In diesem Kapitel

Java Programmierung - Erweiterte Einführung

1.1. Über diesen vertiefenden Teil des Kurses

Über die Grundlagen hinausgehende Java Programmierung ist eine Ergänzung zu den bisher vermittelten grundlegenden Kenntnissen der Java™ Programmiersprache.

Damit Sie die Konzepte dieser Kurseinheit verstehen und davon profitieren, sollten Sie die vorangehenden Kurseinheiten durchgearbeitet haben. Ohne Java Kenntnisse werden Sie kaum auch nur eine Chance haben, vom Kursinhalt zu profitieren.

1.2. Kurs Übersicht

Herzlich willkommen zu dieser Kurseinheit.

In dieser Kurseinheit lernen Sie weiter gehende Java Programmierkonzepte kennen.
Insbesondere geht es um folgende Themen:

- Modul 1 : Fortgeschrittene Sprachkonzepte
  - Lektion 1 - weiterführende Konzepte
  - Lektion 2 - Deprecation
  - Lektion 3 - Innere Klassen
  - Lektion 4 - Die Vector Klasse
  - Übungen zu inneren Klassen
  - Quiz
  - Zusammenfassung
- Modul 2 : Stream I/O und Dateien
  - Einführung
  - Lektion 1 - Stream I/O
  - Lektion 2 - File I/O
  - File I/O Übungen
  - Quiz
  - Zusammenfassung
- Modul 3 : Threads
  - Lektion 1 - Threading in Java
  - Lektion 2 - Ablaufsteuerung von Threads
  - Praktische Übung
  - Lektion 3 - Synchronisation in Java
  - Lektion 4 - Thread Interaktion (wait/notify)
  - Lektion 5 - Producer / Consumer Beispiel
  - Praktische Übung
  - Quiz
  - Zusammenfassung
- Modul 4 : Netzwerkprogrammierung in Java
  - Lektion 1 - Verbindungsaufbau mit Sockets
  - Lektion 2 - UDP Sockets
  - Praktische Übung - Client/Server Beispiel
  - Quiz
  - Zusammenfassung
- Zusammenfassung

1.3. Modul 1 : Fortgeschrittene Sprachkonzepte

In diesem Modul

1.3.1. Einleitung

Sie haben einige grundlegende Konzepte kennen gelernt. Java bietet viel mehr als diese. Aber Sie können nicht alles auf einmal verstehen.
In diesemSinne vertiefen wir hier Ihre Java Kenntnisse, teils mit Wissen, welches Sie wenig brauchen, aber welches die Struktur von Java zeigt, teils mit praxisrelevanten Konzepten:

- Zugriff auf überschriebene Methoden
- Konstruktoren, überladene Konstruktoren und Konstruktoren der Oberklasse
- KlassenvariablenKlassenmethoden
- final, abstrakten Klassen
- Interfaces
- verfeinerte Zugriffskontrollen
- Deprecation - warum? was tun?
- inneren Klassen, anonyme Klassen
- Vectors Klasse
- Übungen, Quiz,Zusammenfassung

Im Teil *Java Grundlagen*, wurden Ihnen einige objekt-orientierte Programmierkonzepte vorgestellt. In diesem Modul vertiefen wir einige der bereits kurz besprochenen Konzepte und ergänzen Ihr Wissen. Speziell geht es um Konstruktoren, Überschreiben von Methoden und Interface. Für spezielle Amwemdungen besprechen wir, wie man den Konstruktor der Oberklasse oder überschriebene Methoden aufrufen kann.
Neue Themen, die wir besprechen, sind unter anderem finale Klassen, finale und statische Methoden und Variablen, abstrakte Methoden und Interfaces, innerere Klassen und verworfene (deprecated) Methoden aus JDK 1.0 und JDK 1.1.

1.3.1.1. Lernziele

Nach dem Durchabeiten dieses Moduls sollten Sie in der Lage sein:

- überschriebene Methoden aus überschreibenden Methoden aufzurufen
- überschriebene Konstruktoren aufzurufen
- den Einsatz der Konstruktoren der Oberklasse zu kontrollieren
- `static` Variables und Methoden zu deklarieren und einzusetzen
- `final` Klassen, Methoden und Variablen zu deklarieren und einzusetzen
- `abstract` Methoden und Interfaces zu deklarieren und einzusetzen
- `protected` Zugriffe zu deklariern und enzusetzen
- mit dem `-deprecation` Flag jene Methoden zu bestimmen, die in der aktuellen Version von JDK im Einsatz sind, und welche verworfen wurden.
- innere Klassen definieren und einsetzen können

1.3.2. Lektion 1 - Java vertieft

1.3.2.1. Einsatz überschriebener Methoden

Oft, wenn Sie eine Methode überschreiben, ist Ihr wahres Ziel nicht, die bestehende Methode zu verbannen. Sie möchten in der Regel eine bestehende Methode in irgend einer Art und Weise erweitern.

Dies können Sie mit Hilfe des `super` Schlüsselwprtes erreichen. Im Beispiel unten verwenden wir dieses Schlüsselwort, um die ursprüngliche Methode, die Methode der Oberklasse auszuführen, indem wir `super.method()` aufrufen. Dabei ist in diesem speziellen Fall die entsprechende Methode in der Oberklasse definiert worden. Allerdings muss dies nicht der Fall sein. Die Methode, die so aufgerufen wird, kann auch bereits von der Obreklasse von einer andern Klasse geerbt worden sein.

```
1    public class Employee {
2        private String name;
3        private int salary;
4
5        public String getDetails() {
6            return "Name: " + name + "\nSalary: " + salary;
7        }
8    }

9    public class Manager extends Employee {
10       private String department;
11
12       public String getDetails() {
13           return super.getDetails() +
14               "\nDepartment: " + department;
15       }
16   }
```

Durch den Einbau der Methode der Oberklasse können wir offensichtlich die eigene Methode stark vereinfachen, lediglich durch die Teile ergänzen, die neu hinzukamen.

Fehlermeldung im obigen Fall:

```
"Child.java": Error #: 458 : method methode() in class
publicprivateunduebeberschreiben.Child cannot override method methode() in
class publicprivateunduebeberschreiben.Parent with weaker access privileges,
was public at line 16, column 16 (oben Zeile 5)
```

1.3.2.2. Regeln betreffend überschriebenen Methoden

Es gibt lediglich zwei Regeln betreffend dem Einsatz überschriebener Methoden:

1. die überschreibende Methode kann nicht geschlossener sein (mit mehr Zugriffseinschränkungen) als die Methode, die sie überschreibt.
2. die überschreibende Methode kann nicht mehr Ausnahmen werfen, als die Methode, die sie überschreibt.

Beide Regeln wurzeln im Polymorphismus, zusammen mit der Anforderung von Java "typesafe" zu sein. Wir zeigen gleich ein illegales Szenario.

Polymorphismus bei Methoden

Eine Methode heisst polymorph, falls sie in mehreren Formen auftritt. Das heisst, die selbe Methode mit der selben Signatur (Datentypen, Argumenteanzahl) tritt in zwei oder mehr Klassen auf. Polymorphe Methoden besitzen dieselbe Semantik (lesen, schreiben oder verändern eines Objekts), aber unterschiedliche Implementationen. Jede Implementation ist der Klasse, in der sie definiert wird, angepasst.

Schauen wir uns nach diesem kurzen Einschub ein falsches Beispiel an, um daraus zu lernen. Wäre es der Methode `method()` in der `Child` Klasse (Zeile 6) erlaubt `private` zu sein, während die überschriebene Methode in `Parent public` wäre (Zeile 2), dann wäre das anschliessende Programm korrekt. Aber bei der Ausführung würde ein Problem auftreten. Die Methode `method()`, welche im Kontext von `p2` (Zeile 14) aufgerufen wird, sollte eine `Child` Version von `method()` sein; diese wurde aber als `private` deklariert. Somit ist der Zugriff nicht gestattet.

Eine analoge Logik gilt für den Fall, dass ein oder mehrere Ausnahmen geworfen werden können.

```
1    public class Parent {
2        public void method() {
3        }
4    }

5    public class Child extends Parent {
6        private void method() {
7        }
8    }

9    public class UseBoth {
10       public void otherMethod() {
11           Parent p1 = new Parent();
12           Parent p2 = new Child();
13           p1.method();
14           p2.method();
15       }
16   }
```

Die Regel betreffend den Ausnahmen steht in der Sprachdefinition. Allerdings mss man beachten, dass Verfeinerungen der Exceptions gestattet sind (Polymorphismus).

Sie finden auf dem Server verschiedene Beispiele, bei denen die spätere Überschreibung mit throwable Methoden möglich ist.

Und hier ein Beispiel, welches zu einer Fehlermeldung führt:

```
class BadPointException extends Exception {
        BadPointException() { super(); }
        BadPointException(String s) { super(s); }
}
class Point {
        int x, y;
        void move(int dx, int dy) { x += dx; y += dy; }
}
class CheckedPoint extends Point {
        void move(int dx, int dy) throws BadPointException {
                if ((x + dx) < 0 || (y + dy) < 0)
                        throw new BadPointException();
                x += dx; y += dy;
        }
}
```

führt zu folgender Meldung:

// "CheckedPoint.java": Error #: 462 : method move(int, int) in class ueberschreibenmitexceptionsgemaesslanguagespec.CheckedPoint cannot override method move(int, int) in class ueberschreibenmitexceptionsgemaesslanguagespec.Point, **overridden method does not throw** ueberschreibenmitexceptionsgemaesslanguagespec**.BadPointExceptio**n at line 13, column 7

Wichtig ist, dass die Methode die Exception wirft und nicht selber abfängt. Sie sehen solche Beispiele auf dem Server: dort treten keine Fehler auf.

1.3.2.3. Einsatz überladener Konstruktoren

Ähnlich wie bei den Methoden kann es auch bei den Konstruktoren passieren, dass Sie mehrere Konstruktoren definiert haben und eigentlich beispielsweise ein Konstruktor mit mehreren Parametern, ein Konstruktor mit einem oder keinem Parameter *ergänzen* wollen.

Sie möchten daher in einem Konstruktor einen andern Konstruktor aufrufen und ergänzen.

Dies können Sie mit Hilfe des `this` Schlüsselwortes, aber als Methodenaufrufwie in den Zeilen 11, 15 und 19 im Beispiel unten, erreichen. Damit können Sie dublizierten Code vermeiden.

Nehmen wir an, wir wollen einen Angestellten neu erfassen, kennen aber sein Gehalt noch nicht. In diesem Fall würden wir en Konstruktor ohne Gehaltsangabe verwenden:

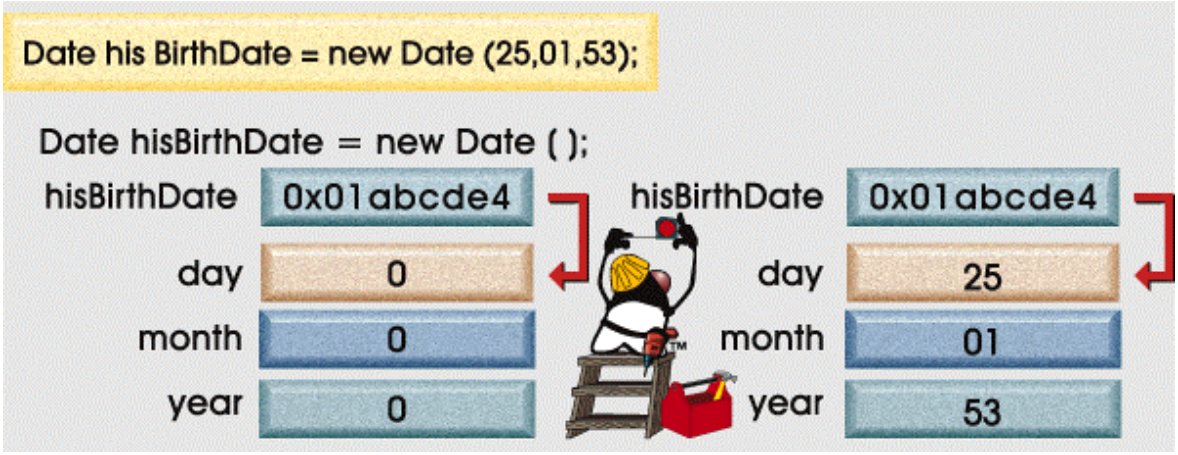
```
Employee rookie = new Employee("Joe");
```

Das Gehalt würde automatisch auf 0 gesetzt, Sie sehen jetzt wieviel ich verdiene! Zeile 10 im Programm zeigt Ihnen dies an. Dort wird der Standardwert des Gehalts auf 0 initialisiert. Gleichzeitig wird ersichtlich, dass eigentlich der übliche Konstruktor eingesetzt wird, also jener mit Gehaltsangabe. Dieser steht bei Zeile 5 und folgende. Der Eingabeparameter `name` ist "Joe" und `salary` ist 0. Wie Sie aus der Klassendefinition erkennen, kann man Gehalt nicht erhöht werden: die nötigen Methoden fehlen.

```
1    public class Employee {
2       String name;
3       int salary;
4
5       public Employee(String n, int s) {
6           name = n;
7           salary = s;
8       }
9
10      public Employee(String n) {
11          this(n, 0); //obiger Konstruktor
12      }
13
14      public Employee(int s) {
15          this("Unknown", s); // Konstruktor aus Zeile 5
16      }
17
18      public Employee() {
19          this("Unknown"); // Konstruktor aus Zeile 10
20      }
21  }
```

1.3.2.4. Aufruf des Konstruktors der Oberklasse

Java ist sehr vorsichtig beim Kreieren neuer Objekte. Die Reihenfolge der einzelnen Schritte haben wir schon einige Male erwähnt, hier trotzdem ein weiteres Mal:



1. Speicherplatz wird angelegt und mit Initialwerten belegt (Nullwerten).
2. für jede Klasse in der Hierarchie (ab *Object*) wird
   - explizit initialisiert
   - ein Konstruktor aufgerufen

Sie können diesen Prozess übrigens sehr schön im Debugger beobachten!
Das Java Sicherheitsmodell verlangt, dass alle Aspekte eines Objekts, welches die Oberklasse beschreibt, zuerst initialisiert werden *bevor* ein Objekt der Unterklasse irgend etwas ausführen kann.

Oft wird ein Default Konstruktor verwendet, um das Objekt der Oberklasse zu kreieren.
Aber Sie können die Konstruktion des Oberklassenobjekts auch steuern. Dies geschieht durch Einsatz des `super` Schlüsselwortes, aber als Methodenaufruf, `super()`, wie im folgenden Beispiel in Zeile 10.

Wichtig ist zu beachten, dass Sie `super()` mit Argumenten aufrufen, die auch in den Konstruktoren der Oberklasse vorkommen. Zudem gilt folgende Regel:

- falls ein Konstruktor einer Oberklasse aufgerufen werden soll, muss `super(...)` die *erste* Anweisung im Konstruktor sein.

Wenn Sie die Aussagen weiter vorne zum schrittweisen Aufbau eines Objekts verstanden haben, sollte Ihnen klar sein warum dies so sein muss: das Objekt der Oberklasse muss vor dem Objekt der aktuellen Klasse eingesetzt werden.

Damit wird der Einsatz des Konstruktors der Oberklasse analog zum Überladen einer Methode. Beide Techniken stellen Wege und Werkzeuge zur Verfügung, um Verdoppelungen zu vermeiden und damit den Code zu vereinfachen und die Wartung zu reduzieren.

```
1    public class Employee {
2        String name;
3        public Employee(String n) {
4            name = n;
5        }
6    }


7    public class Manager extends Employee {
8        String department;
9        public Manager(String s, String d) {
10           super(s); // erste Anweisung im Konstruktor
11           department = d;
12       }
13   }
```

1.3.2.5. Klassen Variablen

Manchmal ist es wünschenswert, dass **alle** Instanzen einer Klasse eine Variable gemeinsam haben, also globale Variablen? Nicht ganz. In diesem Zusammenhang spricht man von Klassenvariabeln (oder wie wir gleich sehen werden, statischen Variabeln).

Solche Variabeln können eingesetzt werden, um Objekte leichter miteinander kommunizieren zu lassen. Auch die Anzahl Instanzen kann damt leicht mit Hilfe eines Zählers, der als Klassenvariable definiert ist, realisiert werden.

Diesen Effekt erreicht man, indem man diese Variable mit dem Schlüsselwort `static` kennzeichnet. In Zeile 3 im Beispiel unten, sehen Sie wie dies aussieht. In diesem Beispiel erhält jedes Objekt eine eindeutige Serienummer. Die letzte vergebene Serienummer wird in der Klasse, nicht im Objekt abgespeichert. Wir können Sie also dort, zentral, jederzeit nachsehen, ohne gezwungen zu sein, uns umzusehen und alle Objekte abzufragen.

Die Variable `counter` wird von **allen** Instanzen gemeinsam genutzt. Jedesmal, wenn der Konstruktor ein neues Objekt kreiert, wird der Zähler um eins erhöht. Das nächste Objekt erhält also die nächst höhere Zahl als Serienummer (`serialNumber`).

```
1    public class Count {
2        private int serialNumber;
3        private static int counter = 0;
4
5        public Count() {       // Konstruktor
6            counter++;
7            serialNumber = counter;
8        }
9    }
```

Eine `static` Variable ist im gewissen Sinn ähnlich wie eine globale Variable in andern Sprachen. Java kennt keine globalen Variabeln. Eine `static` Variable ist eine einfache Variable, auf die von allen Instanzen *einer* Klasse zugegriffen werden kann

Falls eine `static` Variable `public` ist, wie in Zeile 2 im Beispiel, dann kann man von aussen jederzeit darauf zugreifen. Da es sich um eine Klassenvariable handelt, braucht man sogar keine Instanz, um die Variable zu nutzen: man darauf direkt zugreifen, mit Hilfe der Klasse, wie in Zeile 6.

```
1    public class StaticVar {
2        public static int number;
3    }

4    public class OtherClass {
5        public void method() {
6            int x = StaticVar.number; // Klasse.klassenVariable
7        }
8    }
```

1.3.2.6. Klassen Methoden

Manchmal wären Sie froh, wenn Sie auf Daten und Methoden zugreifen könnten, ohne zuerst eine Instanz, ein Objekt bilden zu müssen.

Das können Sie analog zum Zugriff auf Klassenvariabeln mit Hilfe von Klassenmethoden. Methoden, welche mit dem Schlüsselwort `static` gekennzeichnet sind, gestatten diesen Zugriff. Sie sehen im folgenden Beispiel auf Zeile 2 die Definition einer statischen Methode.

`addUp()` ist ein Beispiel für eine Klassenmethode. Weil eine statische Methode ohne die Bildung einer Instanz genutzt werden kann, müssen Sie auch nicht `this` oder irgendwelche andere Tricks anwenden, um die Methode ausführen zu können.

Eine statische Methode kann lediglich auf eigene oder Klassenvariable, statische Variable zugreifen. Jeder Zugriff auf nicht-statische Variablen führt zu einem Fehler beim Übersetzen. In Zeile 17 haben wir einen solchen Fehler eingebaut. Da die Variable `x` nicht statisch ist, wird ein Compiler Fehler resultieren.

```
1    public class GeneralFunction {
2        public static int addUp(int x, int y) {
3            return x + y;      // Zugriff auf Parameterwerte
4        }
5    }

6    public class UseGeneral {
7        public void method() {
8            int a = 9;
9            int b = 10;
10           int c = GeneralFunction.addUp(a, b);
11           System.out.println("addUp() gives " + c);
12       }
13   }

14    public class Wrong {
15        int x;                // nicht static
16        public static void main(String args[]) {
17            x = 9;            // COMPILER FEHLER!
18        }
19   }
```

Sie kennen das Verhalten von statischen Methoden von der `main()` Methode her, wenigstens teilweis. Die `main()` Methode hat aber weitere Eigenschaften (als Startermethode), die andere statischen Methoden nicht haben.

1.3.2.7. Das `final` Schlüsselwort

In Java kann man Klassen oder Variablen als `final` kennzeichnen. Falls man dies tut, kann die Klasse nicht weiter verfeinert werden.

Falls Sie eine Klasse als final deklarieren, dann kann keine Unterklasse mehr definiert werden. Ein Beispiel einer finalen Klasse kennen Sie schon: `java.lang.String` ist eine `final` Klasse.

In diesem Fall kann einer der Gründe seiin, dass man vermeiden wollte, dass jemand das Verhalten, die Methoden, die zu den Zeichenketten gehören, überschrieben werden. In diesem Fall handelt es sich also um eine Sicherheitsmassnahme.

Sie können nicht nur Klassen sondern auch Methoden als `final` kennzeichnen. Solche Methoden können nicht überschrieben werden. Auch hier kann man Sicherheitsaspekte erkennen: wenn Sie wollen, dass zum Beispiel die Abfrage Ihres Gehaltkontos nicht von jemandem überschrieben werden kann, dann erklären Sie die Abfragemethode einfach als `final`. Methoden, die als final deklariert sind, werden auch schneller ausgeführt, da der Compiler nicht zuerst alle virtuellen Methoden überprüfen muss, um zu entscheiden, welche jetzt konkret ausgeführt werden soll.

Falls Sie eine Variable als `final` kennzeichnen, dann heisst das, dass es sich um eine Konstante handelt. Jeder Versuch, eine solche Variable zu verändern führt zu einem Compiler Fehler.

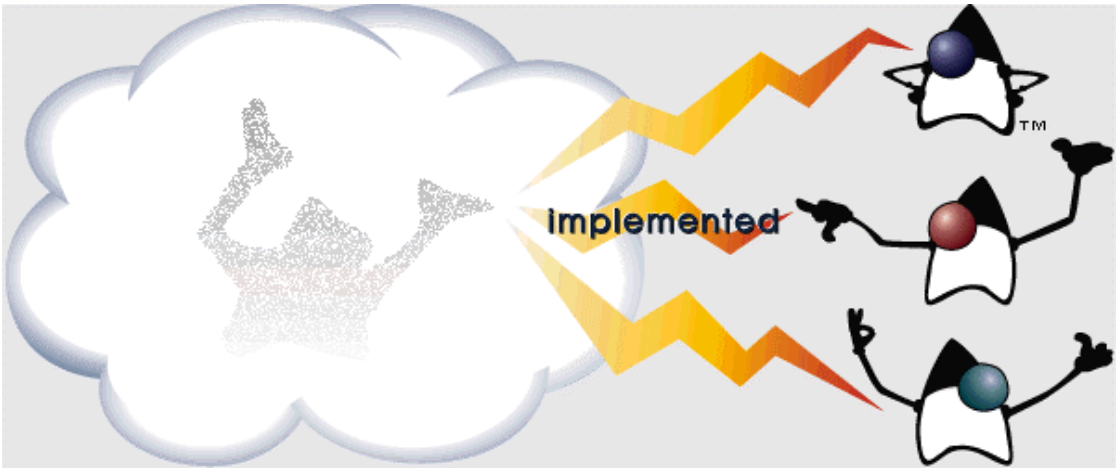
```
package finaleklasse;

public final class FinaleKlasse {

  FinaleKlasse() {
  }

  final int iKONSTANTE=20;
  int iVar;
  //
  public void methode() {
    System.out.println("Aufruf der public Methode in der finalen Klasse");
  }
  //
  public final void finaleMethode(String str) {
    System.out.println("Aufruf der final Methode in der finalen Klasse"+str);
  }
  /*"FinaleKlasse.java": Error #: 454 : class finaleklasse.FinaleKlasse should be declared
abstract; it does not define method abstrakteMethode() in class finaleklasse.FinaleKlasse at
line 12, column 14

 in einer finalen Klasse, die also nicht erweitert werden kann, darf ich nicht abstrakte
Methoden definieren, da
  diese implementiert werden müssten; das geht aber bei finalen Klassen nicht mehr. */
  public abstract void abstrakteMethode(); // Fehler
}
```

1.3.2.8. Abstrakte Klassen

Falls Sie eine ganze Klassenbibliothek entwickeln, kann es Ihnen passieren, dass Sie gezwungen sind das grundlegende Verhalten einer Klasse zu beschreiben, ohne dass Sie die Implementationsdetails kennen. Sie gehen davon aus, dass das Verhalten in Unterklassen implementiert wird.

Ein Beispiel könnte die Behandlung von Maus Events sein. Sie möchten beispielsweise Maus Clicks abfangen können. Diese haben aber kontextsensitiv völlig unterschiedliches Veralten zur Folge.

Eine solche Klasse ist beispielsweise die `MouseAdapter` Klasse, in der die Existenz von Methoden deklariert wird,; aber deren Implementation wird nicht angegeben.Solche Klassen bezeichnet man allgemein als abstrakte Klassen.



In Java werden abstrakte Klassen mit Hilfe des Schlüsselwortes `abstract` gekennzeichnet. Im folgenden Beispiel sehen Sie eine solche Deklaration.

Es sollte nicht der falsche Eindruck entstehen, dass alle Methoden einer abstrakten Klasse abstrakt sein müssen. Im Beispiel ist dies der Fall. Das muss aber nicht sein. Es kann ja durchaus sein, dass Sie ein bestimmtes Verhalten, eine bestimmte Methode in jeder Ihrer Untreklassen einsetzen möchten, eventuell erweiternd. Dann würden Sie diese Methode innerhalb der abstrakten Klasse bereits "vordefinieren", nicht abstrakt sonder sehr konkret.

Instanzieren können Sie eine abstrakte Klasse allerdings nicht. Falls Sie eine Instanz einer Klasse benötigen, müssen Sie zuerst eine Unterklasse definieren.

Wenn Sie im Beispiel unten die abstrakte Klasse MyAdapter instanzieren möchten und folgende Anweisung dazu verwenden:

```
MyAdapter myMouse = new MyAdapter();
```

wird Sie der Compiler nicht so nett grüssen.

Unterklassen einer abstrakten Klasse müssen alle abstrakten Methoden ihrer Oberklasse implementieren. Falls Sie versuchen, mur einzelne Methoden zu implementieren, andere aber

einfach als abstrakt stehen zu lassen (direkt in der Oberklasse), dann wird auch hier der Compiler Sie nett darauf aufmerksam machen, dass Sie

- entweder Ihre Unterklasse als abstrakt deklarieren müssen
- oder Sie alle Methoden implementieren müssen.

```
1    public abstract class MyAdapter {
2        public abstract void mouseClicked(int mouseX, int mouseY);
3        public abstract void mouseDragged(int mouseX, int mouseY);
4    }
5    // fehlerhafte Unterklasse : Methode mouseDragged fehlt
6    // korrekt wäre


7    // public abstract class MeineImpl extends MyAdapter {
8    public class MeineImpl extends MyAdapter {
9        public void mouseClicked(int mouseX, int mouseY) {
10               // ha ha ... ich mach gar nichts
11       }
12   }
13   // Schema erkannt: ich leg das System rein
14   // ich gehe hier von der korrekten Version der obigen Unterklasse aus
15   // also einer Definition wie in Zeile 7, statt 8
16   public class Click extends MeineImpl {
17       public void mouseDragged(int mouseX, int MouseY) {
18               // sowas geht : ich tu nichts, aber die Methode ist def.
19       }
20   }
21
```

1.3.2.9. Interfaces

Ein Interface ist eine Variation der Idee einer abstrakten Klasse. In einem Interface sind alle Methoden abstrakt - keine enthält einen Methodenrumpf.

Datenfelder müssen entweder final sein, also Konstanten, oder aber statisch und damit für alle Instanzen gleich. Gründe dafür gibt es einige. Der Hauptgrund ist, dass Sie ein Interface mehrfach implementieren könnten und dann unklar ist, welche Version benutzt werden soll.

Eine Definition für das Interface `java.awt.event.ActionListener` ist unten abgebildet. Der Vorteil von `interface` ist, dass man sie überall in ein Java language einbinden kann. Eine Klassendefinition gehört im Gegensatz dazu zu einer Klassenhierarchie. Eine Klasse kann genau eine andere Klasse erweitern, aber beliebige Interfaces.

