OPR Zusammenfassung, René Bernhardsgrütter, 12.01.2012

1 Enumeration

Enmumerations werden verwendet, um **Aufzählungen zu speichern**, die wie **Konstanten** verwendet werden (Kleidergrössen). Man kann den Konstanten Werte zuweisen (Strings, Zahlen, etc.) oder sie einfach als Text stehen lassen. Sie fördern Clean Code.

```
enum Zins {
    NUMMERN_KONTO(0.015),
    SALAER_KONTO(0.01);

    double zinssatz;

    private Zins(double zinssatz) {
        this.zinssatz = zinssatz;
    }

    public double getZinssatz() {
        return this.zinssatz;
    }
}

enum Kleidergrössen{ S, M, L, XL; }
```

1.1 Erzeugung (am Beispiel rechts)

1. Enum Name..
2. Verschiedene Typen (gross geschrieben, wie Konstanten) deklarieren. Dort kann man einen Standardwert mit dem richtigen Datentyp eintragen.
3. Zu speichernde Werte für den Datentyp deklarieren, wenn nicht nur Textwerte gespeichert werden sollen.
4. Den Konstruktor deklarieren: Hier muss der Datentyp übergeben werden, der bei den verschiedenen, gross geschriebenen, Typen angegeben wurde. Die übergebenen Werte werden den lokalen Variablen zugewiesen, die man vor dem Konstruktor deklariert hat.
5. Es können auch Methoden deklariert werden. Einige Standardmethoden gibt es (z. B. .toString()), die man überschreiben kann.

1.2 Verwendung

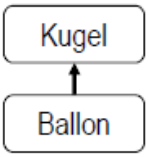
Das Zins-Enum von oben stehe auch im Quellcode.

```
Zins z1 = Zins.NUMMERN_KONTO;        // Nun wird NUMMERN_KONTO(0.015) aufgerufen, welches ein Objekt vom Enum Zins
                                     instanziiert. 0.015 ist dabei ein Parameter, der dem Konstruktor übergeben wird.
                                     Dieser speichert den Wert in der Variable zinssatz. Es ist ein Durchlauf lang
                                     eine Rekursion…
z1.getZinssatz();                    // Ergibt: 0.015;
Zins.z2 = Zins.SALAER_KONTO;         // Ein zweites Enum, diesmal vom Typ Salärkonto.
z2.getZinssatz();                    // Ergibt 0.01
```

2 Vererbung

Subklasse erweitert Superklasse: `class Ballon extends Kugel {}`
Private: Weder von aussen, noch in Subklassen sichtbar.
Protected: Von aussen nicht, in Subklassen jedoch schon sichtbar.
Public: Von aussen und in Subklassen sichtbar.

2.1 @Override



Methoden können damit überschrieben werden. Dann wird von einem Objekt diese Methode ausgeführt, und nicht die der Superklasse. **Achtung**: Falls die Deklaration der Parameter nicht genau mit der der Superklasse übereinstimmt, wird die Methode **überladen anstatt überschrieben**!
Falls man dennoch auf die Methode der Superklasse zugreifen will, muss man diese folgendermassen aufrufen: `super.methodenName();` Dies geht jedoch nur über eine Stufe (`super.super.methodenName();` ist nicht erlaubt).

2.2 Konstruktoren

Wenn man von einer Subklasse ein neues Objekt instanziieren will, wird der Konstruktor der Subklasse aufgerufen. **Wenn in der Superklasse der Standardkonstruktor verwendet wird, wird dieser automatisch aufgerufen** (`super();`)
Ein angepasster Konstruktor der Superklasse muss manuell aufgerufen werden, mit einem

```
super(eveltuelleParameter, ...);
```

gerade **in der ersten Zeile des Konstruktors der Subklasse** aufrufen.
Dies sollte **immer** manuell durchgeführt werden!

