API, Dokumentation, Logik und Kontrakte

Programmierung und Softwareentwicklung
Hörsaalübung

Schnittstellen

Schnittstellen (I) – Aufgabe

Diskutieren Sie mit Ihrem Nachbarn, was eine Schnittstelle sein könnte und welche Arten von Schnittstellen es in einem Programm geben kann.

Schnittstellen (II)

Was ist eine Schnittstelle?

- Was ich von einer Klasse von außen sehe und auf Objekte aufrufe
- Subsysteme interagieren miteinander

  → Schnittstelle legt die Möglichkeit der Interaktion fest
- Bspw: GUI, API, …
- Schnittstellen sind in der Regel für den Endanwender nutzbar
- Klassenschnittstelle sind die Operationen

Schnittstellen (III)

Graphical User Interface (GUI)

- Graphische Benutzeroberfläche, z.B.: mit Knöpfen zur Steuerung des Programms

Application Programm Interface (API)

- Nach außen sichtbaren, aufrufbaren Klassen und Operationen
- Dokumentation der Schnittstelle
- Benutzer kann mit Schnittstelle interagieren, ohne konkrete Kenntnisse über die Implementierung besitzen zu müssen
- https://docs.oracle.com/en/java/javase/11/

Ausschnitt aus String API (I)

Constructors

Constructor	Description
`String()`	Initializes a newly created `String` object so that it represents an empty character sequence.
`String(byte[] bytes)`	Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.
`String(byte[] ascii, int hibyte)`	**Deprecated.** This method does not properly convert bytes into characters.
`String(byte[] bytes, int offset, int length)`	Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.
`String(byte[] ascii, int hibyte, int offset, int count)`	**Deprecated.** This method does not properly convert bytes into characters.
`String(byte[] bytes, int offset, int length, String charsetName)`	Constructs a new `String` by decoding the specified subarray of bytes using the specified charset.
`String(byte[] bytes, int offset, int length, Charset charset)`	Constructs a new `String` by decoding the specified subarray of bytes using the specified charset.

Ausschnitt aus String API (II)

All Methods	Static Methods	Instance Methods	Concrete Methods	Deprecated Methods

Modifier and Type	Method	Description
char	**charAt**(int index)	Returns the char value at the specified index.
IntStream	**chars**()	Returns a stream of int zero-extending the char values from this sequence.
int	**codePointAt**(int index)	Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index)	Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex)	Returns the number of Unicode code points in the specified text range of this String.
IntStream	**codePoints**()	Returns a stream of code point values from this sequence.
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.
boolean	**contains**(**CharSequence** s)	Returns true if and only if this string contains the specified sequence of char values.

Ausschnitt aus String API (III)

concat

`public String concat(String str)`

Concatenates the specified string to the end of this string.

If the length of the argument string is `0`, then this `String` object is returned. Otherwise, a `String` object is returned that represents a character sequence that is the concatenation of the character sequence represented by this `String` object and the character sequence represented by the argument string.

Examples:

```
"cares".concat("s") returns "caress"
"to".concat("get").concat("her") returns "together"
```

Parameters:

`str` - the `String` that is concatenated to the end of this `String`.

Returns:

a string that represents the concatenation of this object's characters followed by the string argument's characters.

Dokumentation

Kommentare

Warum Kommentare?

- Teile im Programmcode, die keine Auswirkungen auf die Funktion des Programms haben
- Dienen der Dokumentation für zukünftige Entwickler, die den Code erweitern oder überarbeiten sollen
- Unerlässlich, weil noch so guter Code in großen Kontexten unverständlich ist
- Fehlende JavaDoc Kommentierung wird mit massivem Punktabzug bestraft. Ihr wurdet gewarnt!
- Drei Arten von Kommentaren: Zeilenkommentare, Blockkommentare, und JavaDoc-Kommentare

Zeilenkommentar

`System.out.println()`
kann etwas auf der Konsole ausgeben

- Kommentierung innerhalb des Codes
- Auch Implementierungskommentar genannt
- Alles hinter // wird für Programmfluss ignoriert

```
public class Raccoon {
    // ...
    public void sayHello() {
        // Prints 'Hello' on command line
        System.out.println("Hello");
    }
}
```