Die Klasse `ReactToButton` implementiert das `ActionListener` Interface und implementiert eine eigene Method `actionPerformed()`.

```
1    public abstract interface ActionListener extends EventListener {
2        public abstract void actionPerformed(ActionEvent e);
3    }

4    public class ReactToButton implements ActionListener {
5       public void actionPerformed(ActionEvent e) {
6           // ... application specific code to react to the event
7       }
8    }
```

Beim Implementieren eines Interfaces wird nur die Struktur vererbt, nicht das Verhalten.
Sie können auch Namen von Interfaces als Referenztypen verwenden. Auf dem Server sehen Sie dies in einem Beispiel. Auch casting, also Typenumwandlung, ist möglich.

```
interface PointInterface {
        void move(int dx, int dy);
}
interface RealPointInterface extends PointInterface {
        void move(float dx, float dy);
        void move(double dx, double dy);
}
```

Und hier einige produzierte Fehler:

```
public interface Interface1 {
  int iA=10;
  final int iKONSTANTE=19;

  public void publicMethode(String s, int i);
  public void publicMethodeMitException(String s, int i) throws
ArrayIndexOutOfBoundsException;

  // "Interface1.java": Error #: 217 : modifier private not allowed here at line 18, column 16
  private void privateMethode(String s, int i);

  public abstract void abstractMethode(String s, int i);

}
```

1.3.2.10. Zugriffskontrolle

Neben `private` und `public` gibt es noch zwei weitere Zugriffslevels: default und `protected`.

Der Standardwert wird angenommen, falls kein expliziter Modifier verwendet wird. In diesem Fall hat jede Methode aus jeder Klasse des gleichen Packages Zugriff.

Im Falle von `protected` ist die Situation leicht komplexer: eine `protected` Methode oder Variable kann von jeder Methode in jeder Klasse verwendet werden, die entweder im selben Package ist, oder aber aus Unterklassen der betreffenden Klasse.

	zugreifbar aus:				
Modifier	**selbe Klasse**	**Klasse im selben Package**	**Unterklassen in unterschiedlichen Packages**	**nicht Subklasse, untersch. Package**	**Beispiel**
public	ja	ja	ja	ja	public int salary;
protected	ja	ja	ja	nein	protected int salary;
(default)	ja	ja	nein	nein	int salary;
private	ja	nein	nein	nein	private int salary;

1.3.3. Lektion 2 - Deprecation

Beim Übergang von Version 1.0 des JDK auf Version 1.1 wurde ein beträchtlicher Aufwand in die Vereinheitlichung von Methodennamen und ähnlichen Verbesserungen gesteckt.

Das Ergebnis sind neue oder verbesserte Methoden und alte, die nicht mehr eingesetzt werden sollten. Viele dieser Umstellungen führen zu einer Vereinfachung; andere stellen echte Verbesserungen dar. Insbesondere bei den Threads, auf die wir nächstens kommen, wwaren grössere Fehler vorhanden.

Schauen wir uns ein Beispiel an:

JDK 1.0 Version von `java.awt.Component`:

- ändern und bestimmen der Grösse einer Komponente
  - `resize()` und `size()`
- ändern und bestimmen der umgebenden Box einer Komponente:
  - `reshape()` und `bounds()`

In der JDK 1.1 Version von `java.awt.Component` sind die obigen Methoden veraltet und wurden durch neue ersetzt. Die neuen haben den Präfix `set` und `get` und erhöhen damit die Lesbarkeit:

- `setSize()` und `getSize()`
- `setBounds()` und `getBounds()`

1.3.3.1. Das Deprecation Flag

Im Moment können Sie die veralteten Methoden und Klassen weiter verwenden. Es könnte aber sein, dass diese eines Tages nicht mehr der Fall ist.

Sie sollten daher den Java Compiler mit dem `-deprecation` Flag verwenden, um Gefahrenherde zu identifizieren:

```
% javac -deprecation DateExample.java
```

Das `-deprecation` Flag bestimmt alle Methoden und Klassen, welche veraltet sind.

Im Beispiel `DateExample.java` mit dem `-deprecation` Flag übersetzt, wird die Compiler Warnung generiert, die weiter unten steht.

```
import java.util.*;

public class DateExample{

   public static void main (String args[]) {
      Date myDate = new Date(74, 5, 3);
      System.out.println (myDate.getYear());
   }

}
```

```
$ javac -deprecation DateExample.java
DateExample.java:6: Note: The constructor
java.util.Date(int,int,int) has been deprecated.
      Date myDate = new Date(74, 5, 3);
                    ^
DateExample.java:7: Note: The method int getYear() in
class java.util.Date has been deprecated.
      System.out.println (myDate.getYear());
                                 ^
Note: DateExample.java uses a deprecated API.  Please
consult the documentation for a better alternative.
3 warnings
```

1.3.3.1.1. Verbesserte Version von **DateExample**

In dieser neuen Version benutzen wir die den `GregorianCalendar` Klasse an Stelle der `Date` Klasse.

```
import java.util.*;

public class DateExample{

   public static void main (String args[]) {
      GregorianCalendar myDate = new GregorianCalendar(74, 5, 3);
      System.out.println (myDate.get(1));
   }
}
```

1.3.4. Lektion 3 - Innere Klassen

Innere Klassen, oft auch als verschachtelte Klassen bezeichnet, wurden in JDK 1.1 eingeführt. In den früheren Versionen des JDKs waren nur top-level Klassen erlaubt.

Ab JDK 1.1 werden nun auch Klassen unterstützt, welche Member einer anderen Klasse sind, lokal in einem Block oder anonym innerhalb eines Ausdrucks. Innere Klassen haben folgende Eigenschaften:

- der Klassennamen darf nur innerhalb des definierten Gültigkeitsbereiches benutzt werden, ausser man verwendet die Klasse qualifiziert.
  Der Name der Klasse muss sich vom Namen der umschliessenden Klasse unterscheiden.
- die innere Klasse kann Klassen- und Instanz-Variablen der lokalen und der umschliessenden Klasse verwenden.
- innere Klassen können auch als `abstract` definiert sein.
- die innerere "Klasse" kann auch ein Interface sein, ein Interface welches von einer anderen inneren Klasse implementiert wird.
- innere Klassen können als `private` oder `protected` definiert werden. Der Zugriff auf umschliessende Klassen und deren Variable ist aber immer möglich.
- inner Klassen, welches als `static` deklariert wurden, werden automatisch top-level Klassen - sie können also nicht weiter auf lokale Variablen der umgebenden Klasse zugreifen und diese modifizieren.
- innere Klassen können keine `static` Members definieren- das können nur top.level Klassen. Eine innere Klasse, welche statische Variablen definieren möchte, muss statisch deklariert werden.

1.3.4.1. Beispiel für eine Innere Klasse

Innere Klassen verwendet man oft um Event Adapter zu kreieren. Das folgende Beispiel `TwoListenInner` ist eine solche Klasse. Der Einsatz der inneren Klasse erhöht die Effizienz der Implementierung. Auch die Grösse der Dateien ist in der Regel kleiner als mit Hilfe separater Klassen.

Folgendes Beispiel finden Sie auf dem Server. Vorsicht, der exit Button fehlt!

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class TwoListenInner {
5      private Frame f;
6      private TextField tf;
7
8      public static void main(String args[]) {
9          TwoListenInner that = new TwoListenInner();
10         that.go();
11     }
12
13     public void go() {
14         f = new Frame("Beispiel mit Inneren Klassen");
15         f.add ("North", new Label ("Bewegen Sie die Maus"));
16         tf = new TextField (30);
17         f.add ("South", tf);
18
19         f.addMouseMotionListener (new MouseMotionHandler());
20         f.addMouseListener (new MouseEventHandler());
21         f.setSize(300, 200);
22         f.setVisible(true);
23     }
24
25     // MouseMotionHandler ist eine innere Klasse
26     public class MouseMotionHandler extends MouseMotionAdapter {
27         public void mouseDragged (MouseEvent e) {
28             String s = "Die Maus wurde bewegt:  X = " +
29                        e.getX() + " Y = " + e.getY();
30             tf.setText (s);
31         }
32     }
33
34     // MouseEventHandler ist eine innere Klasse
35     public class MouseEventHandler extends MouseAdapter {
36         public void mouseEntered (MouseEvent e) {
37             String s = "Die Maus wurd ins Feld bewegt.";
38             tf.setText (s);
39         }
40
41         public void mouseExited (MouseEvent e) {
42             String s = "Die Maus hat das Gebiet verlassen";
43             tf.setText (s);
44         }
45     }
46 }
```

1.3.4.2. Wie funktionieren Innere Klassen?

Innerer Klassen haben Zugriff auf alles, was sich in der umgebenden Klasse befindet, da innere Klassen zu der äusseren Klasse gehören.

Innere Klassen werden aus Kompatibiltätsgründen vom Java Compiler in ganz normale Klassen umgewandelt. Sie sehen in der Ausgabe des Compilers Klassen mit eigenartigen Namen, beispielsweise mit einem $ Zeichen. Dieses $ Zeichen trennt den Klassennamen der inneren Klasse vom Klassennamen von der äusseren Klasse.

In unserem Beispiel werden also folgende Klassendateien generiert:

```
TwoListenInner.class
TwoListenInner$MouseMotionHandler.class
TwoListenInner$MouseEventHandler.class
```

Und nun schauen wir uns an, wie die reverse engineered Java Dateien aussehen:

```
// FrontEnd Plus for JAD
// DeCompiled : TwoListenInner.class

package innereklasse;

import java.awt.*;
import java.awt.event.*;

public class TwoListenInner
{
    public class MouseEventHandler extends MouseAdapter
    {

        public void mouseEntered(MouseEvent e)
        {
            String s = "Die Maus betritt das Geb\344ude";
            tf.setText(s);
        }

        public void mouseExited(MouseEvent e)
        {
            String s = "Die Maus hat das Geb\344ude verlassen";
            tf.setText(s);
        }

        public MouseEventHandler()
        {
        }
    }
```

```
    public class MouseMotionHandler extends MouseMotionAdapter
    {

        public void mouseDragged(MouseEvent e)
        {
            String s = String.valueOf(String.valueOf((new
StringBuffer("Maus Position:  X = ")).append(e.getX()).append(" Y =
").append(e.getY())));
            tf.setText(s);
        }

        public MouseMotionHandler()
        {
        }
    }


    private Frame f;
    private TextField tf;

    public TwoListenInner()
    {
    }

    public static void main(String args[])
    {
        TwoListenInner that = new TwoListenInner();
        that.go();
    }

    public void go()
    {
        f = new Frame("Beispiel mit Inneren Klassen");
        f.add("North", new Label("Klicken und bewegen Sie die Maus"));
        tf = new TextField(30);
        f.add("South", tf);
        f.addMouseMotionListener(new MouseMotionHandler());
        f.addMouseListener(new MouseEventHandler());
        f.setSize(300, 200);
        f.setVisible(true);
    }

}
```

Das war die Java Quelle zur Klassendatei mit dem vollen Namen. Nun folgend die Java Dateien zu den generierten Klassendateien.

```
// FrontEnd Plus for JAD
// DeCompiled : TwoListenInner$MouseEventHandler.class

package innereklasse;

import java.awt.TextField;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

// Referenced classes of package innereklasse:
//            TwoListenInner

public class MouseEventHandler extends MouseAdapter
{

    public void mouseEntered(MouseEvent e)
    {
        String s = "Die Maus betritt das Geb\344ude";
        TwoListenInner.access$0(TwoListenInner.this).setText(s);
    }

    public void mouseExited(MouseEvent e)
    {
        String s = "Die Maus hat das Geb\344ude verlassen";
        TwoListenInner.access$0(TwoListenInner.this).setText(s);
    }

    public MouseEventHandler()
    {
    }
}
```

und schliesslich

```
// FrontEnd Plus for JAD
// DeCompiled : TwoListenInner$MouseMotionHandler.class

package innereklasse;

import java.awt.TextField;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;

// Referenced classes of package innereklasse:
//            TwoListenInner

public class MouseMotionHandler extends MouseMotionAdapter
{

    public void mouseDragged(MouseEvent e)
    {
        String s = String.valueOf(String.valueOf((new StringBuffer("Maus
Position:  X = ")).append(e.getX()).append(" Y = ").append(e.getY())));
        TwoListenInner.access$0(TwoListenInner.this).setText(s);
    }

    public MouseMotionHandler()
    {
    }
}
```

1.3.4.3. Anonyme Klassen

Sie können auch in Ausdrücken eine ganze Klasse verwenden. Diese Klassen haben vom Einsatz her keine Gelegenheit gross definiert und instanziert zu werden; sie werden definiert "bei der Instanzierung".

Innere Klassen können somit keine Konstruktoren besitzen. Sie machen Programmcode auch nicht lesbarer. Aber in vielen Fällen kann man damit die Programme schneller und lürzer schreiben, lesen sicher nicht!

Schreiben wir unsere innere Klasse um und verwenden eine anonyme Klasse:

```
    ...
    ...
13  public void go() {
14      f = new Frame("Two listeners example");
15      f.add ("North", new Label ("Click and drag the mouse"));
16      tf = new TextField (30);
17      f.add ("South", tf);
18
19      f.addMouseMotionListener (new MouseMotionAdapter() {
20    public void mouseDragged (MouseEvent e) {
21       String s = "Mouse dragging:  X = " + e.getX() + " Y = "
22                              + e.getY();
22       tf.setText (s);
23    }
25   } );  // <- KLAMMERN NICHT VERGESSEN!!!!! } für die Klassendefinition; ) für ...Listener(...
26
27   f.addMouseListener (new MouseEventHandler());
28   f.setSize(300, 200);
29   f.setVisible(true);
30 }
  ...
  ...
```

Sie finden auf dem Server und im Begelitbuch weitere Beispiele. Auf dem Server finden Sie Kombinationen von Interface mit inneren Klassen, statische innere Klassen und Zugriffe darauf sowie einiges mehr.

1.3.5. Lektion 4 - Die Vector Klasse

Dieses Kapitel sollte eigentlich leer sein und Sie dazu zwingen, in der Java API Dokumentation nachzuschlagen, was ein Vector ist, wie man ihn konstruiert und welche Members (Function Members: Methoden; und Data Members : Attribute) die Klasse hat.

Da ich zu gutmütig bin, haben ich doch einige Zeilen zum Thema geschrieben.

Die `java.util.Vector` Klasse stellt Methoden zur Verfügung, die es Ihnen erlauben, mit dynamischen Arrays zu arbeiten. Die Klasse ist eine Unterklasse der Klasse `Object` und implementiert das `Cloneable` Interface.

Jedes Vectorobjekt besitzt eine Kapazität, `capacity` und ein Kapazitätsinkrement, `capacityIncrement`. Beim Einfügen von neuen Elementen in den Vektor wird der Vektor erweitert, jeweils und das Inkrement. Die Kapazität des Vektors ist jeweils mindestens so gross wie die Länge des Vektors, in der Regel grösser.

Die Vektorklasse besitzt folgende Konstruktoren (schauen Sie bitte in der API Dokumentation nach ob das auch stimmt; Sie finden diese beispielsweise in der Hilfe m JBuilder [Help -> Help Topics -> Java 2 JDK 1.3 API Documentation : Java 2 Platform Packages java.util -> Class Vector]):

- `public Vector()`
- `public Vector(init initialCapacity)`
- `public Vector(init initialCapacity, int capacityInkrement)`

Fehlt einer? Haben Sie wirklich nachgeschaut?
Ja es fehlt einer: dieser verwendet als Parameter Collections, auf die wir noch eingehen werden. Aber für den Moment lassen wir ihn weg.

Der erste Konstruktor konstruiert einenleeren Vektor, der zweite einen Vektor mit einer Anfangskapazität, der dritte einen Vektor mit einer Anfangskapazität und einer Spezifikations des Zuwachsparameters.

Die Vektorklasse besitzt folgende Klassenfelder:

- `protected int capacityIncrement`
- `protected int elementCount`
- `protected Object elementData[ ]`

Falls Sie das Inkrement auf 0 setzen wird die Grösse des Buffers, des Vektors bei jeder nötigen Vergrösserung einfach verdoppelt.

1.3.5.1. Methoden der `Vector` Klasse

Hier einige der Methoden, die Ihnen zeigen, was man mit der Klasse so alles machen kann.

- `public final int size()`
- `public final boolean contains(Object element)`
- `public final int indexOf(Object element)`
- `public final synchronized elementAt(int index)`
- `public final synchronized void setElementAt(int index)`
- `public final synchronized void removeElementAt(int index)`
- `public final synchronized void addElement(Object obj)`
- `public final synchronized void insertElementAt(Object obj, int index)`

Die Aufgabe der Methoden ergibt sich aus deren Namen. Das synchronisieren des Zugriffs ist nötig, weil sonst, falls mehrere Prozesse oder Threads aktiv sind, beim gleichzeitigen Zugriff auf ein Element (Objekt) dieses in einen undefinierten Zustand gelangen könnte. Synchronisieren verhindert dies: es darf jeweils nur ein Prozess oder Teilprozess (Thread) diese Ressource verwenden. Wir werden dieses Thema im Rahmen der Diskussion der Threads detaillierter kennen lernen.

Selbsttestaufgabe
Welche Methoden der Klasse werfen Ausnahmen und welche?

Hinweis:
Sie finden die Ausnahmen in der API Dokumentation bei der detaillierteren Beschreibung der Methoden.

Beispiel:
`setSize public void` **`setSize`**`(int newSize)`

```
Sets the size of this vector. If the new size is greater than the current
size, new null items are added to the end of the vector. If the new size is
less than the current size, all components at index newSize and greater are
discarded.

Parameters:
newSize - the new size of this vector.

Throws:
ArrayIndexOutOfBoundsException - if new size is negative.
```

1.3.5.2. Beispiel für ein Vektor Template

Das folgende Beispiel verwendet einige Tricks, die über das hinausgehen, was wir bereits besprochen haben. Insbesondere werden alle Basisdatentypen (int, float, ...) in Objekte umgewandelt. Zu jedem Basisdatentyp gibt es eine sogenannte Wrapperklasse,also ein Mantel, der den Basisdatentypen zu einem Objekt macht und viele Methoden zur Verfügung stellt, mit dem dieses Objekt wieder in einen Basisdatentyp umgewandelt werden kann.

Auf diese Klassen werden wir sehr bald eingehen!

```
1   import java.util.*;
2
3    public class MeinVektor extends Vector {
4        public MeinVektor() {
5           super(1,1);        // storage capacity & capacityIncrement
6        }
7
8        public void addInt(int i) {
9            addElement(new Integer(i)); // addElement verlangt Object arg
10       }
11
12       public void addFloat(float f) {
13           addElement(new Float(f));
14       }
15
16       public void addString(String s) {
17           addElement(s);
18       }
19
20       public void addCharArray(char a[]) {
21           addElement(a);
22       }
23
24       public void printVector() {
25           Object o;
26           int length = size();       // vergleiche mit capacity()
27           System.out.println("Anzahl Elemente im Vektor " + length
28                              + "\n ... und hier sind die ELemente:");
29           for (int i = 0; i < length; i++) {
30              o = elementAt(i);
31              if (o instanceof char[]) {
32                  System.out.println(String.copyValueOf((char[]) o));
33              }
34              else
35                  //System.out.println(o.toString()); prints garbage
36                  System.out.println(o.toString());
37           }
38       }
```

```
39       public static void main(String args[]) {
40           MeinVektor v = new MeinVektor() ;
41           int digit = 5;
42           float real = 3.14f;
43           char letters[] = { 'a', 'b', 'c', 'd'};
44           String s = new String ("Das funktioniert ja alles prächtig!");
45
46           v.addInt(digit);
47           v.addFloat(real);
48           v.addString(s);
49           v.addCharArray(letters);
50
51           v.printVector();
52       }
53   }
```

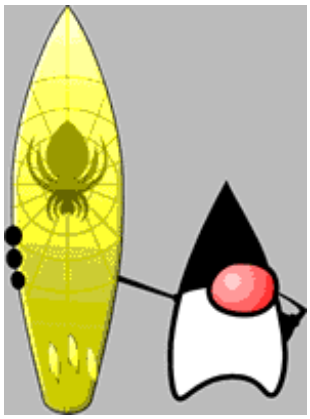
Mit dieser Klasse haben Sie ein brauchbares Hilfsmittel, um auch dynamische Bücherlisten, Videos und vieles mehr in Ihrem Bibliotheksprojekt einzubauen.

Das obige Beispiel, leicht modifiziert, um die Ausgabe besser formattieren zu können, liefert folgende Ausgabe:

```
Anzahl Elemente im Vektor 4
 ... und hier sind die ELemente:
Element 0: 5
Element 1: 3.14
Element 2: Das funktioniert ja alles prächtig!
Element 3: abcd
```

1.3.6. Innere Klassen - Übungen

In dieser Übung erweitern Sie ein Applet, welches wir, wegen der Umstellung des Stoffes, erst nächstes Mal kennen lernen. .



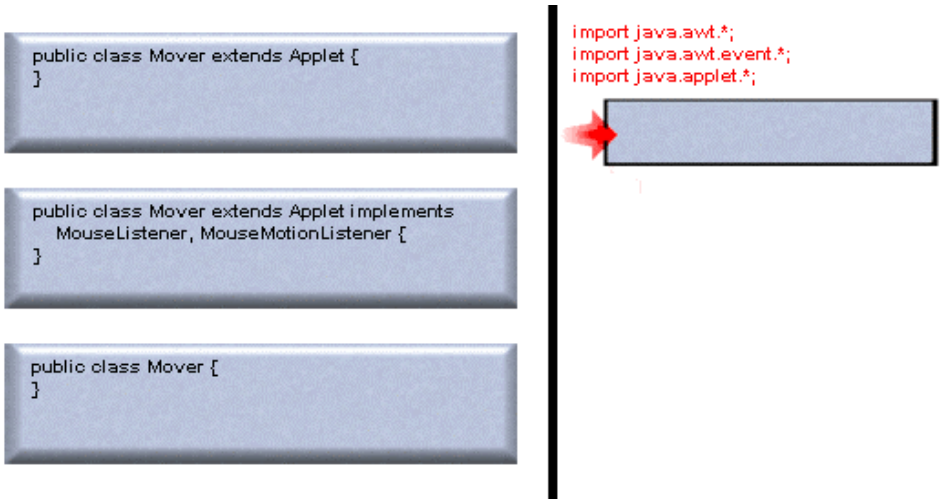
Das Applet spielt ein Audio, falls Sie mit der Maus über Duke fahren. Nun wollen wir das Applet mit inneren Klassen umprogrammieren. Der Einfachheit habe ich den Programmcode des Applets eingefügt. Sie werden ihn ohne grössere Probleme der Idee nach verstehen können. Die Details werden Sie nächstes Mal kennen lernen.

Programmcode für das ursprüngliche Applet:

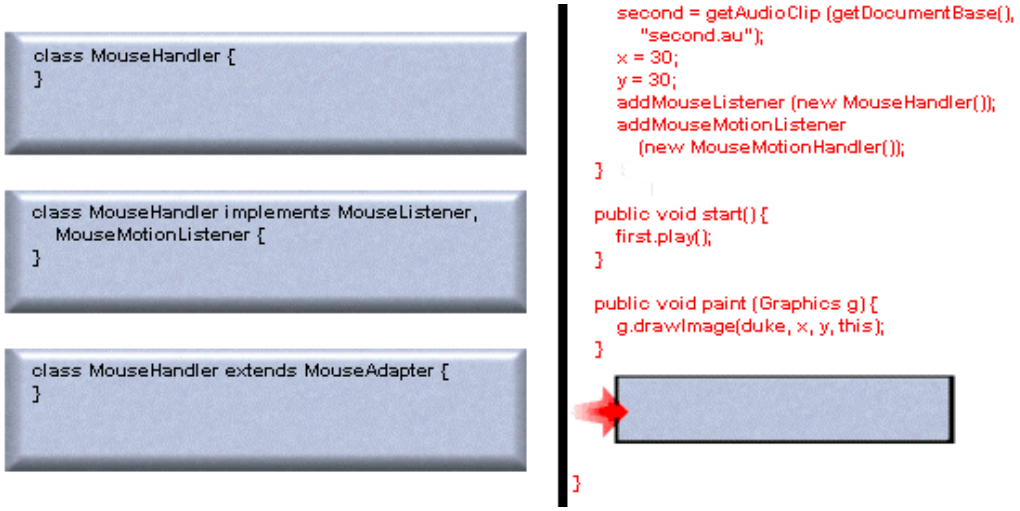
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Mover extends Applet
            implements MouseListener,MouseMotionListener {

      Image duke;
      AudioClip first,second;
      int x,y,imageWidth,imageHeight;

      public void init() {
            MediaTracker tracker = new MediaTracker(this);
            duke = getImage(getDocumentBase(),"duke.gif");
            tracker.addImage(duke,0);
            try {
                  tracker.waitForID(0);
            } catch (Exception e) {}

            imageWidth = duke.getWidth(this);
            imageHeight = duke.getHeight(this);
            first = getAudioClip(getDocumentBase(),"first.au");
            second = getAudioClip(getDocumentBase(),"second.au");
            x = 30;
            y = 30;
            addMouseListener(this);
            addMouseMotionListener(this);
      }

      public void start() {
            first.play();
      }

      public void paint(Graphics g) {
            g.drawImage(duke,x,y,this);
      }
```

```
	/* Mouse Listener */
	public void mousePressed(MouseEvent e) {
	}

	public void mouseReleased(MouseEvent e) {
	}

	public void mouseExited(MouseEvent e) {
	}

	public void mouseEntered(MouseEvent e) {
	}

	public void mouseClicked(MouseEvent e) {
		int mouseX = e.getX();
		int mouseY = e.getY();

		if (mouseX >= x && mouseX <= x+imageWidth) {
			if (mouseY >= y && mouseY <= y+imageHeight) {
				first.play();
			}
		}
	}

	/* Mouse Motion Listener */
	public void mouseDragged(MouseEvent e) {
	}

	public void mouseMoved(MouseEvent e) {
		int mouseX = e.getX();
		int mouseY = e.getY();

		if (mouseX >= x && mouseX <= x+imageWidth) {
			if (mouseY >= y && mouseY <= y+imageHeight) {
				second.play();
				try {
					Thread.sleep(250);
				} catch (Exception e1) {}
			}
		}
	}
}
```

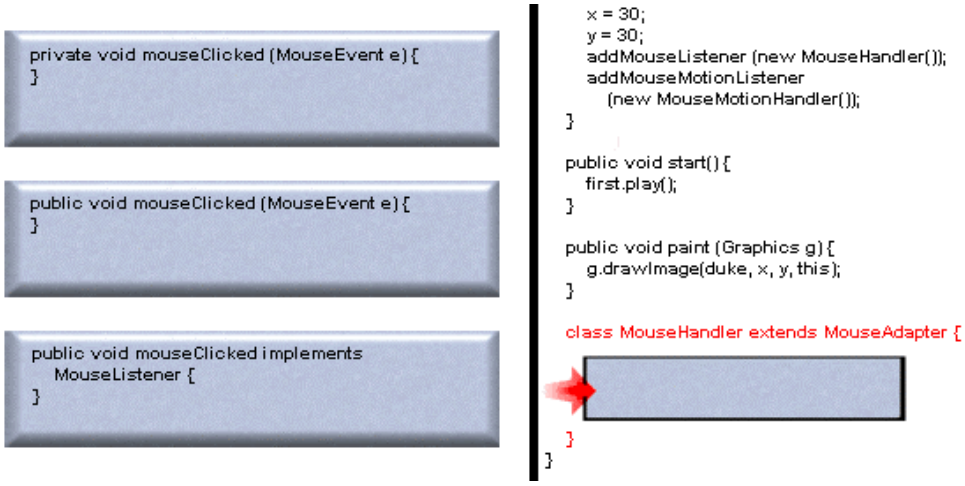
Sie müssen nun jeweils das für Sie plausible Bild auswählen, um den Programmcode zu ergänzen.



Die Antwort sehen Sie gleich im zweiten Schritt. Ergänzen Sie auch hier den Programmcode.