2.3 final

Wenn eine Methode oder eine Instanzvariable als **final** deklariert wurde, kann ihr Wert nicht geändert werden. Bei der Vererbung kann eine final-Variable/Methode in einer Superklasse in der Subklasse **nicht mehr überschrieben werden**. **Wenn eine Klasse als final deklariert wird, kann sie nicht mehr erweitert werden.**

2.4 final static

Wenn eine Variable als **final static** deklariert wurde, wird sie als **Konstante** angesehen: Kann nicht mehr verändert werden (final) und ist eine **Klassenvariable** (static). Selbiges für Methoden. Man kann dann direkt via Klassenname auf die Methode/Konstanze zugreifen. Alle Konstanten werden **durchgehend gross geschrieben**, zB. Math.PI = 3.1415…

2.5 ist-ein-Bezug

Das sollte man stets im Hinterkopf behalten, wenn man mit Vererbung arbeitet. Nummer4=>ist-ein=>Golf=>ist-ein=>Auto=>VW

3 Polymorphismus

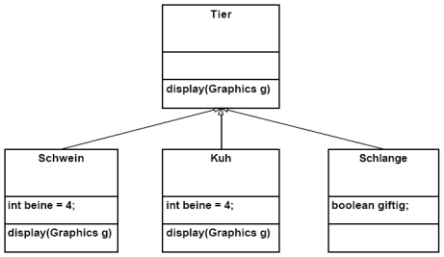
3.1 Prinzipen mit Subprinzipien von OOP

1. **Encapsulation**: Methoden und Daten kapseln.
2. **Inheritance**: Vererbung.
3. **Polymophism**: Objekten mit kompatiblen Datentypen gemischt verwalten.
   3.1. Ein Objekt behält immer die Identität der Klasse, aus der es erzeugt wurde. (Ein Objekt kann nie in ein Objekt einer anderen Klasse konvertiert werden!)
   3.2. Wenn eine Methode auf ein Objekt aufgerufen wird, wird immer die Methode verwendet, die mit der Klasse des Objekt verbunden ist.

3.2 Casting-Up (safe & automatisch)

Das ist der normale Zugriff auf „Subobjekte“.

```
schlangenObjekt.display(g)      //Display von Tier wird aufgerufen.
kuhObjekt.display(g)            Display von Kuh wird aufgerufen
Kuh berta = new Kuh();
Tier tier2 = berta;             // berta ist-ein Tier.
```



Dem tier2, welches vom Typ Tier ist, wurde berta, welche vom Typ Kuh ist, zugewiesen. Das funktioniert ganz normal. Wenn nun `tier2.display(g);` aufgerufen wird, **wird das Display von Kuh ausgeführt**.

Problem: Wenn man auf eine Methode von berta zugreifen will, die in Tier nicht deklariert ist, geht dies via tier2 nicht! Lösung: **Casting Down**.

3.3 Casting-Down (unsafe & manuell)

Man kann es auch anders herum machen. So kann man auf eine Methode oder Variable von (hier) tier2, der Kuh, zugreifen, auch wenn die Methode in Tier nicht deklariert ist.

```
Tier tier2 = new Kuh();      // Der Variable tier2 vom Typ Tier wurde nun ein Kuh-Objekt zugewiesen. Das geht,
                             // denn Kuh extends Tier (=Casting-Up)
Kuh elsa = (Kuh) tier2;      // Wenn man nun das Objekt hinter tier2 einer Kuh-Variable zuweisen will, muss man //
                             zuerst eine explizite Typumwandlung machen. Dies daher, weil man sonst
                             // vermeintlich ein Tier-Objekt einem Kuh-Objekt zuweisen will, was natürlich
                             // nicht geht, denn ~~Tier extends nicht Kuh~~, sondern Kuh extends Tier.
```

3.4 instanceof

Man kann nun eine automatische Typprüfung einbauen:

```
if (berta instanceof Kuh) {
      Kuh kuhKlon = (Kuh) berta;                  // Casting Down
      kuhKlon.melken();                           // Führe Kuh-Spezifisches aus
}
```

4 Regex

*	ax*b	0 od. mehrere x	ab, axb, axxb …	String hw = "Hallo Welt";
				if (hw.matches("H.*W.* ") {}
				String data = "4, 5, 6 2, 8 ,, 100, 18" ;
				String grenzmarke = " ," ;
				StringTokenizer token =
				new StringTokenizer(data, grenzmarke);
				while(token.hasMoreTokens()){
				String blub = token.nextToken();
				}

