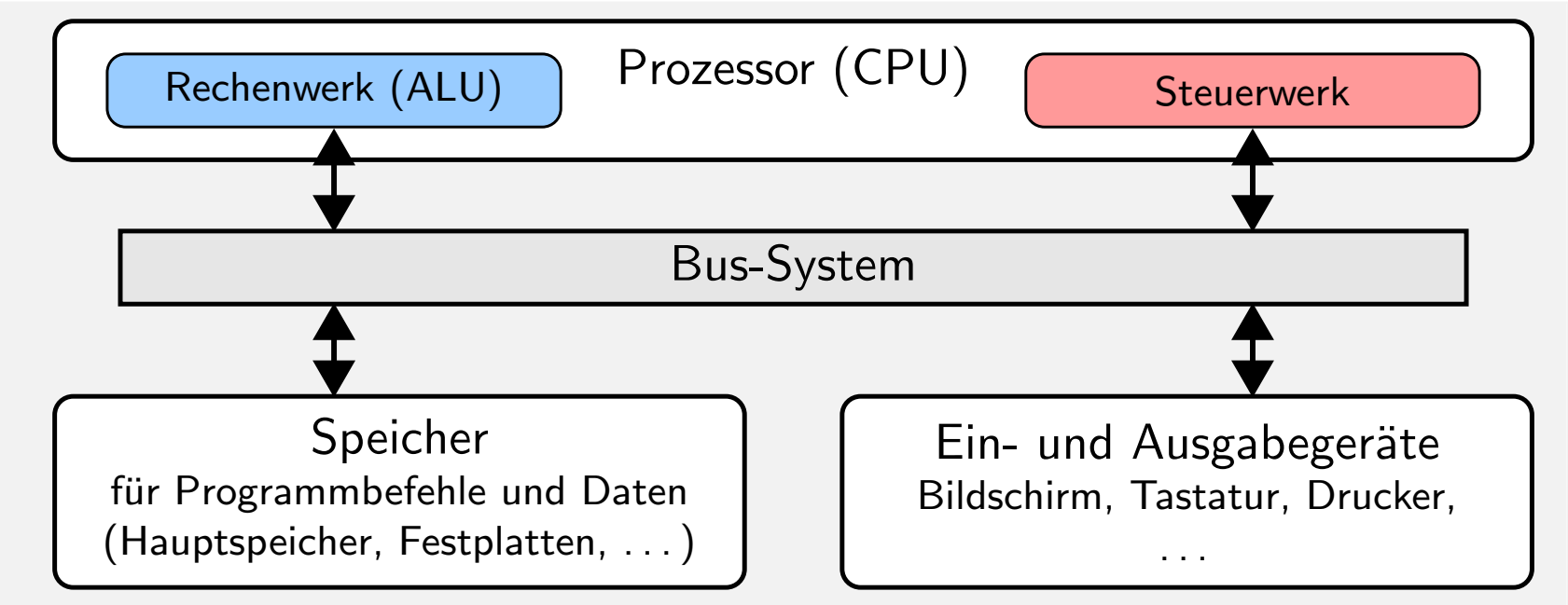
- Teile wiederholt durch 2 mit ganzahliger Division mit Rest, bis der ganzzahlige Anteil zu 0 geworden ist.
- Seien $b_0, b_1, \ldots, b_n$ die Reste dieser Divisionen.
- Dann ist $b_n\, b_{n-1} \ldots b_0$ die Binärdarstellung von D.

Beispiel: Dezimalzahl 26

$$\begin{aligned} 26 : 2 &= 13, \quad \text{Rest } 0 \\ 13 : 2 &= 6, \quad \text{Rest } 1 \\ 6 : 2 &= 3, \quad \text{Rest } 0 \\ 3 : 2 &= 1, \quad \text{Rest } 1 \\ 1 : 2 &= 0, \quad \text{Rest } 1 \end{aligned}$$

Die Binärdarstellung von 26 ist 11010

Aufbau eines Computers (Von-Neumann-Modell)



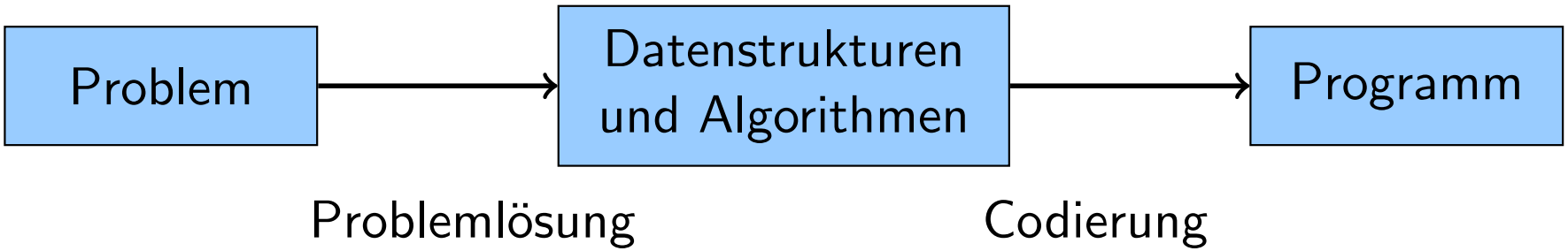
- Rechenwerk (ALU = Arithmetic Logic Unit)
führt Rechenoperationen und logische Verknüpfungen aus
- Steuerwerk führt Programmbefehle aus, verschaltet ALU-Komponenten, steuert den Programmablauf
- Bus-System zur Kommunikation zwischen den Komponenten (Steuerbus, Datenbus, Adressbus)

Programmierung und Softwareentwicklung

Programm

Beschreibung von Datenstrukturen und Algorithmen in einer „dem Computer verständlichen“ Sprache (Programmiersprache)

Programmierung



Softwareentwicklung

Systematische Konstruktion von Programmen und komplexen Softwaresystemen ($\rightarrow$ Systemarchitektur)

Zentraler Begriff: Algorithmus

Wortherkunft des Begriffs „Algorithmus“:

Al'Khwarizmi (ca. 790-840)

- Persischer Universalgelehrter, Mathematiker, Astronom und Geograph
- Author von Hisab al-jabr w'al-muqabala („Das kurz gefasste Buch über die Rechenverfahren durch Ergänzen und Ausgleichen“)
  Der Begriff „Algebra“ stammt von der lateinischen Übersetzung des Buchtitels („Ludus algebrae almucgrabalaeque“)

Zentraler Begriff: Algorithmus (2)

Was ist ein Algorithmus?

Algorithmus

Eine detaillierte, explizite und eindeutige Vorschrift zur schrittweisen Lösung eines Problems

Eigenschaften:

- Jeder Schritt ist eindeutig festgelegt und berechenbar.
- Das Verfahren liefert nach endlich vielen Schritten eine Lösung (es terminiert).

Beispiele:

- Modellbau: Montageanleitung
- Küche: Kochrezept
- Informatik: Such- und Sortieralgorithmen

Gibt es Algorithmen für jedes Problem?

Berechenbarkeit: Verschiedene äquivalente Definitionen, z.B.: Ein Problem ist berechenbar, wenn ...

- ... es durch eine Turing-Maschine berechnet werden kann (Alan Turing 1912 - 1954).
- ... es durch ein While-Programm berechnet werden kann.
- ... es durch einen Ausdruck den λ-Kalküls berechnet werden kann.
- ... es durch einen modernen Rechner mit unendlich viel Speicher berechnet werden kann.

Beachte: **Nicht** alle Probleme sind **berechenbar**!

Beispiel „**Halteproblem**“:

Bestimme, ob ein x-beliebiges Programm für eine x-beliebige Eingabe terminiert.

Beispiel für einen Algorithmus: Sortieren einer Liste

Problembeschreibung:

- Gegeben: Eine Liste von Elementen, die geordnet werden können.
- Gesucht: Liste mit denselben Elementen in aufsteigender Reihenfolge angeordnet

Z. B. soll $5, 33, 12, 13, 8, 1$ zu $1, 5, 8, 12, 13, 33$ sortiert werden

Idee (Sortieren durch Vertauschen):

- Laufe durch die Liste und vergleiche benachbarte Elemente, und vertausche sie, falls falsch geordnet
- Nach einem Durchlauf ist das größte Element ganz am Ende
- Laufe so oft durch wie es Elemente gibt.

Sortieren durch Vertauschen („Bubble Sort“)

Algorithmus

Falls die Liste leer ist, dann fertig.
Ansonsten:

> Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.
>
> Solange „`outer`“ nicht auf das erste Element zeigt:
>
> > Sei „`inner`“ ein Zeiger auf das erste Element der Liste.
> >
> > Solange „`inner` $<$ outer“:
> >
> > > Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.
> > >
> > > Rücke mit „`inner`“ eine Position vorwärts.
> >
> > Rücke mit „outer“ eine Position rückwärts.

Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.

Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

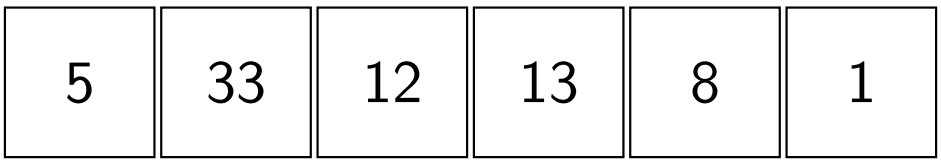
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.

Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

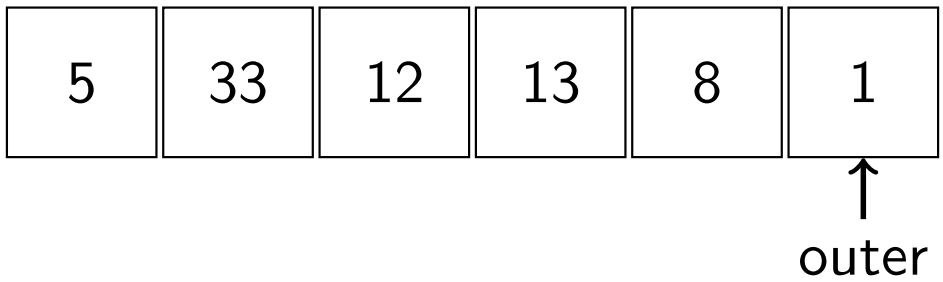
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

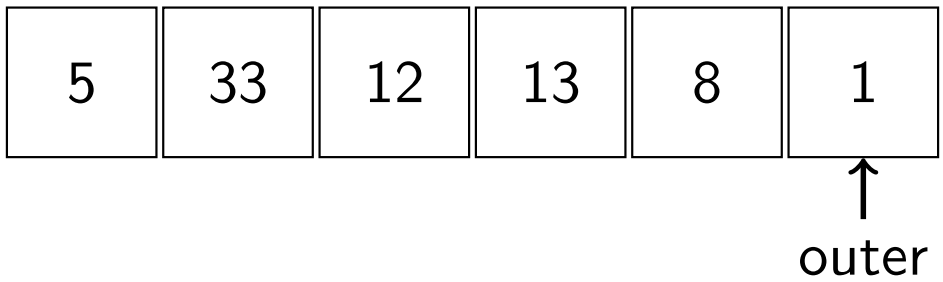
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

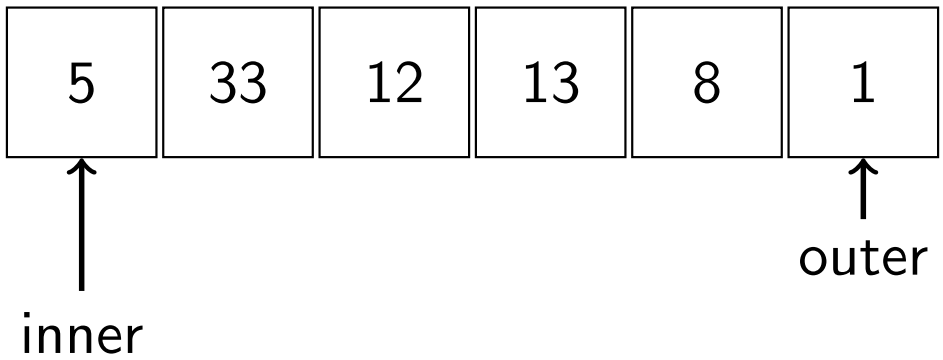
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

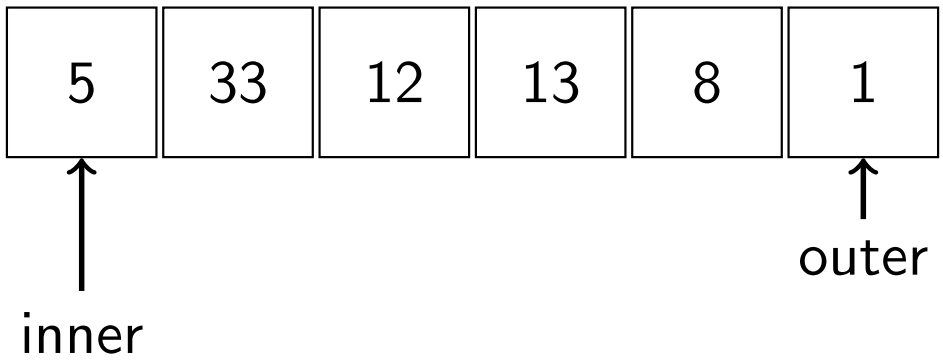
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

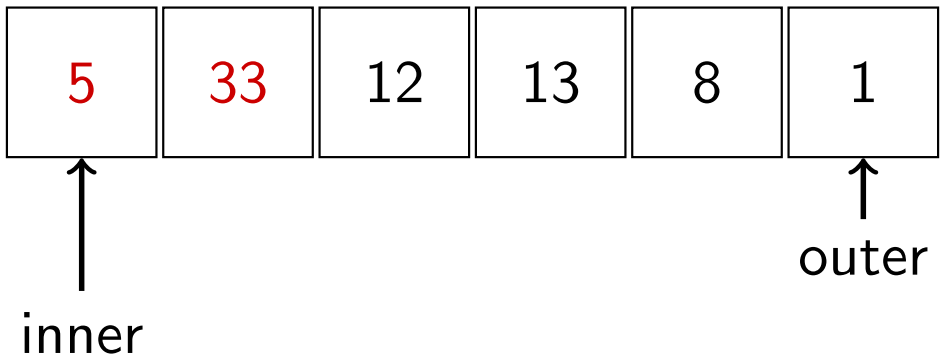
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

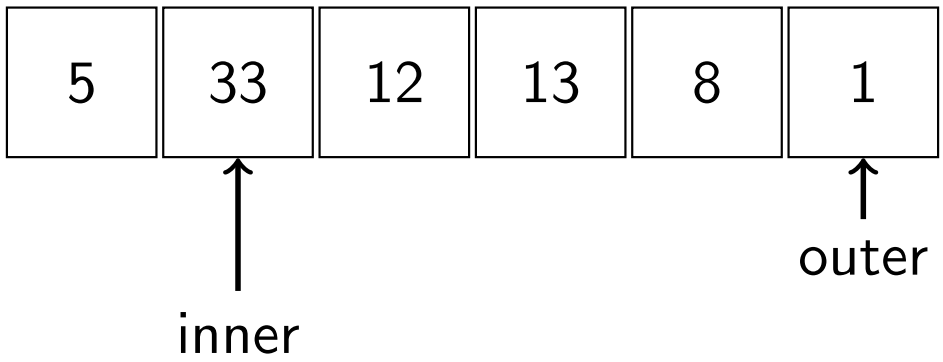
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

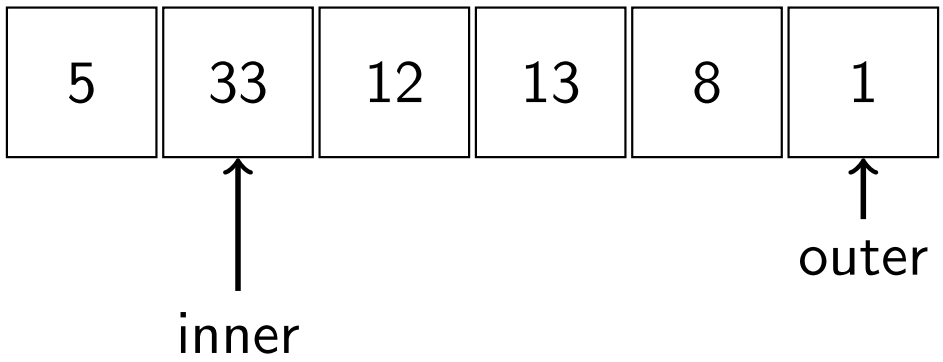
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

