Softwareentwicklung

Java-Grundlagen

Literale: Notation für Werte im Programmtext

Variable: Name für eine Stelle im Speicher, die einen Wert enthält

Bezeichner: Dürfen aus Unicode-Zeichen zusammengesetzt sein

Bitweise Operatoren

~ (Komplement), & (bitweises Und), `|` (bitweises Oder), `^` (bitweise exklusives Oder)

`>>` (vorzeichen-sensitiver Rechtsshift), `>>>` (vorzeichen-insensitiver Rechtsshift – es werden immer 0'en engeschoben), `<<` (Linksshift)

Typen von Grundsymbolen

Primitive Datentypen: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`

Es existieren Wrapper-Klassen für die primitiven Datentypen: `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` mit u.a. den Operationen:

Konstruktoren mit entsprechendem prim. Datentyp oder einem String als Argument

```
static String  toString(int i);
       String  toString();
       int     intValue();
       boolean equals(Integer iObj);
```

`String` (bevorzugt behandelt, aber Verweis auf Objekt!)

+-Operator ist überladen, für alle primitiven Datentypen × String → String (und auch umgekehrt), ebenfalls für alle Klassen, die eine `toString()`-Methode definieren

`String`-Operationen:

```
int length ();
int indexOf (String s);
char charAt (int index);
```
(0-basiert)
```
int compareTo (String s);
```
(negativ wenn s größer, 0 wenn `s` gleich, positiv, wenn `s` kleiner)
```
boolean equals (String s);
char[] toCharArray ();
String substring (int begin, int end);
```
(Extraktion eines Teilstrings aus einem String, von Position `begin` bis vor `end`.)

Operatorrangfolge

hohe Präzedenz	unäre Operatoren: `~ ! -- ++ + -` Type Cast
	`* / %`
	`+ -`
	`>> << >>>`
	`< > <= >= instanceof`
	`== !=`
	`&`
	`^`
	`|`
	`&&`
	`||`
	bedingter Ausdruck: `b ? e1: e2`
geringe Präzedenz	= (Zuweisung)

Typumwandlung

Erforderlich, wenn einengend (durch einen Type Cast)

Ausweitend geschieht implizit (Coercion)

```
byte → short → int → long → float → double
```

Wenn Ganzzahl-Ausdrücke vom Typ kleiner `int` mit einem Operator verbunden werden, passt der Compiler eigenmächtig den Typ auf `int` an, da Operatoren nicht für `short` und `byte` überladen sind:

```
short s1 = 1, s2 = 2, s;
s = s1 + s2;          // Compilerfehler!
s = (short) s1 + s2;  // Das ist OK
```

Arrays: sind Objekte!

```
int [] schach = { 1, 2, 3, 4, 5, 6, 7, 8 };
int auchSchach [];
int [][] schachbrett = new int [8][8];
```

Enthalten Eigenschaft `length`, die die Größe des Arrays angibt

Mehrdimensionale Arrays werden intern als Arrays von Arrays implementiert

`schachbrett.length` ergibt daher 8, nicht 64

Java Eigenheiten

Garbage Collector: Teil des Laufzeitsystems, das sich um das Aufräumen nicht mehr benötigter Objekte/Speicherblöcke kümmert

Java Virtual Machine: führt den Byte Code aus

Programmierstrukturen

Rekursion ↔ Interation

Schleifen: `for`, `while`, `do … while`

Exceptions

```
try { ... }
catch (FirstExceptionTyp e1) { ... }
[catch (SecondExceptionType e2) { ... }
...
catch (nthExceptionType en) { ... }]
```

Alles, was nicht in eckigen Klammern steht, ist nicht optional!

Einfache abstrakte Datentypen & Algorithmen

Sortieren: Insertion Sort, Selection Sort, primitives BubbleSort

Listen, Bäume, Keller, Schlangen

Bäume, Definition bspw.:

```
class BinTree {
    private BinTree left, right;
    private int value;
    ...
}
```

Mit Zeigern (Referenzen) arbeiten

Infix, Präfix, Postfix-Ausgabe von Bäumen

Beispielaufgaben: Kopie eines Baumes erzeugen, Feststellen ob Liste Präfix einer anderen

Hash-Tabelle: Java-Klasse `Hashtable` im Package `java.util`

Operationen:

`void put (Object key, Object value);` (Paar zufügen)

`Object get (Object key);` (Wert zum Schlüssel liefern)

`boolean containsKey (Object key);` (Gibt es ein Paar mit dem Schlüssel?)

`boolean contains (Object value);` (Gibt es ein Paar mit dem Wert?)