Blockkommentar

- Wie Zeilenkommentar, nur für mehrere Zeilen
- Alles zwischen /* und */ wird ignoriert

```
public int nthFibonacci(int n) {
    int first = 1;
    int second = 1;
    /*
     * Adds first and second element and stores the solution in temp.
     * Then assigns second to first and temp to second.
     * This is done until the n-th fibonacci number is reached.
     */
    for(int i = 2; i <= n; i++) {
        int temp = first + second;
        first = second;
        second = temp;
    }
    return second;
}
```

JavaDoc-Kommentar

- Mehrzeilige Kommentare, die der Kommentierung von Funktionen und Klassen dienen
- Alles zwischen /** und */ wird ignoriert
- Aus JavaDoc-Kommentaren kann eine automatische Dokumentation erzeugt werden
- Eclipse interpretiert JavaDoc und zeigt sie beim Eingeben kommentierter Elemente an
- Mit HTML Tags kann der JavaDoc Kommentar formatiert werden

JavaDoc-Kommentar

```java
/**
 * Entity for a raccoon.
 * @author spethso
 * @version 1.0
 */
public class Raccoon {
    // ...
    /**
     * Let's the racoon eat a given food.
     * @param food - The food object to eat
     */
    public void eat(Food food) {
        // ...
    }
    // ...
}
```

JavaDoc-Kommentar

Tags

- Erlaubt ein hinterlegen von Metadaten, z.B.: Namen des Autors
- Optional, aber gute Stil
- Dokumentiert die Klasse oder Operation möglichst einfach

Für Klassen und Interfaces

- `@author` Gibt den Namen des Autors an
- `@version` Gibt die Version der Klasse / des Interfaces an
- `@deprecated` Zeigt an, dass die API nicht mehr in Verwendung ist
- `@since` Gibt das Datum des Releases an

JavaDoc-Kommentar

Tags

- Erlaubt ein hinterlegen von Metadaten, z.B.: Namen des Autors
- Optional, aber gute Stil
- Dokumentiert die Klasse oder Operation möglichst einfach

Für Operationen

- `@param` Gibt den Namen des Parameters und eine Beschreibung dessen an
- `@return` Gibt an, welche Rückgabe zu erwarten ist
- `@throws` Gibt an, welche Exception geworfen werden kann
- `@see` Gibt eine Referenz an

JavaDoc – Styleguide für Operationen

Abstract

- Beschreibung in einem kurzen Satz, was die Operation macht

Detaillierte Beschreibung

- Detaillierte Beschreibung in einem Absatz

Vor- und Nachbedingungen

- Welche Bedingungen müssen erfüllt sein, damit die Operation ausgeführt werden darf?
- Welche Bedingungen gelten, sofern die Vorbedingungen erfüllt waren, und die Operation ausgeführt wurde

Tags, sofern anwendbar

Logik

Boolsche Logik (I)

Was sind Boolsche Werte?

- true
- false

Welche Operationen kann ich auf Boolschen Werten ausführen?

- und (&&), bitweises und (&)
- oder (||), bitweises oder (|)
- Xor (^)
- nicht (!)

Boolsche Logik (II)

A	B	A && B	A || B	A ^ B	!A
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

Boolsche Logik – Spezialfälle

Bitweises Und

- Beide Bereiche werden immer ausgeführt
- Beide Bereiche müssen wahr sein

Bitweises Oder

- Beide Bereiche werden immer ausgeführt
- Einer von beiden Bereichen muss wahr sein

Vergleiche bei Objekten

- Keine Möglichkeit Objekte mit == zu vergleichen (Achtung, kein Compilefehler)

Vergleichsoperationen

Operationen	A	B	Wahrheitswert
<	1	2	true
>	1	2	false
<=	2	2	true
>=	2	3	false
==	2	2	true
!=	2	2	false

Boolsche Logik – Aufgabe (I)

