Einführung in die Informatik

Controlling Behavior

Das `if`-Statement

Wolfram Burgard

Motivation

- Bisher bestanden die Rümpfe unserer Methoden aus einzelnen Statements, z.B. Wertzuweisungen oder Methodenaufrufen.
- Es gibt bisher keine Möglichkeit, Statements nur in Abhängigkeit bestimmter Umstände auszuführen.
- In diesem Kapitel behandeln wir **bedingte Anweisungen**, die es erlauben, Statements in Abhängigkeit davon auszuführen, dass eine bestimmte **Bedingung erfüllt ist**.
- Dadurch können wir **flexiblere Methoden** schreiben und **deutlich mächtigere Modelle** entwickeln.

Das `if`-Statement

- Java stellt mit dem **`if`-Statement** eine Form der **bedingten Anweisung** zur Verfügung.
- Mit Hilfe des **`if`-Statements** können wir eine **Bedingung** testen und, je nach Ausgang des Tests, eine von zwei Anweisungen durchführen.

```
if (axles == 2)
        tollDue = 4;
else
        tollDue = 5 * axles;
```

- Zeile 1 enthält den Test, den wir ausführen.
- Zeile 2 enthält das Statement, das bei erfolgreichem Test ausgeführt wird.
- Zeile 3 enthält das Schlüsselwort `else` und läutet den Teil ein, der ausgeführt wird, wenn der Test fehlschlägt.
- Zeile 4 enthält das Statement, welches bei negativem Ausgang des Tests ausgeführt wird.

Anwendungsbeispiel: Lohn und Gehalt

Aufgabe:

- Modellieren Sie ein Lohnbuchhaltungssystem für Mitarbeiter, die auf einer Stundenbasis bezahlt werden.
- Das System sollte in der Lage sein, den Lohn eines Mitarbeiters für eine Woche aus seinem Grundlohn und den gearbeiteten Stunden zu bestimmen.
- Mitarbeiter, die mehr als 40 Stunden pro Woche gearbeitet haben, bekommen die Überstunden mit 150% bezahlt.
- Falls ein Mitarbeiter in zwei aufeinanderfolgenden Wochen 30 oder mehr Überstunden absolviert hat, soll eine Warnung ausgegeben werden.

Ziel:

- Modellierung dieser Anwendung mit gleichzeitiger Demonstration der Verwendung des `if`-Statements.

Beispielsitzung

- Zunächst wird der Name des Mitarbeiters und sein Gehalt ausgegeben.
- Dann werden die Berechnungen für die einzelnen, aufeinanderfolgenden Wochen durchgeführt und die Ergebnisse ausgegeben.

```
Employee name: Gerald Weiss
Employee rate/hour: 20


Gerald Weiss earned 600 Dollar for 30 hours
Gerald Weiss earned 1100 Dollar for 50 hours
Gerald Weiss has worked 30 or more hours of overtime
Gerald Weiss earned 1400 Dollar for 60 hours
```

Benötigte Objekte

- In unserer Anwendung tauchen die Begriffe *Mitarbeiter*, *Stunden*, *Lohn* und *Gehalt* auf.
- Davon ist *Mitarbeiter* der wichtigste Begriff.
- Um Mitarbeiter zu modellieren, führen wir eine Klasse `Employee` ein:

```
class Employee {
   // benötigte Methoden und Instanzvariablen
...
}
```

Erforderliches Verhalten

- Wir müssen das **Gehalt berechnen**, welches an den Mitarbeiter ausgezahlt werden soll und verwenden dafür eine Methode `calcPay`.
- Wir benötigen den **Namen des Mitarbeiters** und definieren dafür eine Methode `getName`.
- Offensichtlich brauchen wir auch einen **Konstruktor** `Employee`, um `Employee`-Objekte zu **erzeugen**.

Das Schnittstellen des Konstruktors `Employee`

- Wenn wir ein Objekt der Klasse `Employee` erzeugen wollen, müssen wir die Daten wissen, die zur Gehaltsberechnung notwendig sind.
- **In unserer Anwendung** sind das der **Name** und das **Gehalt pro Stunde**.
- Um ein `Employee`-Objekt zu erzeugen, müssen wir also folgenden Code hinschreiben:

```
Employee e;
e = new Employee("Rudy Crew", 10);
```

Die Schnittstellen der Methoden `calcPay` und `getName`

- Um das Gehalt für eine Woche zu berechnen, benötigen wir die Anzahl der gearbeiteten Stunden.
- Das Ergebnis der Lohnberechnung ist der auszuzahlende Betrag.
- Unser System modelliert Beträge durch ganze Zahlen vom Typ `int`.
- Die Methode `calcPay` wird daher folgendermaßen verwendet:

```
int pay;
pay = e.calcPay(30);
```