+	ax+b	1 od. mehere x	axb, axxb, …	
?	ax ?b	x opt.	ab, axb	
|	a|b	a od. b	a, b	
()	x(a|b)x	Gruppierung	xax, xbx	
.	a.b	1 beliebiges Zeichen	aab, acb, azb, a[b, …	
[]	[abc] x	1 Zeichen aus seiner Kl.	ax, bx, cx	
[-]	[a-h]	Zeichenbereich	a, b, c, …, h	

5 Abstrakte Methoden

Wenn man will, dass bei Subklassen eine Methode überschrieben wird, muss man die Methoden in der Superklasse als **abstract** deklarieren. Dabei wird in der Superklasse nur ein Methodenkopf (Signatur) deklariert: `public abstract float getFlaeche();`

Folge: Diese Superklasse muss **selbst auch als abstract** deklariert werden, da sie die abstract-Methode getFlaeche() enthält,:

```
public abstract class Figur {
	public abstract float getFlaeche();
}
```

- Man kann aus dieser Klasse direkt **kein Objekt instanziieren.**
- **Alle Subklassen von Figur müssen alle abstract-Methoden implementieren.**
  - **Alternative: Subklasse ebenfalls als abstract deklarieren.**
- \+ Man kann Design erzwingen.

6 Interfaces

- **Entkoppelt Anwender einer Komponente von der Implementation der Komponente.**
- Anwender verwenden Interface, Implementierer implementieren es mittels Klassen à la `List`.
- Anwender und Implementierer müssen nur Interface kennen.
- Chef des Programmes stellt Interface `IList` zusammen, sodass er die Struktur vorgeben kann.
- Interfaces **aus Sicht des Anwenders**, nicht des Implementieres, **dokumentieren.**

Ein Interface besteht nur aus **public** Methodensignaturen (ohne explizit geschriebenes abstract, **Interfaces sind jedoch immer implizit abstract!**) und `final static`-Konstanten:

```
public interface Ballon {
	public void changeSize(int newDiameter);
	public void move(int newX, int newY);
	public void display(Graphics g);
}
```

Interfaces sind neue Datentypen, genau gleich wie bei Klassen, `Ballon IBallon` funktioniert also. Interfaces können auch andere Interfaces extenden: `public interface Ballon extends Kugel {}`

- Alle Methoden sind **public**, auch wenn sie nicht explizit so bezeichnet sind.
- Man **darf keine static-Methoden** in Interfaces haben.
- Alle **Variablen sind static final**, auch wenn sie nicht explizit so deklariert sind (=Konstanten).
- Man kann keine Interfaces instanziieren.

6.1 Implementation

Interfaces werden von Klassen implementiert, die alle Methoden des Interfaces implementieren. Deren Klassenkopf sieht so aus:

```
class MyClass implements MyInterface {}
```

Wird ein Subinterface erweitert (`public ISubInterface extends ISuperInterface {}`), müssen auch **alle Methoden des ISuperInterfaces implementiert** werden.
Eine Klasse kann auch mehrere Interfaces implementieren:

```
class MyClass implements MyInterface1, MyInterface2, MyInterface3 {}
```

6.2 „Gegen ein Interface programmieren“

Man programmiert mit dem Datentyp des Interfaces:

```
List arry = new ArrayList();
```

List = Name und Datentyp des Interfaces „List“. array = neue Variable, die ein Objekt von der Klasse ArrayList() erhält. ArrayList() implementiert List, ist also vom Datentyp „ArrayList“ und „List“ (da es das implementiert): `array` ist `ArrayList` ist `List`.
Man kann nun die dynamische Liste „ArrayList“ gegen etwas besseres/performanteres austauschen, das ebenfalls das Inteface List implementiert. array kann ganz normal weiter verwendet werden (z. B. „HashList“).