Und so geht's weiter:



Damit sind wir bei dieser Lösung angelangt:

```
public void start() {
   first.play();
}

public void paint (Graphics g) {
   g.drawImage (duke, x, y, this);
}


class MouseHandler extends MouseAdapter {
   public void mouseClicked(MouseEvent e) {
      int mouseX = e.getX();
      int mouseY = e.getY();

      if (mouseX >= x && mouseX <= x+imageWidth) {
         if (mouseY >= y && mouseY <= y+imageHeight)
            first.play();
         }
      }
   }
}
```

1.3.7. Quiz

Gegeben Sei folgendes Programm:

```
public class Test {
    int x = 0;
  static int y = 0;
  public Test () {
      x++;
        y++;
    }

    public static void main (String args []) {
        Test a = new Test ();
        Test b = new Test ();
        System.out.println ("in a, x is " + a.x +
                            " and y is " + a.y);
        System.out.println ("in b, x is " +b.x +
                            " and y is " +b.y);
    }
}
```

Welche Ausgabe erhalten Sie?[1]

a)
```
in a, x is 0 and y is 0
in b, x is 0 and y is 0
```
b)
```
in a, x is 1 and y is 1
in b, x is 1 and y is 1
```
c)
```
in a, x is 1 and y is 1
in b, x is 1 and y is 2
```
d)
```
in a, x is 2 and y is 1
in b, x is 2 and y is 1
```
e)
```
in a, x is 1 and y is 2
in b, x is 1 and y is 2
```

[1] e) ist korrekt

Ein Klassendesign verlangt, dass auf ein bestimmtes Member aus einer Unterklasse zugegriffen werden kann. Wie kann man dies erreichen?[2]

a) die Klasse muss public sein
b) die Klasse muss private sein
c) die Klasse muss protected sein
d) die Klasse benötigt einen anderen Modifier

Sie haben eine Klasse entwickelt, die Sie gerne schützen möchten. Insbesondere soll aus Unterklassen kein "spoofing" möglich sein (vererben und überschreiben, um dadurch den Zugriff zu erschleichen).
Welchen Modifier verwenden Sie?[3]

a) static
b) final
c) private
d) protected

Um auf den Konstruktor der Oberklasse zuzugreifen verwenden Sie folgendes Schlüsselwort:[4]

a) static
b) super
c) synchronized
d) transient

Sie möchten das grundlegende Verhalten der Klasse angeben, ohne die Methoden implementieren zu müssen. Welches Schlüsselwort spielt dabei die entscheidende Rolle?[5]

a) Inner
b) Interface
c) abstract
d) Deprecation

[2] c) **protected** besagt, dass man aus den Unterklassen, die sich auch in anderen Packages befinden können, darauf zugreifen kann.
[3] b) **final** verbietet weitere Verfeinerungen und das Überschreiben
[4] b) **super** bringt Sie in die Oberklasse
[5] c) **abstract** ist der passende *Modifier*

1.3.8. Zusammenfassung

In diesem Modul haben Sie gelernt:

- auf überschriebene Methoden zuzugreifen.
- auf überschriebene Konstruktoren zuzugreifen.
- Konstruktoren der Oberklasse gezielt einzusetzen.
- statische Variablen und Methoden einzusetzen.
- finale Klassen, Methoden und Variablen zu deklarieren und einzusetzen.
- abstrakte Methoden, Klassen und Interfaces zu deklarieren, zu verfeinern und zu implementieren.
- den Zugriffsmodifier protected (Zugriff in Packages und aus vererbten Klassen einzusetzen)
- mit -deprecated festzustellen, ob bestimmte Methoden veraltet sind
- innere Klassen zu beschreiben, zu verfeinern und einzusetzen

Das war's schon! Haben Sie das API der Vector Klasse schon nachgesehen? Sie sollten es tun, weil Sie so lernen, sich schnell eine Übersicht zu verschaffen. Die API Dokumentation ist sehr hilfreich, auch um Ihr Gedächtnis von nachschlagbarem Wissen zu entlasten.

1.4. Modul 2 : Stream I/O und Dateien

In diesem Modul

- Modul 2 : Stream I/O und Dateien
  - Einführung
  - Lektion 1 - Stream I/O
  - Lektion 2 - File I/O
  - File I/O Übungen
  - Quiz
  - Zusammenfassung

1.4.1. Einleitung

In diesem Modul zeigen wir, wie in Java Bytes und Zeichen eingelesen und geschrieben werden können. (I/O). Als erstes betrachten wir den Fall der Standard Eingabe und Ausgabe (`stdin, stdout, stderr`) als Stream basierten I/O.

Dann kümmern wir uns um die Dateien und Eingabe / Ausgabe im Zusammenhang mit Dateien (inklusive Verzeichnisse, ergänzen von Dateien usw.).

1.4.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein,

- das Stream Konzept vom `java.io` Package zu verstehen.
- Dateien zu konstruieren und Filter zu definieren.
- Unterschiede zwischen `Readers` und `Writers` und Streams kennen und wissen, wann welche einzusetzen sind.
- Dateien zu lesen, deren Charakteristiken zu bestimmen und Verzeichnisse anzulegen und zu lesen bzw. zu löschen.
- Daten und Text Dateien zu lesen und anzulegen bzw. zu mutieren.
- Objekte zu serialisieren, mit Hilfe des `Serialization` Interfaces.

1.4.2. Lektion 1 - Stream I/O

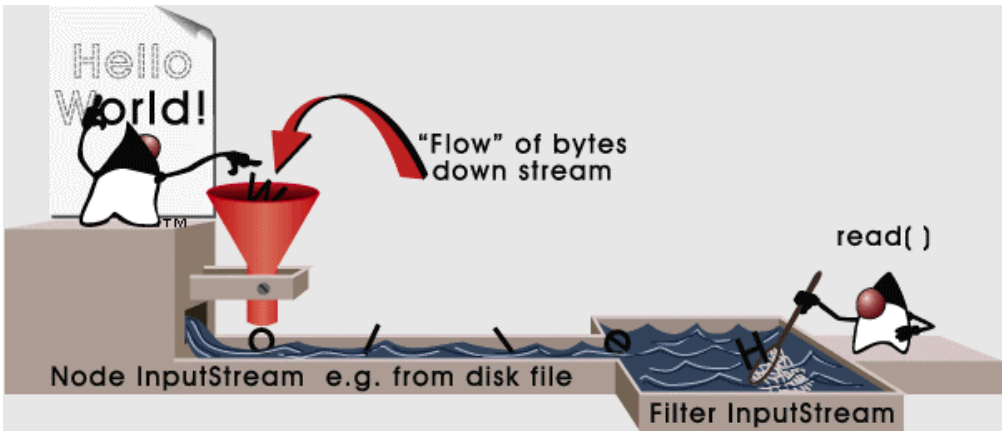
1.4.2.1. Stream Grundlagen

Ein Stream ist entweder eine Quelle oder eine Senke für Bytes und Zeichen. Natürlich ist die Unterscheidung wichtig: Sie können Daten aus einer Tastatureingabe lesen, aber nicht auf die Tastatur schreiben.

Dementsprechend unterscheidet man Input und Output Streams. Input Streams können Sie lesen; aber Sie können nicht in einen Input Stream schreiben. Damit Sie Zeichen aus einem Input Stream lesen können, muss dieser Input Stream eine Quelle besitzen.

Im `java.io` Package sind einige Streams "Knoten"['node'] - Streams, Ströme, die von einer bestimmten Stelle Daten lesen oder Daten an eine bestimmte Stelle schreiben.

Andere Ströme werden als Filter bezeichnet. Ein Filter Input Stream wird immer im Kontext eines Eingabestroms definiert. Dieser Eingabestrom liefert dem Filter die Daten.
Die folgende Illustration zeigt den groben Ablauf schematisch auf.



1.4.2.2. Methoden der Input Streams

Wir können nicht alle Methoden der Eingabeströme im Detail besprechen. Aber einige wichtige Methoden, grundlegende Methoden, sollten Sie schon kennen:

- die folgenden drei Methoden gestatten den grundlegenden Zugriff auf einen Eingabestrom.
  `int read()` liefert einen `int`, welcher ein Byte aus dem Stream enthält;
  falls die Methode -1 liefert, dann ist das Ende der Datei erreicht (EOF).

  `int read(byte[ ])` liefert einen `int`, welcher die Anzahl Bytes im Byte Array enthält, die aus dem Stream gelesen wurden;
  falls die Methode -1 liefert, dann ist das Ende der Datei erreicht (EOF).

  `int read(byte[ ], int, int)` liefert einen `int`, welcher die Anzahl Bytes im Byte Array enthält, die aus dem Stream gelesen wurden, wobei lediglich ein bestimmter Bereich gelesen wird;
  falls die Methode -1 liefert, dann ist das Ende der Datei erreicht (EOF).

  Sie sollten immer möglichst grosse Datenbereiche lesen (aus Effizientgründen)!

- falls Sie einen Strom nicht mehr benötigen, sollten Sie ihn schliessen. Der Garbage Collector hilft Ihnen dabei, aber es ist besser (und in den Demo Programmen ein Muss) die Dateien zu schliessen.

  `public void close()`

  Wenn Sie Filterströme verwenden, dann brauchen Sie nur den Strom zuoberst auf dem Stack zu schliessen, der Rest wird automatisch geschlossen.

- `public int available()`

  zur Verfügung. Die Anzahl hängt mit der internen Pufferung gelesener Daten zusammen. Im Programmbeispiel sehen Sie, dass in unserem Beispiel die genau Anzahl noch verfügbarer Bytes angegeben wird.

- `public long skip(long)`

  erreicht werden. Die Methode hat als Parameter die Anzahl Zeichen, die übersprungen werden sollen; die Methode liefert als Ergebnis die Anzahl Zeichen, die echt überprungen wurden.`boolean markSupported()`
  `void mark(int readlimit)`
  `void reset()`

  Die Grundfunktion ist, die aktuelle Position innerhalb einer Datei festzuhalten (zu markieren), falls diese Funktion unterstützt wird. Sie können zu einer markierten Position zurück kehren, wenn Sie die Methode `reset()` aufrufen.

1.4.2.3. Programmbeispiel

```
package iostreamsread;


import java.io.*;
public class DieReadMethoden {
  public static void main(String[] args) {
    try {
      // int read() ------------------------------------------------------------
      FileInputStream fis = new FileInputStream("Textdatei.txt");
      int i=0;
      while( (i=fis.read()) != -1) {
        System.out.print( (char)i);
      }
      fis.close();
      // int read(buf[ ]) ------------------------------------------------------
      fis = new FileInputStream("Textdatei.txt");
      i = 0;
      byte buf1[] = new byte[150];
      while( (i=fis.read(buf1)) != -1) {
        System.out.println("Anzahl gelesene Bytes: "+i);
        for (int j=0; j<i; j++) System.out.print((char)buf1[j]);
      }
      fis.close();
      // int read(buf[ ]) ------------------------------------------------------
      fis = new FileInputStream("Textdatei.txt");
      i = 0;
      byte buf2[] = new byte[300];
      while( (i=fis.read(buf2)) != -1) {
        System.out.println("\nAnzahl gelesene Bytes: "+i);
        for (int j=0; j<i; j++) System.out.print((char)buf2[j]);
      }
      fis.close();
      // int read(buf[ ], int, int) --------------------------------------------
      fis = new FileInputStream("Textdatei.txt");
      i = 0;
      byte buf3[] = new byte[200];
      while( (i=fis.read(buf3, 0, 200)) != -1) {
        System.out.println("\nAnzahl gelesene Bytes: "+i);
        for (int j=0; j<i; j++) System.out.print((char)buf3[j]);
```

```
      }
      fis.close();
      // int read(buf[ ], int, int) ----------------------------------------
      fis = new FileInputStream("Textdatei.txt");
      i = 0;
      byte buf4[] = new byte[200];
      while( (i=fis.read(buf4, 0, 100)) != -1) {
        System.out.println("\nAnzahl gelesene Bytes: "+i);
        for (int j=0; j<i; j++) System.out.print((char)buf4[j]);
      }
      fis.close();

      // int available() ---------------------------------------------------
      fis = new FileInputStream("Textdatei.txt");
      i = 0;
      byte buf5[] = new byte[200];
      while( (i=fis.read(buf5, 0, 10)) != -1) {
        System.out.println("\nAnzahl gelesene Bytes: "+i);
        System.out.println("\nAnzahl verfügbarer Bytes: "+fis.available());
        for (int j=0; j<i; j++) System.out.print((char)buf5[j]);
      }
      fis.close();
      // int available() ---------------------------------------------------
      fis = new FileInputStream("Textdatei.txt");
      i = 0;
      byte buf6[] = new byte[50];
      while( (i=fis.read(buf6, 0, 10)) != -1) {
        System.out.println("\nAnzahl gelesene Bytes: "+i);
        System.out.println("\nAnzahl verfügbarer Bytes: "+fis.available());
        for (int j=0; j<i; j++) System.out.print((char)buf6[j]);
      }
      fis.close();

    } catch(IOException ioe) {
      System.err.println("Datei Textdatei.txt wurde nicht gefunden");
    }
  try {
      // skip(long) --------------------------------------------------------
      FileInputStream fis = new FileInputStream("Nummerdatei.txt");
      int i=0;
      byte buf7[] = new byte[50];
      while( (i=fis.read(buf7, 0, 20)) != -1) {
        System.out.println("\nAnzahl gelesene Bytes: "+i+"\nÜberspringen von 10 Bytes");
        fis.skip(10);
        System.out.println("\nAnzahl verfügbarer Bytes: "+fis.available());
        fis.skip(10);
        for (int j=0; j<i; j++) System.out.print((char)buf7[j]);
      }
      fis.close();

      // mark..... ---------------------------------------------------------
      fis = new FileInputStream("Nummerdatei.txt");
      BufferedInputStream bis = new BufferedInputStream(fis,1000);
      i=0;
      int j=0;
      if (bis.markSupported()) System.out.println("\n\n\n\nMarkieren wird unterstützt");
      else System.out.println("\n\n\n\nMarkieren wird nicht unterstützt");
      while( (i=bis.read()) != -1) {
        j++;
        if (j==10) {
          bis.mark(10); // max 10 Byte weiterlesen
          System.out.println("Erste Marke bei 10 gesetzt");
          System.out.println("Zeichen an Position 10 : "+ (char)i);
        }
        if (j==25) {
          System.out.println("Zeichen an Position 25 : "+(char)i);
          System.out.println("Rücksprung auf Position 11 [markiert]");
          bis.reset();
          i=bis.read();
          System.out.println("Zeichen an Position reset()+1 : "+ (char)i);
        }
      }
      bis.close(); // top of stack reicht: fis wird auch geschlossen

      System.out.println("markieren und reset erst nachdem zuviele Bytes gelesen wurden");
      try {
        fis = new FileInputStream("Nummerdatei.txt");
        bis = new BufferedInputStream(fis,1000);
        i=0;
        j=0;
        while( (i=bis.read()) != -1) {
          j++;
          if (j==10) {
            bis.mark(5); // max 10 Byte weiterlesen
            System.out.println("Erste Marke bei 10 gesetzt; max 5 weitere Bytes lesen erlaubt
(sonst wird die Marke gelöscht)");
            System.out.println("Zeichen an Position 10 : "+ (char)i);
          }
```

```
          if (j==25) {
            System.out.println("Zeichen an Position 25 : "+(char)i);
            System.out.println("Rücksprung auf Position 11 [markiert]");
            bis.reset();
            i=bis.read();
            System.out.println("Zeichen an Position reset()+1 : "+ (char)i);
          }
        }
        bis.close(); // top of stack reicht: fis wird auch geschlossen
      } catch(IOException ioe) {
        System.err.println("Das mark / reset Spielchen ging schief");
      }

      // die folgende while Schlaufe führt zu einem Fehler, weil fis implizit geschlossen
wurde.
      System.out.println("\nAusgabetest");
      while( (i=fis.read()) != -1) {
        System.out.println( (char)i);
      }
    } catch(IOException ioe) {
      System.err.println("Datei Nummerdatei.txt wurde nicht gefunden");
    }
  }
}
```

1.4.2.4. Ausgabe des Programmbeispiels

Die folgende Ausgabe wurde gekürzt. Sie finden das Programm auf dem Web / Server / CD.

```
iostreamsread.DieReadMethoden

Das ist eine einfache Textdatei.
Ziel dieser Datei ist es, einfach als Eingabedatei für mehrere Testprogramme zu dienen.
Also
in diesem Sinne...
Anzahl gelesene Bytes: 150
Das ist eine einfache Textdatei.
Ziel dieser Datei ist es, einfach als Eingabedatei f?r mehrere Testprogramme zu dienen.
Also
in diesem Sinne...

...<gekürzt>

Anzahl gelesene Bytes: 100
Das ist eine einfache Textdatei.
Ziel dieser Datei ist es, einfach als Eingabedatei f?r mehrere Tes
Anzahl gelesene Bytes: 50
tprogramme zu dienen.
Also
in diesem Sinne...

Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 140
Das ist ei
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 130
ne einfach
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 120
e Textdate
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 110
i.
Ziel d
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 100
ieser Date
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 90
i ist es,
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 80
einfach al
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 70
s Eingabed
Anzahl gelesene Bytes: 10
```

```
Anzahl verfügbarer Bytes: 60
atei f?r m
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 50
ehrere Tes
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 40
tprogramme
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 30
 zu dienen
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 20
.
Also

Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 10
in diesem
Anzahl gelesene Bytes: 10

Anzahl verfügbarer Bytes: 0
Sinne...

...<gekürzt>

Anzahl gelesene Bytes: 20
Überspringen von 10 Bytes

Anzahl verfügbarer Bytes: 31
12345678901234567890
Anzahl gelesene Bytes: 20
Überspringen von 10 Bytes

Anzahl verfügbarer Bytes: -9

1234567890
1234567



Markieren wird unterstützt
Erste Marke bei 10 gesetzt
Zeichen an Position 10 : 0
Zeichen an Position 25 : 3
Rücksprung auf Position 11 [markiert]
Zeichen an Position reset()+1 : 1
markieren und reset erst nachdem zuviele Bytes gelesen wurden
Erste Marke bei 10 gesetzt; max 5 weitere Bytes lesen erlaubt (sonst wird die Marke gelöscht)
Zeichen an Position 10 : 0
Zeichen an Position 25 : 3
Rücksprung auf Position 11 [markiert]
Zeichen an Position reset()+1 : 1

Ausgabetest
Datei Nummerdatei.txt wurde nicht gefunden
```

Da wir den Buffer der Eingabedatei nicht weiter benötigten und gross genug angelegt haben, standen alle Daten nach dem verspäteten Rücksprung immer noch zur Verfügung. Die Fehlermeldung am Schluss zeigt, dass tatsächlich alle Streams geschlossen werden, wenn wir den top of the stack Stream schliessen.

1.4.2.5. Methoden der Ausgabeströme

Die Methoden der Ausgabe sind wieder sehr vielfältig, je nach Stream. Einige wichtige Methoden sind hier zusammengefasst:

- Die Ausgabe geschieht analog zur Eingabe: Byte für Byte oder als Byte Array.

```
public void write(int b) throws IOException
public void write(byte[ ] b)
public void write(byte[ ] b, int off, int len)
```

  Entweder wird ein einzelnes int gechrieben oder ein Byte Array oder ein Teil eines

Byte Arrays.

- Auch beim Schliessen verhält es sich analog wie bei der Eingabe
  `public void close() throws IOException`

- Eine wichtige Funktion ist das Leeren der Buffer bei der Ausgabe, sonst könnte im schlimmsten Fall ein Teil der Ausgabe verloren gehen:
  `public void flush() throws IOException`

1.4.2.6. Programmbeispiel

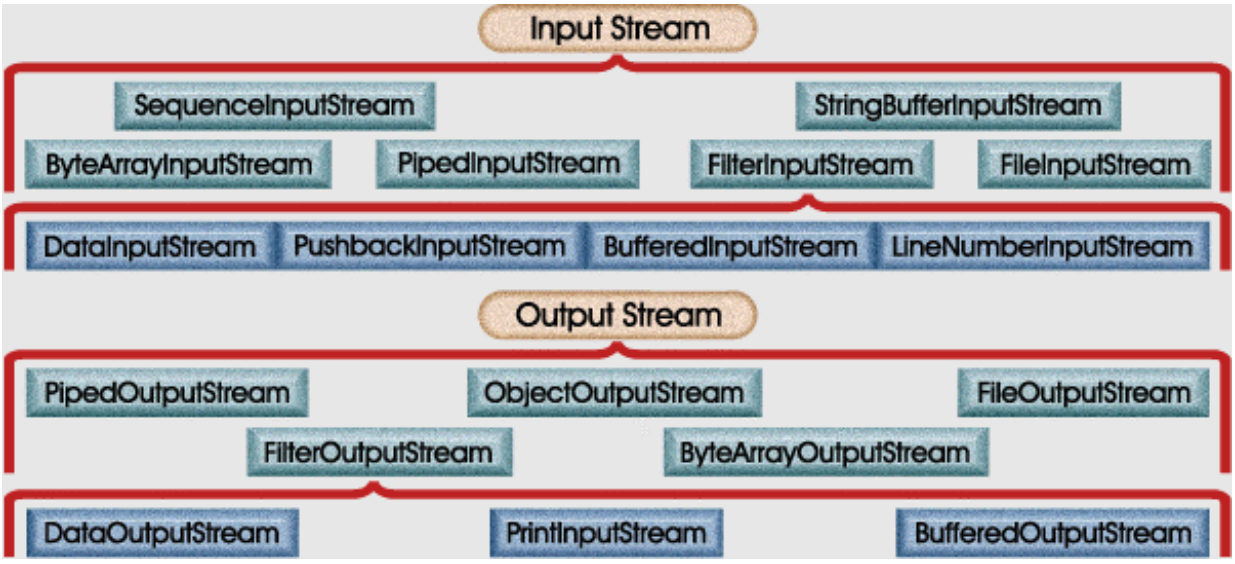
```
package iostreamwrite;

import java.io.*;

public class DieWriteMethoden {
  public static void main(String[] args) {
    try {
      FileOutputStream fos = new FileOutputStream("Ausgabe.txt");
      int i=0;
      fos.write((int)'H');
      fos.write((int)'a');
      fos.write((int)'l');
      fos.write((int)'l');
      fos.write((int)'o');

      fos.write( (int)'\n');

      byte buf1[ ]= {'H','e','u','t','e'};
      fos.write(buf1);

      fos.write( (int)'\n');

      byte buf2[ ]= {'i','s','t',' ','M','i','t','t','w','o','c','h'};
      fos.write(buf2, 4,7);

      fos.flush();
      fos.close();

    } catch(IOException ioe) {
      System.err.println("Fehler bei der Ausgabe");
    }
  }
}
```

1.4.2.7. Ausgabedatei

```
Hallo
Heute
Mittwoc
```

1.4.2.8. Grundlegende Stream Klassen

Im Package `java.io` sind sehr viele unterschiedliche Klassen definiert. Die Hierarchie der einzelnen Klassen sehen Sie im folgenden Chart:



Der Einfachheit halber beschränken wir uns bei der Diskussion auf die gängigen Klassen:

- Die `BufferedInputStream` und `BufferedOutputStream` Klassen kann man einsetzen, um die Effizienz der I/O Operationen zu steigern.

- Die `DataInputStream` und `DataOutputStream` Klassen gestatten es Ihnen Basisdatentypen direkt zu schreiben oder zu lesen, ohne Typenumwandlung. Dementsprechend stehen Ihnen verschiedene Methoden zur Verfügung. Zum Beispiel
  - `DataInputStream` Methoden
    - `byte readByte()`
    - `long readLong()`
    - `double readDouble()`
  - `DataOutputStream` Methoden
    - `void writeByte(byte)`
    - `void writeLong(long)`
    - `void writeDouble(double)`

  Die Methoden der `DataInputStream` und `DataOutputStream` treten entsprechend paarweise auf.

  ACHTUNG:
  die Methoden zum Lesen und Schreiben von Zeichenketten sind unzuverlässig und wurden durch Reader und Writer ersetzt.

- Die `FileInputStream` und `FileOutputStream` Klassen sind 'Node' Streams, also Ströme mit klaren Quellen und Senken, im Speziellen Dateien auf einem Speichermedium. In der Regel können Sie beim Konstruieren dieser Ströme gleich den Pfad im Dateisystem angeben.

  ACHTUNG:
  Falls Sie eine Ausgabedatei anlegen, wird diese jeweils wieder überschrieben.

```
FileInputStream infile = new FileInputStream("Daten.dat");
FileOutputStream outfile = new FileOutputStream("Daten.dat");
// die Datei Daten.dat wird jeweils überschrieben
```

  Wenn Sie Daten **anfügen** wollen, müssen Sie das **Append Flag** im Konstruktor setzen!

```
FileInputStream infile = new FileInputStream("Daten.dat");
FileOutputStream outfile = new FileOutputStream("Daten.dat", true);
// die Datei Daten.dat wird jeweils ergänzt (hinten angehängt)
```

- Piped Streams kann man einsetzen, um zwischen Threads zu kommunizieren.
  Ein `PipedInputStream` Objekt in einem Thread empfängt seine Eingabe aus einem komplementären `PipedOutputStream` Objekt in einem anderen Thread.
  Ein Piped Stream besteht normalerweise aus beiden Teilen, sonst ist das Konstrukt kaum sinnvoll.
  Wir werden ein Beispiel für Piped Streams im Rahmen der Threads kennen lernen.

1.4.2.9. URL Eingabe Streams

Da Java von Grund auf für die Programmierung in verteilten Systemen entwickelt wurde, ist es keine Überraschung, dass Sie auch direkt mit URLs kommunizieren können. Sie verwenden dazu einfach ein `URL` Objekt. Damit können Sie auch auf Audio oder Bilddateien zugreifen. Dies geschieht mit Hilfe der `getDocument()` Methode.

Jedes URL Objekt kann beispielsweise mit einem InputStream verbunden werden.

1.4.2.10. Kopieren einer Bilddatei

Dieses Beispiel setzt voraus, dass Sie einen Web Server lokal installiert haben (`localhost`) und dieser ein Unterverzeichnis `graphics` besitzt und sich darin eine Bilddatei `T1.gif` befindet.

```
package iourl;

import java.net.*;
import java.io.*;

public class LesenAbURL {
  public static void main(String[] args) {
    System.out.println("Lesen einer Bilddatei aus einer URL");
    try {
      URL imageSourceURL = new java.net.URL("http://localhost/graphics/T1.gif");
      System.out.println("Definition des URL InputStreams");
      InputStream is = imageSourceURL.openStream();
      int i=0;
      System.out.println("Definition der lokalen Bilddatei");
      FileOutputStream fos = new FileOutputStream("T1.gif");
      while ( (i=is.read()) !=-1) {
        fos.write(i);
      }
      is.close();
      fos.flush();
      fos.close();
      System.out.println("Die Bilddatei wurde ins lokale Verzeichnis kopiert");

    } catch (MalformedURLException e) {
      System.err.println("Fehler in der URL: ist der lokale Web Server korrekt konfiguriert");
    } catch (IOException ioe) {
      System.err.println("Fehler beim Schreiben der lokalen Datei");
    }
  }
}
```

Beachten Sie, dass Sie die Packages `java.io` und `java.net` importieren müssen!

1.4.2.11. Öffnen eines InputStreams

Sie können natürlich auch mit Applets aus URLs lesen, allerdings nur vom Host, von dem das Applet stammt. Auch in diesem Fall können Sie ein Unterverzeichnis angeben, aus dem die Datei gelesen werden soll.

```
import java.net.*;
import java.applet.*;
import java.io.*;

public class Test extends Applet{

   public  void init() {

      InputStream is;

      String datafile = new String("t");
      byte buffer[] = new byte[24];
      try {
         // new URL throws a MalformedURLException
         // URL.openStream() throws an IOException
```

```
            is = (new URL(getDocumentBase(), datafile)).openStream();
            is.read(buffer, 0, buffer.length);
        } catch (Exception e) {}

    }
}
```

Hier ein voll funktionierendes Beispiel, aus dem Applet Skript:

```
import java.awt.*;
import java.applet.Applet;

public class ShowDuke extends Applet {
      Image duke;
      public void init() {
            duke = getImage(getDocumentBase(),"graphics/T1.gif");
      }

      public void paint(Graphics g) {
            g.drawImage(duke, 25, 25, this);
      }
}
```

1.4.2.12. Unicode

In der Java Technologie verwendet man **Unicode** um Zeichen oder Zeichenketten darzustellen. Jedes Zeichen wird mit 16 Bit dargestellt. Java bietet auch entsprechende Streams, die **`Reader`** und **`Writer`**.