- Die Verwendung der `getName`-Methode wiederum ist einfach:

```
System.out.print(e.getName());
```

Die komplette Schnittstelle

Insgesamt ergibt sich folgende Beispielanwendung:

```
class Payroll {
      public static void main(String a[]) {
            Employee e;
            e = new Employee("Rudy Crew", 10);
            int pay;
            int hours = 30;
            pay = e.calcPay(hours);
            System.out.print(e.getName());
            System.out.print(" earned ");
            System.out.print(pay);
            System.out.print(" Dollars for ");
            System.out.print(hours);
            System.out.println(" hours");
      }
}
```

Die Prototypen unserer Methoden

Aus der Schnittstelle erhalten wir unmittelbar die Prototypen:

```
class Employee {
      public Employee(String name, int rate) {...}
      public int calcPay(int hours) {...}
      public String getName() {...}
      // Instance variables to be supplied
}
```

Erforderliche Instanzvariablen

- Ein `Employee`-Objekt wird erzeugt mit dem **Namen** des Mitarbeiters und seinem **Gehalt**.
- Da **beide Werte von Methoden benötigt** werden, müssen wir sie in **Instanzvariablen** ablegen.
- Hat der Mitarbeiter zu viele Überstunden in den letzten beiden Wochen durchgeführt, so soll eine Warnung ausgegeben werden. Da die **Anzahl der Überstunden in der Vorwoche** nicht an `calcPay` übergeben wird, müssen wir auch diesen Wert in einer **Instanzvariable** ablegen.

```
class Employee {
      // Methods
      ...
      // Instance variables
      private String name;
      private int rate;
      private int lastWeeksOvertime;
}
```

Implementierung: Der `Employee`-Konstruktor

Die **Aufgabe des Konstruktors** ist die **Initialisierung der Instanzvariablen** eines `Employee`-Objektes:

```
public Employee(String name, int rate) {
      this.name = name;
      this.rate = rate;
      this.lastWeeksOvertime = 0;
}
```

Implementierung: Die `getName`-Methode

Die Methode `getName` gibt einfach den **Wert der Instanzvariable** `name` **dieses Objektes** zurück.

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

Implementierung: Die Methode `calcPay` (1)

- Die Methode `calcPay` erfüllt zwei Aufgaben gleichzeitig:
  1. Sie berechnet den Wochenlohn.
  2. Sie gibt eine Warnung aus bei zu vielen Überstunden.
- Der Wochenlohn lässt sich einfach berechnen, wenn **keine Überstunden** vorliegen:

```
pay = hours * this.rate; // No overtime
currentOvertime = 0;
```

- Wenn mehr als 40 Stunden gearbeitet wurden, dann werden die ersten 40 Stunden normal und alle darüber hinausgehenden Stunden mit dem 1.5-fachen Stundenlohn vergütet:

```
pay = 40 * this.rate + (hours - 40)*(this.rate + this.rate/2);
currentOvertime = hours - 40;
```

Implementierung mit einer `if`-Anweisung

```
public int calcPay(int hours){
      int pay, currentOvertime;

      if (hours <= 40) {
            pay = hours * rate;
            currentOvertime = 0;
      }

      else {
            pay = 40*rate+(hours-40)*(rate+rate/2);
            currentOvertime = hours - 40;
      }

      ...
}
```

Implementierung: Die Methode `calcPay` (2)

- Darüber hinaus muss die Methode `calcPay` auch noch berechnen, ob eine Warnung wegen zu vieler Überstunden ausgegeben werden soll.
- Wir verwenden wiederum ein `if`-Statement, um zu bestimmen, ob eine Warnung generiert wird:

```
if (currentOvertime + this.lastWeeksOvertime >= 30) {
        System.out.print(this.name);
        System.out.println(" has worked 30 or more hours of overtime");
}
```

Implementierung: Die Methode `calcPay` (3)

- Dann muss noch der Übertrag der Überstunden berechnet werden.
- Schließlich muss der auszuzahlende Betrag als Ergebniswert zurückgegeben werden.

```
this.lastWeeksOvertime = currentOvertime;
return pay;
```

Die komplette Implementierung (1)

```
class Employee {
    public Employee(String name, int rate) {
        this.name = name;
        this.rate = rate;
        this.lastWeeksOvertime = 0;
    }
    public int calcPay(int hours) {
        int pay;
        int currentOvertime;

        if (hours <= 40) {
            pay = hours * rate;
            currentOvertime = 0;
        }
        else {
            pay = 40*rate+(hours-40)*(rate+rate/2);
            currentOvertime = hours - 40;
        }
```