6.3 Verwendung

Verwendetes Interface:

```
public interface Darstellbar {
	public void setPosition(int x, int y);
	public void anzeigen(Graphics g);
}
```

Eine Methode, die ein Array von Objekten übernimmt, welche vom Datentyp **Darstellbar** sind:

```
public void zeichneLinksbuendig(Graphics g, Darstellbar[] figuren, int xPos, int abstand) {
	for (int i = 0; i < figuren.length; i++){
		figuren[i].setPosition(xPos, i * abstand);
		figuren[i].anzeigen(g);
	}
}
```

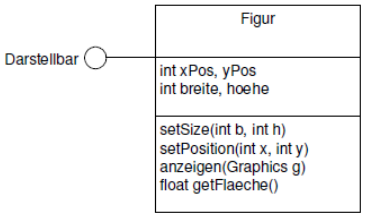
`zeichneLinksbuendig` besitzt einen Array-Parameter vom Typ `Darstellbar`. Die Methode kann beliebige Objekte, die `Darstellbar` implementieren, darstellen.

6.4 UML

Lollipop = Interface:

```
class Figur implements Darstellbar {}
```

Daran kann sich nun etwas „anheften“, wenn es etwas von **Darstellbar** benötigt. Es umschliesst die Kugel zu 50%.

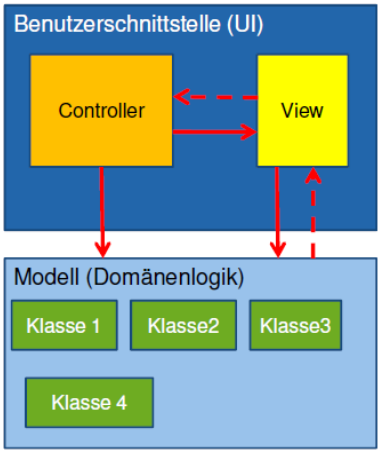


6.5 Unterschied zu abstrakten Methoden/Klassen

- Abstrakte Klassen können implementierte Methoden enthalten, Interfaces nicht.
- Eine Klasse kann mehrere Interfaces implementieren, jedoch nur eine (abstrakte) Klasse erweitern.
- Interfaces = neuer Datentyp.

7 MVC

- Modell muss sauber gehalten werden.
- Modell muss ausgetauscht werden können.
- Modell muss ggf. über PropertyChange mithilfe eines PropertyChangeSupport-Objektes über Änderungen am Modell alle Listener-Objekte informieren, dass sich etwas getan hat (gestrichelte Pfeile).
- Modell enthält Daten und Geschäftslogik.
- View zeigt GUI an und der Zustand der Elemente an (Grafiken, Buttons, Canvas, Scroll-Bars, Checkboxen, Textelemente, Menüs, …).
- Controller nimmt die Benutzereingaben entgegen und leitet sie an die richtige Stelle des Modells weiter.
- Controller führt keine Geschäftslogik aus!
- View führt keine Geschäftslogik aus!



8 Test-Driven-Development

Blackbox-Test: Test **anhand der Spezifikation** erstellen.
Whitebox-Test: Test **anhand des Quellcodes** erstellen.

Primär sollten die **kritischen Grenzwerte** getestet werden. Diese verursachen die **meisten Probleme.** Man sollte auch „normale“ Werte testen. Es muss auf gültige und ungültige Werte geprüft werden. Die Testfällte sollten möglichst viele gültige Äquivalenzklassen (eigenständige Art von Eingabe/-wert) abdecken. **Jede ungültige Äquivalenzklasse sollte nur einen Wert haben**, um die **Fehlerbehandlung nachvollziehen** zu können.

8.1 Regeln zum Bilden von Äquivalenzklassen

- Bilden die gültigen Eingabewerte **einen zusammenhängenden Wertebereich**, so sind ***eine* gültige und *zwei* ungültige Äquivalenzklassen** zu bilden.
- Ist anzunehmen, dass die Elemente einer Äquivalenzklasse verschieden behandelt werden, ist die Äquivalenzklasse entsprechend aufzutrennen.
- Müssen die Eingabewerte eine Bedingung zwingend erfüllen (z.B. erstes Zeichen ist ein Buchstabe), so ist eine gültige und eine ungültige Äquivalenzklasse zu bilden.
- Testdatensätze für ungültige Äquvalenzklassen sollten nur *einen* ungültigen Wert aufweisen.

8.2 Praxis

Man testet Klassen/Objekte => nur deren public-Methoden.

1. Objekte erzeugen.
2. getter-Methoden testen.
3. setter-Methoden testen.
4. Alle restlichen Methoden.

Schlussendlich müssen alle irgendwie möglichen Situationen getestet worden sein.