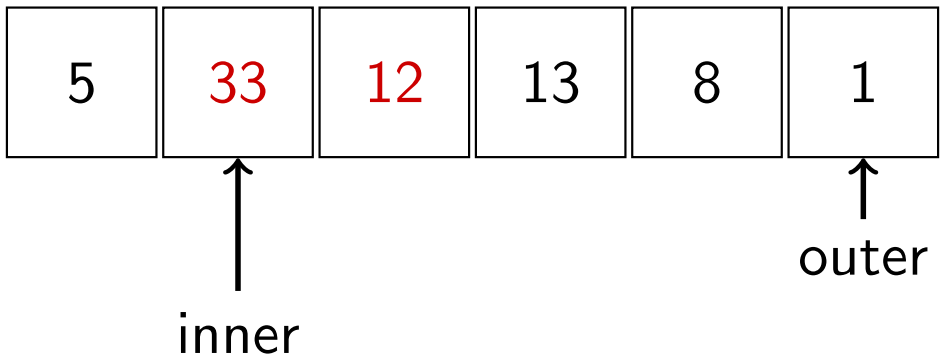
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

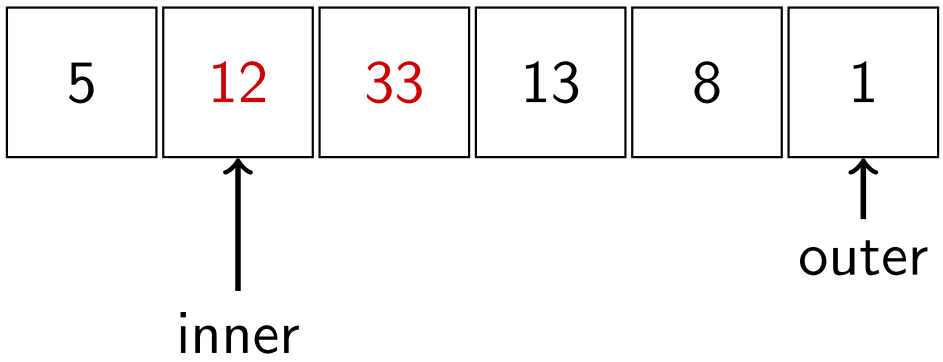
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

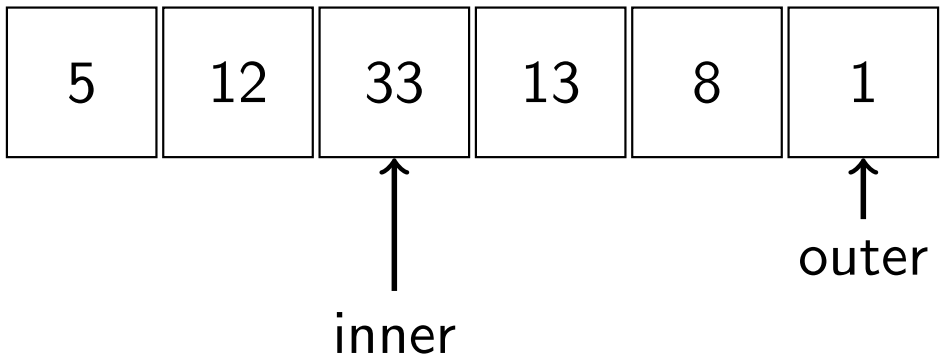
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

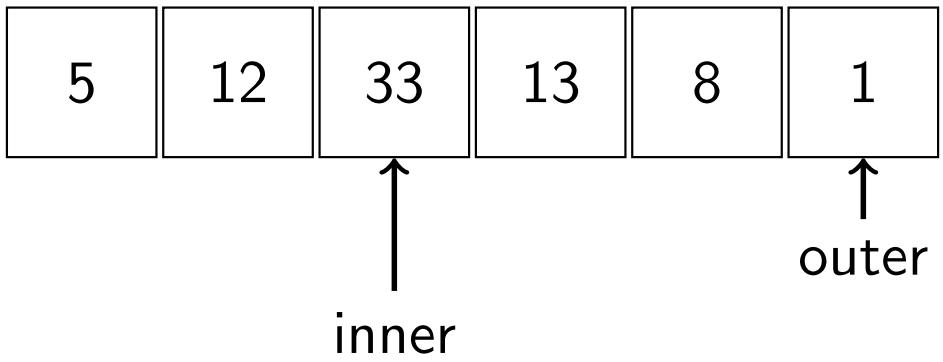
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

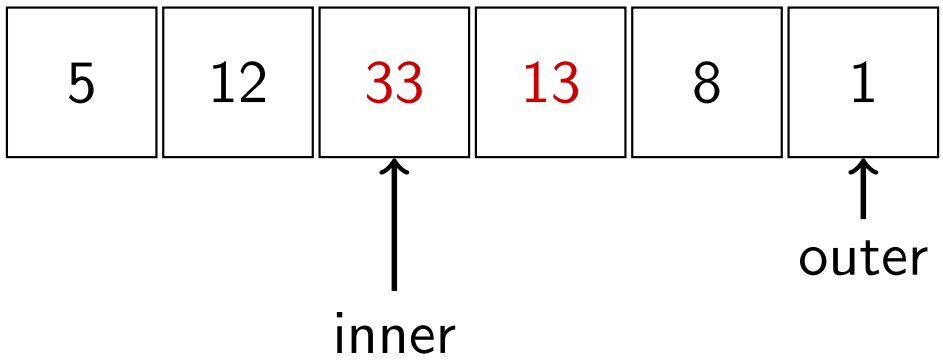
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

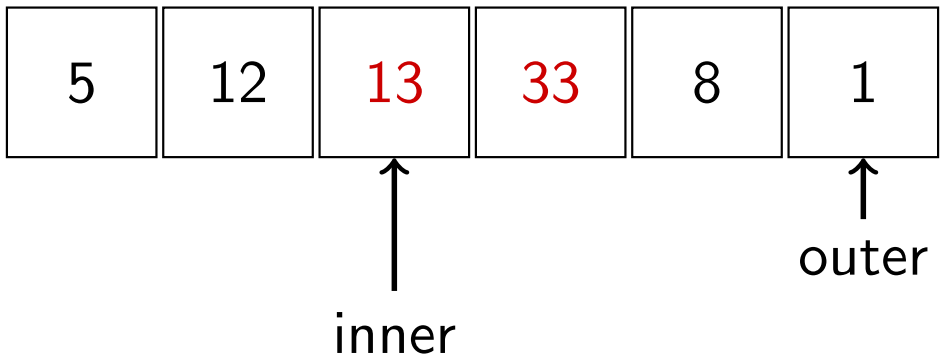
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

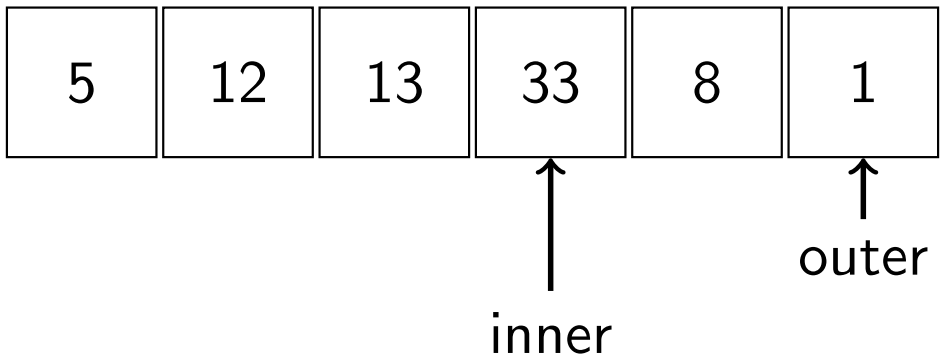
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

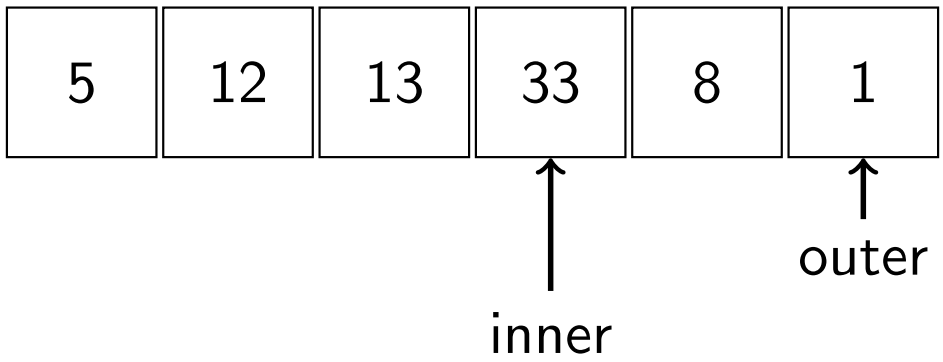
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

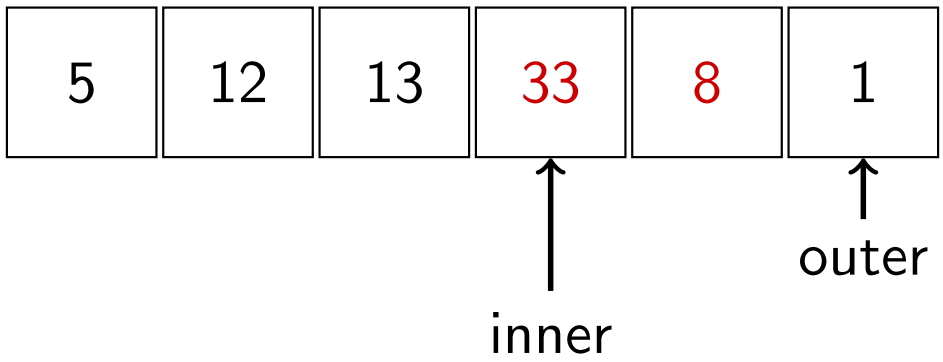
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

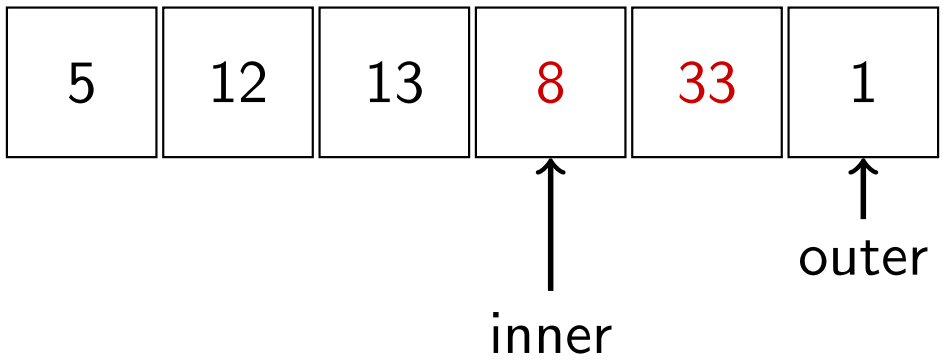
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

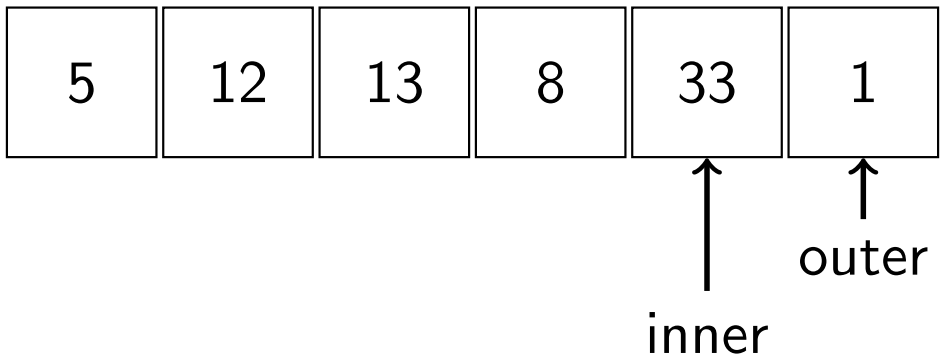
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

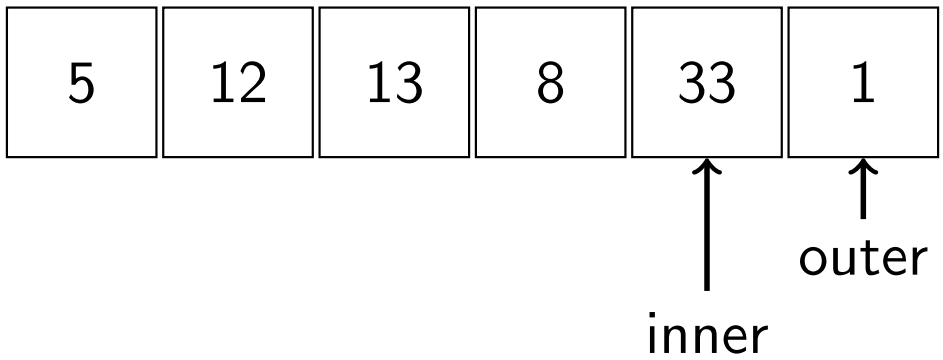
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

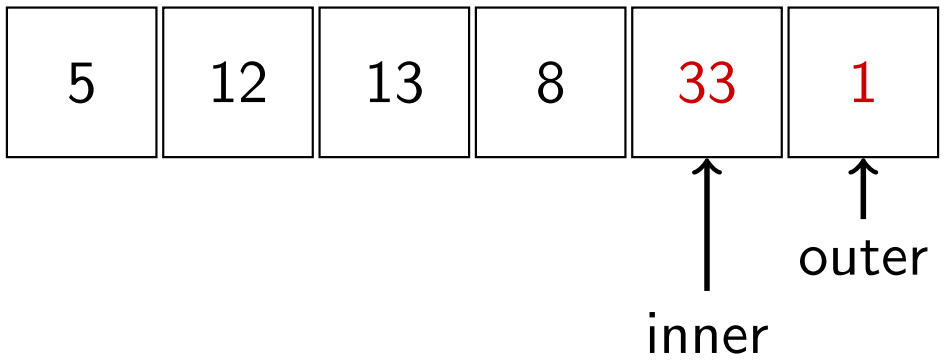
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

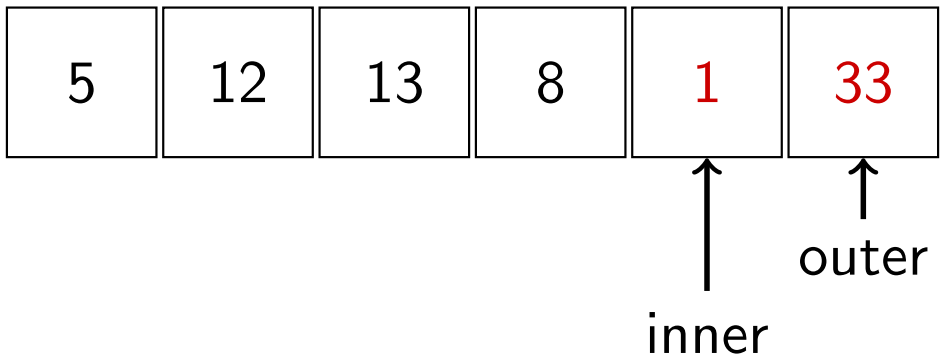
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

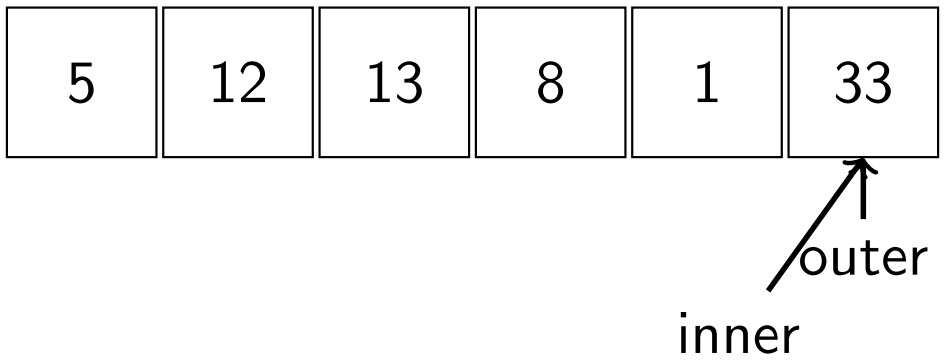
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

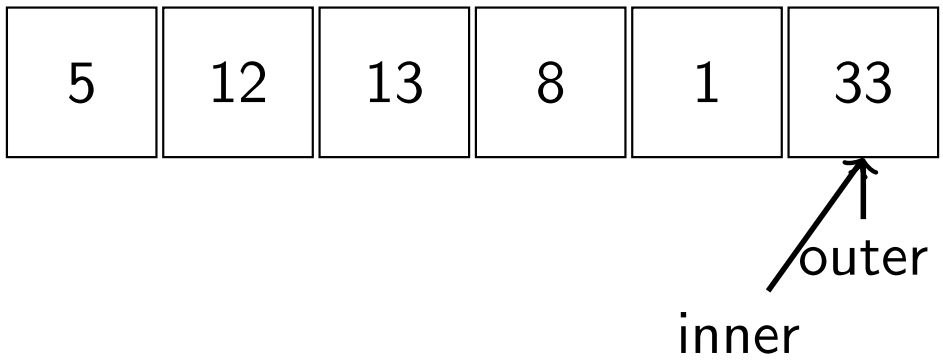
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