Das `java.io` Package verfügt über viele Readers und Writers. Die wichtigsten Varianten sind:

- `InputStreamReader` und
- `OutputStreamWriter`

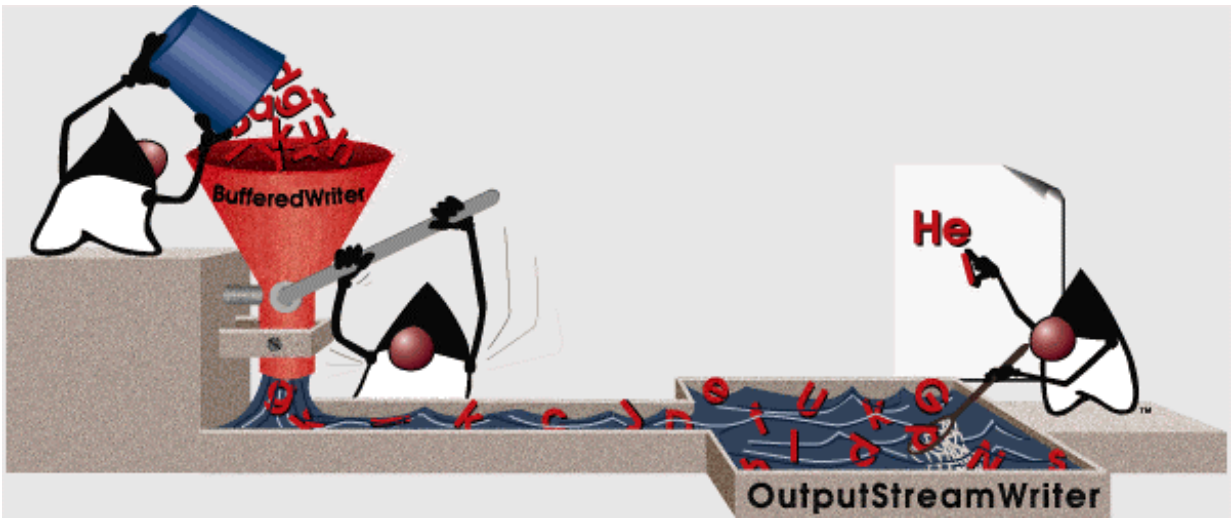
Diese Klassen stellen eine Schnittstelle dar zwischen Byte Streams und Readern und Writern. Zudem werden Konversionen zwischen Unicode und Darstellungen auf anderen Plattformen zur Verfügung gestellt.

```
package iounicode;
import java.io.*;
public class ReaderUndWriter {
  public static void main(String[] args) {
    System.out.println("Bridge zwischen Unicode und Character basierten IO Konstrukten");
    try {
      FileInputStream fis = new FileInputStream("EingabeDatei.txt");
      InputStreamReader isr = new InputStreamReader(fis);
      char cbuf[ ] = new char[100];
      int i;
      // hier wird jetzt ein Character Stream gelesen
      while( (i=isr.read(cbuf,0,100)) != -1) {
        for (int j=0;j<i; j++) System.out.print(cbuf[j]);
      }
```

```
      } catch(IOException e) {
      }
    }
  }
```

1.4.2.13. Buffered Reader und Writer

Die Konversion zwischen Formaten geschieht am effizientesten in grösseren Blöcken, wie die I/O Operationen. Daher ist es klar, dass gepufferte Reader und Writer sicher effizienter sind als normale.



Wie immer bei gepufferten Operationen muss man darauf achten, mit Hilfe der `flush()` Methode vor Abschluss des Programms alle Puffer zu leeren.

Für die Reader und Writer Klassen gibt es die im Wesentlichen gleichen Methoden wie für Streams. Allerdings betreffen die Lese- und Schreib-Methoden in diesem Fall nicht mehr Bytes und Byte-Puffer; an ihre Stelle treten Zeichen und Zeichen-Puffer (char).

Auch die Markierung und Skip Methoden sind genau wie bei den (Byte) Streams vorhanden.

1.4.2.14. Lesen von Eingabe-Zeichenketten

Das folgende Beispiel zeigt, wie auf einfache Art und Weise ab der Konsole (`System.in`) Zeichenketten eingelesen werden können. Wichtig ist das Abbruchkriterium. Hier müssen Sie sich in der Regel etwas einfallen lassen.

```
import java.io.*;

public class IOReaderWriter {
   public static void main(String args[]) throws IOException {
      String s;
      InputStreamReader ir;
      BufferedReader in;

      ir = new InputStreamReader(System.in);
      in = new BufferedReader(ir);

      while ((s = in.readLine()) != null) {
         System.out.println("Read: " + s);
      }
   }
}
```

1.4.2.15. Verwenden unterschiedlicher Zeichen Codierungen

Mit Readers und Writern haben Sie auch die Möglichkeit Character Codierungen anzugeben.

```
ir = new InputStreamReader(System.in, "8859_1")
```

Diese explizite Angabe des ISO Codes (hier ISO 8859_1 : Latin-1 und somit ASCII) garantiert eine korrekte Konversion der Datendarstellung, was insbesondere in verteilten Systemen wichtig sein kann, da die Daten aus unterschiedlichen Ländern stammen können.

1.4.3. Lektion 2 - Datei I/O

1.4.3.1. Kreieren eines File Objekts

Bevor man I/O mit einer Datei ausführen kann, benötigt das Java Laufzeitsystem Informationen über diese Datei. Die `File` Klasse stellt mehrere Hilfsmittel zur Verfügung, um Informationen über Dateien zu erhalten.

Hier einige Beispiele:

a) falls Sie lediglich eine Datei haben, dann ist folgender Konstruktor sinnvoll einsetzbar:

```
File meinFile;
meinFile = new File("meineDatei.dat");
```

b) falls Sie mehrere Dateien aus einem Verzeichnis bearbeiten müssen, ist der folgende Konstruktor der Situation besser angepasst:

```
meinFile = new File("/", "meineDatei.dat"); // Angabe des Verzeichnisses
```

oder

```
File meinDir = new File("/");
meinFile = new File(meinDir, "meineDatei.dat");
```

Mit den `File` Objekten kann man beispielsweise in den `FileInputStream` Objekten weiterarbeiten. Allerdings können Sie beim Konstruktor der `FileInputStream` Klasse auch einen externen Dateinamen in Form einer Zeichenkette angeben.

Der 'Umweg' über das `File` Objekt wird jedoch empfohlen!

1.4.3.1.1. Methoden der File Klasse

Ein File Objekt, eine Instanz der File Klasse, enthält bereits viel Informationen, und Sie können weitere Attribute für diese Dateien setzen. Aus den Methodennamen können Sie leicht deren Funktion herleiten. Für Details sollten Sie jedoch in der API Dokumentation nachschauen:

- File Namen
  - `String getName()`
  - `String getPath()`
  - `String getAbsolutePath()`
  - `String getParent()`
  - `boolean renameTo(File newName)`
- File Tests
  - `boolean exists()`
  - `boolean canWrite()`
  - `boolean canRead()`
  - `boolean isFile()`
  - `boolean isDirectory()`

- `boolean isAbsolute();`

- Generelle File Information und Utilities
  - `long lastModified()`
  - `long length()`
  - `boolean delete()`
- Directory Utilities
  - `boolean mkdir()`
  - `String[] list()`

1.4.3.1.2. Beispielprogramm

```
package iofileklasse;


import java.io.*;
public class FileObjekte {
  public static void main(String[] args) {
    String meinVerz = "d:\\NDKJava\\Programme\\IOFileKlasse";
    File meinFile = new File(meinVerz);

    // Ausgabe : mit Verzeichnis
    for (int i=0; i<meinFile.listFiles().length; i++) System.out.println("
      "+meinFile.listFiles()[i]);
    System.out.println("\n\n\n");
    // Ausgabe : nur Dateinamen, inkl. Verzeichnisse
    for (int i=0; i<meinFile.listFiles().length; i++) System.out.println("
      "+meinFile.list()[i]);
    System.out.println("\n\n\n");

    // Anlegen eines neuen Verzeichnisses
    // falls bereits vorhanden - geschieht nichts
    File neuesVerz = new File("temp");
    boolean bDir = neuesVerz.mkdir();
    for (int i=0; i<meinFile.listFiles().length; i++) System.out.println("
      "+meinFile.listFiles()[i]);
    // temp anlegen geht nur, falls das Verzeichnis vor dem Exit leer ist
    neuesVerz.deleteOnExit();

  }
}
```

1.4.3.2. Random Acces Dateien

Oft wollen Sie eine Datei nicht von Anfang bis zum Ende lesen. Sie möchten unter Umständen einen bestimmten Datensatz lesen, und ab diesem weiterlesen, diesen eventuell mutieren und weitere Datensätze einfügen.

Java stellt Ihnen hierfür die `RandomAccessFile` Klasse zur Verfügung.
Sie können random accesss Dateien auf zwei Arten öffnen:

- mit einem Dateinamen
  `meinRAFile = new RandomAccessFile(String name, String mode);`
- mit einem `File` Objekt
  `meinRAFile = new RandomAccessFile(File file, String mode);`

Das `mode` Argument im Konstruktor bestimmt, ob Sie lediglich lesen oder lesen/schreiben wollen: read `r` oder read/write `rw` Zugriff auf die Datei.
Java bietet Ihnen Hilfe bei der Navigation innerhalb einer Random Access Datei:

- `long getFilePointer();`
  liefert die Position des aktuellen Pointers : wo stehen wir in der Datei.
- `void seek(long pos);`
  setzt den File Pointer an die betreffende Stelle ***in Bytes***, absolut ab Dateianfang. Position `0` markiert den Beginn der Datei.
- `long length();`
  zeigt Ihnen die Länge der Datei. Die Position `length()` markiert das Ende der Datei.

Sie könnten Beispielsweise eine Datenbankdatei folgendermassen öffnen:

```
RandomAccessFile myRAFile;
myRAFile = new RandomAccessFile("db/Kunden.dbf","rw");
```

1.4.3.2.1. Beispielprogramm

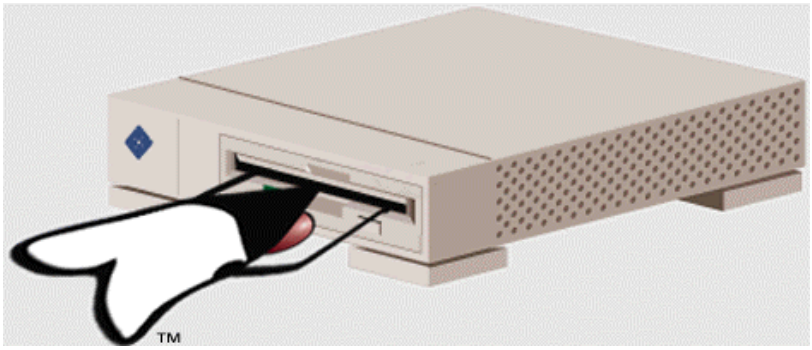
```
package iorandomfileklasse;

import java.io.*;
public class RandomZugriff {
  public static void main(String[] args) {
    try {
      RandomAccessFile meineDB = new
          RandomAccessFile("db/Kunden.dbf","rw");
      // positionieren : Zeichen ab Dateianfang
      meineDB.seek(5);
      System.out.println("Länge der Datei:"+meineDB.length());
      for (int i=0; i<3; i++) System.out.println(" "+meineDB.readLine());
      meineDB.writeUTF("eingefügte Zeile");
      meineDB.close();
    } catch(IOException ioe) {
      System.err.println("Fehler beim Lesen der Random Access Datei");
    }
  }
}
```

WARNUNG:
schauen Sie sich die Funktionsweise des Random Access Mechanismus genau an bevor Sie ihn einsetzen! Falls Sie den Mechanismus nicht 100% verstanden haben, werden Ihre Dateien innert kürzester Zeit unbrauchbar sein!

1.4.3.3. Serialisierung von Objekten

In JDK 1.1 wurde das `java.io.Serializable` Interface eingeführt und die JVM so angepasst, dass Objekte in Streams geschrieben werden können.

Man bezeichnet das Speichern eines Objekts auf permanenten Speicher als *persistence*. Ein Objekt bezeichnet man als persistent, falls man das Objekt auf Disk oder Tape, oder ... abspeichern kann, zu einer anderen Maschine transportieren kann und dort wieder einlesen und weiterverwenden kann... (fast!).



Das `java.io.Serializable` Interface besitzt selbst keine Methoden und dient in diesem Sinne als "Marker", als Kennzeichen dafür, dass ein Objekt serialisiert werden kann. Objekte aus Klassen, die `Serializable` nicht implementieren, können weder gespeichert (serialisiert) noch wiederhergestellt werden.

Gespeichert wird der Zustand, also die Werte der Member Variablen, die Data Members. Die Methoden selbst werden nicht abgespeichert. Damit ist auch der Transpost auf Disketten und problemlose Restart auf einem anderen Rechner nur möglich, falls die Klassenbeschreibung auf beiden Rechnern vorhanden ist.

1.4.3.3.1. Objekt Graphen

Gespeichert wird der Zustand, also die *Daten*, Werte der Member Variablen, die Data Members. Die Methoden selbst werden nicht abgespeichert. Damit ist auch der Transpost auf Disketten und problemlose Restart auf einem anderen Rechner nur möglich, falls die Klassenbeschreibung auf beiden Rechnern vorhanden ist.

Falls die Data Members selbst auch Objekte sind, muss diese Zusatzinformation auch abgespeichert werden. Dies geschieht im sogenannten Objekt *Graphen*, einer Baumstruktur, bei der auch Unterobjekte berücksichtig werden.

Einige Objekte kann man nicht serialisieren:
falls sich die Daten dauernd ändern, macht es wenig Sinn den Zustand amszuspeichern.
Beispiele für nichtserialisierbare Objekte, Klassen, sind `java.io.FileInputStream`, `java.io.FileOutputStream` und `java.lang.Thread`.

Falls ein serialisierbares Objekt eine Referenz auf ein nicht-serialsierbares Objekt enthält, kann das gesamte Objekt nicht serialsiert werden. Allerdings können Sie Objekte explizit mit dem Schlüsselwort `transient` kennzeichnen, um anzugeben, dass Sie keinen Wert auf die aktuellen Werte dieser Variable, dieses Objekts legen.

Die Zugriffsmodifier `public`, `protected` und `private` haben keinen Einfluss auf die Datenfelder bei der Serialisierung.

Die Daten werden als Bytes, Strings als UTF Zeichen abgespeichert. Alles was als `transient` gekennzeichnet wird, wird nicht serialisiert.

In diesem Fall wird die Zeichenkette `customerID` serialisiert:

```
1    import java.io.*;
2    public class MyClass implements Serializable {
3        public transient Thread myThread;
4        private String customerID;
5      private int total;
       ...
```

In diesem Fall wird die Zeichenkette `customerID` **nicht** serialisiert:

```
6    import java.io.*;
7    public class MyClass implements Serializable {
8        public transient Thread myThread;
9        private transient String customerID;
9        private int total;
...
```

1.4.3.3.2. Schreiben eines Objekt Streams

Das Schreiben eines Objekts in einen Datei Stream ist sehr einfach. Das Programmfragment unten zeigt, wie dies aussehen könnte. Sie serialsieren ein `java.util.Date` Objekt in eine Datei.

Zuerst wird ein `Date` Objekt `d` kreiert und ein Ausgabestrom `f` mit dem Namen `date.ser` instanziert.

Dann wird ein Objekt Ausgabestrom `s` kreiert und an den Datei-Ausgabestrom `f` angehängt. Schliesslich wird mit der `writeObject()` Method das `Date` Objekt in die `date.ser` Datei geschrieben.

```
import java.io.*;
import java.util.*;

public class MyClass implements Serializable {

    Date d = new Date();

    public void write() {
       try {
          FileOutputStream f = new FileOutputStream("date.ser");
          ObjectOutputStream s = new ObjectOutputStream (f);
          s.writeObject (d);
          s.close ();
       } catch (IOException e) {
          e.printStackTrace();
       }
    }
}
```

1.4.3.3.3. Lesen eines Objekt Streams

Das Lesen eines Objekts, welches serialsiert wurde, ist genau so einfach. Das Programmfragment verwendet die `readObject()` Methode und liefert ein generellles `Object`, welches noch passend umgewandelt (gecastet) werden muss.

```
import java.io.*;
import java.util.*;
public class MyClass implements Serializable {
    Date d = new Date();
    public void read() throws ClassNotFoundException {
       try {
          FileInputStream f = new FileInputStream("date.ser");
          ObjectInputStream s = new ObjectInputStream (f);
          d = (Date)s.readObject();
       } catch (IOException e) {
          e.printStackTrace();
       }
       System.out.println("Date serialized at " + d);
    }
}
```

1.4.3.3.4. Vollständiges Serialsierungsbeispiel

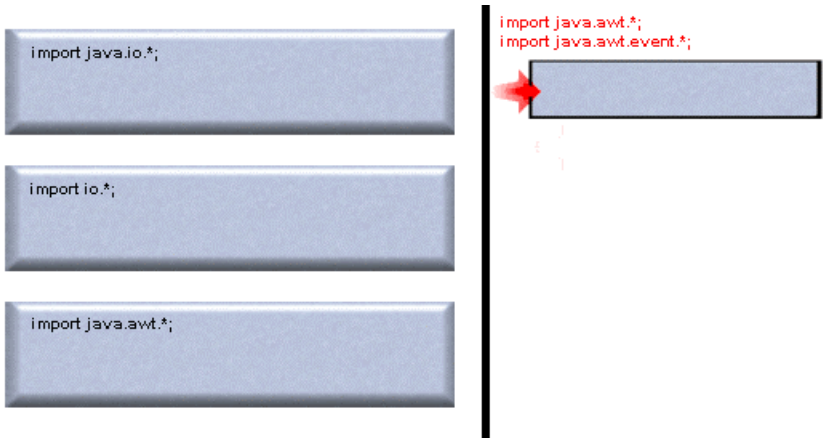
```
package ioobjektserialisierung;

import java.io.*;
import java.util.*;
public class SchreibenUndLesenEinesObjekts implements Serializable{
  public static void main(String[] args) {
    Date d = new Date();
    System.out.println("Aktuelles Datum : "+d+" exakt: "+d.getTime());
    try {
      FileOutputStream f = new FileOutputStream("date.ser");
      ObjectOutputStream s = new ObjectOutputStream (f);
      s.writeObject (d);
      s.close ();
    } catch (IOException e) {
      e.printStackTrace();
    }
    // zum Testen des ersten Teile einfach exit aktivieren
    //System.exit(0);
    // warte zwei Sekunden
    try {
      Thread.sleep(2000);
    } catch (InterruptedException ie) {
      System.err.println("Der Thread wurde im Schlaf erschlagen");
    }
    try {
      FileInputStream f = new FileInputStream("date.ser");
      ObjectInputStream s = new ObjectInputStream (f);
      d = (Date)s.readObject();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException cnfe) {
      System.err.println("Pech: die Klasse wurde nicht gefunden");
      cnfe.printStackTrace();
    }

    System.out.println("Das Objekt wurde am "+d+" serialisiert, exakt : "+d.getTime());
    d = new Date();
    System.out.println("Datum und Zeit jetzt : "+d+" exakt : "+d.getTime());

  }
}
```

```
Aktuelles Datum : Thu Mar 01 11:18:53 GMT+01:00 2001 exakt: 983441933583
Das Objekt wurde am Thu Mar 01 11:18:53 GMT+01:00 2001 serialisiert, exakt : 983441933583
Datum und Zeit jetzt : Thu Mar 01 11:18:56 GMT+01:00 2001 exakt : 983441936407
```

1.4.4. File I/O Übung

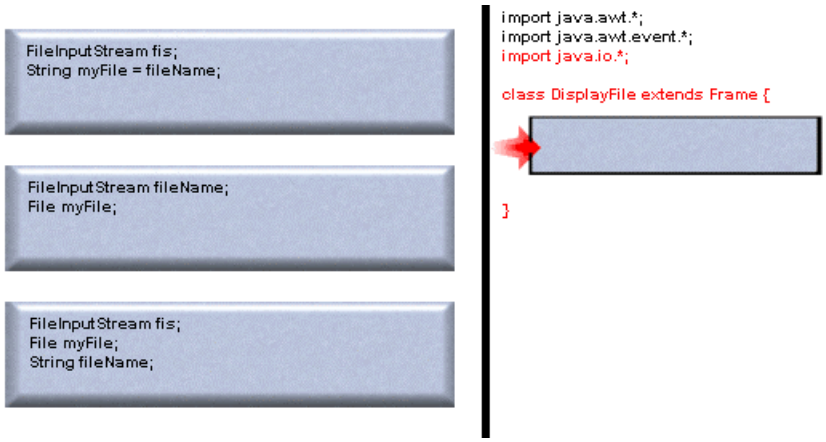
In dieser Übung können Sie nachvollziehen, wie eine Applikation den Inhalt einer Datei anzeigt.

a) welches Programmfragment müssen Sie einfügen, um Dateien anbinden zu können?

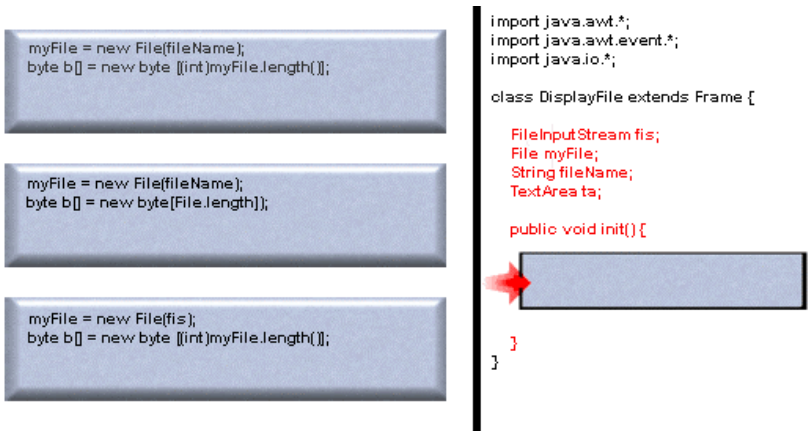


Sicher haben Sie die korrekte mittlere Variante ausgewählt!

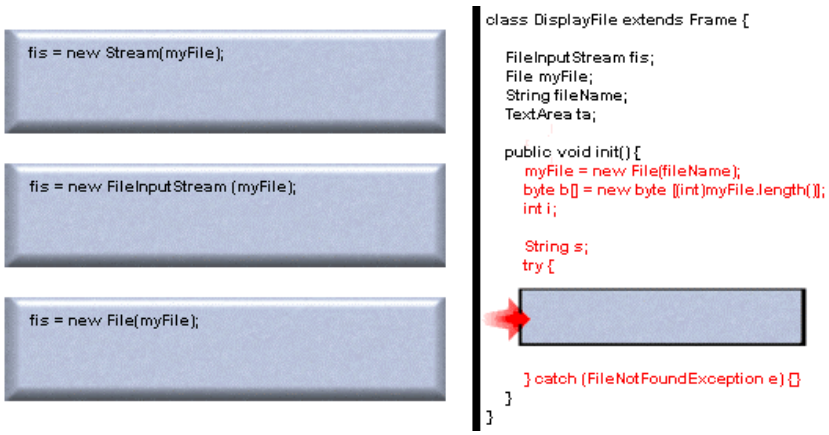
b) als nächstes müssen Sie schrittweise das Programm ergänzen, vergessen Sie die Zeichenkette für das Fileobjekt nicht!



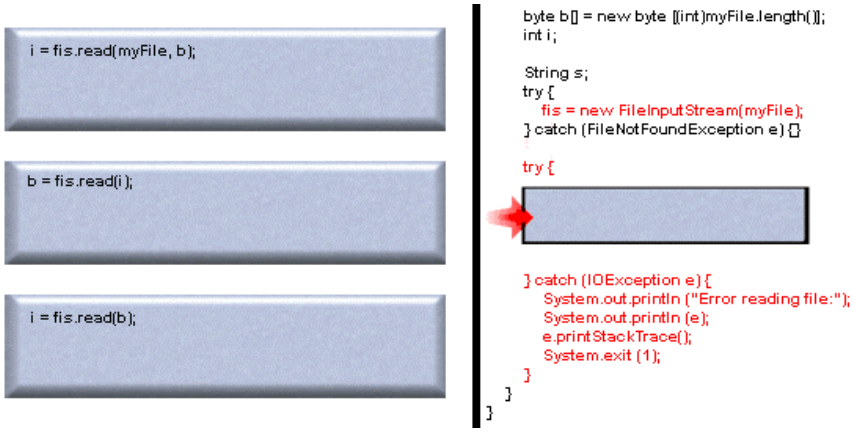
c) wie geht's jetzt wohl weiter?



d) jetzt geht's dem Ende zu



e) zum Lesen benötigen Sie



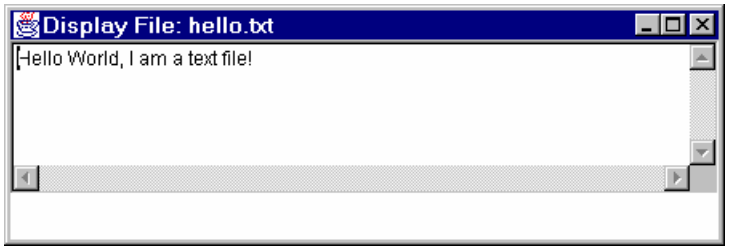
f) das war's

```
    TextArea ta;

    public void init() {
        myFile = new File(fileName);
        byte b[] = new byte [(int)myFile.length()];
        int i;

        String s;
        try {
            fis = new FileInputStream(myFile);
        } catch (FileNotFoundException e) {}

        try {
            i = fis.read(b);
        } catch (IOException e) {
            System.out.println ("error reading file:");
            System.out.println (e);
            e.printStackTrace();
            system.exit(1);
        }
    }
}
```

Jetzt wollen wir mal sehen, wie ein Programm ausehen müsste, welches nach folgenden Befehlen

```
javac DisplayFile.java
java -cp . DisplayFile hello.txt
```

folgende Ausgabe liefert:



Ich mach's Ihnen einfach - auf der folgenden Seite finden Sie die Lösung!

Aufgabe (ohne Musterlösung):
schreiben Sie ein Programm, welches eine Zeichenkette (als Objekt) serialisiert abspeichert, einmal normal und einmal transient und ändern Sie die Zeichenkette vor dem Wiedereinlesen.

```
package iodisplayfile;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

class DisplayFile extends Frame {
    FileInputStream fis;
    File myFile;
    String fileName;
    TextArea ta;

    public void init() {
        myFile = new File(fileName);
        byte b[] = new byte [(int)myFile.length()];
        int i;
        String s;

        try {
            fis = new FileInputStream(myFile);
        } catch (FileNotFoundException e){ }

        try {
            i = fis.read(b);
        } catch(IOException e){
            System.out.println ("Fehler beim Lesen der DAtei:");
            System.out.println (e);
            e.printStackTrace();
            System.exit (1);
        }
        s = new String(b);
        ta = new TextArea(s, 5, 40);
        add (ta, "North");

        setSize(450, 150);
        setVisible(true);
        addWindowListener (new WindowHandler());
    }

    public DisplayFile(String file) {
        super("Display File: " + file);
        fileName = file;
    }

    public static void main (String args[]) {
        if (args.length < 1) {
            System.out.println ("Usage:");
            System.out.println ("java DisplayFile <file>");
            System.exit (1);
        }
        DisplayFile disp = new DisplayFile(args[0]);
        disp.init();
    }

    private class WindowHandler extends WindowAdapter {
        public void windowClosing (WindowEvent e) {
            setVisible(false);
            dispose();
            System.exit(0);
        }
    }
}
```

1.4.5. Quiz

a) Erklären Sie Ihrem Nachbarn, was man unter einer Wrapper Class bei den Eingabe / Ausgabe Klassen versteht. Anders ausgedrückt: wie gelangen Sie von InputStream zu

BufferedInputStream? Was geschieht eigentlich?

b) Was versteht man unter Nodeklassen im Zusammenhang mit Input / Output?

c) Mit welchen Streams können Sie zwischen Threads kommunizieren?
Kennen Sie ein ähnliches Konzept auf Betriebsystemebene (Windows/DOS, Unix)?

d) Mit welcher Methode können Sie den Namen einer Datei, eines Fileobjekts bestimmen?

e) Welche *mode* Argumentewerte gibt es für Random Access Dateien?

1.4.6. Zusammenfassung

In diesem Modul haben Sie folgende Konzepte kennen gelernt:

- das Stream Konzept des java.io Packages
- die Konstruktion von Dateien und Streams
- den Unterschied zwischen Streams und Reader/Writer
- die Manipulation von Dateien und Verzeichnissen
- lesen, schreiben und mutieren von Textdateien und Datendateien
- Serialisieren von Objekten und das Einlesen von Objekt(daten).