I. Welche Ausgabe liefert das Programm?

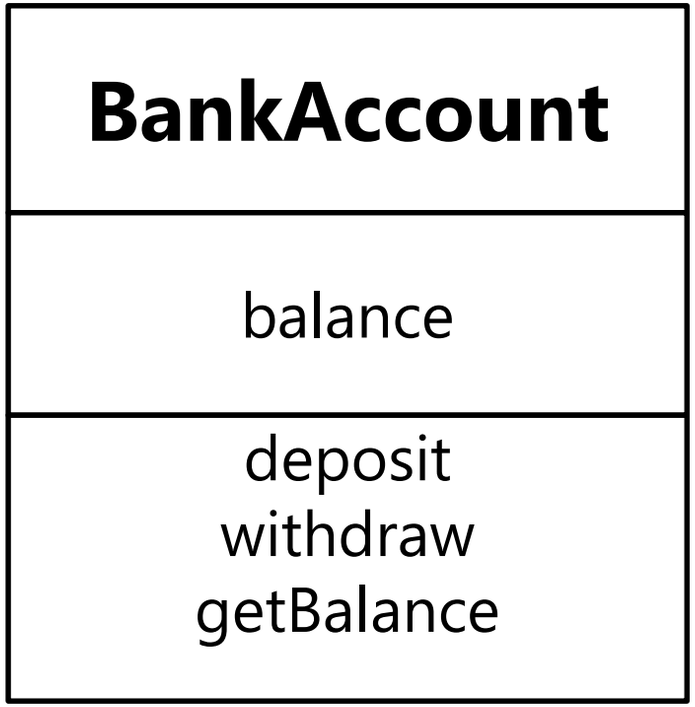
```
foo bar          result: true
foo              result: false
foo bar          result: true
foo bar          result: false
foo              result: true
foo bar          result: true
foo bar          result: true
foo bar          result: true
foo bar          result: false
foo bar          result: true
```

```java
public class BooleanLogic {

    public static boolean foo(boolean b) {
        System.out.print("foo ");
        return b;
    }

    public static boolean bar(boolean b) {
        System.out.print("bar ");
        return b;
    }

    public static void main(String[] args) {
        // And
        System.out.print("\tresult: " + (foo(true) && bar(true)) + "\n");
        System.out.print("\t\tresult: " + (foo(false) && bar(true)) + "\n");

        // Bitwise-and
        System.out.print("\tresult: " + (foo(true) & bar(true)) + "\n");
        System.out.print("\tresult: " + (foo(false) & bar(true)) + "\n");

        // Or
        System.out.print("\t\tresult: " + (foo(true) || bar(false)) + "\n");
        System.out.print("\tresult: " + (foo(false) || bar(true)) + "\n");

        // Bitwise-or
        System.out.print("\tresult: " + (foo(true) | bar(false)) + "\n");
        System.out.print("\tresult: " + (foo(false) | bar(true)) + "\n");

        // Xor
        System.out.print("\tresult: " + (foo(true) ^ bar(true)) + "\n");
        System.out.print("\tresult: " + (foo(true) ^ bar(false)) + "\n");
    }
}
```

Problem

Was passiert, wenn bei `withdraw` der `BankAccount` nicht genügend Geld besitzt?

Was passiert, wenn bei `withdraw` ein negativer Betrag abgehoben werden soll?



Was passiert, wenn bei `deposit` ein negativer Betrag eingezahlt werden soll?

Kontrakte

Kontrakte

Kontrakte

- Geben eine Art Vertrag mit der Schnittstelle an
- Dokumentiert die Schnittstelle mit semantischen Bedingungen
  - Benutzer muss seine Seite des Vertrags erfüllen, um Schnittstelle nutzen zu dürfen
  - Benutzer weiß sicher, welcher Zustand das System nach der Nutzung haben kann
- Hilfreich für Debugging
- Drei Arten von Kontrakten
  - Vorbedingung
  - Nachbedingungung
  - Klasseninvarianten
- Problem: Java bietet nativ keine Möglichkeit Kontrakte anzugeben

Java Modeling Language

Java Modeling Language (JML)