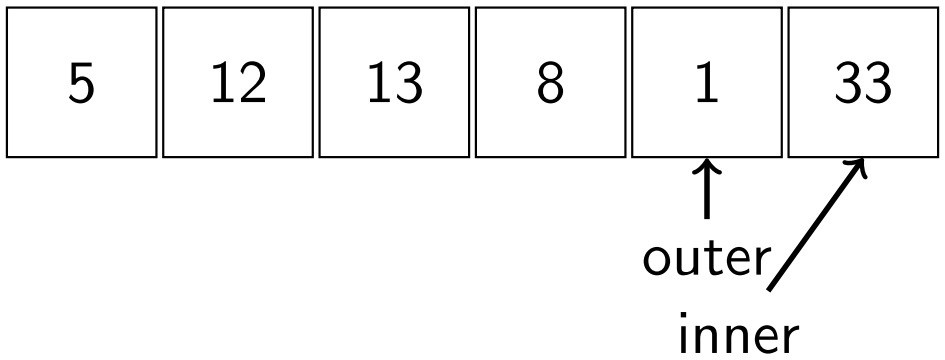
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

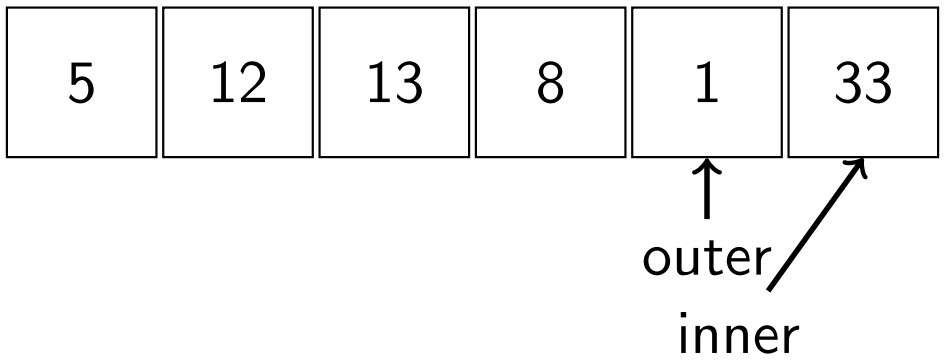
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

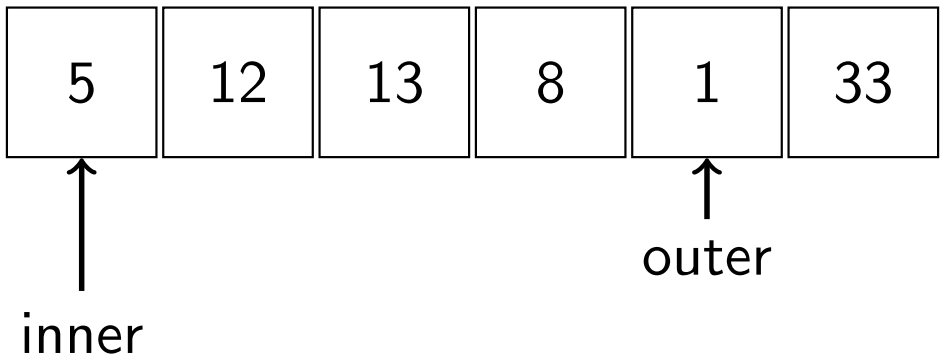
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

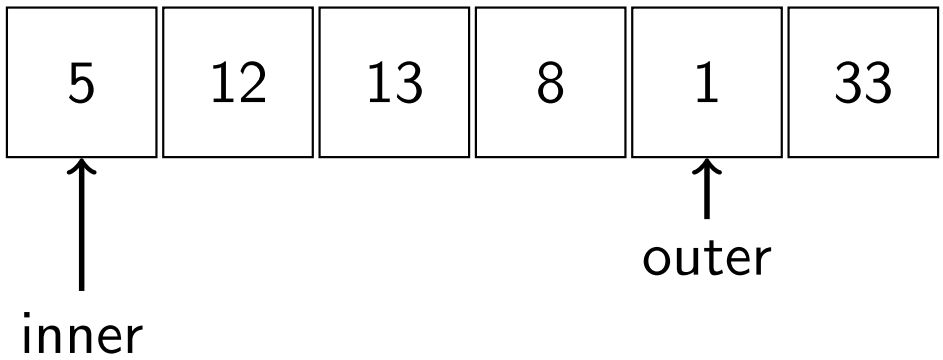
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

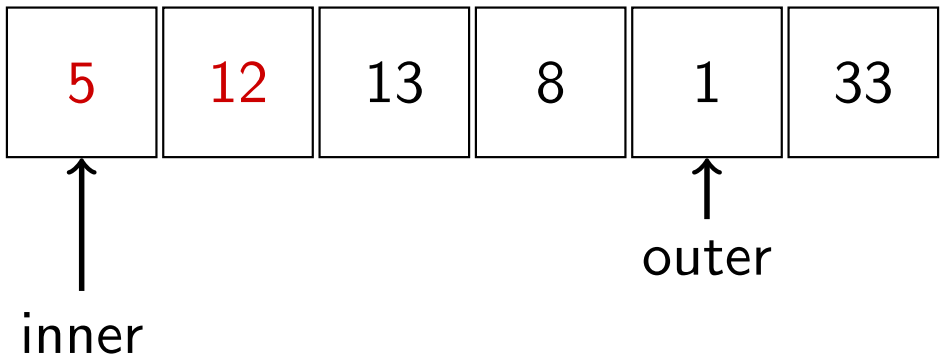
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

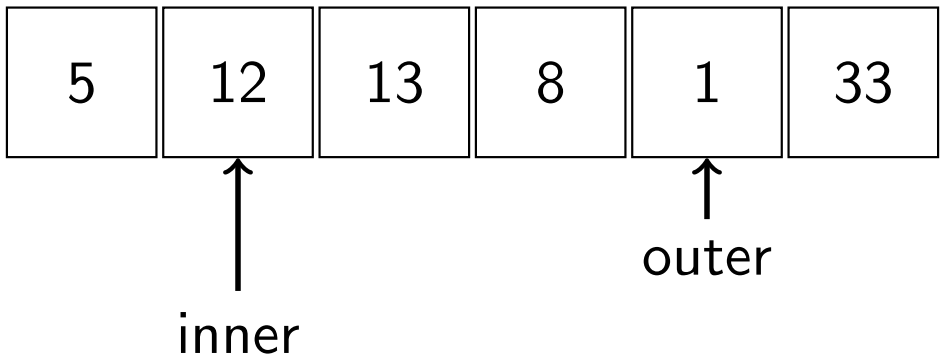
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

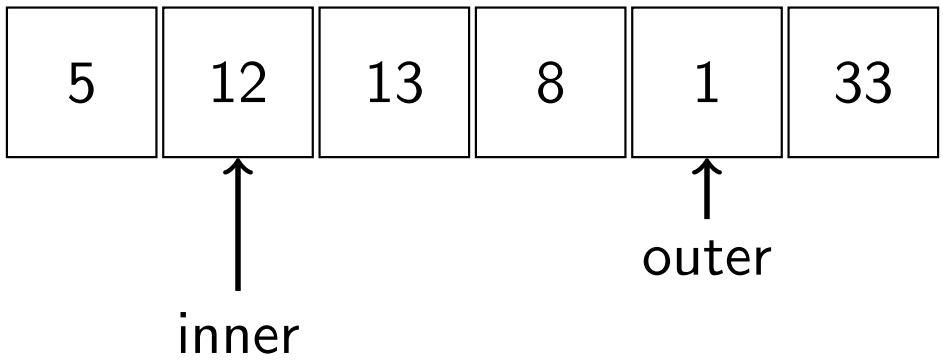
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

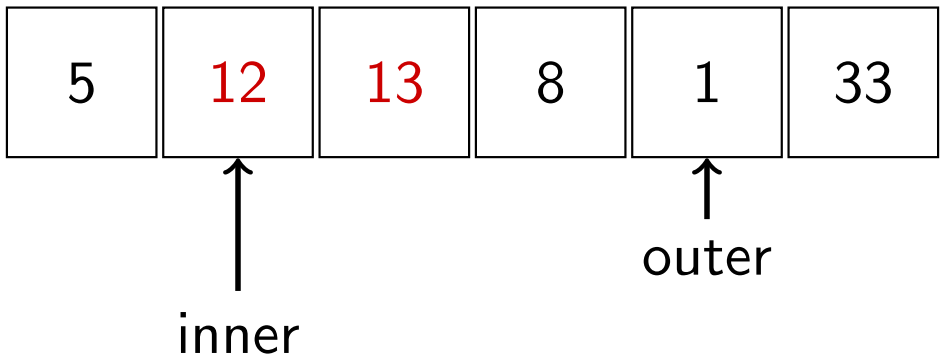
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

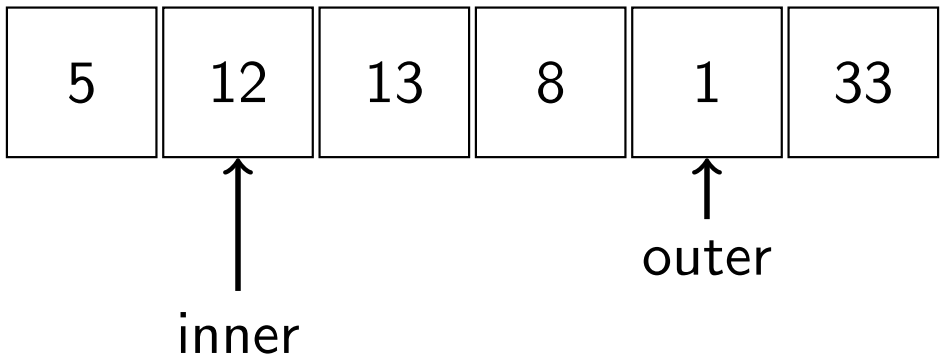
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

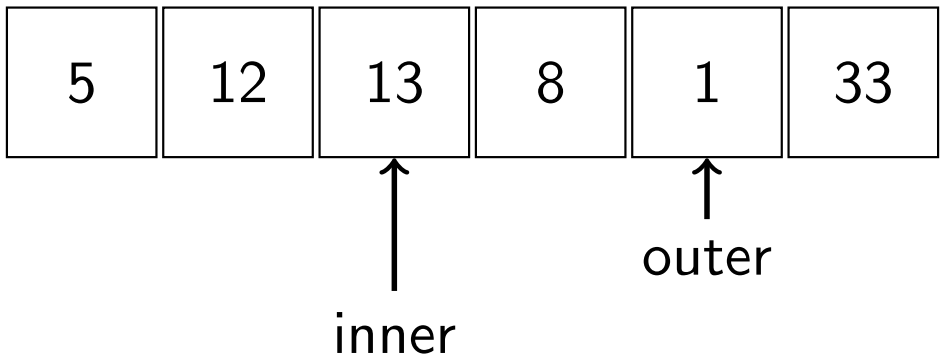
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

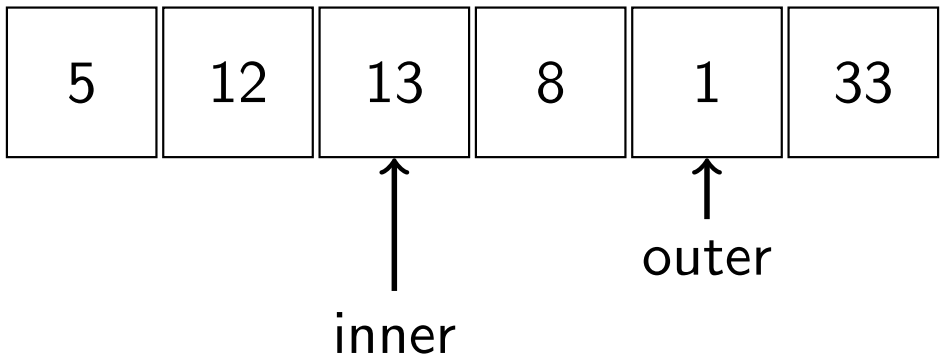
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

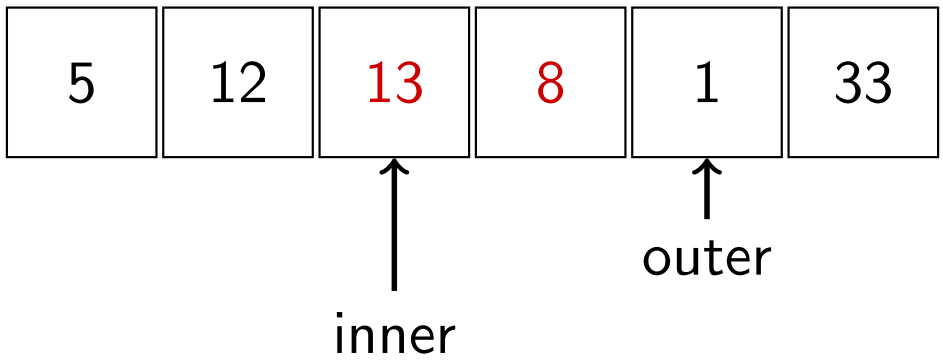
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

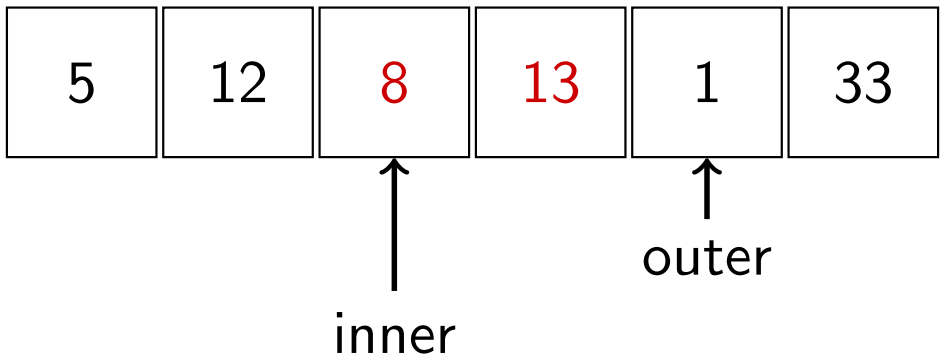
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

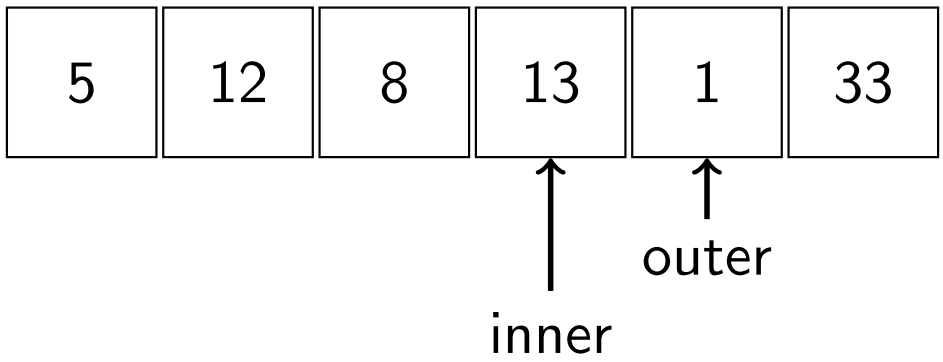
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

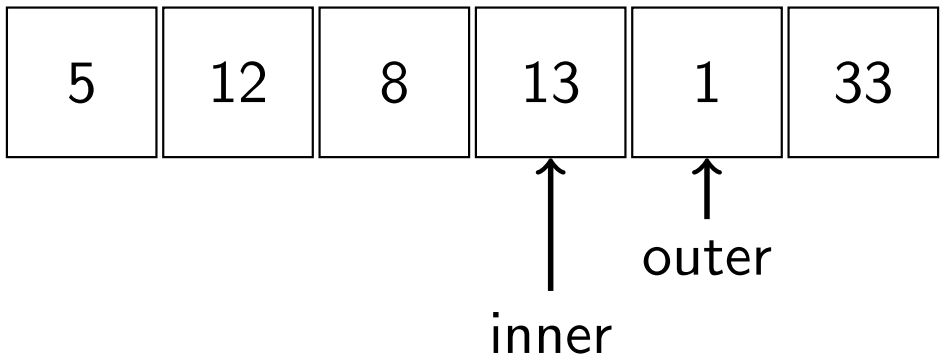
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