1.5. Modul 3 : Threads

In diesem Modul

- Modul 3 : Threads
  - Lektion 1 - Threading in Java
  - Lektion 2 - Ablaufsteuerung von Threads
  - Praktische Übung
  - Lektion 3 - Synchronisation in Java
  - Lektion 4 - Thread Interaktion (wait/notify)
  - Lektion 5 - Producer / Consumer Beispiel
  - Praktische Übung
  - Quiz
  - Zusammenfassung

1.5.1. Einleitung

Viele Software Probleme kann man am besten lösen, wenn man mehrere nebenläufige Kontrollflüsse (Threads) zulässt. Zum Beispiel:

- interaktive Spiele;
- das gleichzeitige Herunterladen und Anzeigen
- Kommunikation (senden und empfangen)

1.5.1.1. Lernziele

In diesem Modul lernen Sie die Java Beschreibung solcher Nebenläufigkeiten mit Java Threads kennen. Ziel ist es, dass Sie erkennen, wo die Probleme liegen und wie man einige davon vermeiden und andere lösen kann.

Nach dem Durcharbeiten dieses Moduls sollten Sie:

- wissen was ein Threasd ist.
- wissen wie man Threads kreiert, deren Ausführung kontrolliert und wie der Zugriff auf gemeinsame Daten synchronisiert werden.
- Thread Programme schreiben können, die (weitestgehend) plattformunabhängig sind.
- einige der Probleme verstehen, die in nebenläufigen Systemen auftreten können, falls Daten gemeinsam genutzt werden.
- das Schlüsselwort `synchronized` verstehen, mit dem in Java vermieden werden kann, dass Daten in einen undefinierten Zustand gelangen können.

1.5.2. Lektion 1 - Threading in Java

1.5.2.1. Was sind Threads?

Eine einfaches aber brauchbares Bild eines Rechners besteht darin, dass eine CPU Berechnungen durchführt, im ROM bestimmte Steuerprogramme gespeichert werden, die CPU im RAM Daten und Programme zwischenspeichern kann und Daten aus persistenten Speichern bearbeitet und verarbeitet werden.

Dieses einfache Beispiel hat eine Schwachstelle: es wird jeweils nur eine Tätigkeit ausgeführt. Die Realität ist komplexer: in der Regel kann das Betriebsystem dafür sorgen, dass mehrere Programme jweils eine bestimmte Zeitscheibe der CPU zur Verfügung erhalten und der Reihe nach schrittweise ausgeführt werden

Aus der Sicht des Programmierers sieht die Situation so aus, dass er am liebsten mehrere Prozesse nebeneinander starten und laufen lassen möchte. Er möchte also sozusagen gleichzeitig mehr als einen Rechner oder mindestens mehrere CPUs zur Verfügung haben. In diesem Sinne kann man unter *Thread* oder *Ausführungskontext* eine *virtuelle* CPU plus eigener Programmcode und Daten verstehen. Vereinfacht gesagt handelt es sich um bei Threads um 'leichtgewichtige' Prozesse, Kontrollflüsse.

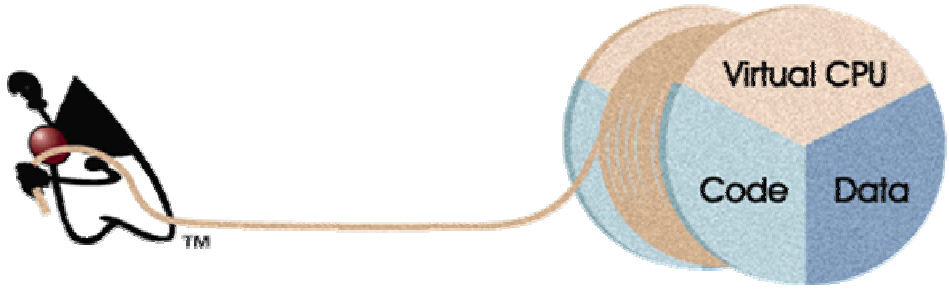
Die Klasse `java.lang.Thread` enthält die wichtigsten Threading Bausteine, Methoden, mit denen Threads kreiert und kontrolliert werden können. Der Präfix zeigt, dass es sich um Klassen handelt, die zum Kernsystem von Java gehören, also nicht nachträglich irgendwie noch dazu gekommen sind.

`Thread` wird im Folgenden verwendet, um diese Klasse `java.lang.Thread` zu beschreiben; mit Thread bezeichnen wir den Ausführungskontext.

1.5.2.2. Bestandteile eines Threads

Ein Thread besteht aus den folgenden drei Hauptbestandteilen:

- einer virtuellen CPU.
- dem Programmcode, der von der CPU ausgeführt wird.
- die Daten, mit denen der Programmcode arbeitet.



Die virtuelle CPU wird in Java durch die `Thread` Klasse repräsentiert. Falls ein Thread konstruiert wird, erhält er den Programmcode und die Daten im Konstruktor mitgeteilt.

Die drei Bestandteile des Threads sind effektiv voneinander unabhängig. Im Falle des Programmcodes dürfte dies klar sein: jeder Thread wird wohl seine eigene Kontrollflussbeschreibung in Form des Programmcodes haben.

Unterschiedliche Thread können auf die selben oder unterschiedliche Daten zugreifen.

1.5.2.3. Kreieren eines Threads

Schauen wir uns genauer an, wie ein Thread kreiert wird und welche Argumente der Konstruktor aufnehmen kann.

Sie können Threads auf zwei Arten kreieren, wie Sie sehr schnell feststellen werden:

- auf der einen Seite können Sie Threads als Instanzen der Klasse `Thread` definieren.
- auf der anderen Seite können Sie das Interface `Runnable` implementieren. `Runnable` enthält nur eine Methode: `run()`.

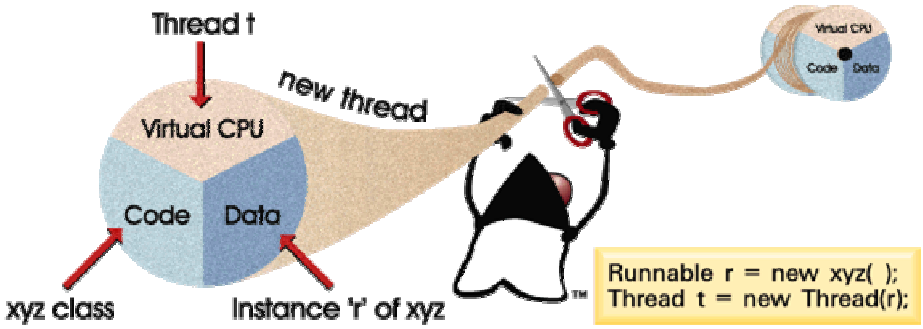
Die erste Vorgehensweise erscheint Ihnen vielleicht naheliegender. Die zweite ist aber flexibler. Daher werden wir uns zuerst mit der zweiten beschäftigen.

Der Konstruktor der Klasse `Thread` akzeptiert unsere Implementierung des `Runnable` Interfaces, eine Instanz von `java.lang.Runnable`.

```
1    public class xyz implements Runnable {
2        int i=0;
3        public void run() {
4            while (i<10) {
5                System.out.println("Hello " + i++);
6            }
7        }
8    }

9    Runnable r = new xyz();
10   Thread t = new Thread(r);
```

Der Thread `t`, als Instanz der Threadklasse, wird den Programmcode ausführen, der in der `run()` Methode der Definition der Klasse `xyz` beschrieben ist. Die Definition von `Runnable` verlangt, dass `public void run()` implementiert wird. Die Daten werden durch die Instanz der Klasse `xyz` also in unserem Falle durch `r` an den Konstruktor des Threads übergeben.

Sie referenzieren also mittels einer Instanz der Klasse `Thread` auf Ihren Thread. Der Programmcode stammt aber vom Argument des Konstruktors. Dieses Argument muss das `Runnable` Interface implementieren. Die Daten für den Thread stammen von der Instanz der Klasse, die Runnable implementiert.



1.5.2.4. Starten des Threads und Scheduling mehrerer Threads

Nachdem Sie einen Thread kreiert haben, bleibt er inaktiv, er tut rein gar nichts! Es liegt an Ihnen, den Thread zu starten und damit zum Leben zu erwecken. Die Klasse `Thread` stellt zum Starten die Methode `start()` zur Verfügung (Sie sollten nicht die ebenfalls vorhandene Methode `run()` der Klasse `Thread` anwenden!):

```
t.start();
```

Nach dem Starten wird sozusagen die virtuelle CPU gestartet, der Thread beginnt aktiv zu werden. Aber eventuell startet er nicht gleich! Da das System nur eine physische CPU hat, kann die virtuelle CPU eventuell nicht gleich gestartet werden: sie muss warten, bis der Scheduler der JVM ihr eine Chance gibt, zu starten.

Je nach JVM Implementation wird der Scheduler unterschiedlicher implementiert. Grundsätzlich sollte der Scheduler in der Regel 'pre-emptiv' sein, nicht notwendigerweise zeitscheibenbasiert.

Pre-Emptiv besagt, dass mehrere Threads bereits sein können, aber lediglich einer ausgeführt wird. Dieser Thread wird entweder bis zu Ende ausgeführt oder ein Thread mit höherer Priorität wird aktiv und schmeisst den Thread mit tieferer Priorität raus, er wird pre-empted durch den Thread mit höherer Priorität.

Ein Thread kann aus vielen unterschiedlichen Gründen nicht mehr ausführbereit sein. Er kann beispielsweise schlafen:

```
Thread.sleep()
```

Er kann aber auch auf externe Geräte, Festplatten, Modem, ... oder Benutzereingaben warten.

Alle Threads, welche lauffähig sind aber nicht laufen, werden in Warteschlangen verwaltet, je eine Warteschlange pro Priorität (zusätzlich existieren weitere Warteschlangen für bestimmte Threadzustände). Wenn ein Thread seine Ausführung unterbricht, wird er am Ende der 'lauffähig' Warteschlange eingetragen.
Ein Thread, der lauffähig wird, nachdem er blockiert war (schlafend oder wartend auf I/O), wird ebenfalls am Ende der Warteschlange eingetragen.

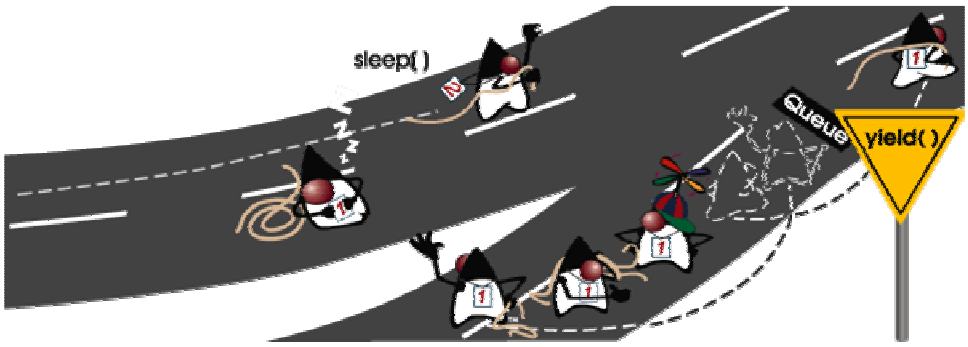
Da die Steuerung der Threads nicht notwendigerweise mit Zeitscheibensteuerung geschieht (time slicing), liegt es an Ihnen, dafür zu sorgen, dass Ihre Thread anderen Threads eine Chance gibt, aktiv zu werden. Das folgende Programmfragment zeigt, wie dies erreicht werden kann. Ein einfacher Weg ist das Ausführen der `sleep()` Methode, immer dann, wenn Ihr Thread mal Pause machen könnte. Da der Thread 'im Schlaf erschlagen' werden könnte, also eine Ausnahme eintreten könnte, muss diese Methode in einem `try ... catch` Block drin stehen. `sleep()` ist eine statische Methode der `Thread` Klasse. Sie können also direkt `Thread.sleep()` aufrufen, ohne eine Instanz der Klasse `Thread` bilden zu müssen. Das Argument der Methode gibt an, wieviele Millisekunden gewartet, geschlafen werden soll. Die Zeit müssen Sie als minimale Schlafzeit auffassen, da der Thread unter Umständen nicht gleich aktiv werden kann (vielleicht ist ein anderer Thread mit höherer Priorität noch eine Weile aktiv).

Und hier das erwähnte Programmfragment:

```
1    public class xyz implements Runnable {
2       public void run() {
3           while (true) {
4               // hier passierts
5               ...
6               // und hier kriegt ein anderer Thread eine Chance
7               try {
8                   Thread.sleep(10);
9               } catch (InterruptedException e) {
10                  // der Thread wurde sozusagen im Schlaf umgebracht
11              }
12          }
13      }
14  }
```

Eine andere, aus meiner Sicht bessere Methode, die Systemressourcen für andere Threads freizugeben, liefert die Methode `yield()` der Klasse `Thread`. Die Methode legt den Thread nicht einfach so auf plumpe Art und Weise schlafend. Die Methode funktioniert folgendermassen:

- falls andere Threads mit der selben Priorität lauffähig sind, wird der aktuelle Thread ganz hinten an die Warteschlange gestellt.
- falls kein anderer Thread mit derselben Priorität läuffähig ist, geschieht nichts.



Der Unterschied zwischen sleep() und yield() ist der, dass mit sleep() auch Thread mit einer tieferen Priorität eine Chance haben aktiv zu werden.

1.5.3. Lektion 2 - Ablaufsteuerung von Threads

1.5.3.1. Einen Thread beenden

Falls ein Thread seine `run()` Methode ausgeführt hat, dann ist seine Arbeit getan, er endet. Er kann dann aber auch *nicht wieder gestartet* werden.

Sie können einen Thread erzwungenermassen Stoppen, mit der `stop()` Methode (obschon Sie dies besser anders programmieren: diese Methode wurde berechtigterweise verworfen).

Trotzdem - hier ist ein Beispiel für ein Thread Steuerungsprogramm unter Einsatz der `stop()` Methode:

```
1    public class xyz implements Runnable {
2        // hier steht so allerlei : alles was der Thread tun soll
3    }


4    public class tTest {
5        public static void main(String args[]) {
6            Runnable r = new xyz();
7            Thread t = new Thread(r);
8            t.start();
9            // da passiert was
10           if (time_to_kill)
11               t.stop();
12       }
13   }
```

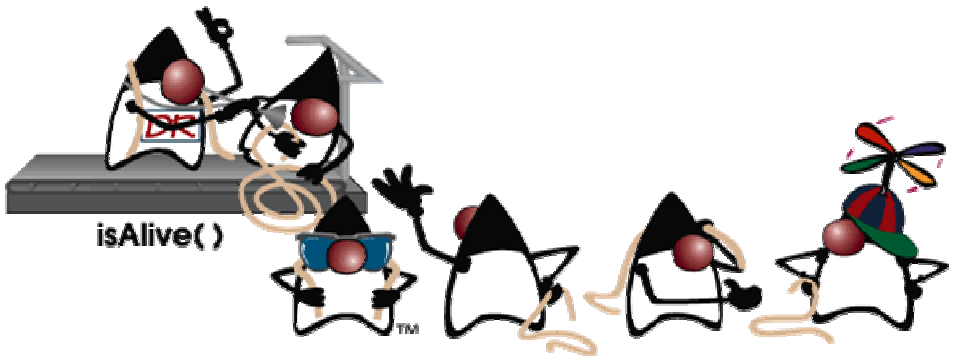
Im obigen Beispiel stoppen Sie einen bestimmten Thread aus dem Hauptprogramm heraus. Natürlich können Sie auch den Thread, der gerade ausgeführt wird stoppen: der Thread stoppt sich selbst, er zerstört damit auch seinen Prozesskontext, es kann also keine weitere Anweisung aus der `run()` Methode ausgeführt werden. Um die Identität von sich selbst festzustellen, muss der Thread seine Identität mit `Thread.currentThread()` feststellen. Anschliessend kann er sich selbst stoppen: `Thread.currentThread().stop()`.

```
1    public class xyz implements Runnable {
2        public void run() {
3            while (true) {
4                // hier passierts
5                if (time_to_die)
6                    Thread.currentThread().stop();
7            }
8        }
9    }
```

1.5.3.2. Testen eines Threads

Unter Umständen kann es passieren, dass nicht klar ist, in welchem Zustand sich ein Thread befindet, beispielsweise, wenn mehrere Threads sich gegenseitig beeinflussen. Dann kann es hilfreich sein, festzustellen, on der Thread überhaupt noch am Leben ist. Dafür stellt die `Thread` Klasse die Methode `isAlive()` zur Verfügung.

Falls die Methode den Wert `true` liefert, heisst das nicht, dass der Thread am Laufen ist. Es besagt nur, dass der Thread einmal gestartet wurde und weder gestoppt wurde, noch beendet ist.



1.5.3.3. Thread on Hold

Die Threadklasse definiert mehrere Methoden, die einen Thread temporär unterbrechen können. Die meisten dieser Methoden sind eher gefährlich und Sie sollten sich genaustens überlegen, ob Sie diese Methoden wirklich einsetzen wollen.

Eine Methode gestattet es Ihnen den Thread zu unterbrechen, und eine weitere Methode erlaubt es Ihnen anschliessend den Thread wieder weiter laufen zu lassen. Es sieht von aussen so aus, als ob der Thread ein Anweisung einfach sehr langsam ausgeführt habe:

- `sleep()`
- `suspend()` und `resume()`

```
1    public class xyz implements Runnable {
2        public void run() {
3            // hier passierts
4
5            // warte, bis jemand Bescheid sagt
6            Thread.currentThread().suspend ();
7            // und dann geht's weiter
8        }
9    }
10       ...
11       Runnable r = new xyz();
12       Thread t = new Thread(r);
13       t.start();
14       // jetzt lassen wir den Thread eine Weile arbeiten
15       // und nehmen an er sei suspended...
16       Thread.sleep(1000);
17
18       // Schonzeit abgelaufen:  xyz muss weiter arbeiten
19       t.resume();
20
21       // für alle Fälle: gib ihm eine Chance
22       Thread.yield();
```

Die `sleep()` Methode wurde schin sehr früh eingeführt und dient der Unterbrechung eines Threads, der Pausierung für eine bestimmte Zeitdauer. Vergessen Sie nicht, dass der Thread in der Regel nach Ablauf der Schlafzeit nicht gleich aktiv werden kann, weil eventuell noch ein Thread mit höherer Priorität aktiv sein kann, ausser:

a) der Thread wacht auf und besitzt eine höhere Priorität
b) der laufende Thread wird aus irgend einem Grund blockiert, beispielsweise, weil er auf Daten warten muss.
c) timeslicing ist aktiviert.

Manchmal ist es wichtig einenThread für unbestimmte Zeit anzuhalten. In diesem Fall benötigen Sie einen weiteren Thread, der den suspendierten Thread wieder aktivieren kann, mit. Die Methoden `suspend()` und `resume()` wurden dafür definiert.

Beachten Sie, dass ein Thread von irgend einem Stück Programmcode suspendiert werden kann, welches eine Referenz auf den Thread hat.
Aber wieder gestartet kann der suspendierte Thread nur von einem anderen Thread, da er selbst keine Zeile Programmcode, *keine* Programmanweisung ausführen kann.

Neben den `sleep()`, `suspend()` und `resume()` Methoden finden Sie in der `Thread` Klasse eine weitere, mit der die Ausführung eines Threads temporär gestoppt werden kann:

- `join()`

```
1   TimerThread tt = new TimerThread (100);
2   tt.start ();
3   ...
4   public void timeout() {
5       // warten, bis der Timer Thread beendet ist
6       tt.join ();
7       ...
8       // jetzt kann dieser Thread weiterfahren
9       ...
10  }
```

Die `join()` Methode bewirkt, dass der aktuelle Thread wartet, bis der Thread auf den die join Methode angewandt wurde, beendet ist

Im obigen Beispiel, in Zeile 6, wird festgehalten, dass der aktuelle Thread warten soll, bis der Timer Thread `tt` beendet ist.

`join(long millis)` kann auch mit einem Timeout Wert in Millisekunden angegeben werden. Mit dieser `join()` Methode wird der Thread entweder für "timeout" Millisekunden oder bis der Thread `tt` beendet wurde, was auch immer zuerst eintrifft.

1.5.3.4. Alternativen zum Kreieren von Threads

Bisher haben wir Threads mittels der Schnittstelle `Runnable` implementiert. Allerdings haben wir bereits erwähnt, dass ein Thread auch mit Hilfe einer Klasse implementiert werden kann,welche die Klasse `Thread` erweitert, statt zu `Runnable` implementieren.

Das folgende Beispiel zeigt, wie eine solche Implementation aussehen könnte:

```
package threadskreieren;

public class EinEinfacherThread extends Thread {

   public void run() {
      int i=100, j=0;
      while(true) {
         System.out.println ("Thread ist running "+j++);
         try{

            System.out.println ("Thread schläft gleich "+i+" Millisekunden");
            sleep(i);
         } catch (InterruptedException e) {
            System.out.println("InterruptedException");
         }
      }
   }

   public static void main (String args[]) {
      Thread t = new EinEinfacherThread();
      t.start();
   }
}
```

Das gleiche Programm lässt sich auch mit Runnable implementieren, mit einer leichten Änderung (da die sleep() Methode im Runnable Interface nicht definiert wird:

```
package threadrunnable;

public class ThreadAlsRunnable implements Runnable {

   public void run() {
      int i=100, j=0;
      while(true) {
         System.out.println ("Thread ist running "+j++);
         try{

            System.out.println ("Thread schläft gleich "+i+" Millisekunden");
            Thread.sleep(i);
         } catch (InterruptedException e) {
            System.out.println("InterruptedException");
         }
      }
   }

}

package threadrunnable;

public class ThreadRunnableStarter {

  public static void main(String[] args) {
    Runnable r = new ThreadAlsRunnable();
    Thread t = new Thread(r);
    t.start();
  }
}
```

1.5.3.5. Runnable versus extends Thread

Nachdem Sie zwei unterschiedliche Möglichkeiten haben, Threads zu implementieren, stellt sich die Frage, wann die eine oder die andere Technik besser geeignet ist.

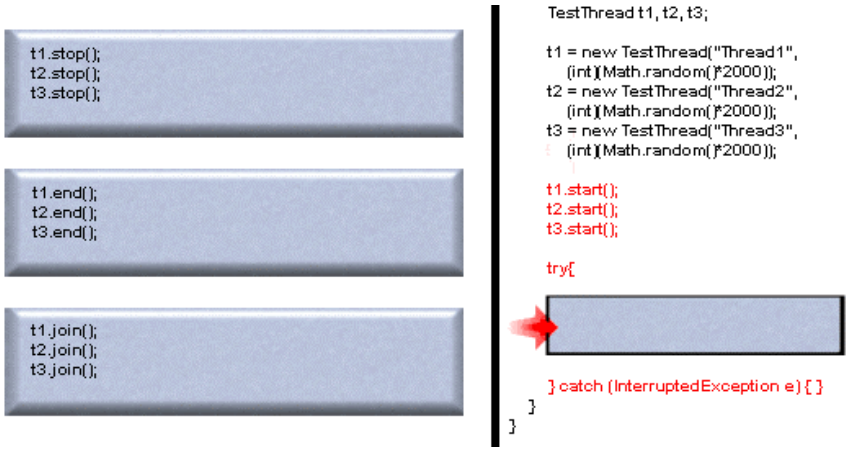
Hier einige Punkte, die bei der Entscheidung für die eine oder die andere Lösung sprechen:

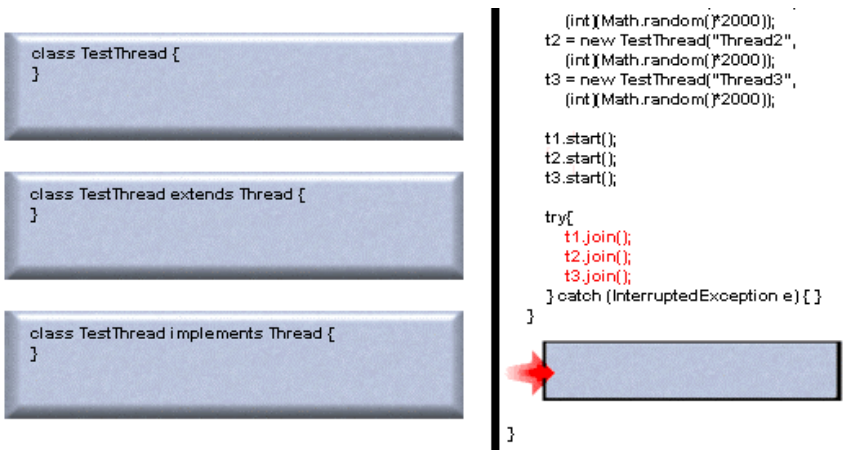
- *spricht für* `implements Runnable`*:*
  - aus objektorientierter Sicht ist die `Thread` Klasse eine Kapselung der virtuellen CPU. Als solche sollte sie nur dann erweitert werden, falls das Verhalten dieser virtuellen CPU erweitert werden muss.
    In der Regel wird dies aber nicht unser Ziel sein.
    Falls Sie die Dreiteilung (virtuelle CPU, Code, Daten) als sinnvoll erachten, dann werden Sie auch eher die Schnittstelle implementieren.
  - Java erlaubt keine Merfachvererbung. Wenn Sie beispielsweise auch noch die Klasse `Applet` erweitern müssen, können Sie nicht die Klasse `Thread` als Basis für Ihre Thread Beschreibung verwenden. Auch in diesem Fall werden Sie `Runnable` implementieren.
  - Vielleicht sind Sie auch einfach nur stur: da Sie sowieso in einzelnen Fällen zwingend `Runnable` implementieren müssen, wenden Sie diese Technik durchgehend an?
- *spricht für* `extends Thread`*:*
  - Die Beschreibung der Threads wird einfacher, weil Sie nicht dauernd auf den Thread referenzieren müssen: `Thread.currentThread().suspend();` wird einfach zu `suspend();`
    Weil der resultierende Programmcode einfacher wird, macht es Sinn, die `Thread` Klasse direkt einzusetzen und passend zu erweitern.
  - Da Java keine Merfachvererbung kennt, müssen Sie in einzelnen Fällen auf diese Technik der Implementation Ihrer Threads verzichten!

1.5.4. Praktische Übung

In dieser Übung geht es darum, einfache Ergänzungen zu einem Programmvorschlag zu finden. Die Lösung der Aufgabe sehen Sie jeweils gleich im nächsten Schritt.

Insgesamt soll die Lösung drei Threads umfassen, welche zufällig und unabhängig voneinander laufen sollen. Alle geben die Zeit und die Thread ID aus und enden nach einer zufälligen Zeitdauer.






```
    try{
      t1.join();
      t2.join();
      t3.join();
    } catch (InterruptedException e) { }
  }

  class TestThread extends Thread {
    private String whoami;
    private int delay;

    public TestThread (string s, int d) {
      whoami = s;
      delay = d;
    }

    [                                    ]

  }
}
```


```
class TestThread extends Thread {
  private String whoami;
  private int delay;

  public TestThread (string s, int d) {
    whoami = s;
    delay = d;
  }

  public void run() {
    try {

    [                                    ]

      } catch (InterruptedException e) {}
      System.out.println("Hello World! " + whoami + " "
        new Date());
    }
  }
}
```

```
      t3.join();
    } catch (InterruptedException e) { }
  }

  class TestThread extends Thread {
    private String whoami;
    private int delay;

    public TestThread (string s, int d) {
      whoami = s;
      delay = d;
    }

    public void run() {
      try {
        sleep(delay);
      } catch (InterruptedException e) {}
      System.out.println("Hello World! " + whoami + " "
        new Date());
    }
  }
}
```

Das war ja ganz nett. Nun wollen Sie sicher den vollständigen Programmcode sehen und auch die Ausgabe. Die Lösung befindet sich auf dem Server / Web / CD.

```
package threadsloesungderuebung;

import java.util.*;

public class DreiThreads {

    public static void main(String args[]) {
        TestThread t1, t2, t3;
        t1 = new TestThread("Thread1",(int)(Math.random()*2000));
        t2 = new TestThread("Thread2",(int)(Math.random()*2000));
        t3 = new TestThread("Thread3",(int)(Math.random()*2000));

        t1.start();
        t2.start();
        t3.start();

        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) { }
    }
}


class TestThread extends Thread {
    private String whoami;
    private int delay;

    public TestThread(String s, int d) {
      whoami = s;
      delay = d;
    }

    public void run() {
        try {
          sleep(delay);
      } catch (InterruptedException e) { }
      System.out.println("Hello World! " + whoami + " " + new Date());
    }
}
```

Hello World! Thread1 Fri Mar 02 13:30:59 GMT+01:00 2001
Hello World! Thread2 Fri Mar 02 13:31:00 GMT+01:00 2001
Hello World! Thread3 Fri Mar 02 13:31:00 GMT+01:00 2001

1.5.5. Lektion 3 : Synchronisation in Java

1.5.5.1. Das Synchronisations-Problem

Jetzt besprechen wir im Wesentlichen das `synchronized` Schlüsselwort. Damit wird in Java der grundlegende Synchronisationsmechanismus, die Kontrolle der Zugriffe mehrerer Threads auf gemeinsame Daten, geregelt.