Die komplette Implementierung (2)

```
        if (currentOvertime + this.lastWeeksOvertime >= 30) {
            System.out.print(this.name);
            System.out.println(
            " has worked 30 or more hours of overtime");
        }

        this.lastWeeksOvertime = currentOvertime;
        return pay;
    }

    public String getName() {
        return this.name;
    }

    private String name;
    private int rate;
    private int lastWeeksOvertime;
}
```

Verwendung der Klasse Employee

```
class Payroll {

      public static void main(String arg[]) {
            Employee e;
            e = new Employee("Rudy Crew", 10);
            int pay;
            int hours = 30;
            pay = e.calcPay(hours);
            System.out.print(e.getName());
            System.out.print(" earned ");
            System.out.print(pay);
            System.out.print(" Dollars for ");
            System.out.print(hours);
            System.out.println(" hours");
      }
}
```

Ausgabe: `Rudy Crew earned 300 Dollars for 30 hours`

Eine längere Beispielanwendung

```
e = new Employee("Rudy Crew", 10);
int pay;
int hours = 30;
pay = e.calcPay(hours);
System.out.print(e.getName());
...
hours = 50;
pay = e.calcPay(hours);
System.out.print(e.getName());
...
hours = 60;
pay = e.calcPay(hours);
System.out.print(e.getName());
...
```

Ausgabe:

```
Rudy Crew earned 300 Dollars for 30 hours
Rudy Crew earned 550 Dollars for 50 hours
Rudy Crew has worked 30 or more hours of overtime
Rudy Crew earned 700 Dollars for 60 hours
```

Diskussion der Modellierung

- Durch die Festlegung von `Employee` als das wichtigste Objekt konnten alle notwendigen Methoden und Eigenschaften sinnvoll in einer Klasse definiert werden.
- Die anderen in diesem Modell betrachteten Einheiten, wie z.B. `name` und `rate` haben eine spezielle Beziehung zu der Klasse `Employee`.
- Wir bezeichnen diese Beziehung als eine **has-a**-Relation: Ein Mitarbeiter **hat einen** Namen etc.
- Da die Wochenarbeitszeit nicht einen längerfristig gleichen Wert hat wie z.B. `name` und `rate`, wurden die Stunden, die sich permanent ändern, nicht in einer Instanzvariablen abgelegt (obwohl das prinzipiell möglich wäre).

Die zwei Formen des `if`-Statements

- Die erste Variante des `if`-Statements verzweigt in zwei unterschiedliche
- Programmstücke, die in Abhängigkeit von dem Test ausgeführt werden:

```
if (condition)
    statement1
else
    statement2
```

- Die zweite Version lässt den `else`-Teil aus und führt das Statement nur aus, wenn der Test erfolgreich ist.

```
if (condition)
    statement
```

  In dieser Version wird das Statement nur ausgeführt, wenn die Bedingung wahr ist, sonst wird die Anweisung ausgelassen.

Mehrere Anweisungen in `if`-Statements

- In der Grundversion des `if`-Statements können nur einzelne Statements im `then`**-Teil** und `else`-Teil verwendet werden.
- Sollen **mehrere Statements** ausgeführt werden, muss man diese zu einem **Block zusammenfassen**, indem man sie in Klammern ({ und }) einschließt.

```
if (x>y)  {
   System.out.print(x);
   System.out.print(" is greater than ");
   System.out.println(y);
}
else      {
   System.out.print(x);
   System.out.print(" is not greater than ");
   System.out.println(y);
}
```

zusammen-gesetzte Statements

Bedingungen in `if`-Statements

- Die **Bedingung** eines `if`-Statements muss ein **Ausdruck** sein, der entweder wahr oder falsch ist.
- Im Moment schränken wir uns auf Vergleiche zwischen Zahlwerten ein.
- Java stellt folgende **Operatoren für den Vergleich von Zahlen** zur Verfügung:

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

Kaskadierte `if`-Statements

- In der Praxis tauchen häufig Situationen auf, in denen es mehrere Alternativen gibt und in denen das Programm in mehrere Teile verzweigen muss.
- Eine einfache Variante sind so genannte Multiway-Tests, bei denen ein Ausdruck mehr als zwei Werte haben kann.
- Ein typisches Beispiel hierfür ist

```
if (hours <= 40)
      System.out.println("No overtime");
else if (hours <= 60)
      System.out.println("Overtime");
else                  // Hours > 60
      System.out.println("Double-overtime");
```

Geschachtelte `if`-Statements

- `if`-Statements lassen sich nicht nur kaskadieren sondern auch **schachteln**.
- Dadurch wird in manchen Fällen **kompakterer Code** erreicht.
- Nehmen wir an, wir wollten nur etwas ausgeben, wenn Überstunden gemacht wurden:

```
if (hours > 40)
      if (hours <= 60)
            System.out.println("Overtime");
      else
            System.out.println("Double-overtime");
```

Zu welchem `if` gehört ein `else`?