`void remove (Object key);` (Paar mit dem Schlüssel entfernen)

Objektorientiertes Programmieren

Objekte und Hierarchien

Objektvariable, -methoden: gehören nur einem speziellen Objekt, können auf Klassenvariable/-methoden zugreifen

Klassenvariable, -methoden: gekennzeichnet durch `static`-Schlüsselwort

Konstruktor: Wird (ausschließlich!) implizit aufgerufen

`public`, `protected`, `private`

Standard ist `public`. `private`-Variablen/Methoden werden nicht nicht vererbt

`private`-Variablen/Methoden sind in anderen Objekten oder von Unterklassen aus nicht sichtbar

`protected`-Variablen/Methoden sind in anderen Packages nicht sichtbar

Vererbung : Ober-/Unterklassen

Definition einer Unterklasse durch:

`class` *`Unterklasse`* `extends` *`Oberklasse`*

`super` bezeichnet die direkte Oberklasse eines Objektes, Zugriff auf `protected` und `public`-Eigenschaften/Methoden möglich

Aufruf des Konstruktors der Oberklasse mit `super(`*`ArgumentListe`*`)`, muss erste Anweisung sein! Notwendig, wenn Oberklasse keinen parameterlosen Konstruktor enthält.

Überschreiben: Methoden einer Unterklasse, die mit gleichem Namen, gleichen Typen von Parametern und Ergebnis auch in einer Oberklasse definiert sind. Sie überschreiben die der Oberklasse. Sie sollten konzeptuell „das gleiche" leisten, nur spezieller.

Überladen: Methoden mit gleichem Namen, aber verschiedenen Eingabeparametertypen. Abhängig von den Parametertypen wird die jeweils „passende" Methode aufgerufen

Dynamische Methodenbindung: Welche Methode aufgerufen wird bestimmt die Klassenzugehörigkeit des Objektes zur Laufzeit - nicht der Typ der Variablen, die die Objektreferenz enthält.

Interface: Enthält keine Implementierung!

Ein Klasse, die ein Interface erfüllt (`extends ...`), muss die Methoden als `public` verfügbar implementieren - also komplette Methoden mit passenden Methodenköpfen enthalten. (Ausnahme: Die Klasse ist abstrakt, dann gilt die Aussage jeweils für wenigstens eine Ihrer Unterklassen.)

abstrakte Klasse: Schlüsselwort `abstract`

Kann nicht instanziiert werden

Wenn alle enthaltenen Funktionen abstract sind (fast) äquivalent zu Interfaces. Unterschied: Abstrakte Klassen können `protected` Funktionen definieren.

Innere Klassen: Erlaubt die Verwendung von Variablen/Methoden der äußeren Klasse

anonyme Klassen: Klasse mit nur einem Exemplar, ohne Typdefinition

Beispielaufgabe: Klassenhierarchie aufstellen

Swing

Einbindung druch `import javax.swing.*;`

Swing-Hierarchie:

```
java.lang.Object
+--java.awt.Component
   +--java.awt.Container
      +--java.awt.Window
      |  +--java.awt.Frame
      |     +--javax.swing.JFrame
      +--javax.swing.JComponent
         +--javax.swing.AbstractButton
         |  +--javax.swing.JButton
         |  +--javax.swing.JMenuItem
         |     +--javax.swing.JMenu
         +--javax.swing.JPanel
         +--javax.swing.JMenuBar
```

Swing-Klassen:

Bei Klassen, die mit J beginnen, wird im Konstruktor der Titel des Objekts übergeben

`JFrame` (Fenster)

```
public void addWindowListener(WindowListener l);
```
(oft mit anonymen Klassen verwendet)
```
public Container getContentPane();
```

`JButton`

```
public void addActionListener(ActionListener l);
public void setActionCommand(String command);
```

`JComboBox`

```
public JComboBox(Object[] items);
```
(für gewöhnlich Array von Strings)
```
public void addActionListener(ActionListener l);
public void addItem(Object anObject);
public void removeItem(Object anObject);
public void setEditable(boolean aFlag);
public int getSelectedIndex();
```
(analog: `getSelectedItem`)

`JLabel` (Beschreibungsfeld)

`JMenu`

```
public JMenuItem add(JMenuItem menuItem);
public JMenuItem add(Component c);
```
(`JMenu` ist `Component`)

`JMenuBar`

```
public JMenu add(JMenu c);
```

`JMenuItem`

```
public void addActionListener(ActionListener l);
```

`JPanel`

```
JPanel(LayoutManager layout);
```
(z.B. `new FlowLayout()`)
```
public void paintComponent(Graphics g);
```