Schauen wir uns der Einfachheit halber einen Stack an. Der erste Entwurf unten scheint brauchbar zu sein. Die Klasse kümmert sich weder um Stack Overflow noch Underflow. Auch die Kapazität des Stacks ist sehr begrenzt. Aber diese Aspekte sind für uns im Moment völlig unwichtig!

In dieser Klasse zeigt die Variable `idx` auf die nächste *leere* Zelle im Stack. Dabei verwenden wir den "pre-decrement, post-increment" Approach.

Was passiert, wenn zwei Threads auf diesen Stack gleichzeitig zugreifen? Beide Threads besitzen eine Referenz auf eine und dieselbe Instanz der Klasse. Ein Thread schreibt Daten auf den Stack; der andere mehr oder weniger unabhängige Thread liest Daten vom Stack und löscht sie. Das Ganze sieht also recht gut organisiert aus. Aber es gibt ein Problem!

```
1    class stack {
2        int idx = 0;
3        char [] data = new char[6];
4
5        public void push(char c) {
6            data[idx] = c;
7            idx++;
8        }
9
10       public char pop() {
11           idx--;
12           return data[idx];
13       }
14   }
```

Stellen Sie sich vor, Thread `a` schreibe ein Zeichen und Thread `b` lese ein Zeichen und lösche dieses Zeichen. Thread `a` hat gerade sein Zeichen auf den Stack gelegt aber den Index Zähler noch nicht hochgezählt. Dieser Thread wird nun pre-empted, aus was für Gründen auch immer, das heisst, der Thread wird unterbrochen und ein anderer Thread wird aktiv. Unser Datenmodell ist zu diesem Zeitpunkt inkonsistent!

```
buffer |p|q|r| | | |
idx = 2     ^
```

Im Speziellen sollte entweder `idx=3` sein oder das Zeichen sollte noch nicht auf dem Stack sein.

Wenn nun Thread `a` seine Ausführung weiterführt, kann es sein, dass in der Zwischenzeit nichts passiert ist und die Daten konsistent werden.

Aber genausogut könnte es sein, dass in der Zwischenzeit Thread `b` darauf wartete ein Zeichen vom Stack abzuholen und in der Wartezeit von Thread a dies auch tun konnte! In diesem Fall wird die `pop()` Methode auf eine inkonsistente Daten Situation angewandt. Die `pop()` Methode reduziert den `idx` Wert um Eins:

```
buffer |p|q|r| | | |
idx = 1   ^
```

Damit wird das Zeichen `r` im Stack ignoriert. Das macht aber scheinbar noch nichts. Ausser Thread a wird jetzt aktiv und beendet seine `push()` Methode. Im Speziellen wird also der Index um Eins erhöht.

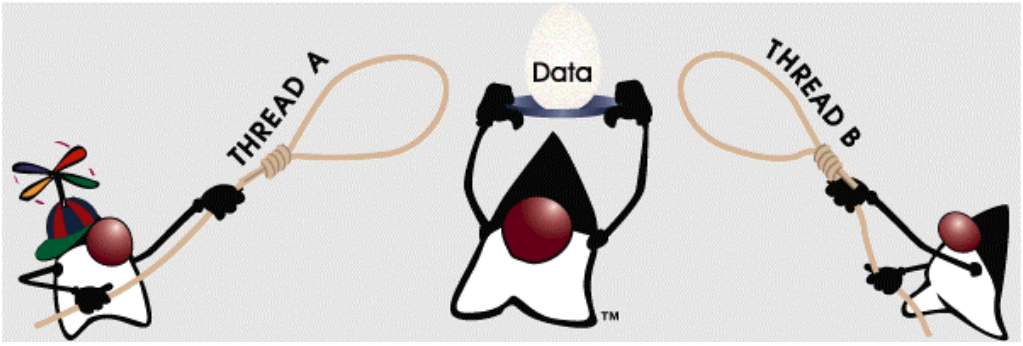
```
buffer |p|q|r| | | |
idx = 2     ^
```

Gemäss dieser Situation ist q ein gültiger Eintrag und die nächste leere Zelle enthält r. Sie werden also r nie lesen können. q dagegen unter Umständen zweimal.

Dieses einfache Beispiel zeigt auf ein generelles Problem, welches in Systemen mit mehreren Threads und gemeinsamen Daten entstehen kann. Wir benötigen einen Mechanismus, mit dem wir unsere Daten vor unerlaubten Zugriffen schützen können.

Ein solcher Mechanismus könnte darin bestehen, dass jeder Thread sobald er einen kritischen Bereich (*critical section*) betritt, nicht unterbrochen werden darf. Diese Atomizität wird bei Betriebsystemen und Datenbanken eingesetzt. Mehrbenutzersysteme bauen in der Regel darauf auf und verwenden übergeordnete Techniken.

Java stellt einen Mechanismus zur Verfügung, mit dem man den Zugriff auf gemeinsame Daten koordinieren kann.

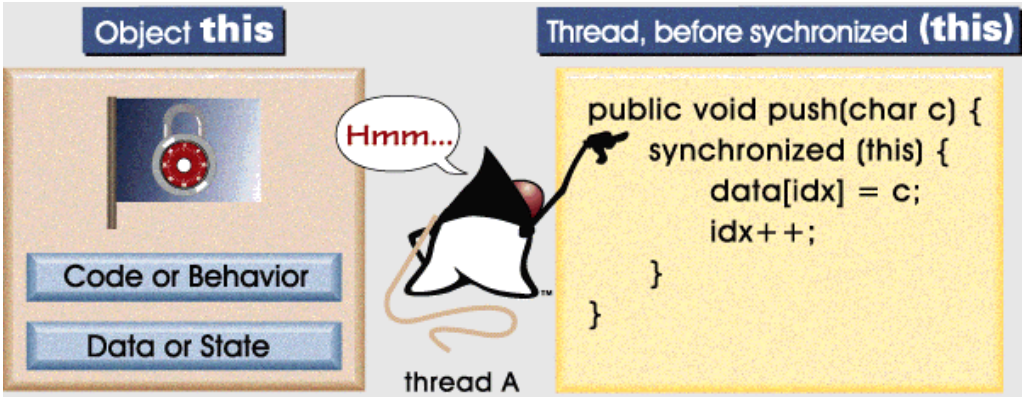


Das Konzept setzt auch mehr oder weniger voraus, dass der Zugriff auf die Daten mit Hilfe von Methode, also nicht direkt erfolgt: die Zugriffsmethoden werden einfach synchronisiert. Falls die Methoden komplex sind, sollte man nicht die gesamte Methode synchronisieren, sondern nur den kritischen Bereich, als synchronisierten Block.

1.5.5.2. Das Objekt Lock Flag

In Java besitzt jede Instanz, jedes Objekt also, ein Flag, welches wir als "lock flag" bezeichnen können. Mit Hilfe des `synchronized` Schlüsselworts kann man mit diesem Lock Flag arbeiten.

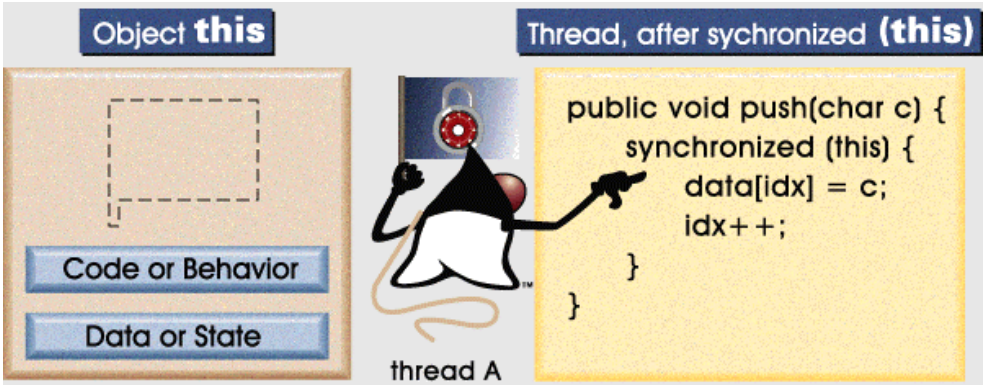
Anschaulich stellen wir uns das etwa folgendermassen vor:



Sobald der Thread in der Ausführung beim Schlüsselwort `synchronized` anlangt, prüft der Thread das Objekt, welches als Argument von `synchronized` angegeben wurde und versucht das Lock Flag dieses Objekts zu besetzten. Sie können sich das Lock Flag auch als Token vorstellen: bei `synchronized` erhält ein und nur en Thread diesen Token.

Mit diesem Flag bzw. dem synchronized Schlüsselwort für die `push()` Methode haben wir unser Problem mit dem Stack Object (`this`) nicht gelöst:

falls jemand die unveränderte Methode `pop()` einsetzt, *riskieren wir immer noch die Konsistenz der Daten des Objekts zu vernichten.*

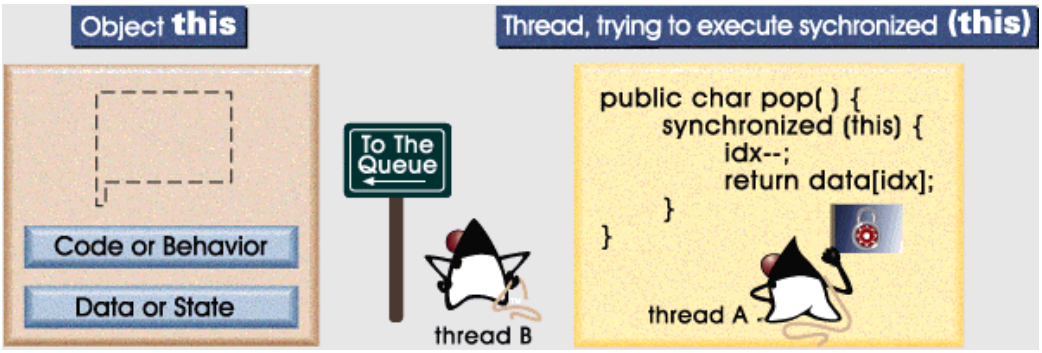


Was wir zu tun haben ist, auch die `pop()` Methode entsprechend zu modifizieren. Dazu ergänzen wir die Methode durch ein `synchronized(this)` um den kritischen Bereich der `pop()` Methode, genau wie bei der `push()` Methode.

Was passiert, falls ein Thread versucht eine synchronisierte Methode auszuführen, diese aber gesperrt ist?

Wenn der Thread versucht die Anweisung `synchronized(this)` auszuführen, versucht es den Token zu erhalten, das Lock Flag des Objekts `this`. Da das Flag nicht vorliegt /der Token sich woanders befindet, kann der Thread seine Ausführung nicht fortsetzen.

Der Thread muss nun warten: er wird in eine Warteschlange mit weiteren Threads eingetragen. Diese Warteschlange ist dem Objekt Flag so zugeordnet, dass dann, wenn das Flag an das Objekt zurück gegeben wird, jener Thread, der in der Warteschlange am vordersten steht, das Flag erhält und seine Ausführung fortsetzen kann.



1.5.5.3. Freigabe des Lock Flags

Da ein Thread, welcher auf ein Flag wartet, nicht fortfahren kann, bis er das Flag erhält, ist es wichtig, dass das Flag so früh wie möglich nachdem es nicht mehr benötigt wird, zurück gegeben wird.

Die Rückgabe des Flags an das Objekt geschieht automatisch: sobald der synchronisierte Blaock verlassen wird, erhält das Objekt das Flag zurück. Java kümmert sich darum, dass das Flag auch immer korrekt zurück gegeben wird! Wenn also beispeilsweise im synchronisierten Block eine Ausnahme geworfen würde oder aus dem synchronisierten Block herausgesprungen würde (mit `break` oder `continue`) sorgt Java dafür, dass das Flag korrekt zurückgegeben wird.

Falls ein Thread versucht ein Objekt zweimal zu synchronisieren, verschachtelt:

```
synchronized(obj) {
      synchronized(obj) {

      }

}
```

dann wird das Flag erst beim Verlassen des äusseren Blocks zurückgegeben und die Synchronisation des inneren Blocks effektiv ignoriert.

Dieses Synchronisationskonzept vereinfacht die Behandlung critical sections im Vergleich zu anderen Konzepten, wie etwa *binären Semaphoren*. Eine Semaphore ist eine ganzzahlige Variable, mit deren Hilfe der Zugriff auf Daten kontrolliert wird. Bevor ein Prozess die geschützten Daten einsetzt, muss er den Wert der Semaphore abfragen. Falls der Wert grösser als 0 ist, vermindert der Prozess atomar den Wert um eins, greift auf die Daten zu und erhöht anschliessend den Wert atomar wieder um eins. Falls die Semaphore binär ist, kennt sie nur die Werte 0 und 1, wie eine Ampel (grün, rot).

1.5.5.4. Synchronisation insgesamt

Der Synchronisierungsmechanismus funktioniert nur, falls der Programmierer die Anweisungen an der korrekten Stelle plaziert. Jetzt wollen wir uns der Frage widmen: wie stellt man denn eine korrekt synchronisierte Klasse zusammen?

Dazu müssen wir uns überlegen, auf welche Daten man überhaupt zugreifen darf. Falls Daten als privat markiert sind, brauchen wir uns keine weiteren Gedanken zu machen: darauf können wir sowieso von ausserhalb der Klasse nicht zugreifen! Damit schützen wir unsere Daten sehr effizient. Also: Sie sollten so viele Daten wie möglich als privat deklarieren.

Betrachten wir nun Methoden, die auf Daten, auch private, zugreifen. Dann haben wir gesehen, dass wir den Zugriff schützen können, indem wir die Zugriffe auf das Objekt synchronisieren:

```
public void push(char c) {
        synchronized(this) {
        ...
        }
    }
```

In der Regel werden wir als Argument `this` verwenden (wir könnten auch irgend ein Objekt verwenden, welches sonst keine Funktion hat) führten die Entwickler der Sprache Java die folgende kürzere Notation ein:

```
public synchronized void push(char c) {
        ...
    }
```

Die beiden Schreibweisen sind in diesem einfachen Beispiel identisch. Im Allgemeinen besteht aber ein ein Unterschied:

- falls Sie eine Methode synchronisieren, kann es passieren, dass das Lock Flag vom Thread länger blockiert wird als nötig.
- falls Sie einen Block synchronisieren, wird das Lock Flag nur während der Ausführung dieses Blocks vom Thread benötigt.

Eine Finesse, die in komplexen Projekten aber wichtig sein kann ist, dass ein Modifier in die `javadoc` Dokumentation aufgenommen wird; ein `synchronized(this)` jedoch nicht.

1.5.5.5. Deadlock

Falls Sie mit Programmen arbeiten, in denen mehrere Threads gleichzeitig aktiv sein können, besteht die Gefahr, dass Verklemmungen, *Deadlocks*, auftreten. Diese treten immer auf, falls ein Thread auf den Lock Token wartet, den gerade ein anderer Thread besitzt; aber dieser Thread wartet auf den Lock Token für ein Lock, welches gerade vom ersten Thread besetzt wird. Somit kann keiner der Threads weiterfahren: beide warten jweils auf die Freigabe.

Java behebt dieses Problem nicht; Sie müssen darauf achten, dass diese Situation nicht eintrifft! Falls Sie Systeme mit mehreren synchronisierten und verschachtelten Bereichen haben, sollten Sie global die Reihenfolge der Ausführung selbst festlegen.

1.5.6. Lektion 4 - Thread Interaktion (wait/notify)

1.5.6.1. Koordination von Threads

Threads werden sehr oft Tätigkeiten ausüben, die nicht unabhängig voneinander sind. Daher benötigen wir neben der Synchronisation der Zugriffe auf gemeinsame Daten, weitere Konzepte. In der Theorie und der Praxis existieren viele verschiedene Konzepte für diese Koodination der Threads. Die einzelnen Verfahren können in der Regel ineinander transformiert werden, oder die andern Konstrukte umschrieben werden.

In diesem Abschnitt prüfen wir den Mechanismus, den Java offeriert und wie man ihn sinnvollerweise einsetzt.

Warum benötigen wir einen solchen Mechanismus?
Betrachten wir ein einfaches Beispiel:
zwei Personen arbeiten zusammen in der Küche. Die eine Person spült, die andere trocknet ab. Die Personen repräsentieren unsere Threads. Beide haben gemeinsame Daten, das Abtropfbrett. Beide möchten eigentlich lieber mit ihren Bekannten im Internet chatten, haben also keine Lust, ihre Arbeit zu tun. Der Abtrockner kann nicht arbeiten, falls keine Gegenstände auf dem Abtropfbrett sind. Falls das Brett voll ist, kann der Abwascher nicht weiterfahren.

Also benötigen wir einen Mechanismus, der uns hilft, diese Wechselwirkung der zwei Threads sinnvoll zu beschreiben.

1.5.6.2. Lösung des Koordinationsproblems

Falls wir dumm wären, würden wir einfach mit `suspend()` und `resume()` Methoden arbeiten, dumm, weil die Methoden verworfen wurden und das wohl aus gutem Grunde.
Zudem müssten wir in diesem Fall dafür sorgen, dass jeder Thread eine Referenz auf die anderen Threads besitzt, also ein eher kompliziertes Unterfangen.

Java bietet einen Kooperationsmechanismus, der auf Warteschlangen basiert:
- jedes Objekt besitzt zwei Thread- Warteschlangen.
  - Die erste Warteschlange wird für die auf das Lock Flag wartenden Threads benutzt.
  - Die zweite Warteschlange wird für die Kooperationsmethoden `wait()` und `notify()` verwendet.

Auf der obersten Objektebene, `java.lang.Object`, werden drei Methoden `wait()`, `notify()` und `notifyAll()` definiert.

Betrachten wir zuerst die zwei Methoden `wait()` und `notify()` am Geschirrwäscher-Beispiel:

Thread `a` ist der Geschirrwäscher, Thread `b` ist der Geschirrtrockner.
Beide haben Zugriff auf das gemeinsame Abtropfbrett, das `drainingBoard` Objekt.
Nehmen wir an, Thread `b` (der trocknende Thread) möchte einen Gegenstand abtrocknen; das Abtropfbrett ist aber leer:

```
if (drainingBoard.isEmpty())
    drainingBoard.wait();
```

Der Thread `b` führt die `wait()` Anweisung aus, führt aberanschliessen keine weiteren Anweisungen aus, ist also nicht mehr laufend, runnable. Der Thread `b` wird in die wait-Warteschlangeit des `drainingBoard` Objekts eingetragen.

Der Thread kann nicht weiterfahren, bis er aus der Warteschlange entfernt wird.

Wie könnte der trocknende Thread wieder aktiviert werden?
Es liegt in der Verantwortung des spühlenden Threads `a` den wartenden Thread zu informieren, sobald dieser etwas sinnvolles tun kann!

Dies wird durch den Aufruf der Methode `notify()` an das Abtropfbrett erreicht (dort ist ja der Thread `b` am Warten):

```
drainingBoard.addItem(plate);
drainingBoard.notify();
```

Damit wird dem ersten Thread in der wait-Warteschlange signalisiert, dass er wieder aktiv werden könnte, sofern er CPU Zeit erhält.

Die `notify()` Methode wird unabhängig davon, ob Thread warten oder nicht, an das Objekt gesandt. Man konnte nun argumentieren, dass hier Optimierungspotential drin liegt. Darüber wollen wir in diesem Zusammenhang aber weniger Nachdenken! Der Methodenaufruf bewirkt einfach nichts, falls die Warteschlange leer ist.

Die Methode `notify()` hat eine Schwäche: wenn der vorderste Thread aus irgend einem Grund nicht aktiv werden kann, könnte im schlimmsten Fall ein deadlock resultieren. Diese

Situation könnte zum Beispiel auftreten, wenn nach der Freigabe der Thread eine atomare Aktion ausführen möchte, aber vom Scheduler nicht genug Zeit erhält, diese auszuführen und daher nichts tun kann.

Es könnte aber sehr wohl sein, dass ein weiterer Thread in der Warteschlange aktiv werden könnte. Die Methode `notifyAll()` kann an Stelle der einfacheren Methode notify() eingesetzt werden. Damit werden alle wartenden Threads informiert.

Nun können wir das Geschirrwäscherproblem in Java formulieren:
- immer wenn wir eine Operation ausgeführt haben, welche garantiert, dass ein anderer Thread sinnvolle Arbeiten erledigen kann, führen wir die `notify()` Methode des Abtropfbretts aus.
- immer wenn wir versuchen Arbeiten auszuführen, aber nicht fortfahren können, führen wir die `wait()` Methode des Abtropfbretts aus.

1.5.6.3. Die Wahrheit

Der eben beschriebene Mechanismus beschreibt nur das grundlegende Konzept. Aber die Java Implementierung ist etwas komplexer:

- die Verwaltung der Warteschlangen muss *atomar* geschehen, da sonst Probleme beim Eintragen oder Löschen auftreten könnten.

- Dies kann man erreichen, indem man vor der Ausführung des `wait()`, `notify()` oder `notifyAll()` auf einem Objekt das Lock Flag dieses Objekts erhält. Daher müssen diese Methoden in `synchronized` Blöcken ausgeführt werden.

```
1    synchronized(drainingBoard) {
2        if (drainingBoard.isEmpty())
3            drainingBoard.wait();
4    }
```

und entsprechend:

```
5    synchronized(drainingBoard)  {
6        drainingBoard.addItem(plate);
7        drainingBoard.notify();
8    }
```

Das Konstrukt sieht nach einer ungeschickten Verschachtelung aus:
- der erste Thread führt beispielsweise die Anweisung 1 aus, erhält das Lock Flag und kontrolliert damit den Zugang zum Objekt `drainingBoard`
- der zweite Thread möchte die Anweisungen 5-8 ausführen, benötigt dazu aber das Lock Flag des `drainingBoard` Objekts. Das Lock Flag ist aber beim ersten Thread?

Dieses Problem wird in Java automatisch gelöst:
- sobald die `wait()` Methode aufgerufen wird, wird automatisch das Lock Flag an das Objekt zurück gegeben
- sobald die `notify()` Methode ausgeführt wird, wird ein Thread automatisch aus der wait in die Lock Flag Warteschlange verschoben.
- falls `interrupt()` ausgeführt wird, wirft `wait()` eine Exception (daher `try... catch`)

1.5.7. Lektion 5 - Producer / Consumer Beispiel

1.5.7.1. Beschreibung des Problems

Nun machen wir uns an die echten Informatikthemen heran... An Stelle von Tellern und Tellerwäschern beschäftigen wir uns mit Zeichen und einem Stack Objekt. Aber das Problem ist im Prinzip das selbe. Wir könnten auch eine Bierbrauerei und Studenten als Kunden beschreiben (mein Standard Beispiel), oder einen Supermarkt mit Kassen, Einkaufswagen und vielen Theken (Käse, Fleich [zur Zeit immer leer, wegen BSE, MKS, und vielem mehr]) Eingängen und Ausgängen.

Das Stack Problem ist ein klssisches Informatik Problem zur Beschreibung der Consumer - Producer Situation (Konsumenten / Produzenten Problem).

Als erstes wenden wir uns dem Stack und dessen Beschreibung zu. Dann betrachten wir die Details der Konsumenten- und Produzenten- Threads. Schliesslich werden wir uns dann noch die Details des Stacks anschauen: Zugriffsschutzmechanismen und Thread Kommunikation.

Wir nennen unsere Klasse `SyncStack`, um sie von der Standardklasse `java.util.Stack` unterscheiden zu können. Unser Stack API sieht folgendermassen aus:

```
public void push(char c);
public char pop();
```

1.5.7.2. Der Produzent

Der Produzent besteht aus den Methoden, die Sie im ersten Entwurf unten sehen. Er wird 20 zufällige Grossbuchstaben generieren und diese auf den Stack lagen, mit zufälliger Verzögerung zwischen den einzelnen Operationen. Die Verzögerungen liegen im Bereich 0..100 Millisekunden. Das Schreiben eines Zeichens auf den Stack wird auf der Konsole angezeigt (auf ein GUI verzichten wir).

```
public class Producer implements Runnable {

   SyncStack theStack;

   Producer (SyncStack stack) { // Konstruktor
      theStack = stack;
   }

   public void run() {
      char c;
      for (int i = 0; i < 20; i++) {
         c = (char)(Math.random() * 26 + 'A');
         theStack.push(c);
         System.out.println("Produziert: " + c);
         try {
            Thread.sleep((int)(Math.random() * 100));
         } catch (InterruptedException e) {
                // was auch immer
         }
      }
   }
}
```

1.5.7.3. Der Konsument

Der Konsument wird in etwa die Struktur haben, wie der Entwurf unten. Der Konsument wird 20 Buchstaben aus dem Stack lesen, mit zufälligen Verzögerungen zwischen den einzelnen Lesebefehlen (`pop()`). Die Verzögerung liegt im Bereich 0...2 Sekunden.

Damit wird der Stack langsamer geleert als gefüllt und somit vermutlich einigermassen schnell voll.

```
public class Consumer implements Runnable {

   SyncStack theStack;

   Consumer (SyncStack stack) { // Konstruktor
      theStack = stack;
   }

   public void run() {
      char c;
      for (int i = 0; i < 20; i++) {
         c = theStack.pop();
         System.out.println("Konsumiert: " + c);
         try {
            Thread.sleep((int)(Math.random() * 1000));
         } catch (InterruptedException e) {
             // was soll's
         }
      }
   }
}
```

1.5.7.4. Der synchronisierte Stack

Für die Beschreibung der Stack Klasse benötigen wir ein Puffer, als Array. Der Puffer braucht nicht dynamisch zu sein, da wir lediglich die Threading- Konzepte erläutern möchten. Der Stack muss auch nicht viele Objekte oder Zeichen aufnehmen können, aus dem selben Grunde.

Wir legen die Anzahl Zeichen im Puffer (Stack) auf 6 fest. Ein neu konstruierter `SyncStack` sollte leer sein. Die Initialisierung führen wir explizit durch, obschin dies auch noch weggelassen werden könnte.

Damit haben wir die Bausteine für unseren synchronisierten Stack!
Den Konstruktor lassen wir weg, eigentlich keine besonders gute Idee...

```
1    class SyncStack {
2        private int index = 0;
3        private char [] buffer = new char[6];
4
5        public synchronized char pop() {
6        }
7
8        public synchronized void push(char c) {
9        }
10   }
```

Als nächstes müssen wir die pop und push Methoden definieren:
- diese müssen `synchronized` sein, um die sensiblen Daten (Zeichen, Puffer) zu schützen;
- zusätzlich müssen wir `wait()` einbauen, für den Fall, dass die Methode nicht weiter ausgeführt werden kann.
- und `notify()` , um mitteilen zu können, dass Arbeit vorliegt

Die Definition der `pop()` Methode sehen Sie weiter unten. Der Aufruf von `wait()`geschieht explizit als `this.wait()`. Der Einsatz von `this` ist redundant, aber hier wird diese Referenz verwendet, um zu unterstreichen, dass das Rendez -Vous *auf diesem Stack Objekt* statt findet.

Da ,wie wir oben erwähnt haben, `wait()` durch einen Aufruf von `interrupt()` abgebrochen werden könnte (und eine Ausnahme geworfen wird), müssen wir die Anweisung in einen `try ... catch` Block fassen.