- In unserem Beispiel könnte das `else` entweder zu dem Test `hours > 40` oder zu dem Test `hours <= 60` gehören.
- Jede dieser Varianten hat ein eigenes Verhalten des Programms zur Folge.
- **Ein `else` gehört immer zu dem letzten `if`, für das noch ein `else` fehlt**.
- Unser Beispiel entspricht daher:

```
if (hours > 40) {
      if (hours <= 60)
            System.out.println("Overtime");
      else
            System.out.println("Double-overtime");
}
```

Wirkung von Klammern bei Geschachtelten `if`-Statements

- In bestimmten Fällen **müssen Klammern gesetzt werden** , um das **korrekte Verhalten** zu erreichen.
- Bei der folgenden Variante würden falsche Ausgaben erzeugt, wenn die Klammern fehlten (Double-overtime bei höchstens 40 Stunden):

```
if (hours <= 60) {
    if (hours > 40)
        System.out.println("Overtime");
    }
else
    System.out.println("Double-overtime");
```

- Beachten Sie die **Einrückung der Statements**. Diese **sollte die Zuordnung der Statements widerspiegeln**.

Das `switch`-Statement

- Für den Fall, dass in einem kaskadierten `if`-Statement **nur Tests auf Gleichheit** vorkommen, kann man mit dem `switch`-Statement eine einfachere Konstruktion verwenden.
- Die Struktur des `switch`-Statements ist

```
switch (x) {
      case value1: statement1;
                   break;
      case value2: statement2;
                   break;
      ...
      case valuen: statementn;
                   break;
      default:     statement;
                   break;
}
```

Anwendung des `switch`-Statements

- Die einzelnen **Werte** müssen **Konstanten** oder **Literale** sein.
- Im Gegensatz zum `if`-Statement werden **keine Klammern** benötigt, um verschiedene **Statements** innerhalb eines Falles zu verwenden.
- Der `default`**-Fall ist optional**.
- Jede Sequenz von Anweisungen für einen Fall soll mit der Anweisung `break` abschließen.
- Fehlt das `break`-Statement, wird mit den Anweisungen des nächsten Falles fortgefahren.

Beispielanwendung

```
switch (dayOfWeek) {
  case 2:
  case 4:  System.out.println("Heute ist Informatik-Vorlesung!");
           break;
  case 1:
  case 3:
  case 5:
  case 6:
  case 7:  System.out.println(
                    "Heute ist keine Informatik-Vorlesung!");
           break;
  default: System.out.print(
                    "Fehler: dayOfWeek hat unzulaessigen Wert: ");
           System.out.println(dayOfWeek);
           break;
}
```

`if`-Statement versus `switch`-Statement

- Im Prinzip kann jedes `switch`-Statement durch kaskadierte `if`-Statements ersetzt werden.
- `switch-Statements` sind jedoch einfacher zu lesen.
- Darüber hinaus werden `switch`-Statements häufig schneller ausgewertet als äquivalente `if`-Statements.

Anwendung des `if`-Statements: Testen auf Ende des Inputs

- Viele Methoden haben **Return-Werte**.
- Bei manchen von diesen Methoden kann jedoch nicht garantiert werden, dass tatsächlich ein Wert zurückgegeben werden kann.
- Ein Beispiel ist das Einlesen aus Dateien oder dem Internet: Wenn die Datei oder die Ressource keine Zeile (mehr) enthält, kann die Methode `readLine` kein `String`-Objekt zurückliefern.
- Für solche Fälle gibt es das **Schlüsselwort `null`**.
- Während jeder Wert einer Referenzvariablen ein Objekt referenziert, steht der Wert **`null`** für den Fall, dass **kein Objekt referenziert** wird.

```
String s;
s = br.readline();
if (s == null)
      ...                          // nothing can be read
else
      ...                          // something could be read
```

Der Typ `boolean`

- Für logischen Werte **wahr** und **falsch** gibt es in Java einen primitiven Datentyp `boolean`
- Die **möglichen Werte** von Variablen dieses Typs sind `true` und `false`.
- Wie Integer-Variablen kann man auch Variablen vom Typ `boolean` vereinbaren.
- Diesen Variablen können **Werte logischer Ausdrücke** zugewiesen werden.