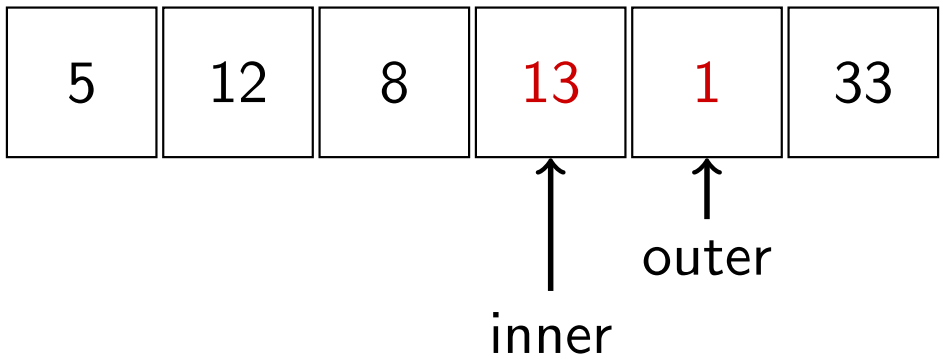
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

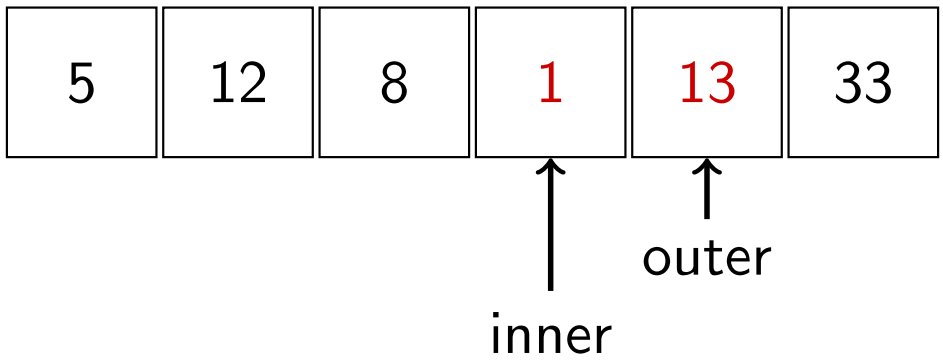
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

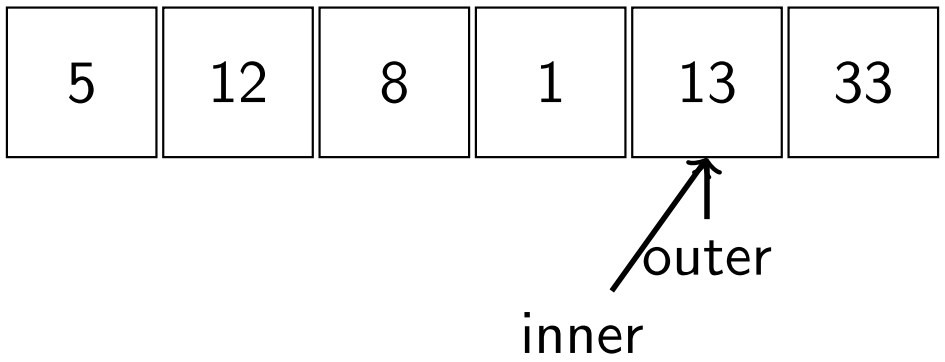
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

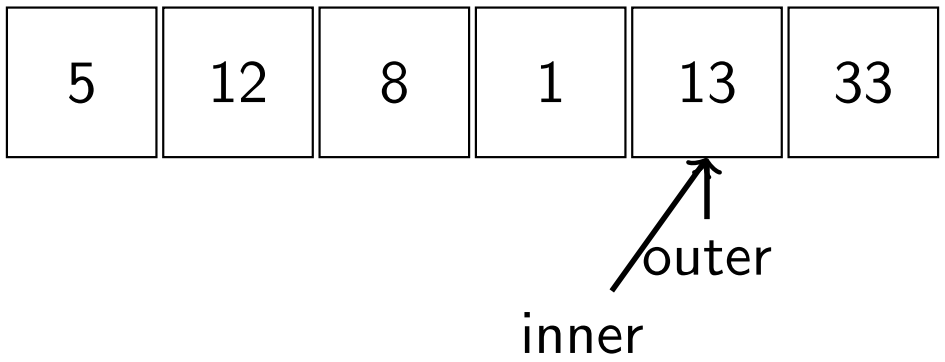
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

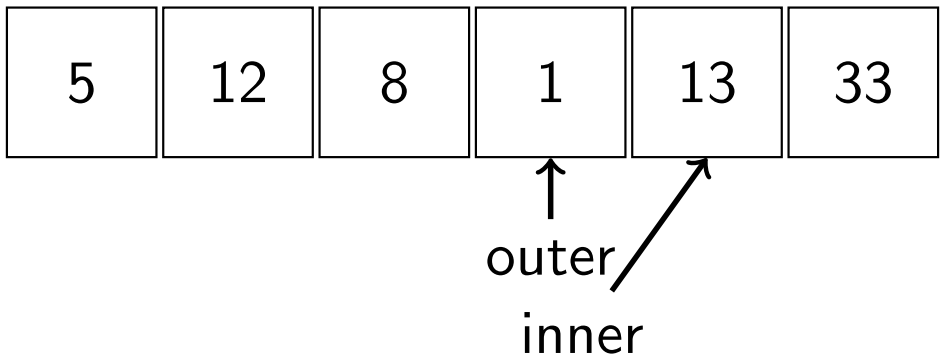
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

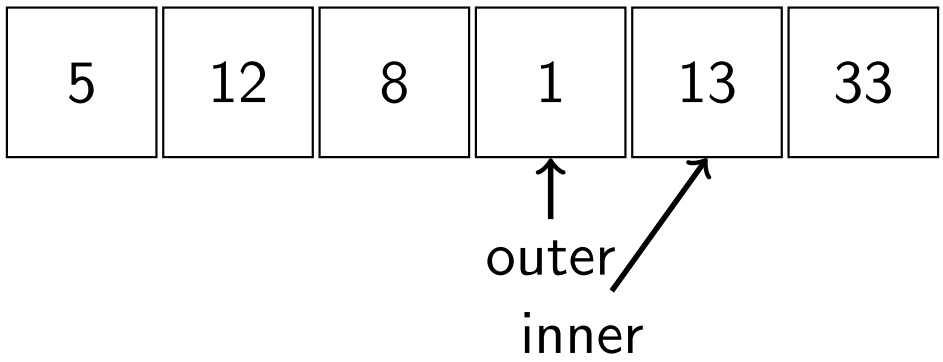
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

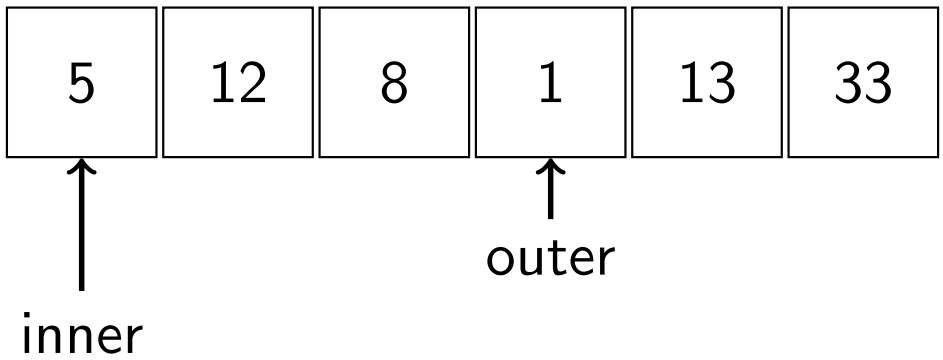
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

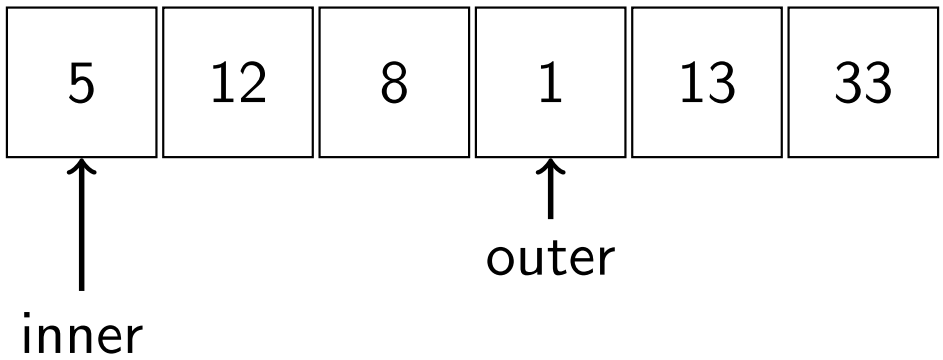
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

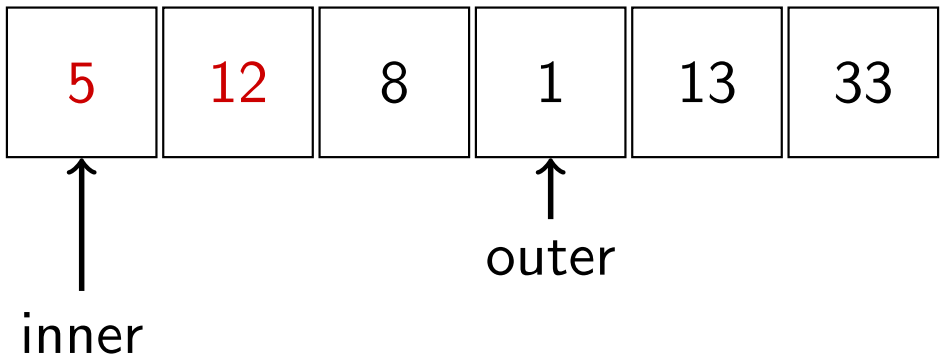
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

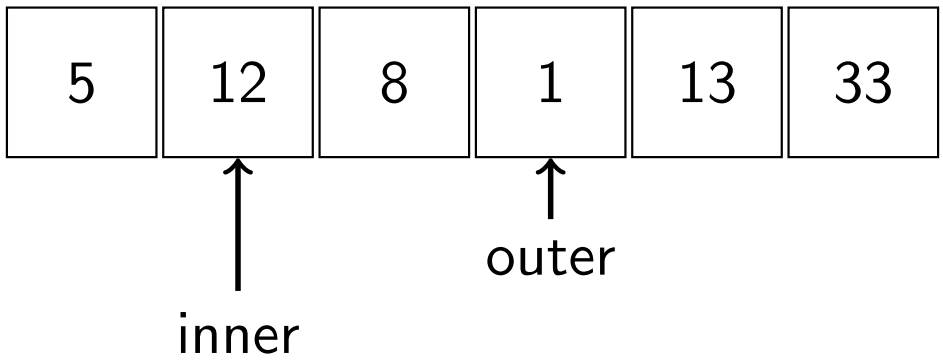
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

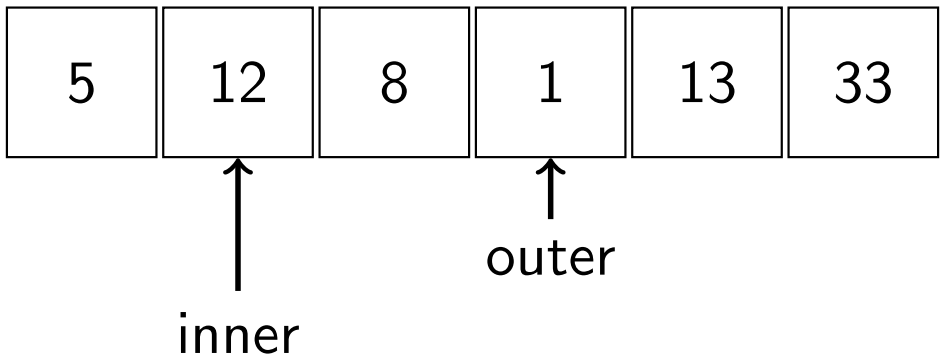
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

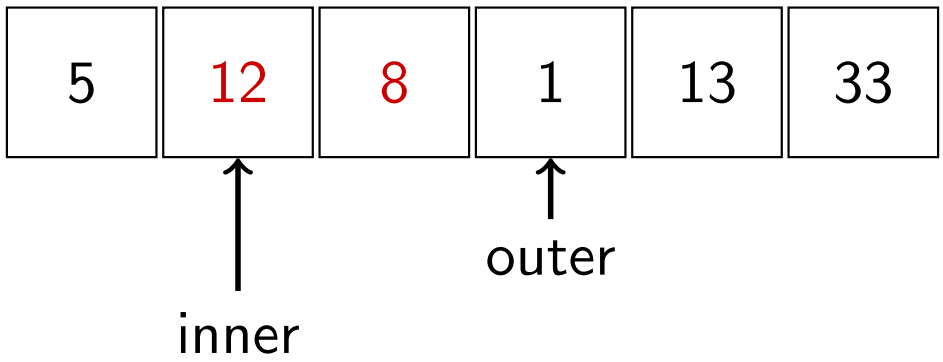
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

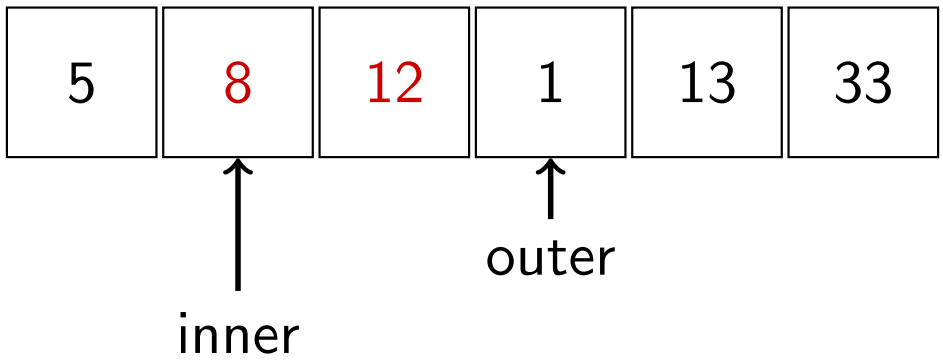
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

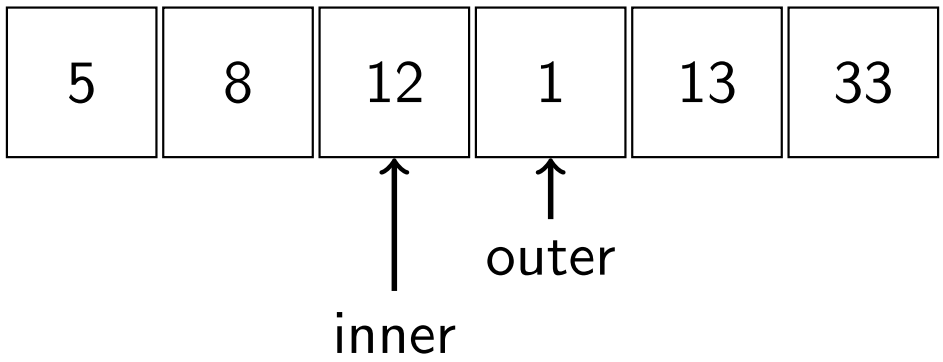
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

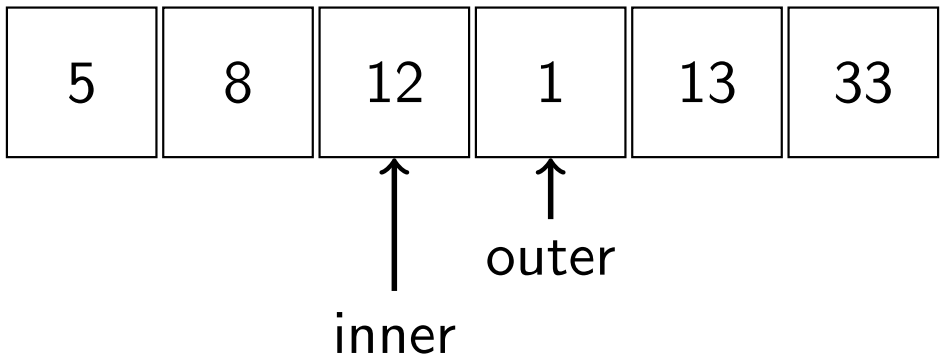
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

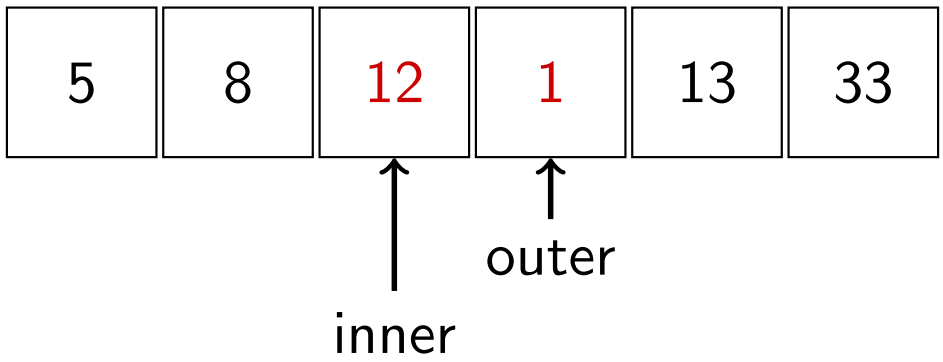
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