8.3 JUnit

Für automatisiertes Testen. JUnit arbeite mit konkreten Testdaten.

8.3.1 Annotations

`@Test`	Kennzeichnet Methoden als ausführbare Testfälle.
`@Before, @After`	Markiert Setup- bzw. Teardown-Aufgaben, die für jeden Testfall wiederholt werden sollen.
`@BeforeClass, @AfterClass`	Markieren Setup- bzw. Teardown-Aufgaben, die nur einmal pro Testklasse ausgeführt werden sollen.
`@Ignore`	Kennzeichnet temporär nicht auszuführende Testfälle.

8.3.2 Assert-Methoden

`assertTrue(boolean condition)`	Prüft, ob condition wahr ist
`assertFalse(boolean condition)`	Prüft, ob condition falsch ist
`assertEquals(Object exp, Object act)`	Verifiziert, ob die zwei Objekte **`expected`** und **`actual`** *gleich* sind.
`assertEquals(Object[] exp, Object[] act)`	Verifiziert die *Gleichheit* zweier Arrays.
`assertNull(Object object)`	**Verifiziert, ob eine Referenz `null` ist:**
`assertNotNull(Object object)`	**Verifiziert, ob eine Objektreferenz nicht `null` ist**

8.3.3 Beispiel-Test

Hier wird in der Testklasse `FrankenTest` geprüft, ob die Methode `add` von der Franken-Klasse wie erwartet arbeitet.

```
@Test
public void adding() {
        Franken two = new Franken(2.00);
        Franken sum = two.add(two);
        assertEquals(new Franken(4.00),sum);
}
```

8.4 Praxis

Klasse schreiben, Tests schreiben, testen, nächste Klasse, usw. **ODER Tests schreiben, Klasse schreiben, nächste Klasse**, usw. (= **TDD**) Tests sollten die **5 FIRST-Regeln** erfüllen:

1. **Fast**: Tests sollten **schnell** ablaufen, sodass sie häufig durchgeführt werden.
2. **Independent**: Jeder Test sollte **unabhängig** von den anderen durchgeführt werden können und die Tests sollten in beliebiger Reihenfolge durchgeführt werden können.
3. **Repeatable**: Tests sollten in **jeder Umgebung** wiederholbar sein (Entwicklungsumgebung, Produktivumgebung, Laptop, mit und ohne Netzwerk).
4. **Self-Validating**: Tests sollten ein **binäres Resultat** liefern (ok oder nok). Es sollte keine manuelle Inspektion von Test-Logs nötig sein.
5. **Timely**: Tests sollen zur richtigen Zeit geschrieben werden. D.h. gerade **bevor der Produktivcode geschrieben wird**.

9 Eigenständige Programme

main-Methode (static) wird bei Programmaufruf ausgeführt. Erzeugt als Erstes ein Objekt der Applikation. Dort drin läuft dann das gesamte Programm.

```
public class Applikation extends Frame implements WindowListener {
        public static void main(String[] args) {
            Applikation a = new Applikation();
            a.tuEtwas();
        }
        public Applikation() {
            super();
            …
        }
        private void tuEtwas() {
            …
        }
}
```

10 Exceptions

Checked: Müssen behandelt werden. Sollte man weiter geben via `throws XyzException` hinter Methodenname:
`public void methode() throws XyzExcepiton {}`

So müssen sie von aufrufender Stelle auch weiterbehandelt werden. Werden meistens durch Benutzer ausgelöst (**FileNotFoundExcepiton, IOException**)
Unchecked: Können behandelt werden (sind in der Regel Bugs). Z.B. **NullPointerException**.