`JTextField`

```
public String getText();
```
(Entsprechend: setText)
```
public void addActionListener(ActionListener l);
public void selectAll();
```

`Color`

```
public Color(float r, float g, float b);
```
(jeweils im Bereich 0.0 - 1.0)

`Component`

```
public void setVisible(boolean b);
public Graphics getGraphics();
public void update(Graphics g);
public Dimension getSize();
public void setSize(int width, int height);
public void addMouseListener(MouseListener l);
```

`Container`

```
public Component add(Component comp);
public void setLayout(LayoutManager mgr);
```
(z.B. `new FlowLayout()`)
```
public void paint(Graphics g);
```
(zum Überschreiben gedacht)

`Graphics`

```
public abstract void drawRect(int x, int y,
int width, int height);
```
(ebenso `drawOval`, `fillRect`, `fillOval`)
```
public abstract void drawString(String str,
int x, int y);
```
(String wird so gezeichnet, dass die Basislinie des linkesten Zeichens an (x,y) steht.)
```
public abstract void setColor(Color c);
```

`WindowAdapter` (ggf. sinnvoll als innere Klasse)

```
public void windowClosing(WindowEvent e);
```
(zum Überschreiben gedacht)

`MouseAdapter`

```
public void mouseClicked(MouseEvent e);
```
(ebenso `…Pressed`, `…Released`, `…Entered`, `…Exited`)

`MouseEvent` enthält u.a. `public Point getPoint();` `public Object getSource();`

Swing-Interfaces

`ActionListener`

```
void actionPerformed(ActionEvent e);
```
`ActionEvent` hat u.a. die Methoden: `public String getActionCommand(); public Object getSource();`

Steuerelemente hinzufügen: Objektvariablen für Steuerelemente, Menüs, etc. anlegen

Im Fenster zeichen: `paintComponent()`-Methode eines `JPanels` verwenden

Nebenläufige Programmierung

Thread-Klasse ↔ Thread-Interface: Verwendung je nach Anforderung

Interface

Runnable

```
public void run();
```
(muss von der implementierenden Klasse überladen werden)

Initialisierung des Threads mit:

```
Thread auftrag = new Thread (new Aufgabe (...));
```
wobei Aufgabe das Interface `Runnable` implementiert

Start mit:

```
auftrag.start();
```

Empfehlenswert, wenn der abgespaltene Prozeß nicht weiter beeinflußt werden muss.

Klassen

`Thread`

```
public Thread()
public Thread(Runnable target)
public void run();
public static void sleep(long millis) throws
InterruptedException;
public final void join() throws
InterruptedException;
```
(Auf Terminierung des Threads warten.)

Zu verwenden, wenn der abgespaltene Prozeß weiter beeinflußt werden soll.

Start des Threads über die Methode `start`();

Funktionen zur Nebenläufigkeit in `Object`

```
public final void notify();
public final void notifyAll();
public final void wait() throws
InterruptedException;
```

Monitor: Jedes `Object` in Java kann als Monitor dienen

Es wird ein lock count verwaltet, der beim Eintreten in einen `synchronized` Abschnitt um 1 erhöht wird, und beim Verlassen um 1 verringert wird. Eintreten in einen solchen Block ist nur möglich, wenn der lock count gleich 1 ist, oder der aktuelle Thread bereits im Besitz des Monitors ist

`synchronized`-Blöcke/Methoden

Aufrufe von `synchronized` -Blöcken und Methoden von mehreren Threads, die den gleichen Monitor verwenden, werden unter gegenseitigem Ausschluß ausgeführt.

Zu jedem Zeitpunkt kann mit synchronized Methoden höchstens aus einem Thread auf Objektvariable zugegriffen werden.

Producer ↔ Consumer: Monitor erforderlich, um die gemeinsam benutzen Ressourcen zu verwalten (d.h. exklusive Verwendung zu garantieren).

Verklemmungen: können eintreten, wenn folgende Bedingungen gelten:

Jeder Prozeß fordert mehrmals nacheinander Ressourcen an.

Jeder Prozeß benötigt bestimmte Ressourcen exklusiv für sich allein.

Die Relation „Prozeß p benötigt eine Ressource die Prozeß q hat“ ist zyklisch.

Java-Gently

`Stream`

```
public Stream(InputStream i);
```
(meistens: `System.in`)
```
public Stream(String filename, int how);
```
(`how` = `Stream.READ`, `Stream.WRITE`)
```
public String readLine () throws IOException;
public int readInt () throws IOException;
```
(ebenso für `…char`, `…Double`, `…String`)