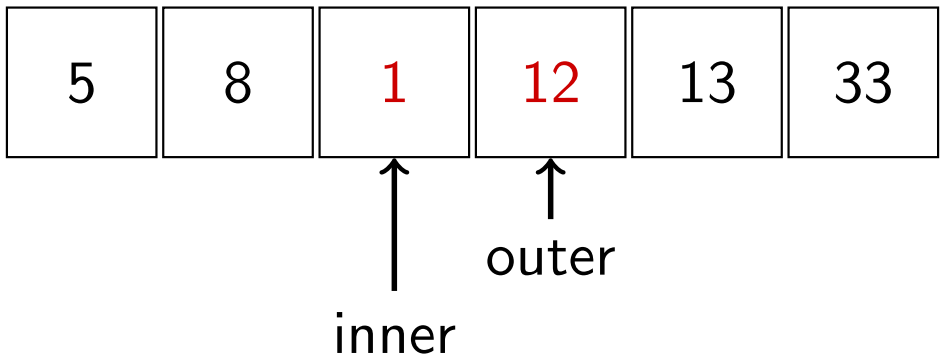
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

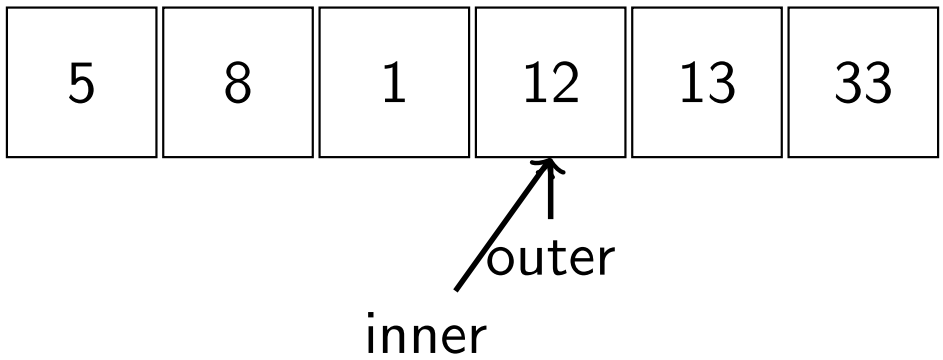
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

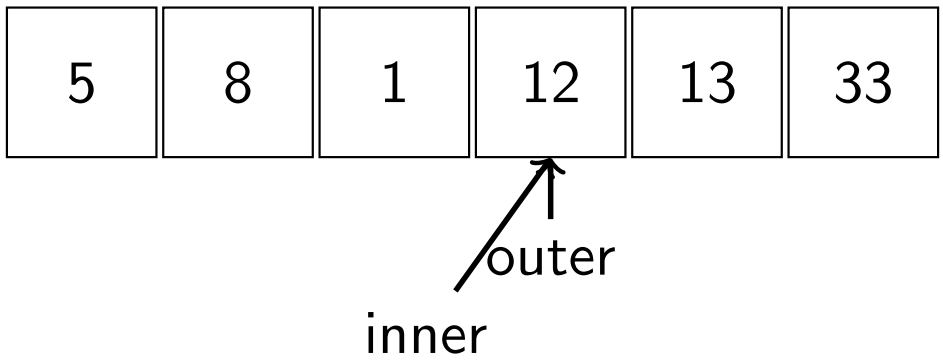
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

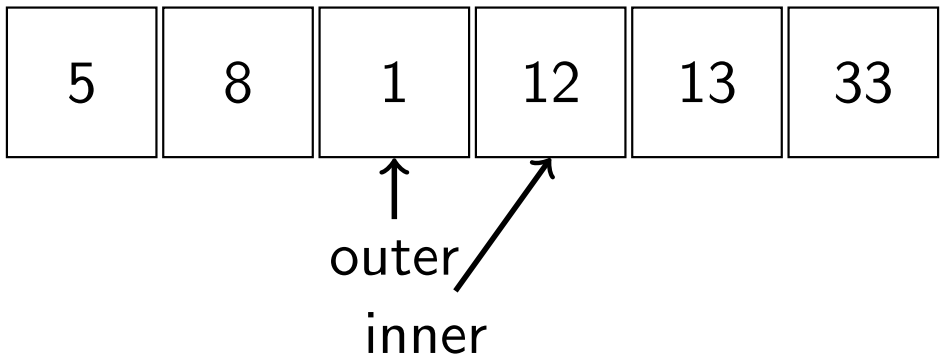
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

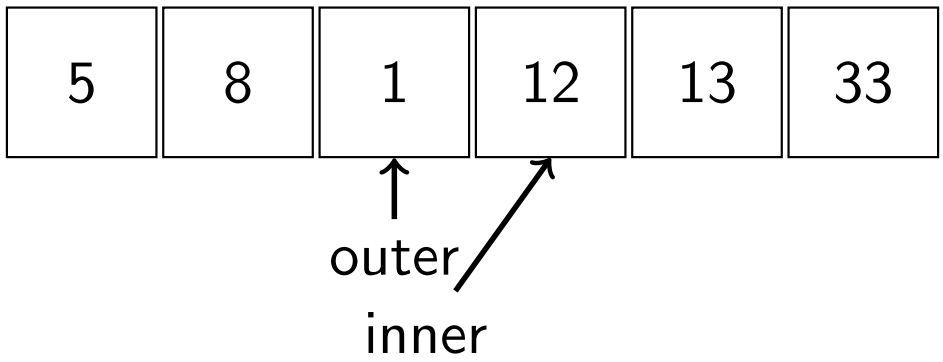
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

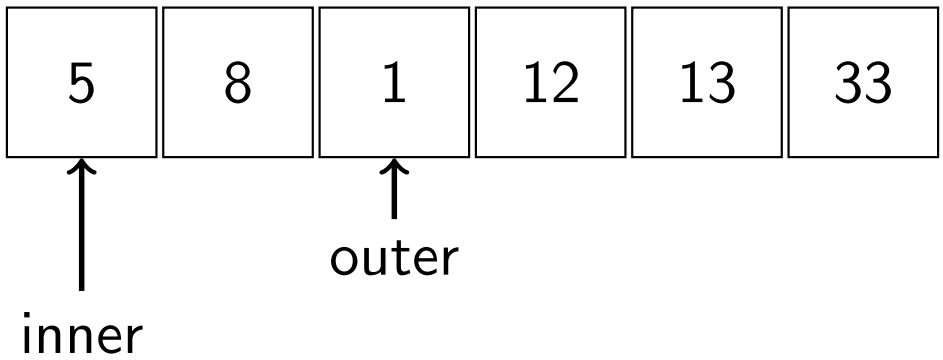
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

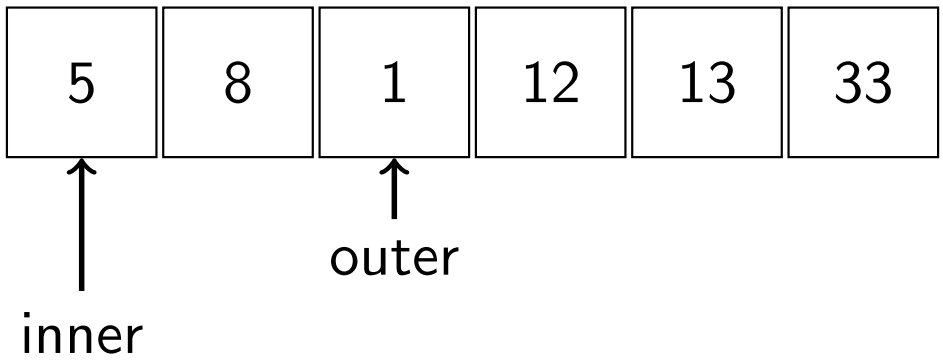
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

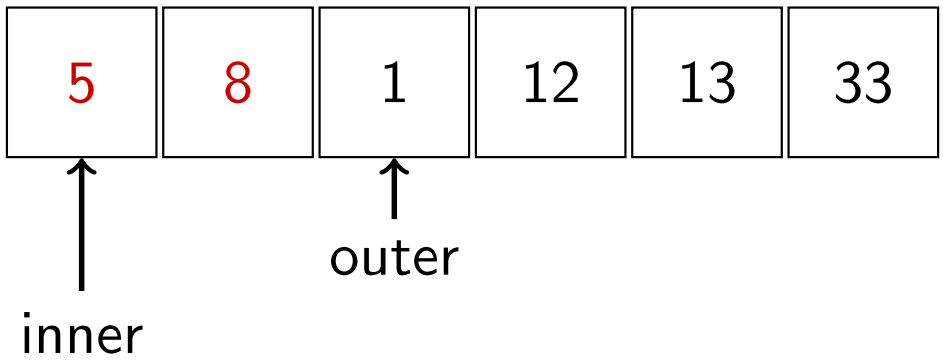
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

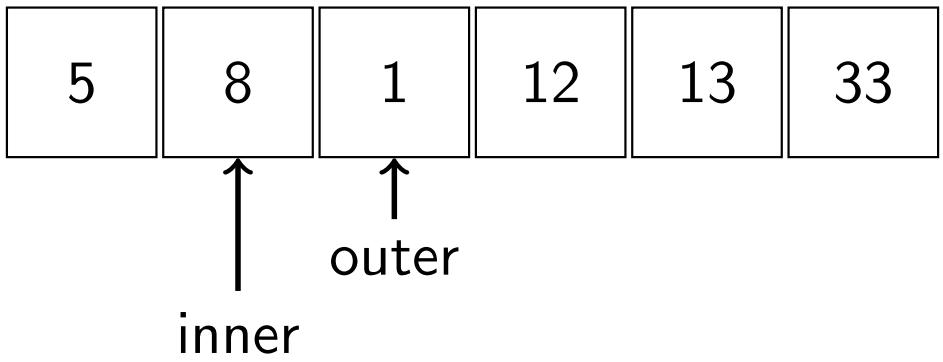
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

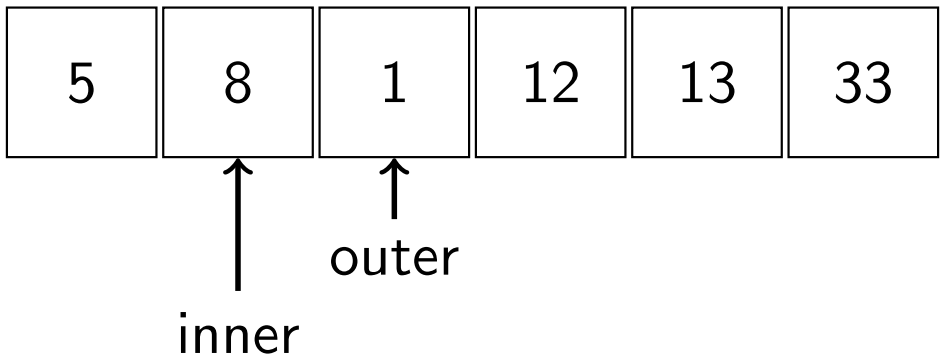
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

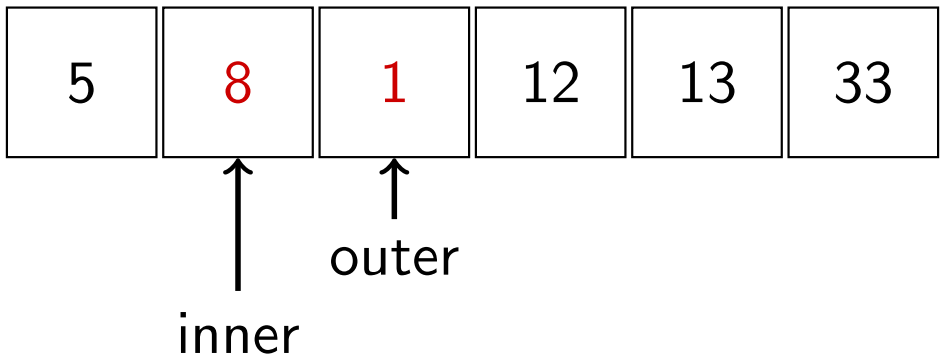
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

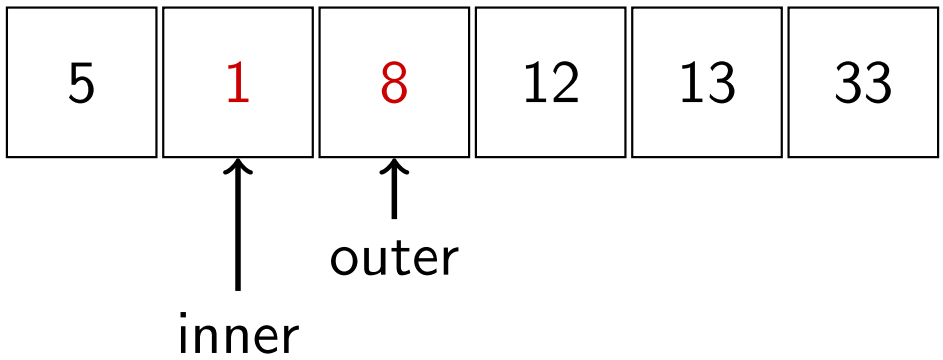
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

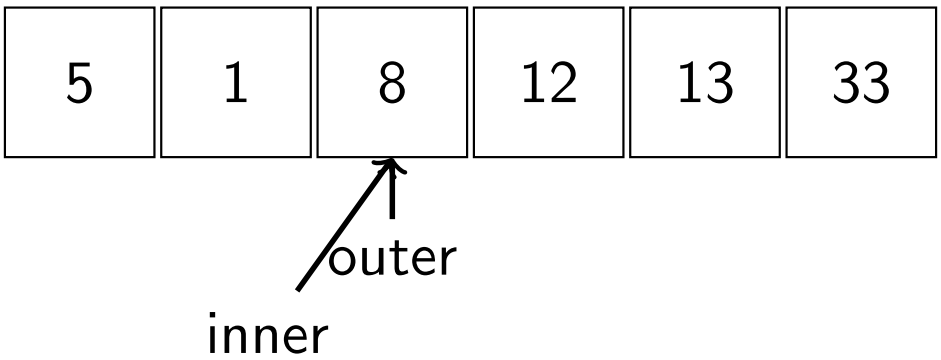
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

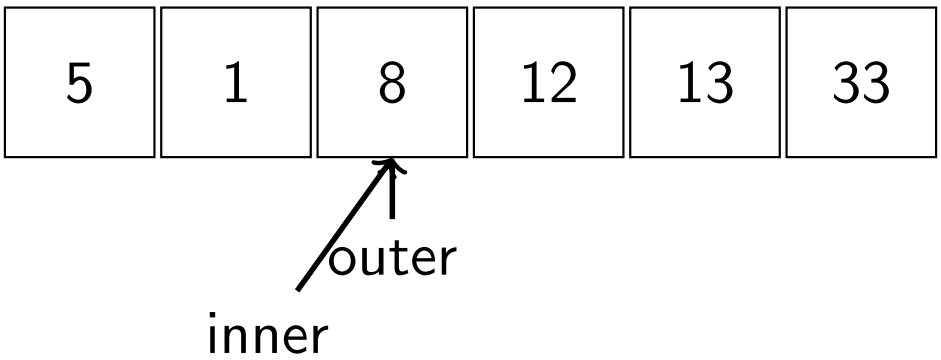
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

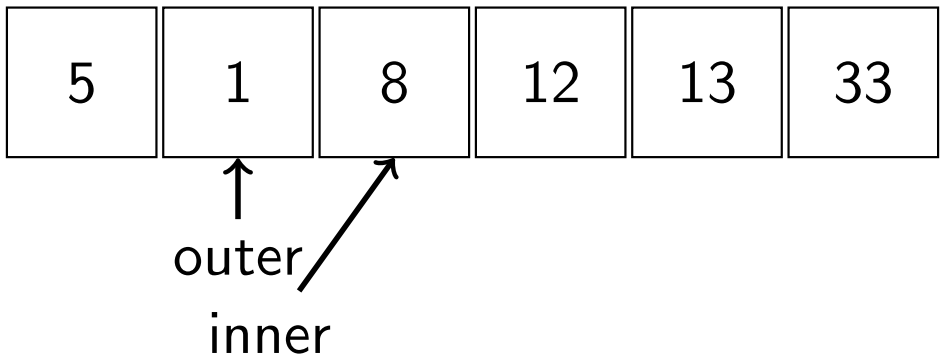
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

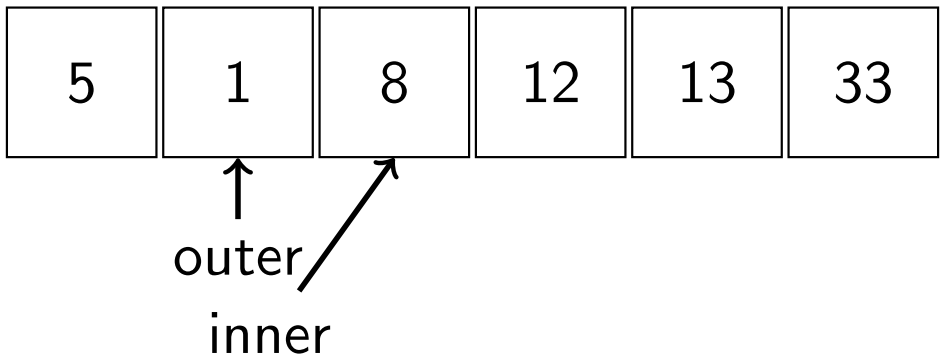
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.

Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.

Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.

Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.

Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

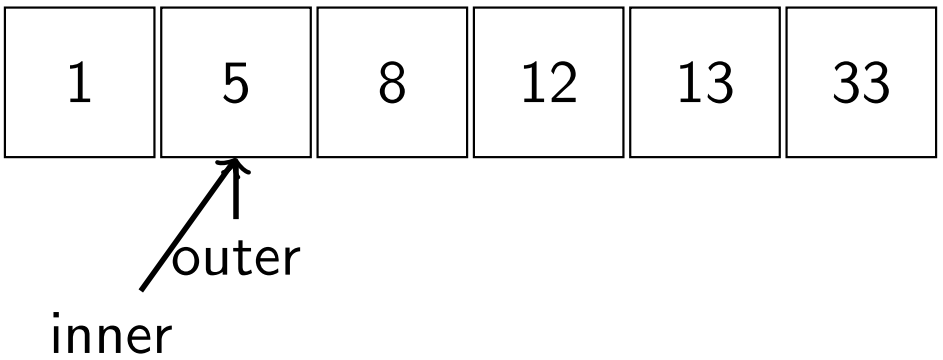
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

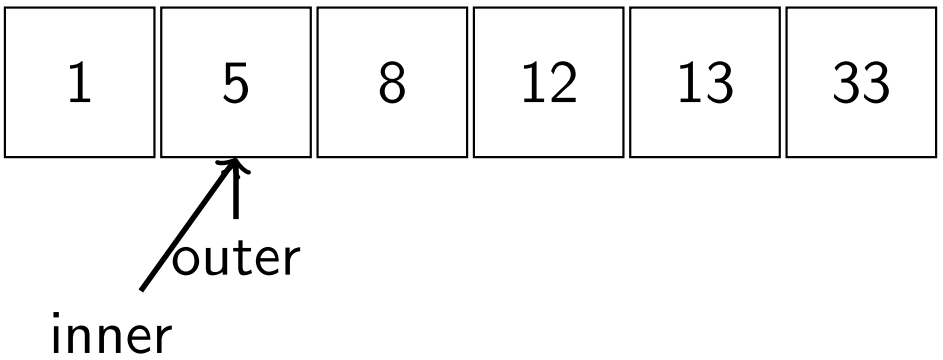
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

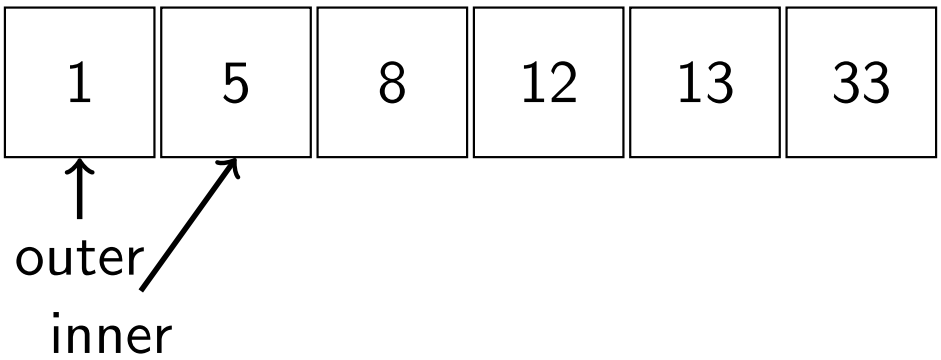
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



Ablauf am Beispiel

Falls die Liste leer ist, dann fertig.
Ansonsten:

Sei „`outer`“ ein Zeiger auf das letzte Element der Liste.

Solange „`outer`“ nicht auf das erste Element zeigt:

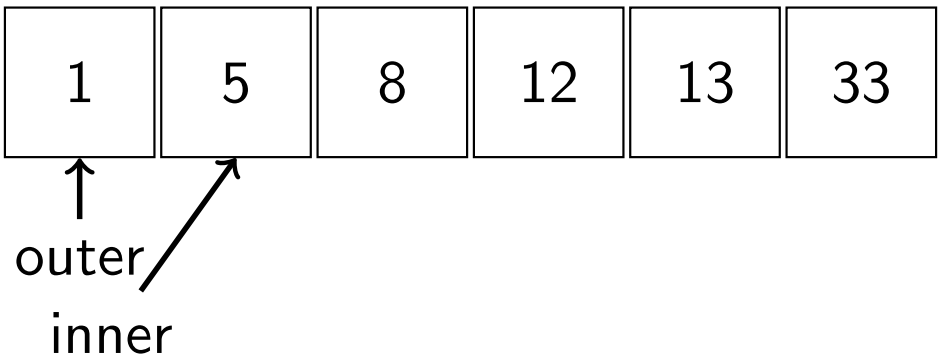
Sei „`inner`“ ein Zeiger auf das erste Element der Liste.

Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner+1`“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.



BubbleSort in Java

```
static void bubbleSort(int[] a){
  for (int outer = a.length - 1; outer > 0; outer--) {
    for (int inner = 0; inner < outer; inner++) {
      if (a[inner] > a[inner + 1]) {
          // tausche a[inner] und a[inner + 1]
        int temp = a[inner];
        a[inner] = a[inner + 1];
        a[inner + 1] = temp;
      }
    }
  }
}
```

Programmiersprachen

Maschinenprogramme

- bestehen aus Bit-Folgen (0en und 1en),
- Für den Mensch nahezu **unverständlich**
- Verständlicher, aber immer noch zu kleinschrittig: Assemblercode

Programmiersprachen

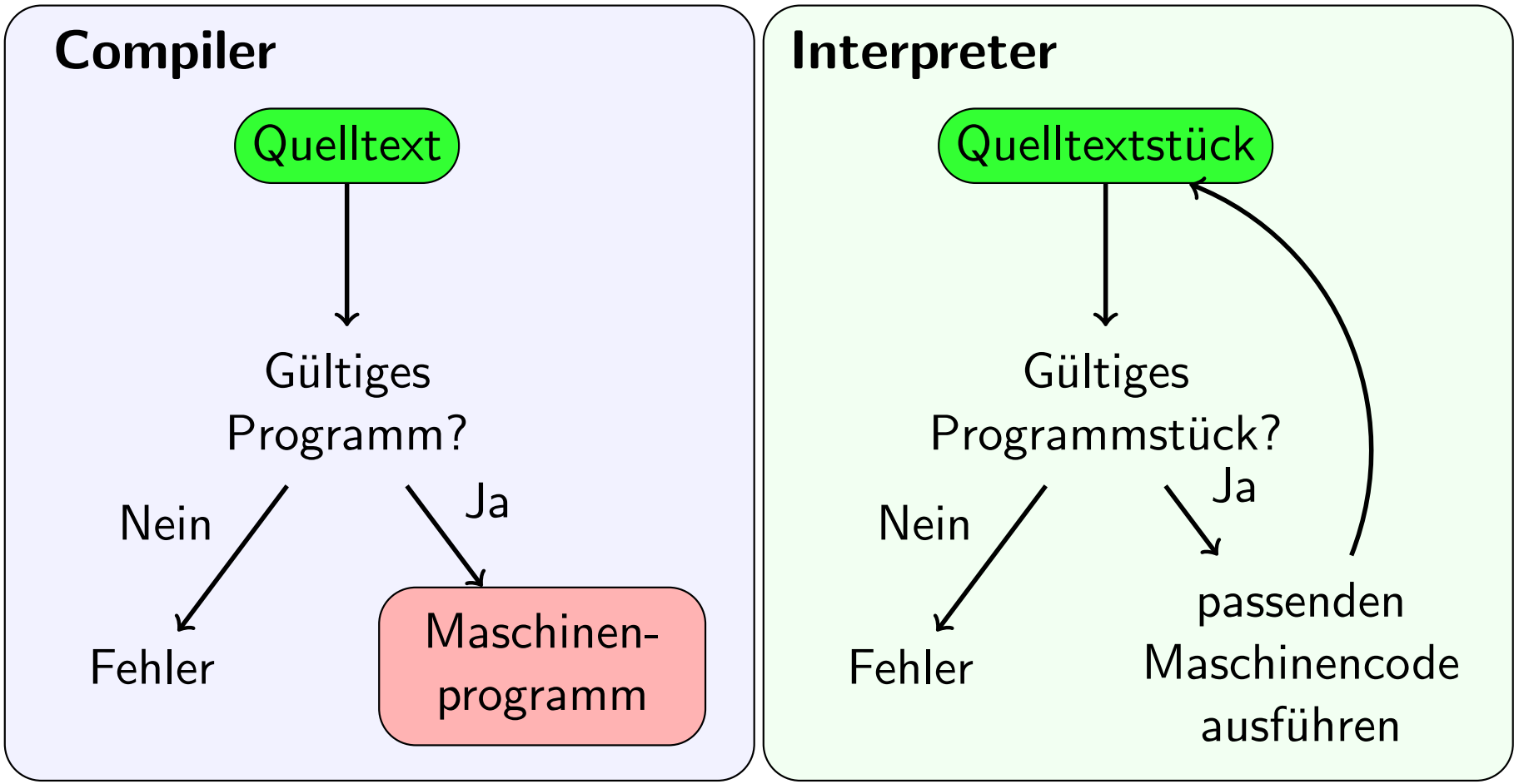
Maschinenprogramme

- bestehen aus Bit-Folgen (0en und 1en),
- Für den Mensch nahezu **unverständlich**
- Verständlicher, aber immer noch zu kleinschrittig: Assemblercode

Höhere Programmiersprachen

- Für den Mensch (meist) verständliche formale Sprache, mit der Algorithmen und Datenstrukturen beschrieben werden
- Abstraktere Konzepte, nicht genau am Rechner orientiert
- Der Rechner versteht diese Sprachen **nicht**
- Ein Übersetzer wird benötigt
- **Quelltext** = Programm in höherer Programmiersprache

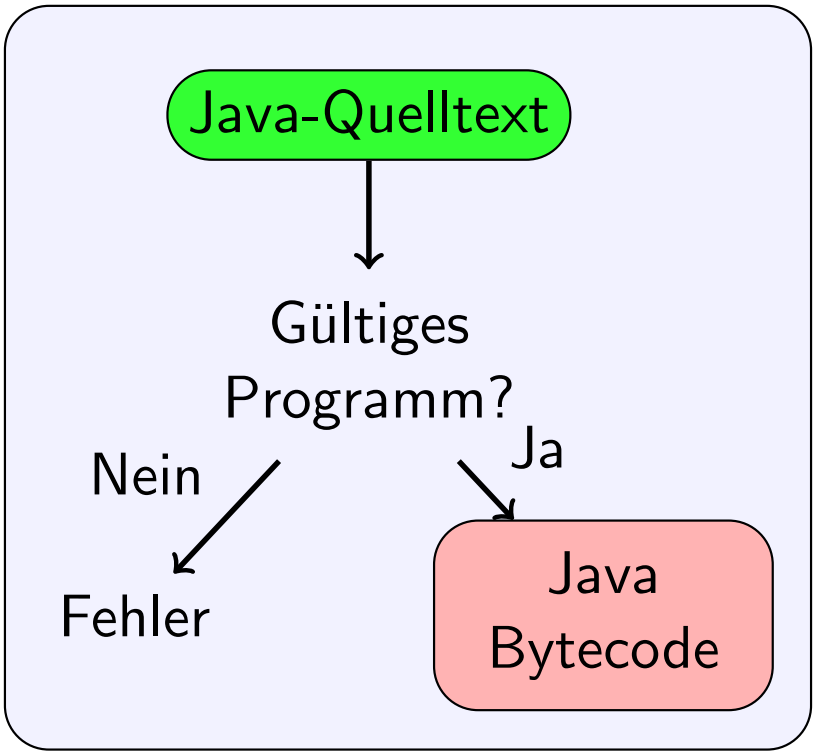
Compiler / Interpreter



- Langsame Übersetzung auf einmal
- Ausführung schnell

- Schnelle Übersetzung e. kleinen Stücks
- Ausführung eher langsam

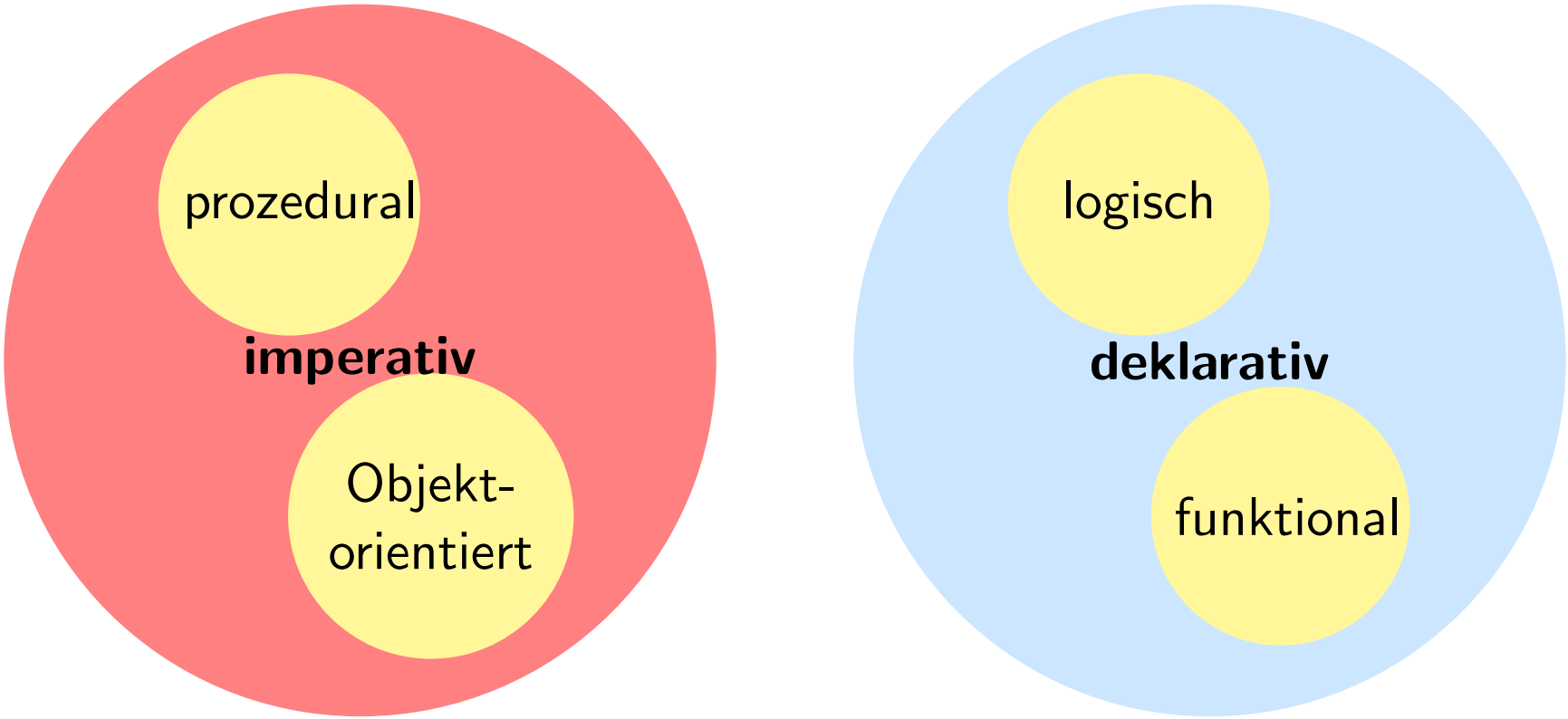
Java Compiler



- Java-Compiler erzeugt **Java Bytecode** d.h. ein **Maschinenprogramm** für die **Java Virtual Machine**
- Ausführung:
  Die Java Virtual Machine (JVM) **interpretiert** Java Bytecode
- Für jedes Betriebssystem ist eine eigene JVM implementiert.
- Vorteil: **Plattform-unabhängig**:
  Der Bytecode läuft auf jeder Plattform, für die es eine JVM gibt
- Nachtteil: **Langsamer** als nativer Maschinecode.

Programmierparadigmen

Es gibt viele verschiedene höhere Programmiersprachen!



Imperative Programmierung

- imperare = befehlen
- genau kleineschrittige Anweisungen (Befehle), **wie** das Problem gelöst wird
- Variablen mit Zuweisung
- Schleifen (Iteration)
- Varianten imperative Programmiersprachen:
  - **Prozedurale Sprachen**
    Imperative Sprachen mit Strukturierung durch Prozeduren (= Wiederverwendbare Codeabschnitte)
    Beispiele: Fortran, Algol, Pascal, Modula, C, ...
  - **Objektorientierte Sprachen**
    Imperative Sprachen mit Objekten, Klassen und Vererbung
    Beispiele: Simula, Smalltalk, C++, C#, Java, ...