Anwendung vom Typ boolean

Typische Situation:

```
boolean hasOvertime;
if (hours > 40)
      hasOvertime = true;
else
      hasOvertime = false;
...
if (hasOvertime)  // same as: if (hasOvertime == true)
      ...
```

Alternative:

```
boolean hasOvertime;
hasOvertime = (hours > 40);
...
if (hasOvertime)
      ...
```

Logische Ausdrücke

- Die Bedingung `temp < 32` ist ein Ausdruck, der einen Vergleichsoperator enthält.
- Da das Ergebnis eines solchen Vergleichs ein **Boolescher Wert** ist, bezeichnen wir Ausdrücke dieser Art als **Boolesche** oder **logische Ausdrücke**.
- Im **Bedingungsteil** einer `if`-Anweisung können beliebige **Boolesche Ausdrücke** stehen.

Logische Operatoren und zusammengesetzte logische Ausdrücke

- Häufig besteht eine Bedingung aus **mehreren Teilbedingungen** die gleichzeitig erfüllt sein müssen.
- Beispielsweise hat ein Mitarbeiter normale Überstunden absolviert, wenn er über 40 und höchstens 60 Stunden pro Woche gearbeitet hat.
- Java erlaubt es, mehrere Tests mit Hilfe **logischer Operatoren** zu einem Test zusammenzusetzten:

```
hours > 40 && hours <= 60
```

- Der **&&-Operator** repräsentiert das logische **Und**.
- Der **||-Operator** realisiert das logische **Oder**.
- Der **!-Operator** realisiert die **Negation**.

Präzedenzregeln für logische Operatoren

- Der **!-Operator** hat die höchste Präzedenz von den logischen Operatoren. Zweithöchste Präzedenz hat der **&&-Operator**. Schließlich folgt der **||-Operator**.
- Der Ausdruck

```
if (this.hours < hours ||
    this.hours == hours && this.minutes < minutes)
```

  hat daher die gleiche Bedeutung wie

```
if (this.hours < hours ||
    (this.hours == hours && this.minutes < minutes))
```

- **Durch Klammern werden Ausdrücke leichter lesbar!**

Das `if`-Statement und das Logische Und

- Der **&&-Operator** für das **logische Und** kann auch durch eine **geschachtelte `if` -Anweisung** realisiert werden.

```
if (condition1)
    if (condition2)
        statement
```

und

```
if (condition1 && condition2)
    statement
```

haben die gleiche Wirkung.

- In der Regel sind zusammengesetzte Ausdrücke jedoch einfacher zu lesen als geschachtelte `if`-Anweisungen.

Das `if`-Statement und das Logische Oder

- Sind Statements in **aufeinanderfolgenden Then-Teilen** von **kaskadierten `if`-Anweisungen** identisch, kann die `if`-Anweisung durch Verwendung des **`||`-Operators** für das **logische Oder** **vereinfacht** werden.
- Beispielsweise kann

```
if (condition1)
    statement
else if (condition2)
    statement
```

ersetzt werden durch

```
if (condition1 || condition2)
    statement
```

- Auch hier ist die zweite Variante vorzuziehen, um Wiederholungen derselben Anweisung(sfolge) zu vermeiden.

Zusammenfassung (1)

- **Bedingte Anweisungen** erlauben es, in Abhängigkeit von der Auswertung einer Bedingung im Programm **verschiedene Anweisungen durchzuführen**.
- Dadurch kann der Programmierer den **Kontrollfluss steuern** und in seinem Programm entsprechend **verzweigen**.
- Mit einem `if`**-Statement** kann man **zwei Fälle** unterscheiden.
- Das `switch`**-Statement** erlaubt das Verzweigen in **mehr als zwei Alternativen**.
- Mit Hilfe der bedingten Anweisung kann man **testen**, ob das **Ende einer Datei** erreicht ist.
- In diesem Fall liefert die `readLine`**-Methode** den Wert `null`.

Zusammenfassung (2)

- Bedingungen sind **Boolesche Ausdrücke**, die zu `true` oder `false` ausgewertet werden..
- In Java gibt es dafür den primitiven Datentyp `boolean` mit den beiden Werten `true` und `false`.
- Einfache **Boolesche Ausdrücke** können mit den **Vergleichsoperatoren** `<`, `>`, `<=`, `>=`, `==`, und `!=`, die auf Zahltypen operieren, definiert werden.
- **Komplexere Boolesche Ausdrücke** werden mit den **logischen Operatoren** `&&`, `||` und `!` zusammengesetzt.