Nun zu `notify()`. Auch hier verwenden wir zur Verdeutlichung `this.notify()`.
Warum wird `notify()` überhaupt bereits an dieser Stelle aufgerufen?
Der Aufruf der Methode geschieht bereits bevor die Daten verändert wurden. Warum ist dies nicht ein Fehler?
Ein Thread im `wait()` Zustand kann nicht fortfahren, bis er auch aus der Lock Flag Warteschlange draussen ist, wie wir oben gesehen haben. Damit können wir die `notify()` Methode bereits vorzeitig ausführen, sie wird eh erst später aktiv werden können.

```
1    public synchronized void push(char c) {
2        while (index == buffer.length) {
3            try {
4                this.wait();
5            } catch (InterruptedException e) {
6                // vergiss es ..
7            }
8        }
9        this.notify();
10       buffer[index] = c;
11       index++;
12   }
```

Eine letzte Bemerkung zur `push()` Methode:
- wir haben keine Fehlerbehandlung eingebaut, um einen Overflow abzufangen (falls mehr als 6 Elemente auf den Stack gelegtwerden sollten)
- dieser Fehler wird durch das `wait()` Konstrukt verhindert!

```
1    public synchronized char pop() {
2        while (index == 0) {
3            try {
4                this.wait();
5            } catch (InterruptedException e) {
6                  // vergiss es ..
7            }
8        }
9        this.notify();
10       index--;
11       return buffer[index];
12   }
```

1.5.7.5. Das vollständige Stack Beispiel

Nun müssen wir nur noch die einzelnen Bausteine zusammenfügen, wie beim LEGO. Als Driver für die Klassen verwenden wir `SyncTest.java`.Sie finden den Programmcode unten. Auch der Konsument und der Produzent sind als vollständige Klassen wiedergegeben. Einzig import Anweisungen und der Package Name fehlen.
Sie finden das vollständige Beispiel auf dem Server / Web / der CD.

1.5.7.5.1. Das Hauptprogramm SyncTest

```
1    // Test Programm für das Producer-Consumer Problem
2    public class SyncTest {
3
4        public static void main(String args[]) {
5            SyncStack stack = new SyncStack();
6            Runnable source = new Producer(stack);
7            Runnable sink = new Consumer(stack);
8            Thread t1 = new Thread(source);
9            Thread t2 = new Thread(sink);
10           t1.start();
11           t2.start();
12       }
13   }
```

1.5.7.5.2. Der Konsument Consumer

```
1    package ...;
2    public class Consumer implements Runnable {
3
4        SyncStack theStack;
5
6        public Consumer(SyncStack s) {
7            theStack = s;
8        }
9
10       public void run() {
11           char c;
12           for (int i = 0; i < 20; i++) {
13               c = theStack.pop();
14               System.out.println("Konsumiert: " + c);
15               try {
16                   Thread.sleep((int)(Math.random() * 1000));
17               } catch (InterruptedException e) {
18                   // vergiss es ..
19               }
20           }
21       }
22   }
```

1.5.7.5.3. Der Produzent Producer

```
1    package ...;
2    public class Producer implements Runnable {
3
4        SyncStack theStack;
5
6        public Producer(SyncStack s) {
7           theStack = s;
8    }
9
10       public void run() {
11           char c;
12           for (int i = 0; i < 20; i++) {
13               c = (char)(Math.random() * 26 + 'A');
14               theStack.push(c);
15               System.out.println("Produziert: " + c);
16               try {
17                   Thread.sleep((int)(Math.random() * 100));
18               } catch (InterruptedException e) {
19                   // vergiss es ..
20               }
21           }
22       }
22   }
```

1.5.7.5.4. Der synchronisierte Stack SyncStack

```
1    package ...;
2    public class SyncStack {
3
4        private int index = 0;
5        private char [] buffer = new char[6];
6
7        public synchronized char pop() {
8            while (index == 0) {
9                try {
10                   this.wait();
11               } catch (InterruptedException e) {
12                   // vergiss es ..
13               }
14           }
15           this.notify();
16           index--;
17           return buffer[index];
18       }
19
20       public synchronized void push(char c) {
21           while (index == buffer.length) {
22               try {
23                   this.wait();
24               } catch (InterruptedException e) {
25                   // vergiss es ..
26               }
27           }
28           this.notify();
29           buffer[index] = c;
30           index++;
31       }
32   }
```

1.5.7.5.5. Mögliche Ausgaben

Eine detailliertere Ausgabe zeigt etwas genauer was passiert (alle Erweiterungen stehen in den `pop()` und `push()` Methoden vor `wait()` und `notify()`):

```
Info an den Produzent [es geht weiter]
Produziert: N
Info an den Konsumenten [es geht weiter]
Konsumiert: N
Info an den Produzent [es geht weiter]
Produziert: I
Info an den Produzent [es geht weiter]
Produziert: M
Info an den Produzent [es geht weiter]
Produziert: N
Info an den Konsumenten [es geht weiter]
Konsumiert: N
Info an den Produzent [es geht weiter]
Produziert: T
Info an den Konsumenten [es geht weiter]
Konsumiert: T
Info an den Produzent [es geht weiter]
Produziert: G
Info an den Produzent [es geht weiter]
Produziert: M
Info an den Produzent [es geht weiter]
Produziert: P
Info an den Produzent [es geht weiter]
Produziert: T
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Info an den Produzent [es geht weiter]
Produziert: N
Konsumiert: T
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Konsumiert: N
Info an den Produzent [es geht weiter]
Produziert: Y
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Info an den Produzent [es geht weiter]
Produziert: I
Konsumiert: Y
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Konsumiert: I
Info an den Produzent [es geht weiter]
Produziert: B
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Info an den Produzent [es geht weiter]
Produziert: Z
Konsumiert: B
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Konsumiert: Z
Info an den Produzent [es geht weiter]
Produziert: R
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Info an den Produzent [es geht weiter]
Produziert: V
Konsumiert: R
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Konsumiert: V
```

```
Info an den Produzent [es geht weiter]
Produziert: F
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Info an den Produzent [es geht weiter]
Produziert: J
Konsumiert: F
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Konsumiert: J
Info an den Produzent [es geht weiter]
Produziert: U
Der Produzent muss warten [Stack ist voll]
Info an den Konsumenten [es geht weiter]
Info an den Produzent [es geht weiter]
Produziert: Z
Konsumiert: U
Info an den Konsumenten [es geht weiter]
Konsumiert: Z
Info an den Konsumenten [es geht weiter]
Konsumiert: P
Info an den Konsumenten [es geht weiter]
Konsumiert: M
Info an den Konsumenten [es geht weiter]
Konsumiert: G
Info an den Konsumenten [es geht weiter]
Konsumiert: M
Info an den Konsumenten [es geht weiter]
Konsumiert: I
```

Sie können das Modell leicht erweitern oder modifizieren, beispielsweise dadurch, dass Sie dem Konsumenten gleiche Wartezeiten zuordnen. Dannn kann es auch passieren, dass der Konsument warten muss, weil keine Zeichen mehr auf dem Stack vorhanden sind:

```
...
Produziert: V
Info an den Konsumenten [es geht weiter]
Konsumiert: V
Der Konsument muss warten [Stack ist leer]
Info an den Produzent [es geht weiter]
Info an den Konsumenten [es geht weiter]
...
```

1.5.7.6. Das Philosphenproblem - Aushungern oder Starvation

Neben Deadlocks, Verklemmungen, ist ein weiteres grundsätzliches Problem bei nebenläufigen Prozessen die Möglichkeit, dass das System still steht, weil alle beteiligten Threads sich gegenseitig zu fair und höflich sind.

Das Standardbeispiel dafür ist das Philosophenproblem:
mehrere Philosophen sitzen an einem runden Tisch und möchten Spagetti essen. Dazu benötigen Philosophen zwingend zwei Gabeln.

Zwischen je zwei Philosophen beindet sich jeweils eine Gabel. Insgesamt fehlt also eine Gabel.

Wenn alle Philosophen sehr höflich sind, werden sie sich gegenseitig bitten, doch zu essen... mit dem Effekt, dass keiner zum Essen kommt, da ja eine Gabel fehlt. Die Philosophen werden aus Höflichkeit verhungern!

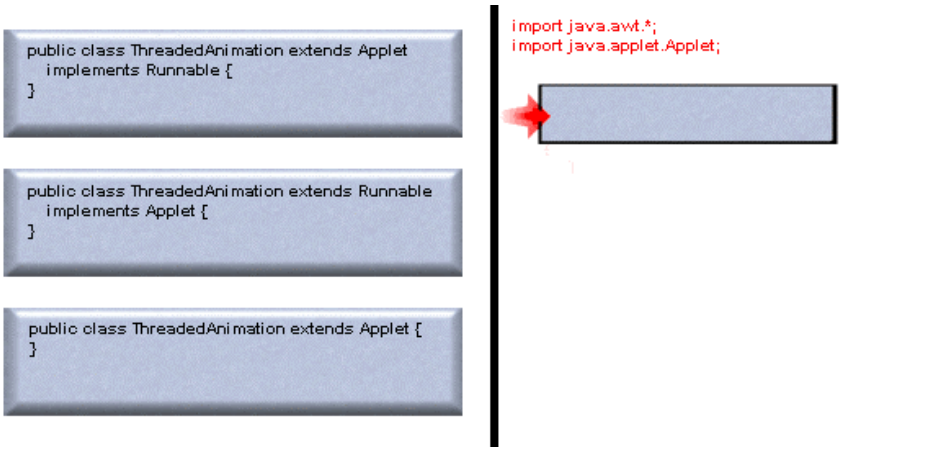
Sie finden viele unterschiedliche Implementationen diese Grundproblems nebenläufiger Systeme im Web, inklusive Schmatzgeräuschen und animierten Bildern.

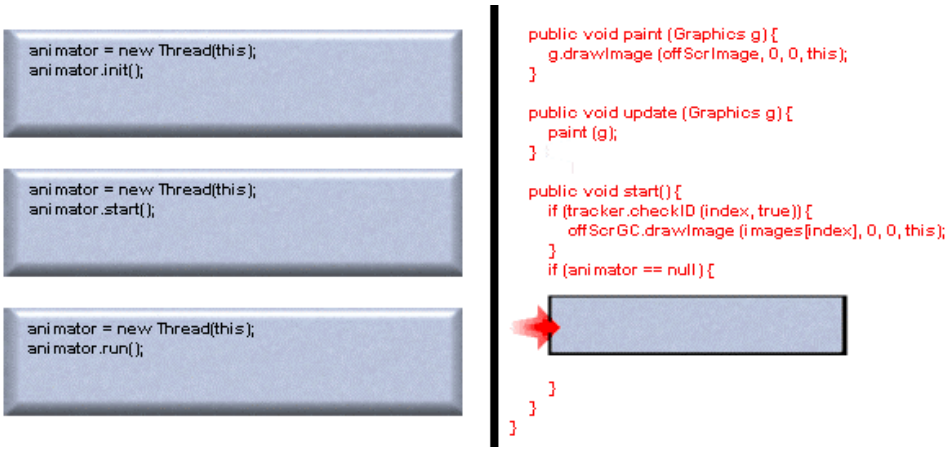
Viel Spass beim zuschauen!

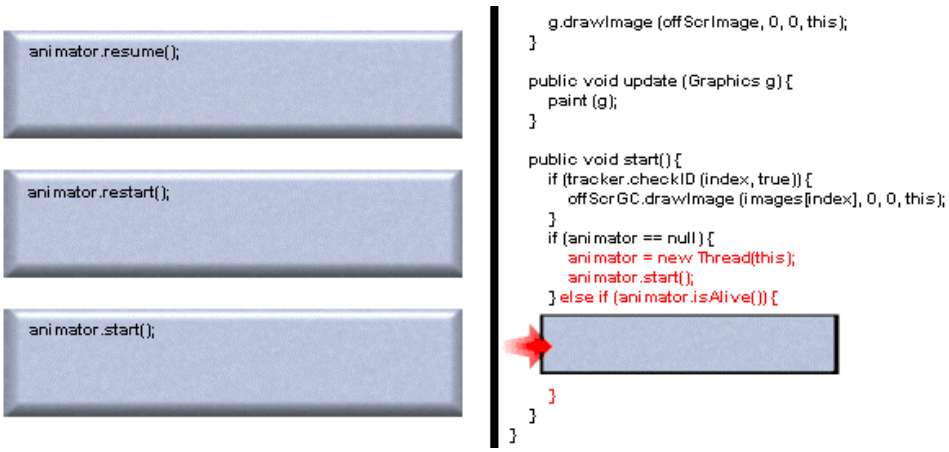
1.5.8. Praktische Übung - Konstruktion einer Animation

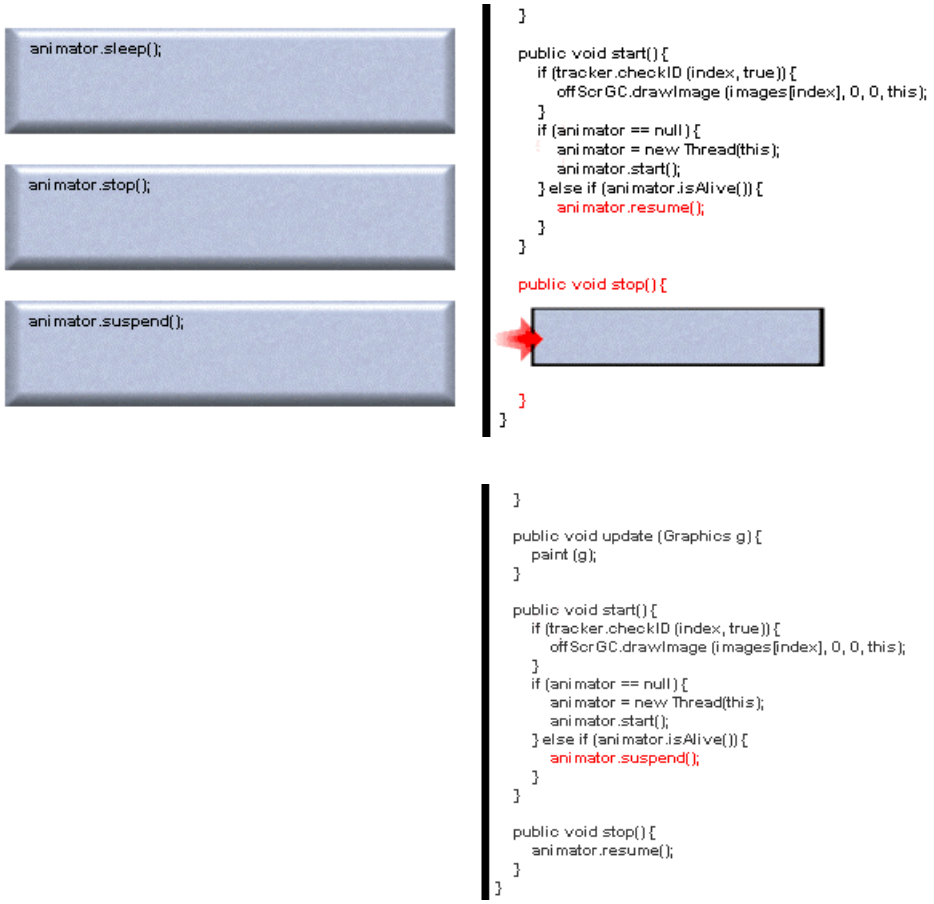
In dieser Übung können Sie die Entwicklung eines Applets nachvollziehen, welches eine Animation anzeigt (als Bildfolge). Die Animationbeginnt bereits, bevor alle Bilder geladen sind.

Wie üblich müssen Sie zuerst Programmteile auswählen, dass sehen Sie das Programm als Ganzes und anschliessend können Sie es testen und genauer studieren. Sie finden auf dem Server / Web / der CD weitere Beispiele, zum Teil vereinfachte, zur Generierung von Animationen.

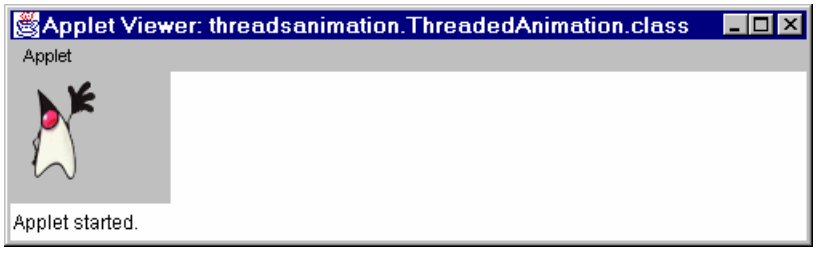








Und hier das Ergebnis:



1.5.8.1. Der Programmcode

```
package threadsanimation;

import java.awt.*;
import java.applet.Applet;

public class ThreadedAnimation extends Applet implements Runnable {

    Image images[];
    MediaTracker tracker;
    int index = 0;
    Thread animator;
    boolean isStopped = false;
    int maxWidth, maxHeight;
    Image offScrImage;
    Graphics offScrGC;

    public void init() {
        tracker = new MediaTracker(this);

        maxWidth = 100;
        maxHeight = 100;
        images = new Image[10];

        try {
            offScrImage = createImage(maxWidth, maxHeight);
            offScrGC = offScrImage.getGraphics();
            offScrGC.setColor(Color.lightGray);
            offScrGC.fillRect(0, 0, maxWidth, maxHeight);
            resize(maxWidth, maxHeight);
        } catch (Exception e) {
            e.printStackTrace();
        }

        for (int i=0; i < 10; i++) {
            String imageFile = new String ("graphics/Duke/T" +
                String.valueOf(i+1) + ".gif");
            images[i] = getImage(getDocumentBase(), imageFile);
        }
    }

    public void paint(Graphics g) {
        g.drawImage(offScrImage, 0, 0, this);
    }

    public void update (Graphics g) {
        paint (g);
    }

    public void start() {
        if (tracker.checkID (index, true)) {
            offScrGC.drawImage (images[index], 0, 0, this);
        }
        if (animator == null) {
            animator = new Thread(this);
            animator.start();
        } else if (animator.isAlive()) {
            animator.resume();
        }
    }

    public void stop() {
        animator.suspend();
    }

    public void run() {
```

```
        while (animator != null) {
            offScrGC.fillRect(0,0,100,100);
            offScrGC.drawImage (images[index], 0, 0, this);
            index++;
            if (index == images.length) {
                index = 0;
            }

            try {
                 animator.sleep(200);
            } catch (InterruptedException e) { }
                repaint ();
        }
    }
}
```

1.5.9. Quiz

Hier einige Fragen, die Sie zum Teil vermutlich nur beantworten können, wenn Sie sich die Dokumentation des Threading APIs nachschauen.

1. Welche der folgende Aussagen sind korrekt?[6]
   a) Die Priorität eines Threads wird erst festgelegt, nachdem er gestartet wurde.
   b) Unkoordinierte Änderungen an Daten kann zu inkonsistenten Daten führen.
   c) Das Thread Objekt wird zerstört sobald der Thread beendet ist.
   d) Threads können miteinander kommunizieren.
   e) Threads, welche aus der selben Klasse stammen werden auch zusammen beendet.

2. Gegeben sei eine Klasse xyz, welche Runable implementiert. [7]
   Welche der folgenden Programmfragmente ist korrekt?
   a) Thread t = new Thread();
   b) Runnable r = new xyz();
      Thread t = new Thread(r);
   c) Runnable r = new xyz();
      Thread t = new Thread();
   d) Runnable r = new xyz();
      Thread t = new Thread(t);

3. Geben Sie zwei Argumente, die für ... implements Runnable sprechen.
   Geben Sie zwei Argumente, die für ... extends Thread sprechen.

4. synchronized als Methoden Modifier bewirkt, dass ...

5. Die Thread Kommunikation in Java geschieht mit Hilfe von [8]
   a) suspend() ... resume()
   b) sleep() ... wake()
   c) join() ... sync()
   d) wait() ... notify()

[6] b) und d) sind korrekt;
[7] b)
[8] d)

1.5.10. Zusammenfassung

In diesem Modul haben Sie:

- das Konzept 'Thread' kennen gelernt.
- gelernt Threads zu konstruieren und deren Ausführung zu kontrollieren sowie deren Zugriff auf gemeinsam genutzte Daten zu koordinieren
- einige der Schwierigkeiten kennen gelernt, die bei der Verwendungvon Threads auftreten können, vorallem bei der Nutzung gemeinsamer Daten, verschachtelten Aufrufen (Verklemmung, Deadlock Gefahr) oder dem gegenseitigen Warten (Aushungern, Starvation [Philosophen Problem]).
- Sie kennen die Bedeutung des Schlüsselwortes `synchronized` in Java

1.6. Modul 4 : Netzwerkprogrammierung

In diesem Modul

- Modul 4 : Netzwerkprogrammierung in Java
  - Lektion 1 - Verbindungsaufbau mit Sockets
  - Lektion 2 - UDP Sockets
  - Praktische Übung - Client/Server Beispiel
  - Quiz
  - Zusammenfassung

1.6.1. Einleitung

Wir haben bei der Besprechung der Eingabe und Ausgabe Klassen und Methoden ein einfaches Beispiel gesehen, wie man in Java an Stelle lokaler auch Web oder TCP basierte Systeme einbinden kann. Das entsprechende Beispiel lud eine Bilddatei aus einer URL auf den lokalen Rechner.

Netzwerk Programmierung gestattet es Ihnen, Ihre Applikationen auf primär TCP basierte Netzwerke auszudehnen. Java stellt dafür einige Basisklassen zur Verfügung. Diese befinden sich in im `java.net` Package. Diese stellt Ihnen plattformunabhängige Abstraktionen verschiedener Netzwerkkonzepte wie Web Protokolle und Sockets (eine Erweiterung des Dateikonzepts aus Unix auf Netzwerke) zur Verfügung.

Java stellt daneben noch viel tiefergehende Netzwerkkonzepte zur Verfügung. Wir werden diese später besprechen. Hier geht es darum, Ihnen einen ersten Einblick zu gewähren.

Mit dem Wissen aus Modul 2 betreffend Eingabe und Ausgabe Streams werden Sie nun in die Lage versetzt auch über Netzwerke Dateien zu lesen und Daten flexibel auszutauschen.

1.6.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- einen minimalen TCP/IP Server in Java zu schreiben
- einen minimalen TCP/IP Client in Java zu schreiben
- zu verstehen, wie UDP Sockets in Java implementiert sind

Natürlich werden Sie auch die entsprechenden Begriffe kennen lernen!

1.6.2. Lektion 1 - Verbindungsaufbau mit Sockets

1.6.2.1. Sockets - was ist das?

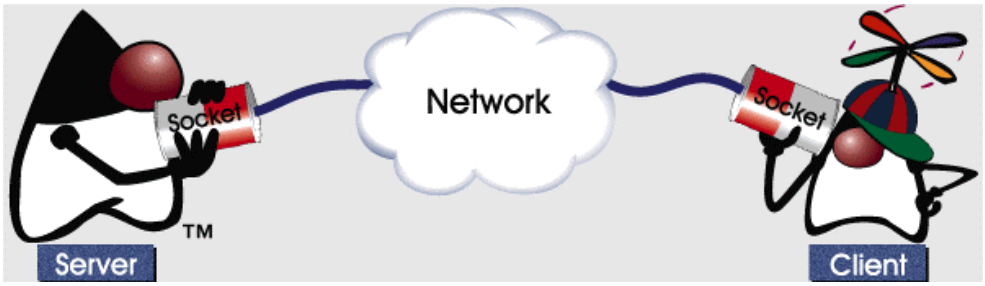
"Sockets" steht für die Endpunkte einer Kommunikationsverbindung zwischen Prozesse in einem bestimmten Programmiermodell. Der Begriff ist so populär, dass er nicht einheitlich verwendet wird. Aber ein Socket ist immer der Endpunkt einer Verbindung, sozusagen die Steckdose.

Java verwendet auch in der Netzwerkkommunikation das Streams Modell für den Datenaustausch. In diesem Fall werden aber pro Socket zwei Streams verwendet: einen Eingabe Stream und einen Ausgabe Stream:

- Ein Prozess sendet seine Daten an einen anderen Prozess über das Netzwerk einfach indem er die Daten in den Ausgabestrom des Sockets schreibt.

- Der Prozess empfängt seine Daten von einem anderen Prozess über das Netzwerk einfach inden er Daten aus dem Ausgabestrom des Sockets liest.

Nachdem die Verbindung einmal aufgebaut ist, merken Sie kaum einen Unterschied mehr zwischen einem lokalen Dateizugriff und der Kommunikation über ein Netzwerk mit Hilfe von Streams.



1.6.2.2. Aufsetzen einer Verbindung und deren Adressierung

Um eine Verbindung aufbauen zu können, muss auf der einen Seite der Kommunikationsverbindung ein Programm gestartet werden, der Server, welches auf Verbindungsanfragen wartet.
Auf der anderen Seite der der Kommunikationsverbindung muss ein weiteres Programm, der Client, versuchen das erste (den Server) zu erreichen.

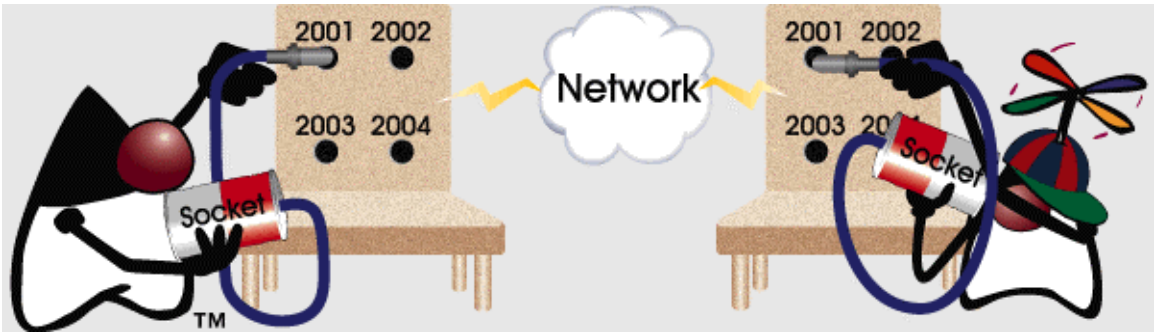
Eigentlich unterscheidet sich dies kaum von einer Telefonverbindung: der eine Kommunikationspartner muss die Nummer des anderen Partners anwählen. Der Gesprächspartner auf der anderen Seite muss auf den Anruf warten und passend reagieren.

Falls Sie eine Telefonverbindung aufbauen möchten, müssen Sie die Telefonnummer Ihres Gesprächspartners kennen.

Beim Aufbau einer Netzwerkverbindung müssen Sie die Adresse oder den Namen des remote Rechners kennen.

Kommunikationsverbindungen sind oder waren zumindest teuer. Daher muss man sie möglicht gut nutzen. Daher hat man neben dem remote Host noch ein weiteres Unterscheidungsmerkmal eingeführt, den Port (gekennzeichnet durch eine Nummer, die Portnummer).
Die Portnummer entspricht beim Telefon der internen Nummer (extension number). Nicht jeder Telefonapperat besitz eine externe Nummer, aber jedes interne Telefon ist erreichbar: auch hier haben Sie es mit einer Kostenoptimierung und Kommunikationsoptimierung zu tun.

Der Port liefert Ihnen eine Unterscheidung des Kommunikationzwecks: hinter jeder Portnummer steht eine bestimmte Anwendung.



Die Analogie zur Telefonie wäre, dass Sie beim Anruf einer bestimmten Firma eine Sammelnummer wählen, aber eigentlich mit dem Kundendienst sprechen möchten.

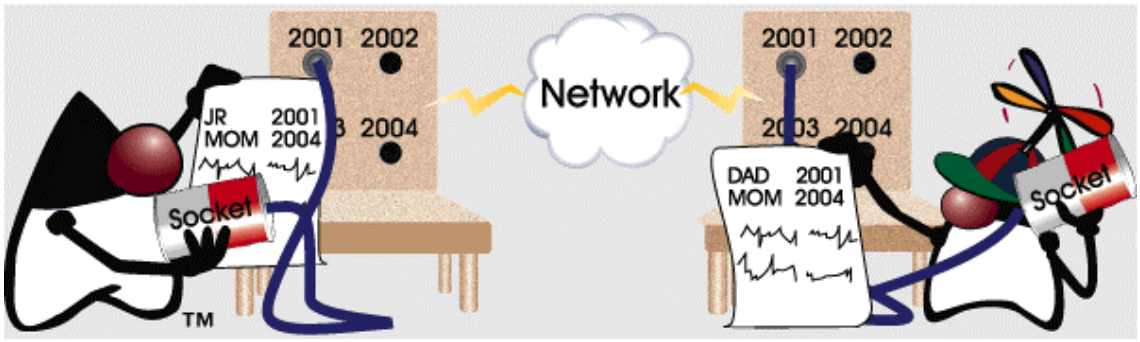
Die Sammelnummer entspricht dem Host Namen, die Abteilung entspricht dem konkreten Port. Die Funktion der Abteilung entspricht einem 'Dienst', *service*.