10.1 Werfen

Mit dem Schlüsselwort **`throw`** können **neue Exceptions geworfen** werden. Das neue impliziert, dass es **neue Exception-Objekte** sein müssen (**new**). Wenn nichts weiter gemacht wurde, müssen diese von der aufrufenden Stelle **nicht** abgefangen werden (**unchecked**). Wenn man nach dem Methodenname **`throws Exception1, Exception2`** anhängt, **müssen** diese Exceptions von der aufrufenden Stelle **abgefangen werden (checked) bzw. weitergereicht werden (wieder mit `throws`).**

```
public int rechne(int a, int b) throws IllegalArgumentException, Exception {       // Liste für checked Exceptions.
      if ((a + b) < 32) {
            return ((int) Math.pow(2, (a + b))) / (a - b);
      } else if (irgendwas) {
            throw new IllegalArgumentException("(a + b) muss < 32 sein");         // Ein neues Exception-Objekt
      } else {                                                                    // wird geworfen.
            throw new Exception(„Fehler aufgetreten…“);
      }
}
```

10.2 Abfangen

```
public void callRechne(Graphics g) {
      try {                                                       // Versurche g.drawString();
            g.drawString("Versuch 1");
      } catch (IllegalArgumentException e1) {                     // IllegalArgumentExceptions abfangen.
            g.drawString ("Fehler: Etwas Falsches eingegeben");
      } catch (Exception e2) {                                    // Catches kann man anhängen, hier werden
            g.drawString("Unbekannter Fehler: Eingabe ungültig."); // alle restlichen Exceptions abgefangen.
      } finally {}                                                // Wird immer ausgeführt.
}
```

11 Dateien

Nur blockweise lesen/schreiben. Byteweise ist sehr langsam! Dateidialog: Klasse `FileDialog`

```
InputStream (FileInputStream)                                     // Lesen von beliebigen Daten.
OutputStream (FileOutputStream)                                   // Schreiben beliebiger Daten.
Reader (BufferedReader/InputStreamReader (FileReader))            // Lesen von Character-Streams.
Writer (PrintWriter, OutputStreamWriter (FileWriter))             // Schreiben von Character-Streams.
```

Lesen

```
try {
    String dateiInhalt;
    inFile = new BufferedReader( new FileReader(fileName) );
    while( ( line = inFile.readLine() ) != null) {
          dateiInhalt.append(line+"\n");
    }
    inFile.close();
}
catch (IOException){…}
```

Schreiben

```
PrintWriter outfile;
try{
   outFile = new PrintWriter(new FileWriter("out.txt"), true);
   outFile.print( inputTextArea.getText() );
   outFile.close();
}
catch (IOException e) {
   System.err.println("File Error: " + e.toString());
   System.exit(1);
}
```

12 Zugriffsmodifikatoren

Modifikator	Gleiche Klasse	Gleiches Package	Subklasse in anderem Package	Andere Klasse in anderem Package
Private	Ja	-	-	--
Keiner (package)	Ja	Ja	-	-
Protected	Ja	Ja	Ja	-
Public	Ja	Ja	Ja	Ja

13 Events

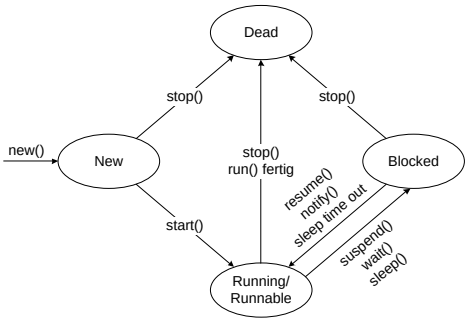
1. Listener definieren `...implements ActionListener{…}`
2. Objekt definieren `Button button = new Button(„Button“);`
3. Button hinzufügen `add(button);`
4. Als Listener registrieren `button.addActionListener(this);`
5. Eventmethode überschreiben `public void actionPerformed (ActionEvent e) {…}`

14 Threads

14.1 Erstellen

```
class Ball implements Runnable {}
Ball ball = new Ball();
Thread ballThread = new Thread(ball);
ballThread.start();
```

14.2 Zustände



`New` Thread erzeugt (mit new), aber noch nicht gestartet (mit start()).
`Running` Thread läuft in CPU.
`Runnable` Thread bereit zum Laufen, aber anderer Thread läuft zurzeit.
`Blocked` Thread kann zurzeit nicht weiterfahren, z.B. wegen **sleep.**
`Dead` run() ist fertig abgelaufen.