- Dokumentation der Kontrakte in Kommentaren
- Kommentarzeilen, die mit @ Annotiert sind, starten einen JML Ausdruck

OpenJML

- https://www.openjml.org/
- Verifikationstool für Java Programme
  - Überprüft die Annotierten Kontrakte der Java Programme

Vorbedingung

- Gibt Zustände an, die vor der Ausführung der Operation erfüllt sein müssen
- Annotation des Kommentars mit `requires` Schlüsselwort
- Operationen ohne `requires` Schlüsselwort erfüllen immer die Vorbedingung
  - Sind immer aufrufbar

```
/*@
 @ requires amount >= 0.0;
 */
public void deposit(double amount) {
    // ...
}
```

Vorbedingung

Prinzip

- Aufrufe, ohne Erfüllung der Vorbedingung sind fehlerhaft
- Aufrufende Einheit sollte vor dem Aufruf sicherstellen, dass die Vorbedingung erfüllt ist

Nachbedingung

- Gibt Zustände an, die nach der Ausführung der Operation erfüllt sein werden
- Annotation des Kommentars mit `ensures` Schlüsselwortes
  - `\old` Schlüsselword ermöglicht Zugriff auf den Wert einer Abfrage, vor der Ausführung des Programms
  - `\result` beschreibt Ergebnis (return Wert) der Operation

```
/*@
 @ ensures getBalance() == \old(getBalance()) + amount;
 */
public void deposit(double amount) {
    // ...
}
```

Nachbedingung

Prinzip

- Programme, die die Erfüllung einer Nachbedingung nicht sicherstellen können, sind fehlerhaft
- Operation muss sicherstellen, dass Nachbedingung erfüllt werden wird, falls Vorbedingung erfüllt wurde

Vor- und Nachbedingung

- Es können mehrere Bedingungen vorhanden sein
- Ausdrücke können logisch verknüpft werden

```
/*@
 @ ensures !d.isLocked() ==>
 @ d.getOpenStatus() != \old(d.getOpenStatus())
 @ && d.getOpenStatus() == "open";
 @ ensures d.isLocked() ==>
 @ d.getOpenStatus() == \old(d.getOpenStatus())
 @ && d.getOpenStatus() == "closed";
 */
public boolean tryToOpenDoor(Door d) {
    // ...
}
```

Klasseninvarianten

- Eigenschaften, die zwischen der Ausführung der Operationen immer erfüllt sein müssen
- Annotation des Kommentars mit `invariant` Schlüsselwort
  - Zeile muss mit @ beginnen

```
//@ public instance invariant getBalance() >= 0;
```

Weitere JML Ausdrücke und Schlüsselwörter

Ausdrücke

- `\foreall`

Schlüsselwörter

- `assignable` zeigt welche Attribute gesetzt werden können
- `pure` gibt an, dass die Operation keine Seiteneffekte besitzt, z.B.: Getter
- `non_null` Objekt ist nicht null
- `signals` kann Nachbedingungen für Exceptions ausdrücken
- `normal_behavior` gibt das normale Verhalten an
  - Alle Exceptions werden dadurch ausgeschlossen

Aufgabe 1

Implementieren Sie die Klasse `BankAccount`. Geben Sie dabei insbesondere die Vor- und Nachbedingungen der Operationen, sowie Klasseninvarianten von `BankAccount` an.

Diskutieren Sie mit Ihrem Nachbarn die Lösung und passen Sie gegebenenfalls Ihre Kontrakte an.

Anmerkungen

- `balance` ist vom Typ `double`
- `withdraw` bekommt ein `double` übergeben und gibt ein boolschen Wert zurück
- `deposit` bekommt ein `double` übergeben
- `getBalance` gibt den Wert von `balance` zurück

Aufgabe 2

Wir befinden uns im Kontext der PSE Klausur.

a) Überlegen Sie sich, welche Vorbedingungen Sie erfüllen müssen, um an der PSE Klausur teilnehmen zu dürfen
b) Überlegen Sie sich, welche Nachbedingungen die Teilnahme an der Klausur besitzen
c) Gibt es irgendwelche Invarianten, die erfüllt sein müssen?