1.6.2.3. Die Port Nummer

In TCP / IP bestehen Port Nummern aus 16 Bit (0 ..65535).

Die Port Nummern unterhalb 1024 sind für vordefinierte Dienste reserviert. Sie sollten diese Portnummern nicht einsetzen, ausser Sie wollen einen dieser Dienste nutzen (TELNET, SMTP, FTP, POP3, NEWS...).

Damit ein Client mit einem Server kommunizieren kann, müssen Sie im voraus festlegen über welchen Port Sie kommunizieren wollen, sonst können die beiden nicht miteinander kommunizieren! Wir werden später auch Port Scanner bauen, mit denen Sie Portbereiche abfragen können und vieles mehr. Im Moment geht es aber darum die grundlegenden Begriffe kennen zu lernen!



Sie finden auf Ihrem Rechner, mindestens falls Sie TCP / IP instaliert haben, eine Datei Services, die Sie nicht verändern sollten, da die meisten Ports eine fixe Bedeutung haben (Standardisiert durch die Internet Behörde).

Bei Windows NT finden Sie die Datei im Verzeichnis WinNT/System32/Drivers/etc:

```
# Format:
#
# <Dienstname> <Anschluánummer>/<Protokoll> [Alias...] [#<Kommentar>]
#

echo                 7/tcp
echo                 7/udp
...
chargen             19/tcp    ttytst source
chargen             19/udp    ttytst source
ftp-data            20/tcp
ftp                 21/tcp
telnet              23/tcp
smtp                25/tcp    mail
time                37/tcp    timserver
time                37/udp    timserver
..
pop                109/tcp    postoffice
pop2               109/tcp                    # Post Office
pop3               110/tcp    postoffice
...
```

1.6.2.4. Das Java Netzwerkmodell

In Java werden TCP/IP Socket Verbindungen mit Hilfe von Klassen im Package `java.net` implementiert. Ziel ist es, sichere Verbindungen aufbauen zu können. Daher gestattet Ihnen Java nicht bestimmte Subprotokolle von TCP einzusetzen, beispielsweise jene, von denen man weiss, dass sie oft zum Hacken von Systemen eingesetzt werden. Java abstrahiert und setzt höher an. Sie könnten zwar viele Kommunikationsparameter selbst einstellen. In der Regel werden Sie dies aber nicht benötigen, sich also oberhalb dieses Kommunikationslayers bewegen.

Schematisch sieht eine Kommunikation zwischen einem Client und einem Server folgendermassen:

- der Server starten an einem bestimmten Host und einer bestimmten Port Nummer. Sobald der Client eine Verbindung verlangt, öffnet der Server die Socketverbindung mit der `accept()` Methode.
  Der Server verwendet amAnfang einen speziellen Socket, einen Serversocket. Sobald die Verbindung zu Stande kommt, wechselt der Server auf normale Sockets!

- der Client baut eine Verbindung zum Host an dessen Port *port_number* auf.

- Der Server und der Client wechseln auf eine private Port Nummer: dies geschieht automatisch und wird durch TCP erledigt.
  Aus Sicht des Anwendungsprogrammierers kommunizieren Client und Server einfach mittels `InputStream` und `OutputStream`.

Programmbeispiele für Client und Server finden Sie auf den nächsten Seiten.

1.6.2.5. Ein minimaler TCP/IP Server

Eine TCP/IP Server Applikation basiert auf den `ServerSocket` und `Socket` Netzwerkklassen in `java.net`.

Die `ServerSocket` Klasse übernimmt dabei den Grossteil der Arbeit für den Aufbau eines Servers.

```
1    import java.net.*;
2    import java.io.*;
3
4    public class EinfacherSocketServer {
5        public static void main(String args[]) {
6            ServerSocket s = null;
7            Socket s1;
8            String sendString = "Hello Net World!";
9            OutputStream s1out;
10           DataOutputStream dos;
11
12           // registrieren des Dienstes an Port 5432
13           try {
14               s = new ServerSocket(5432);
15           } catch (IOException e) { }
16
17           // "..warte und lose.."
18           while (true) {
19               try {
20                   // warten auf eine Verbindung
21                   s1=s.accept();
22                   // bestimme den Kommunikationsstream
23                   // dieses Sockets : Socket, nicht ServerSocket!!
24                   s1out = s1.getOutputStream();
25                   dos = new DataOutputStream (s1out);
26
27                   // Senden des String!
28                   dos.writeUTF(sendString);
29
30                   // Verbindung schliessen
31                   // Der Serversocket bleibt offen
32                   dos.close();
33                   s1out.close();
34                   s1.close();
35               } catch (IOException e) { }
36           }
37       }
38   }
```

1.6.2.6. Ein minimaler TCP/IP Client

Ein TCP/IP Client wird mit Hilfe der `Socket` Klasse gebaut. Auch hier wird die eigentliche Arbeit durch die fix fertig definierte Klasse `Socket` erledigt.

Der Client baut eine Verbindung zum Server aus dem vorigen Abschnitt auf und schickt die empfangene Zeichenkette an `stdout`.

```
1    import java.net.*;
2    import java.io.*;
3
4    public class EinfacherSocketClient {
5        public static void main(String args[]) throws IOException {
6            int c;
7            Socket s1;
8            InputStream s1In;
9            DataInputStream dis;
10
11           // Verbindungsaufbau zum Host und dessen Port 5432
12           s1 = new Socket("meinServer",5432);
13           // bestimmen des Inputstreams
14           // des Sockets und lesen des Inhalts
15           s1In = s1.getInputStream();
16           dis = new DataInputStream(s1In);
17
18           String st = new String (dis.readUTF());
19           System.out.println(st);
20
21           // schliessen der Verbindung und Abbau der Verbindung
22           dis.close();
23           s1In.close();
24           s1.close();
25       }
26   }
```

Um Client und Server richtig in Action sehen zu können, müssen Sie an Stelle von "meinServer" beispielsweise die IP Adresse eingeben und dann Ihren Nachbarn bitten den Client zu starten.

Schauen Sie die Verbindung genauer an, indem Sie in einem DOS Fenster den Netzwerkstatus abfragem: netstat mit einigen Flags.

Versuchen Sie herauszufinden, auf welche Port Nummer die Verbindung switched, nachdem die Verbindung aufgebaut ist. Eventuell müssen Sie dafür Client und Server längere Zeit laufen lassen (eine Schleife ausführen lassen).

1.6.3. Lektion 2 - UDP Sockets

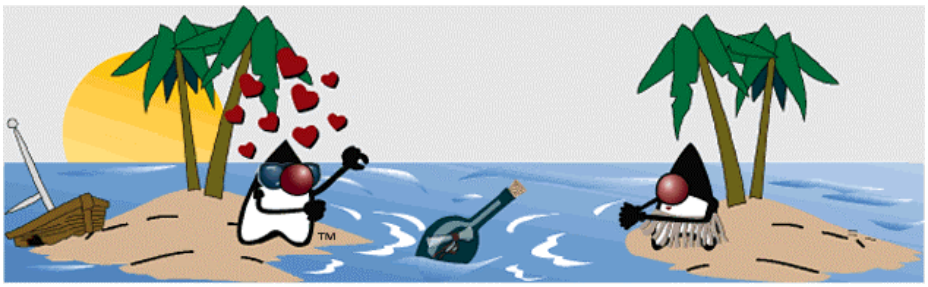
1.6.3.1. Was ist ein UDP Socket?

TCP / IP Sockets sind verbindungsorientiert, besitzen also klare Quellen und Senken, Endpunkte der Kommunikation.

UDP, User Datagram Protocol, dagegen ist ein "verbindungsloses" Protokoll.

In Analogie zum Telefon könnte man sagen, UDP Kommunikation sieht eher aus wie Briefpost. Briefpost hat die tolle Eigenschaft, dass falls Sie zwei Karten vom selben Ferienort an Ihre Bekannten senden, die eine Karte nach zwei Tagen, die andere ins Nachbardorf erst nach drei Wochen ankommt. Wenn Sie mehrere Karten hintereinander, sagen wir mal täglich eine, an jemanden senden, kann es leicht vorkommen, dass diese Reihenfolge beim Empfänger nicht mit der Reihenfolge des Absendens übereinstimmt! Einzelne Karten werden eventuell nie ankommen.

Im Gegensatz zu UDP oder eben der Briefpost, haben Sie bei der synchronen Kommunikation, wie sie TCP/IP oder das Telefon darstellt, die Garantie, dass die einzelnen Pakete, in die die Daten für die Kommunikation zerlegt werden, in der richtigen reihenfolge ankommen. Stellen Sie sich ein Telefongespräch vor, bei dem die einzelnen Worte nicht notwendigerweise in der richtigen Reihenfolge ankommen...



Das User Datagram Protokoll wird dem Programmierer mit Hilfe der beiden Java Klassen DatagramSocket und DatagramPacket zur Verfügung gestellt.

Das `DatagramPacket` ist eine Message, welche Informationen über den Sender, die Länge der Meldung und die Meldung selbst enthält. Die Übermittlung vom Sender zum Empfänger geschieht einzig und allein aufgrund der im Packet enthaltenen Information.

Der `DatagramSocket` ist der Sende- oder Empfangs-Punkt für einen Paket Lieferdienst. Jedes Paket wird individuell adressiert und geroutet. Mehrere Pakete an den selben Empfänger können unterschiedlich geroutet werden. Die Sende- und die Empfangs- Reihenfolge brauchen nicht übereinzustimmen.

In Ihrer Service Datei (siehe oben) sehen Sie auch welche Dienste von TCP, welche von UDP unterstütztwerden. Oft werden Dienste (daytime,...) von beiden Protokollen unterstützt, benötigen aber eben unterschiedliche Klassen bei der Java Implementierung.

1.6.3.2. Die DatagramPacket Klasse

Die Klasse `DatagramPacket` besitzt zwei Konstruktoren - einen um Daten zu empfangen, und einen weiteren um Daten zu senden:

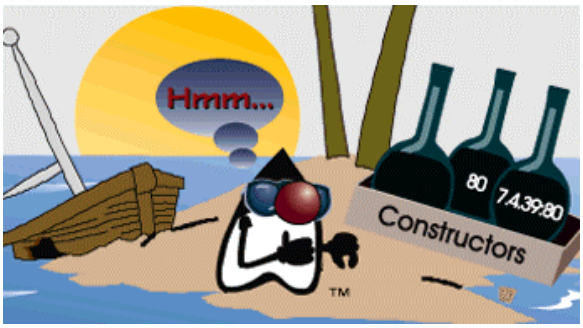
- `DatagramPacket (byte [] recvBuf, int readLength)`
  Mit diesem Konstruktor können Sie eine UDP Verbindung aufbauen, über welche Sie in einem Byte-Array ein UDP Paket empfangen können. Das Byte-Array wird leer an den Konstruktor übergeben; mit der Variable `readLength` geben Sie an, wieviele Bytes Sie lesen wollen (natürlich weniger als im Byte- Array Platz haben).

- `DatagramPacket (byte[] sendBuf, int sendLength, InetAddress iaddr, int iport)`
  Mit diesem Konstruktor können Sie ein UDP Paket für die Übermittlung zusammenstellen. Hierbei sollte die Länge der Meldung `sendLength` nicht länger als der `sendBuf` Byte- Array sein.
  Wie Sie oben sehen, werden auch noch die Internet Adresse (**`InetAddress)`** und der Port (**`int iport`**) angegeben. Diese Angaben betreffen natürlich den Empfänger Host.

1.6.3.3. Die DatagramSocket Klasse

Die DatagramSocket Klasse wird eingesetzt, um UDP Pakete zu lesen oder zu schreiben / senden. Die Klasse besitzt drei Konstruktoren, welche die Angabe der Internet Adresse und des Ports gestatten:

- `DatagramSocket ()`
  ohne Angaben wird ein Standard Port am `localhost` Host ausgewählt (ein freier Port)
- `DatagramSocket (int port)`
  der Socket wird an den Port `port` des `localhost` Host gebunden
- `DatagramSocket (int port, InetAddress iaddr)`
  der Socket wird an den Port `port` des Hosts mit der Internet Adresse `iaddr` gebunden.



1.6.3.4. Ein UDP Daytime Server

```
//Titel:        UDP Daytime Server Klasse
import java.net.*;
import java.io.*;
import java.util.Date;

public class UDPDaytimeServer {

  protected static int defaultPort = 7230;
  protected static int defaultBufferLength = 65507;

  public static void main(String[] args) {
    System.out.println("Start UDPDaytimeServer");
    DatagramPacket incoming;

    int port;
    int len;

    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
      port = defaultPort;
    }
    try {
      len = Integer.parseInt(args[1]);
    }
    catch (Exception e) {
      len = defaultBufferLength;
    }

    try {
      DatagramSocket ds = new DatagramSocket(port);
      byte[] buffer = new byte[len];
      while (true) {
        incoming = new DatagramPacket(buffer, buffer.length);
        try {
          ds.receive(incoming);
          respond(ds, incoming); // siehe unten
          System.out.println("Datagramm empfangen und beantwortet");
        }
        catch (IOException e) {
          System.err.println(e);
        }
      } // end while
    }  // end try
    catch (SocketException se) {
      System.err.println(se);
    }  // end catch
    System.out.println("Ende UDPServer");
  }  // end main

  public static void respond(DatagramSocket ds, DatagramPacket dp) {
    System.out.println("UDPDaytimeServer.respond()");
    DatagramPacket outgoing;
    Date now = new Date();
    String s = now.toString();
    byte[] data = new byte[s.length()];
    s.getBytes(0, s.length(), data, 0);
    try {
      System.out.println("Time: "+s);
      outgoing = new DatagramPacket(data, data.length, dp.getAddress(), dp.getPort());
      ds.send(outgoing);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

1.6.3.5. Ein UDP Daytime Client

```
package udpdaytimeclient;

//Titel:        UDP Klasse
//Beschreibung:  ein allgemeiner Client
```

```
import java.net.*;
import java.io.*;

public class UDPClient {

  protected static int defaultPort = 13;   //
  protected static int bufferLength = 8192; // NFS abgeschaut

  public static void main(String[] args) {
    System.out.println("Start UDPDaytimeClient");

    String hostname;
    int port;
    int len;

    if (args.length > 0) {
      hostname = args[0];
    }
    else {
      hostname = "localhost";
      port = defaultPort;
      len = bufferLength;
    }
    try {       // zur Abwechselung mal eine neue Variante
      port = Integer.parseInt(args[1]);
    }
    catch (Exception e) {
      port = defaultPort;
    }
    try {
      len = Integer.parseInt(args[2]);
    }
    catch (Exception e) {
      len = bufferLength;
    }

    try {
      System.out.println("Host="+hostname+"; Port="+port);
      DatagramSocket ds = new DatagramSocket(0);
      InetAddress ia = InetAddress.getByName(hostname);

      DatagramPacket outgoing = new DatagramPacket(new byte[512], 1, ia, port);
      DatagramPacket incoming = new DatagramPacket(new byte[len], len);
      ds.send(outgoing);
      System.out.println("DatagramSocket : Datagramm gesendet");
      ds.receive(incoming);
      System.out.println("DatagramSocket : Datagramm empfangen");
      System.out.println(new String(incoming.getData(), 0, 0, incoming.getLength()));
    }  // end try
    catch (UnknownHostException e) {
      System.err.println(e);
    }  // end catch
    catch (SocketException e) {
      System.err.println(e);
    }  // end catch
    catch (IOException e) {
      System.err.println(e);
    }  // end catch
    System.out.println("Ende UDPClient");
  }  // end main
}
```

1.6.3.6. Die Ausgaben von Client und Server

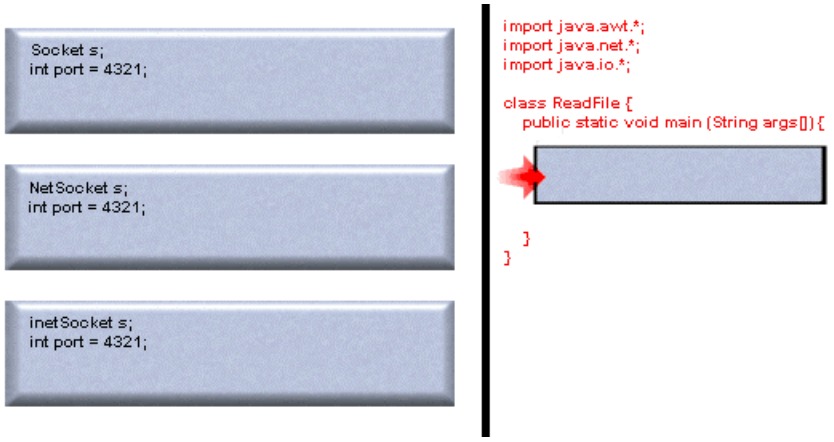
Start **UDPDaytimeServer**
UDPDaytimeServer.respond()
Time: Sat Mar 03 15:34:19 GMT+01:00 2001
Datagramm empfangen und beantwortet

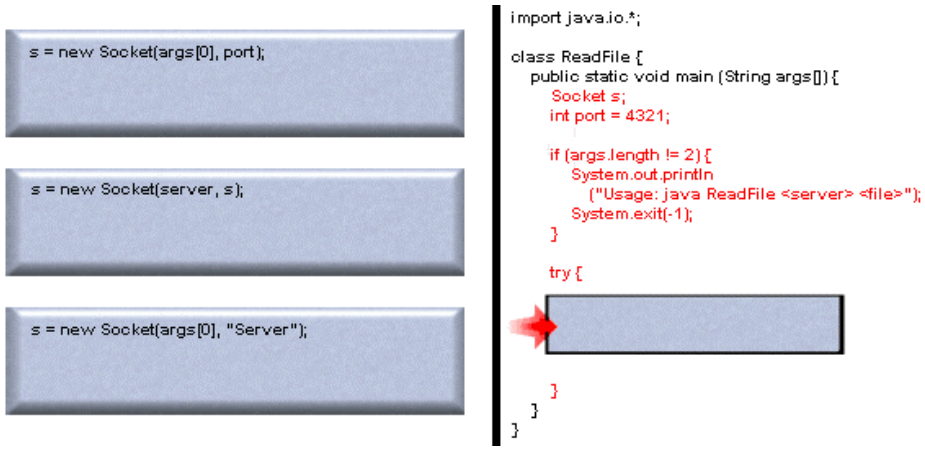
Start **UDPDaytimeClient**
Host=localhost; **Port=13**

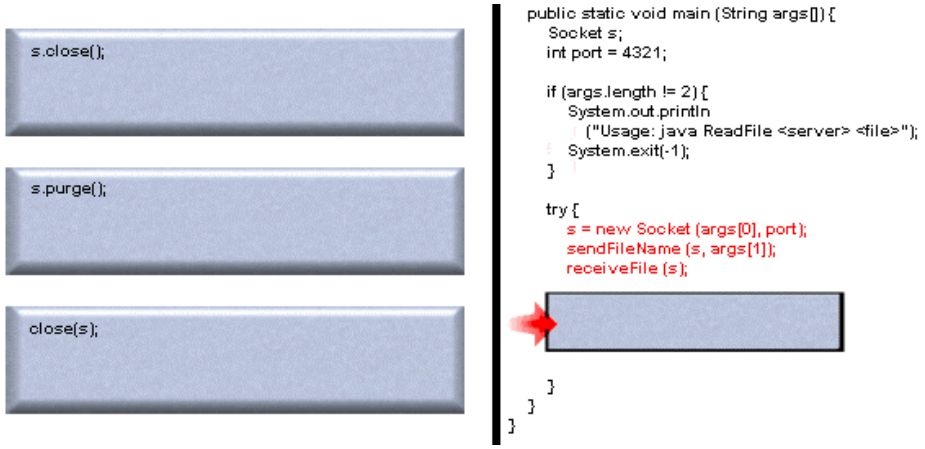
```
DatagramSocket : Datagramm gesendet
DatagramSocket : Datagramm empfangen
Sat Mar 03 15:34:19 GMT+01:00 2001
Ende UDPClient
```

1.6.4. Praktische Übung - Client/Server Beispiel

Die folgenden zwei Übungen können Sie wieder Schritt für Schritt eine kleine Applikation zusammenbauen. An Schluss finden Sie den Programmcode. Die Lösung sehen Sie schrittweise.
Der Server empfängt einen Dateinamen und sendet diese Datei zurück.







Damit haben wir folgenden Server:

```
class ReadFile {
   public static void main (String args[]) {
      Socket s;
      int port = 4321;

      if (args.length != 2) {
         System.out.println
            ("Usage: java ReadFile <server> <file>");
         System.exit(-1);
      }

      try {
         s = new Socket (args[0], port);
         sendFileName (s, args[1]);
         receiveFile (s);
         s.close();
      } catch (IOException e) {
         System.out.println ("Connection failed");
      }
   }
}
```

Die vollständige Lösung finden Sie auf den folgenden Seiten. Lauffähige Versionen sind auf dem Server / Web / der CD.

1.6.4.1. Das Server Programm

```
package netsocketdateiserver;

import java.awt.*;
import java.net.*;
import java.io.*;

class SocketDateiServer {

    public static void main(String args[]) {
        System.out.println("Der Socket Datei Server wird gestartet");
        ServerSocket s=(ServerSocket) null;
        Socket s1;
        byte inbuf [] = new byte [100];
        String fileName;

        try {
            s = new ServerSocket(4321,1);
        } catch (IOException e) {
            // Zeitüberschreitung : das könnte man umgehen
            System.out.println ("\nServer timed out!");
            System.exit(-1);
        }

        while (true) {
            try {
                System.out.println("Der Socket Datei Server wartet auf Anfragen...");
                s1=s.accept();
                // bestimmen des Dateinamens
                fileName = getFileName (s1);
                sendFileToClient (s1, fileName);
                s1.close();
            } catch (IOException e) {
                System.out.println ("Fehler - " + e.toString());
            }
        }
    }

    public static String getFileName (Socket s1) throws IOException {
        InputStream s1in;
        DataInputStream d1In;
        String sfile;

        s1in = s1.getInputStream();
        d1In = new DataInputStream (s1in);

        sfile = d1In.readLine();
        System.out.println ("Folgende Datei wird gelesen: " + sfile);
        return (sfile);
    }

    public static void sendFileToClient (Socket s1, String sfile)
        throws IOException  {

        int c;
        FileInputStream fis;
        OutputStream s1out;

        s1out = s1.getOutputStream();
        File f = new File (sfile);

        if (f.exists() != true) {
            String error = new String ("Die Datei " + sfile + " existiert nicht ...\n");
            int len = error.length();
            for (int i = 0; i < len; i++) {
                s1out.write((int)error.charAt(i));
            }
            System.out.println (error);
            return;
        }

        if (f.canRead()) {
            fis = new FileInputStream (sfile);
            System.out.println ("Sending: " + sfile);
            while ((c = fis.read ()) != -1) {
                s1out.write (c);
            }
            fis.close();

        } else {
            String error=new String("Die Datei "+ sfile + " kann nicht gelesen werden...\n");
            int len = error.length();
```

```
            for (int i = 0; i < len; i++) {
                slout.write((int)error.charAt(i));
            }
                System.out.println (error);
        }
    }
}
```

Beispiel Protokoll

Der Socket Datei Server wird gestartet
Der Socket Datei Server wartet auf Anfragen...
Folgende Datei wird gelesen: NetSocketDateiServer.html
Sending: NetSocketDateiServer.html
Der Socket Datei Server wartet auf Anfragen...
Folgende Datei wird gelesen: NetSocketDateiServer.html
Sending: NetSocketDateiServer.html
Der Socket Datei Server wartet auf Anfragen...

Der SocketDateiClient wird gestartet ...
Usage: java SocketDateiClient <server> <file>
Verbindungsaufbau : Server localhost; Port 4321
Senden des Dateinamens : NetSocketDateiServer.html
Empfangen der Datei ...
<HTML>
...
</HTML>
Alle Daten wurden empfangen.
Verbindungsabbau ...
Der SocketDateiClient wird beendet.

1.6.4.2. Das Client Programm

```
package netsocketdateiclient;

import java.awt.*;
import java.net.*;
import java.io.*;

class SocketDateiClient {

    public static void main(String args[]) {
        System.out.println("Der SocketDateiClient wird gestartet ...");
        Socket s;
        int port = 4321;
        String host = "localhost";
        String fname = "NetSocketDateiServer.html";

        if (args.length != 2) {
            System.out.println ("Usage: java SocketDateiClient <server> <file>");
            //System.exit (-1);
        } else {
          host = args[0];
          fname= args[1];
        }

        try {
            System.out.println("Verbindungsaufbau : Server "+host+"; Port "+port);
            s = new Socket(host, port);
            System.out.println("Senden des Dateinamens : "+fname);
            sendFileName (s, fname);
            System.out.println("Empfangen der Datei ...");
            receiveFile (s);
            s.close();
            System.out.println("Alle Daten wurden empfangen.");
            System.out.println("Verbindungsabbau ...");
        } catch (IOException e) {
            System.out.println("Verbindungsaufbau schlug fehl");
        }
        System.out.println("Der SocketDateiClient wird beendet.");
    }

    public static void sendFileName (Socket s, String fileName)
        throws IOException {
        OutputStream sOut;
        DataOutputStream dOut;

        sOut = s.getOutputStream ();
        dOut = new DataOutputStream (sOut);
        String sendString = new String (fileName + "\n");
        dOut.writeBytes (sendString);
    }

    public static void receiveFile (Socket s) throws IOException {
        int c;
        InputStream sIn;

        sIn = s.getInputStream();
        while ((c = sIn.read()) != -1) {
            // nicht empfehlenswert für den Fall, dass Sie auch
            // binäre Dateien übertragen wollen!!
            System.out.print((char)c);
        }
    }
}
```

1.6.5. Quiz

Hier einige Fragen zu diesem Modul:

1) Erklären Sie folgende Begriffe:
   a) Server
   b) Client
   c) Datagram
   d) Socket

2) Um TCP Verbindungen aufzubauen verwendet Java[9]
   a) ServerSockets
   b) DatagramSockets
   c) DatagramPackages
   d) Sockets

3) Um UDP Verbindungen aufzubauen verwendet Java[10]
   a) ServerSockets
   b) DatagramSockets
   c) DatagramPackages
   d) Sockets

4) Um die Netzwerk- Verbindung aufbauen zu können, benötigen Sie:[11]
   a) den Systemtyp
   b) die Adresse des remote Hosts
   c) die Port Nummer
   d) den Socket
   e) das Datagram

[9] a) und d): die Frage war TCP! Datagramme werden nur in UDP verwendet.
[10] b) und c) : UDP verwendet Datagramme und DatagramSockets
[11] b) und c)

1.6.6. Zusammenfassung

In diesem Modul haben Sie gelernt,

- wie ein einfacher Socket Server aussieht und aufgebaut ist
- wie ein einfacher Socket Client aussieht und aufgebaut ist
- wie UDP Verbindungen in Java aufgebaut werden können.

1.7. Zusammenfassung des Kurses

In diesem Kurs haben Sie gelernt, wie

- auf überschriebene Methoden aus den überschreibenden Methoden zugegriffen werden kann.
- überschriebene Konstruktoren verwendet werden können.
- Konstruktoren der Oberklassen eingesetzt werden können.
- `static` Variablen und Methoden definiert und eingesetzt werden können.
- `final` Klassen, Methoden und Variablen sinnvoll deklariert und eingesetzt werden können.
- `abstract` Methoden und Interfaces definiert und eingesetzt werden können.
- Zugriffsrechte mit `protected` und weiteren Zugriffsmodifiern gesteuert werden kann.
- wit dem `-deprecation` Flag mit dem Java Compiler festgestellt werden kann, ob alte APIs verwendet werden.
- innere Klassen beschrieben und eingesetzt werden.
- das `java.io` Package aufgebaut ist und wie mit Streams die Eingabe und Ausgabe gesteuert werden kann.
- Dateien und Filter Treams definiert und mutiert werden.
- durch Objekt Serialisierung der Zustand eines Objekts persistent abgespeichert werden kann
- in Java Threads definiert werden und welche grundlegenden Methoden Ihnen zu deren Einsatz zur Verfügung stehen.
- grundlegende Thread Aufgaben gelöst werden können.
- Probleme beim nicht koordinierten Zugriff auf Daten entstehen können.
- durch Einsatz des Schlüsselworts `synchronized` Daten vor der Korruption gerettet werden können.
- einfache TCP/IP Client und Server aufgebaut sind.
- UDP Sockets in Java implementiert wurden.