Tagesprogramm

Ein- und Ausgabe

Zusicherungen und Programmverstehen

Standardein- und -ausgabe

Standardausgabe:

```
System.out.print(...);
System.out.println(...);
```

Standardfehlerausgabe:

```
System.err.print(...);
System.err.println(...);
```

Standardeingabe:

```
Scanner scanner = new Scanner(System.in);
String line = scanner.nextLine();
```

Programmparameter

```
public static void main(String[] args) {...}
```

Parameter in `args` vor der Verwendung prüfen

Dateien öffnen

ungepuffert: einfacher zu verwenden

```
FileReader rawin = new FileReader("InName");
FileWriter rawout = new FileWriter("OutName");
```

gepuffert: effizienter

```
BufferedReader in = new BufferedReader(rawin);
BufferedWriter out = new BufferedWriter(rawout);
```

Streams: `rawin`, `rawout`, `in`, `out` müssen am Ende geschlossen werden:

```
in.close(); out.close();
```
(auf `rawin` und `rawout` weitergeleitet)

I/O-Exceptions müssen abgefangen werden (auch die von `close`):

```
... catch(IOException ex) {...}
```

Aufgabe: Wie funktioniert Numbered?

Such Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Fragen:

Warum braucht Numbered zwei try-Anweisungen?

Warum wurden in und out zuerst mit null initialisiert?

Werden in und out auch geschlossen wenn beim Lesen oder Schreiben eine Exception geworfen wird?

Zeit: 3 Minuten

Programm verstehen $\neq$ nachvollziehen

Programmablauf nachvollziehen:

dynamischer Ablauf
nur für bestimmte Datenmengen
durch Ausprobieren überprüfbar
(kann sehr aufwendig sein)

Programm verstehen:

statisches Verstehen des Programmcodes
für alle möglichen Datenmengen
durch formale Beweise überprüfbar
(für bestimmte Aspekte effizient)

Hoare-Logik

{A} P {B}

P ist ein **Programmstück**;
wenn **Vorbedingung** A vor Ausführung von P gilt,
dann gilt **Nachbedingung** B nach Ausführung von P

Beispiel: $\{y \geq 1\}$ `x = y + 1;` $\{x > 1 \land y \geq 1\}$

wenn zwei der drei Teile (Vorbedingung, Programmstück, Nachbedingung) bekannt, kann ein passender dritter Teil gefunden bzw. berechnet werden

Bedingungen können Parameter (im mathematischen Sinn) enthalten

Zusammensetzen von Programmstücken

Sequenz:

wenn $\{A\}\ P\ \{B\}$ und $\{B\}\ Q\ \{C\}$ gelten,
dann gilt auch $\{A\}\ P;Q\ \{C\}$

Auswahl:

wenn $\{A \wedge B\}\ P\ \{C\}$ und $\{A \wedge \neg B\}\ Q\ \{C\}$ gelten,
dann gilt auch $\{A\}$ `if(B){P}else{Q}` $\{C\}$

Wiederholung:

wenn $\{A\}\ P\ \{A\}$ gilt, (**Schleifeninvariante A**)
dann gilt auch $\{A\}$ `while(B){P}` $\{A \wedge \neg B\}$

Schleifen und Zusicherungen

```
// berechne 1 + ··· + n
// n > 0
public static int sum(int n) {
    int s = n;
    int i = n - 1;
    // i >= 0 und 1 + ··· + n == (1 + ··· + i) + s
    while (i != 0) {
        // i > 0 und 1 + ··· + n == (1 + ··· + i) + s
        s = s + i;
        i = i - 1;
        // i >= 0 und 1 + ··· + n == (1 + ··· + i) + s
    }
    // i == 0 und 1 + ··· + n == (1 + ··· + i) + s
    // daraus folgt: 1 + ··· + n == s
    return s;
}
```

In der Praxis

Zusicherungen zwischen Programmzeilen meist nur gedacht,
aber bei Methodenköpfen stehen explizite Zusicherungen
(als Beschreibungen der Methoden)

Grundvoraussetzung für Programmverstehen:

gute Schleifeninvarianten finden

komplizierte Schleifeninvarianten explizit hinschreiben

`assert`-Anweisungen reichen meist nicht aus

Aufgabe: Zusicherungen

Versehen Sie dieses Programmstück mit Zusicherungen (als Kommentare):

```
// array ist aufsteigend sortiert;  ist elem in array?
public static boolean binarySearch(int[] array, int elem) {
    int first = 0;
    int last = array.length - 1;
    while (first <= last) {
        int current = (first + last) / 2;
        if (array[current] > elem) {
            last = current - 1;
        } else if (array[current] == elem) {
            return true;
        } else {
            first = current + 1;
        }
    }
    return false;
}
```