Deklarative Programmierung

- declarare = erklären
- „Beschreibung des Problems“ statt exakter „Befehlsfolge zur Lösung“
- eher **was** gelöst werden soll, statt **wie**
- Varianten deklarativer Programmiersprachen:
  - **Funktionale Sprachen**
    Programme sind Ausdrücke: Anwendung (insbes. rekursiver) Funktionen auf Argumente,
    Ausführung: Funktionsauswertung
    Beispiele: LISP, ML, Haskell, OCaml, F#, Erlang, Clojure, ...
  - **Logische Sprachen**
    Programme bestehen aus Fakten und Regeln.
    Ausführung: logische Deduktion von neuen Fakten
    Beispiele: Prolog, Mercury, Curry, ...

Typisierung bei Programmiersprachen

Was ist ein Typ?

- Ein Typ beschreibt einen Wertebereich für Objekte.
- Beispiel: Der Typ Integer beschreibt alle Objekte, die zu Ganzzahlen von -2^{31} bis $2^{31}-1$ ausgewertet werden können.

Sprechweisen:

- Man sagt:
  - „X ist vom Typ Y“
  - „X hat den Typ Y“
  - „X gehört zum Typ Y“
- die Werte selbst, z.B. -100, 42, . . . „sind vom Typ Integer“, aber auch z.B. Ausdrücke wie 10+5*6 „sind vom Typ Integer“

Eigenschaften der Typisierung von Programmiersprachen

statisch vs. dynamisch

- **Statische Typisierung**: Typisierung des Quelltexts (Variablen, Werte, Funktionen, . . . haben einen Typ), Typcheck von Operationen durch den Compiler
- **Dynamische Typisierung**: Typisierung von Daten während der Laufzeit, z.B. Prozeduren prüfen die Typen ihren Eingabedaten.

Eigenschaften der Typisierung von Programmiersprachen

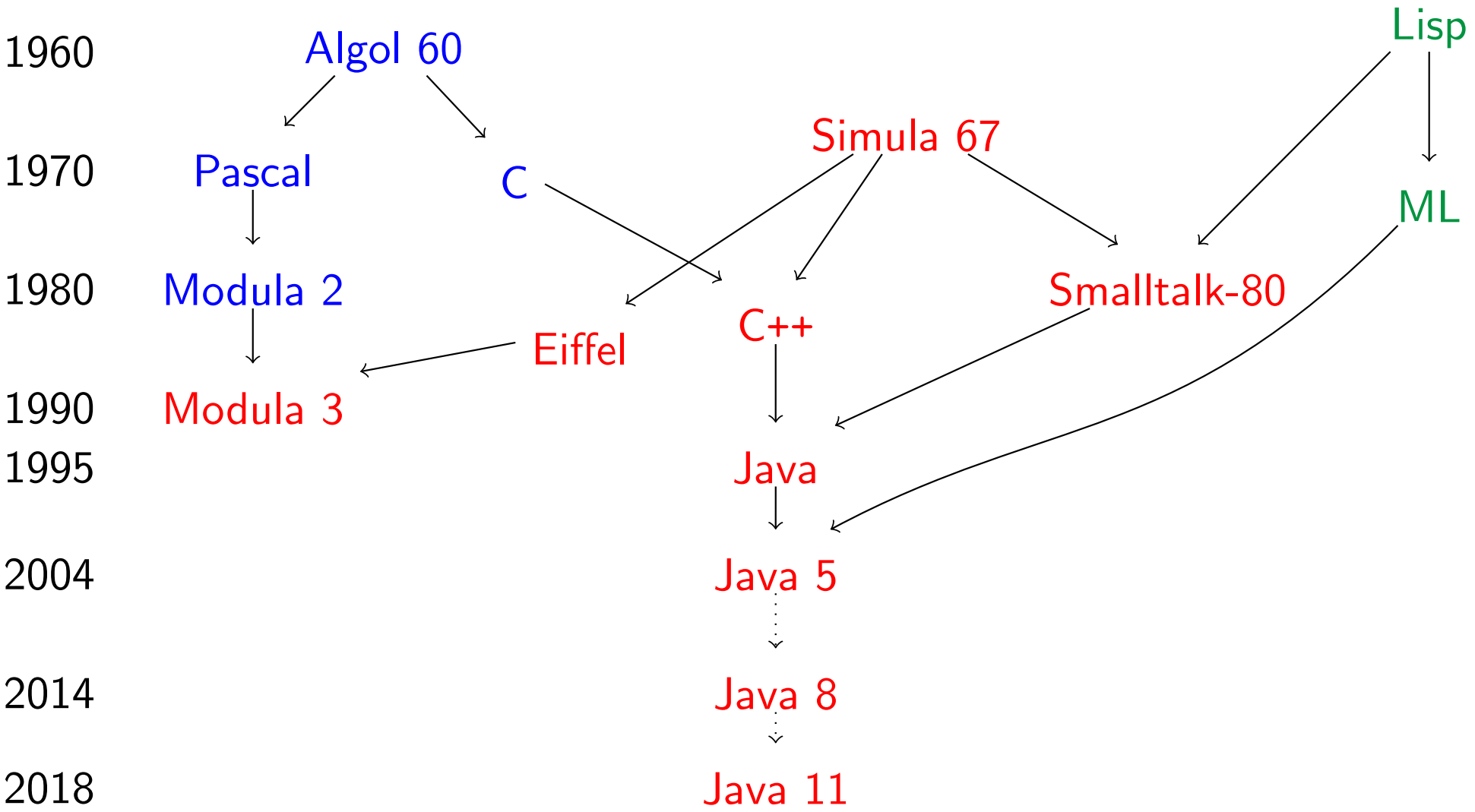
statisch vs. dynamisch

- **Statische Typisierung**: Typisierung des Quelltexts (Variablen, Werte, Funktionen, ... haben einen Typ), Typcheck von Operationen durch den Compiler
- **Dynamische Typisierung**: Typisierung von Daten während der Laufzeit, z.B. Prozeduren prüfen die Typen ihren Eingabedaten.

stark vs. schwach

- **Starke (strenge) Typisierung**: Zur Laufzeit sind keine Typfehler erlaubt, d.h. der Compiler findet diese schon zur Compilezeit.
- **Schwache Typisierung** Es sind Programme erlaubt, die Typfehler während der Laufzeit haben können.

Entwicklungsgeschichte von Java



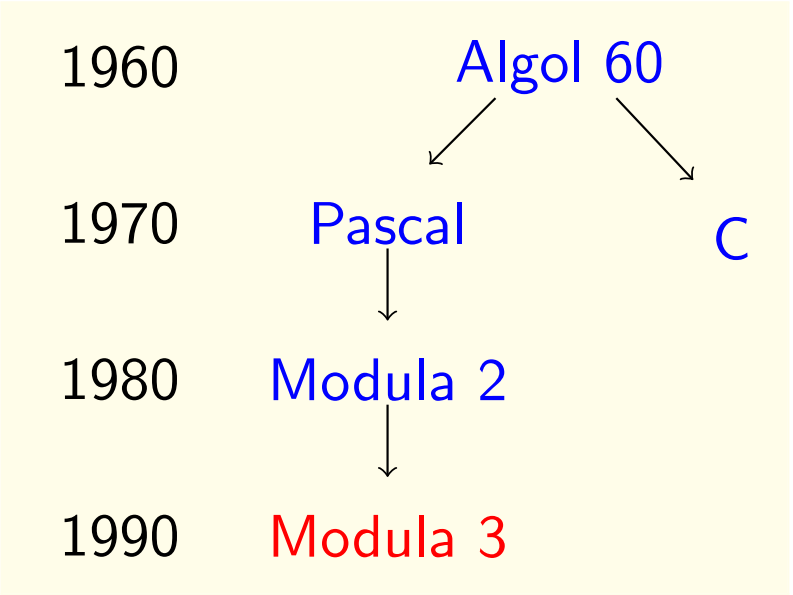
Entwicklungsgeschichte von Java (2)

Algol 60 (Algorithmic Language 1960)

- Entwickelt von John Backus, Friedrich L. Bauer, John McCarthy, Peter Naur, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson
- Imperative Sprache mit Blockkonzept
- Syntaxdefinition in Backus-Naur-Form

Pascal

- Entwickelt von Niklaus Wirth
- Prozedurale Sprache
- Weiterentwicklung von Algol 60
- Verbunddatentypen (Record)



C

- imperative und prozedurale Programmiersprache
- entwickelt von Dennis Ritchie
- eine der am weitesten verbreiteten Programmiersprachen

Entwicklungsgeschichte von Java (3)

Simula 67

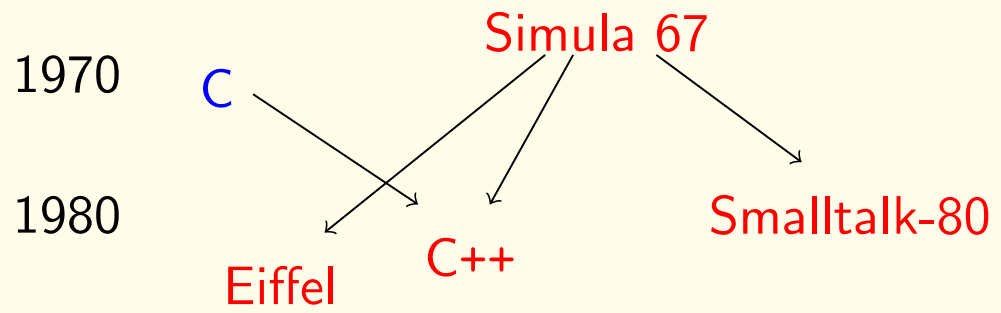
- „erste“ objektorientierte Sprache
- entwickelt von Kristen Nygaard und Ole-Johan Dahl.
- Baut auf Algol 60 auf

Smalltalk-80

- entwickelt von Alan Kay und Adele Goldberg
- Erste dynamische rein-objektorientierte Sprache
- mit Betriebssystem und Entwicklungsumgebung

C++

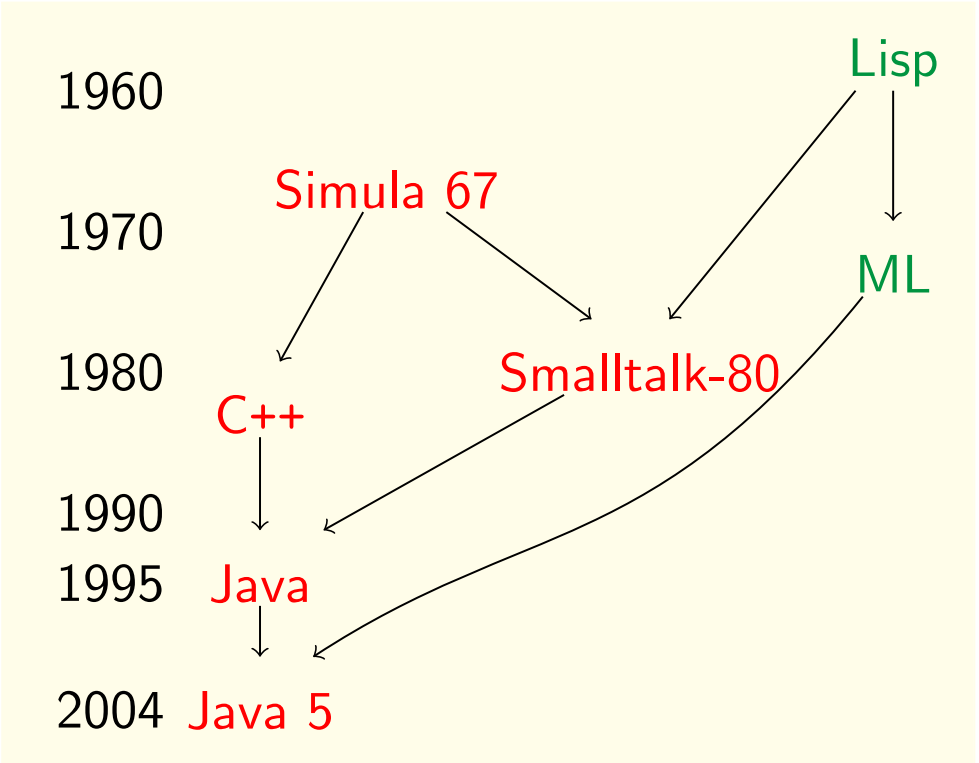
- entwickelt von Bjarne Stroustrup
- Effiziente objektorientierte Programmiersprache
- Erweiterung von C



Entwicklungsgeschichte von Java (4)

Lisp

- List Processing
- funktionale Programmiersprache
- entwickelt von John McCarthy



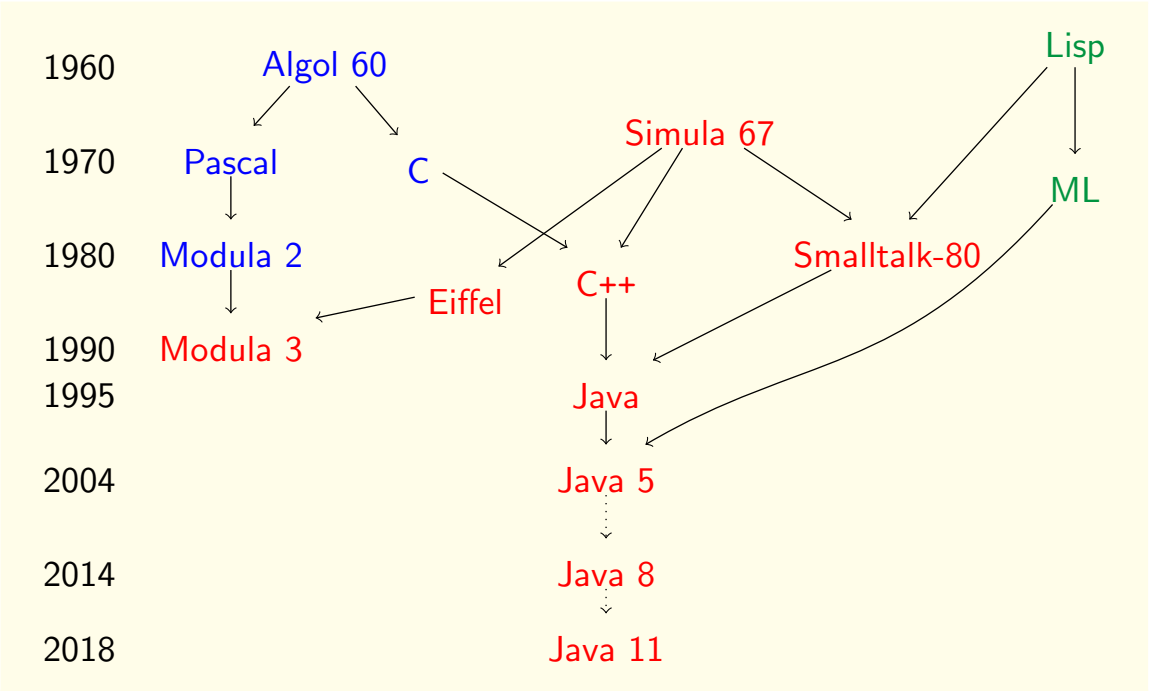
ML

- ML = Meta-Language
- funktionale Programmiersprache
- Ursprünglich von Robin Milner als Teil eines Theorem-Beweisers entwickelt
- Varianten: Standard ML, OCaml, ...
- Starke, statische Typsysteme, Generisches Programmieren, ...