14.3 Thread-safe

Falls Threads auf gemeinsame Ressource zugreifen, darf die Ressource **nur einen Zugriff aufs Mal zulassen** (gegenseitiger Ausschluss)!
Methoden müssen **synchronized** implementieren: `public synchronized void increment() {…}`
Es wird ein **Monitor für das Objekt** erzeugt, welcher einen **Lock für das Objekt** verwaltet. **Lock ist ein Schlüssel** für eine Zimmertür. Nur eine Person gleichzeitig im Zimmer.

14.4 TickTock von Uhr-Programm

```
public class TickTock {
	private boolean tickHappens = false;

	public synchronized void waitForTick(){
		while (!tickHappens){
			try {                                   // try/catch, wenn der Thread interrupt()ed wird
				wait();                             // Thread wartet auf tickHappens und gibt
			}                                       // währenddessen Lock frei.
			catch (InterruptedException e) {
				System.err.println("Exception");
			}
		}
		tickHappens = false;                        // Setzte tickHappens wieder auf false
		notifyAll(); // Wecke alle wartenden Threads dieses Ojbkets auf
	}
	public synchronized void tick() {
		while(tickHappens){                         // Wenn tickHappens bereits true, warte.
			try {
				wait();                             // Thread wartet bis Konsument tickHappens
			}                                       // abgeholt hat
			catch (InterruptedException e) {
				System.err.println("Exception");
			}
		}
		tickHappens = true;
		notifyAll(); // Wecke alle wartenden Threads dieses Objekts auf
	}
}
```

14.5 notify() und notifyAll()

Werden verwendet, um Threads aufzuwecken. Die **schlafenden Threads haben auf einem synchronized-Objekt this.wait() aufgerufen und warten**. Dabei haben sie das **Lock wieder freigegeben**. Ein anderer Thread hat nur **this.notifyAll()** aufgerufen, um alle anderen Threads, die auf dem **Objekt gelockt sind, weiter auszuführen**.

14.6 Critical Section

In diesem Threadabschnitt wird auf eine gemeinsame Ressource zugegriffen. Sollte möglichst kurz sein, da die Threads dabei warten müssen.

14.7 Consumer-Producer ohne Queue

```
// Klasse/Thread Produzent
while(weiterso){
	Item produzieren;
	Warten, bis letztes Item abgeholt;
	// Critical Section
	Gemeinsames Objekt blockieren;
	Item bereitstellen;
	Gemeinsames Objekt freigeben;
	Konsument wecken;
}
```

```
// Klasse/Thread Konsument
while(weiterso){
	Warten, bis nächstes Item produziert ist;
	// Critical Section
	Gemeinsames Objekt blockieren;
	Item abholen;
	Gemeinsames Objekt freigeben;
	Produzent wecken;
}
```

Probleme: Geht nur, wenn es eine ganz einfache Situation ist.

14.8 Consumer-Producer mit Queue

```
// Produzent
while(weiterso){
	Item produzieren;
	Warten, falls Queue voll ist;
	Gemeinsames Objekt blockieren;
	Item in Queue ablegen;
	Gemeinsames Objekt freigeben;
	Konsument wecken;
}
```

```
// Konsument
while(weiterso){
	Warten, falls Queue leer ist;
	Gemeinsames Objekt blockieren;
	Item aus Queue abholen;
	Gemeinsames Objekt freigeben;
	Produzent wecken;
}
```

14.9 interrupt()

Wenn eine Variable/Flag (z. B. das `weiterso` von oben) als **`volatile`** deklariert wird, werden **alle lesenden Threads nach einer Änderung des Wertes** von der Variable/Flag **ebenfalls den neuen Wert haben**.
Falls ein Thread, der diese Variable/Flag hat, gerade **schläft** (von wait()), kann der den **geänderten Wert nicht lesen**. Man kann ihn **mit interrupt() wecken**. **Das wirft eine InterruptedException().** Programmierer muss wissen, was er damit machen will.

14.10 Deadlock, wenn Threads sich gegenseitig blockieren

Wenn Threads sich gegenseitig blockieren und das Programm so still steht (oder die Threads). Kann passieren, wenn man auf eine Ressource zugreift und sich dabei unglückliche Konstellationen ergeben können.

- Threads sollten in geordneter Reihenfolge auf Ressourcen zugreifen.
- Auf alle möglichen Arten testen!