Entwicklungsgeschichte von Java (5)

Java

- entwickelt von Sun Microsystems (2010 von Oracle gekauft)
- Entwickler: James Gosling, Bill Joy, Patrick Naughton, u.a.
- erste plattformunabhängige objektorientierte Sprache
- eine der populärsten Programmiersprachen
- insbes. zur Programmierung von Internet-Applikationen
- Wir verwenden Java 8



Charakteristika von Java

- Imperativ
- Objektorientiert: Klassenkonzept, starke und statische Typisierung
- Unabhängig von Plattform: Durch Übersetzung in Bytecode, der von einer Virtuellen Maschine (VM) interpretiert wird.
- Unterstützt parallele Ausführungen (Nebenläufigkeit)
- Besitzt eine reichhaltige Klassenbibliothek (API, „Application Programming Interface“) zur Wiederverwendung von Programmen

Grober Aufbau eines Java Programms

- Java Programme bestehen aus einer **Menge von Klassen**.
- Eine Klasse besteht aus
  - **Attributen**: Beschreiben charakteristische Merkmale / Eigenschaften der Objekte dieser Klasse
  - **Methoden**: Beschreiben Operationen in Form von Algorithmen

Syntax eines (sehr) einfachen Java-Programms

```
public class <Klassenname> {

    public static void main(String[] args) {
        <Anweisungen>
    }

}
```

Wir nehmen das zunächst einmal so hin, und konzentrieren uns auf den Teil mit den `<Anweisungen>`.
Folgendes Beispiel-Programm gibt den Text „Hallo!“ aus:

```
public class Hallo {

    public static void main(String[] args) {
        System.out.println("Hallo!");
    }

}
```

Methodenaufruf

Methodenaufruf allgemein:

```
object.methodName(parameters);
//bei statischen Methoden:
        class.methodName(parameters);
```

Beispiel:

```
System.out.println("Hallo!");
```

Leerzeichen und Formatierung

Anstelle von

```
public class Hallo {

    public static void main(String[] args) {
        System.out.println("Hallo!");
    }

}
```

hätten wir auch schreiben können

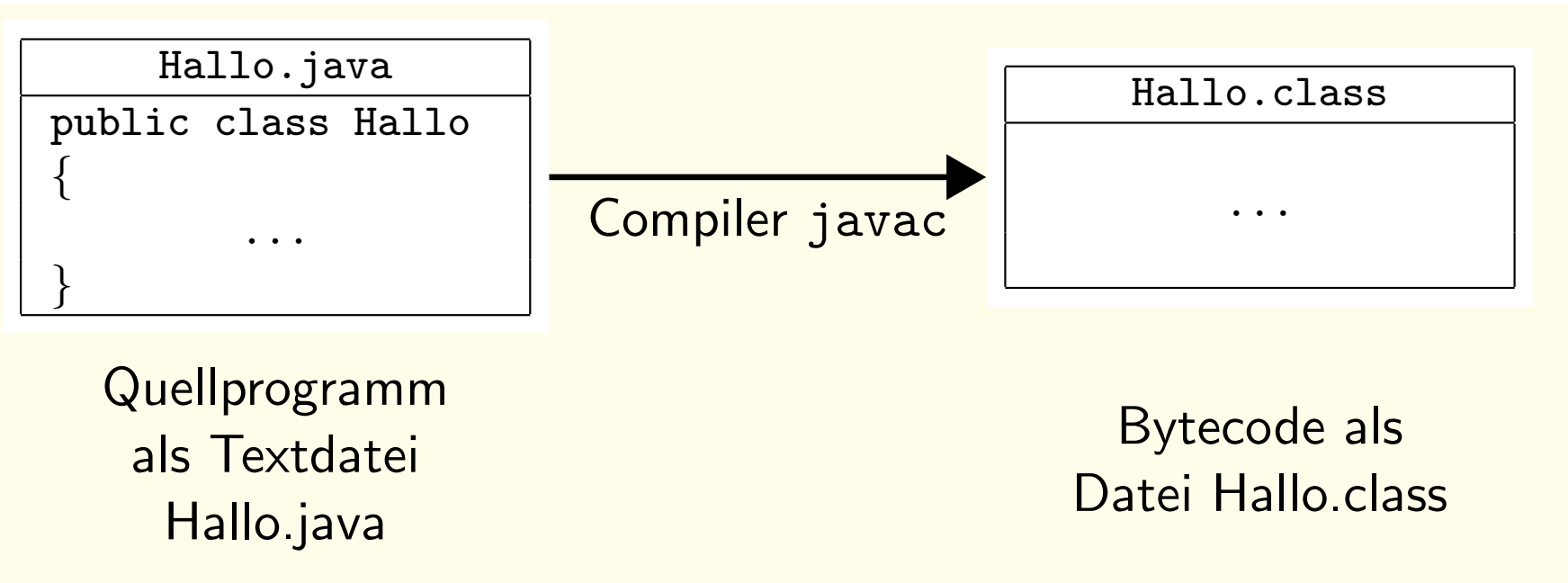
```
 public class
 Hallo { public static void
     main(String[] args
) {System.out.println("Hallo!"); }          }
```

Meist folgt man bestimmten Konventionen für die Formatierung.

Übersetzung von Java-Programmen

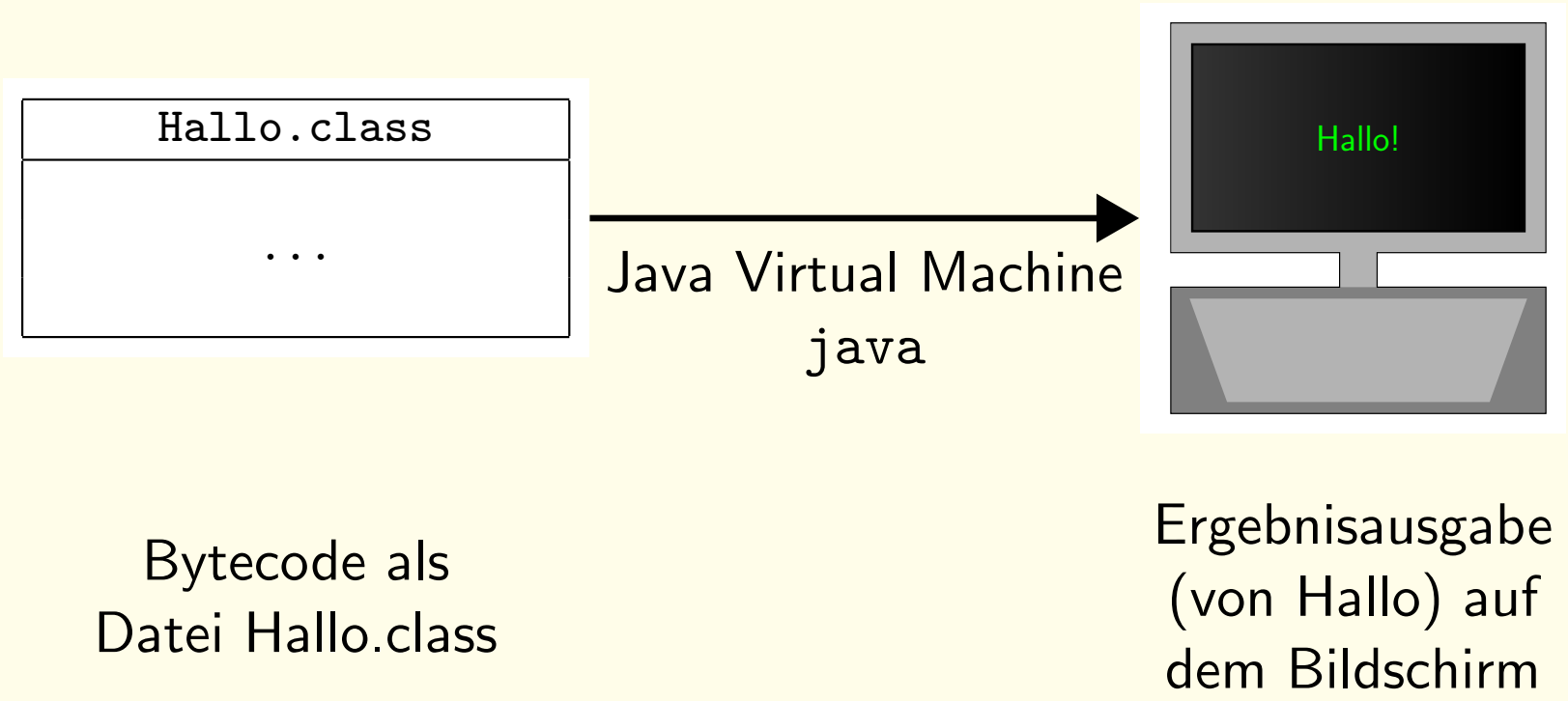
Übersetzung in Bytecode (für die Java Virtual Machine (JVM))

- Aus einer Textdatei mit Endung „`.java`“ erzeugt der Compiler `javac` eine Datei mit gleichem Namen, aber Endung „`.class`“
- Diese enthält den Bytecode für die JVM.



Ausführung von Java-Programmen

Die Datei mit dem Bytecode wird der JVM übergeben und von der JVM ausgeführt.



Übersetzung und Ausführung von Hallo.java

Übersetzung von Hallo.java:

```
> javac Hallo.java
```

Unter Windows (falls Systemvariable zum Auffinden von javac nicht gesetzt sind):

```
> "C:\Program Files\Java\jdk1.8.0_181\bin\javac.exe" Hallo.java
```

Ausführung von Hallo.class:

```
> java Hallo
```

Gibt auf Bildschirm aus:

```
Hallo!
```

Korrektheit von Programmen

- **Syntaktische Korrektheit:** Ein Programm ist syntaktisch korrekt, wenn es gemäß den (syntaktischen) Regeln der Programmiersprache geschrieben ist.
- **Semantische Korrektheit:** Ein Programm ist semantisch korrekt, wenn bei der Ausführung des Programms die gewünschte Wirkung erzielt wird.

Beachte: Syntaktisch korrekte Programme sind häufig semantisch nicht korrekt.

Man spricht von

- normaler Software bei bis zu 25 Fehlern pro 1000 LOC (lines of code), also 2,5% Defektniveau,
- guter Software bei bis zu 2 Fehlern pro 1000 LOC (lines of code), also 0,2% Defektniveau.

Fehler

Analog zur Korrektheit kann man auch die **Fehler** unterscheiden:

- Syntaxfehler: Der Quellcode entspricht nicht der Syntax der Programmiersprache; Syntaxfehler nennt man auch Parsefehler (parse error))
  Z.B. falsches Zeichen, fehlende Klammern, ...
- Logische / Semantische Fehler: Das Programm implementiert die falsche Funktionalität
- Typfehler: Der Code ist syntaktisch korrekt, aber die Typen passen nicht.

Fehler (2)

Unterscheiden nach Zeitpunkt des Auftretens

- Compilezeitfehler: Fehler, die bereits vom Compiler / Interpreter entdeckt werden, und daher in einer Fehlermeldung enden.
- Laufzeitfehler: Fehler, die erst zur Laufzeit auftreten, und daher nicht vom Compiler/Interpreter schon erkannt werden. Z.B. Division durch 0, Datei lesen, die nicht existiert, etc.

Java und Fehler

- In Java werden viele Fehler schon beim Compilieren entdeckt
- Der Compiler liefert Fehlermeldungen $\Rightarrow$ genau lesen!

Beispiel: Syntax-Fehler

```
public class DreiSyntaxFehler {
    public static void main(String[] args {
        System.out.println("Hallo!")
    }
```

Java-Compiler erkennt Syntax-Fehler:

```
> javac     DreiSyntaxFehler.java:
DreiSyntaxFehler.java:2: error: ')' expected
   public static void main(String[] args {
                                         ^
DreiSyntaxFehler.java:3: error: ';' expected
       System.out.println("Hallo!")
                                       ^
DreiSyntaxFehler.java:4: error: reached end of file while parsing
   }
    ^
3 errors
```

Beispiel: Typ-Fehler

```
public class TypFehler {
    public static void main(String[] args) {
        System.out.println("Hallo!" - 10);
    }
}
```

Java-Compiler erkennt (oft) Typ-Fehler:

```
> javac TypFehler.java
TypFehler.java:3: error: bad operand types for binary operator '-'
       System.out.println("Hallo!" - 10);
                                   ^
  first type:  String
  second type: int
1 error
```

Beispiel: Logischer Fehler

```
public class LogikFehler {
    public static void main(String[] args) {
        System.out.print("7 zum Quadrat ergibt: ");
        System.out.println(2*7);
    }
}
```

Java-Compiler erkennt keine Fehler:

```
> javac LogikFehler.java

> java LogikFehler
7 zum Quadrat ergibt: 14
```

Beachte: Logik-Fehler findet der Compiler i.A. nicht.

Laufzeitfehler

```
public class Laufzeitfehler {
    public static void main(String[] args) {
        System.out.println(4 / 0);
    }
}
```

Java-Compiler erkennt keine Fehler:

```
> javac Laufzeitfehler.java

> java Laufzeitfehler
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Laufzeitfehler.main(Laufzeitfehler.java:3)
```

Offensichtlich: Laufzeitfehler erkennt der Compiler **nicht**
(sonst wären sie Compilezeitfehler)

Kommentare in Programmen

> *"The view that documentation is something that is added to a program after it has been commissioned seems to be wrong in principle, and counterproductive in practice. Instead, documentation must be regarded as an integral part of the process of design and coding."*

Tony Hoare: Hints on Programming Language Design (1973)

Kommentare in Java-Programmen: Zeilenkommentare

Durch

```
// Hier steht ein Kommentar
```

wird der Text ab // **bis zum Ende der Zeile** zum Kommentar

Beispiel:

```
...
int kontostand = 0; // Kontostand mit 0 initalisieren
kontostand +=  100; // Kontostand um 100 erhoehen
...
```

Kommentare in Java-Programmen: Mehrzeilenkommentare

Mehrzeilige Kommentare fangen mit /* an und hören mit */ auf.

Beispiel:

```
public class HalloMehrzeiligerKommentar {
    public static void main(String[] args) {
        System.out.println("Hallo!");
        /* Wir schreiben hier einen Kommentar
           der ueber mehrere Zeilen geht.
           Wo hoert er nur auf?
        System.out.println("Nocheinmal: Hallo!");
           Was druckt dieses Programm nur aus?
        */
        System.out.println("Ein letztes Mal: Hallo!");
    }
}
```

Kommentare in Java-Programmen: Mehrzeilenkommentare

Mehrzeilige Kommentare fangen mit /* an und hören mit */ auf.

Beispiel:

```
public class HalloMehrzeiligerKommentar {
    public static void main(String[] args) {
        System.out.println("Hallo!");
        /* Wir schreiben hier einen Kommentar
           der ueber mehrere Zeilen geht.
           Wo hoert er nur auf?
        System.out.println("Nocheinmal: Hallo!");
           Was druckt dieses Programm nur aus?
        */
        System.out.println("Ein letztes Mal: Hallo!");
    }
}
```

Die Klasse Hallo kommentiert

```
/*
  Diese Klasse dient zum Anzeigen des
  Strings "Hallo!" auf dem Bildschirm
*/
public class Hallo {
  /*
    Die Methode main gibt aus "Hallo!"
  */
  public static void main(String[] args) {
      System.out.println("Hallo!"); // drucke Hallo
  }
}
```

HTML-Dokumentation erzeugen

Das Programm `javadoc` erzeugt aus Java-Quellcode eine HTML-Dokumentation.
Dabei werden nur sogenannte Doc-Kommentare sichtbar

```
/**
  * Ein Doc-Kommentar
  */
```

Javadoc Tags Spezielle Schlüsselwörter, die mit @ beginnen und in der Dokumentation verwendet werden können

`@author`, `@param`, `@see`, `@version`, ...

Beispiel mit JavaDoc-Kommentaren

```
/**
  * Diese Klasse dient zum Anzeigen des
  * Strings "Hallo!" auf dem Bildschirm
  *
  * @author D. Sabel
  * @version 1.0
  *
  */
public class HalloDokumentiert {
    /**
      * Die Methode main gibt aus "Hallo!"
      */
      public static void main(String[] args) {
          System.out.println("Hallo!");
      }
}
```

Aufruf

```
> javadoc -author -version HalloDokumentiert.java
```

Erzeugte Webseite

Class HalloDokumentiert

java.lang.Object
    HalloDokumentiert

```
public class HalloDokumentiert
extends java.lang.Object
```

Diese Klasse dient zum Anzeigen des Strings "Hallo!" auf dem Bildschirm

Version:
1.0

Author:
D. Sabel

Constructor Summary

Constructors

Constructor and Description
`HalloDokumentiert()`

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method and Description
static void	`main(java.lang.String[] args)` Die Methode main gibt aus "Hallo!"

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

HalloDokumentiert

```
public HalloDokumentiert()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Die Methode main gibt aus "Hallo!"

Parameters:
`args` - Ein Parameter für Kommandozeilenparameter

JavaDoc-Kommentare reichen nicht aus!

JavaDoc-Kommentare dienen zur Dokumentation

- der Funktionalität von Programmen
- den Schnittstellen
  (wie verwendet man die Klasse, Methode, . . .)

Aber:

- Sie dokumentieren **nicht**, wie die interne Implementierung funktioniert.
- Hierfür sind „normale“ Java-Kommentare notwendig.
- Kommentare damit, der Code **für andere Programmierer** sowohl **nachvollziehbar** als auch **wartbar** und **anpassbar** ist.

Zusammenfassung (1)

Wesentliche, einführende Begriffe:

- Informatik
- Information
- Binärdarstellung
- Von-Neumann-Rechner
- Programm
- Algorithmus
- Compiler
- Interpreter

Zusammenfassung (2)

- Programmierparadigmen (imperativ, prozedural, objektorientiert, deklarativ, funktional, logisch)
- Typisierung (statisch, dynamisch, stark, schwach)
- Java: Entwicklungsgeschichte und wichtige Eigenschaften (imperativ, objektorient, plattformunabhängig)
- Grundgerüst eines Java-Programms
- Übersetzung und Ausführung von Java-Programmen
- Korrektheit von Programmen (syntaktisch, semantisch)
- Fehler: Syntaxfehler, Typfehler, Logischer Fehler
- Java-Programme kommentieren und dokumentieren