J Zusammenfassung der 2. Übung

- Grundlagen der OO mit Java (Fortsetzung)
  - Vererbung
- weitere OO-Konzepte mit Java
  - abstrakte Klassen
  - Interfaces
  - Packages
  - Sichtbarkeitsattribute
  - statische Elemente
  - Konstanten
  - innere Klassen
- JUnit

K Überblick über die 3. Übung

- Lösung der Aufgabe 2
- Exceptions
- Streams
- Hinweise zur 3. Aufgabe
  - StringTokenizer

K.1 Lösung der 2. Aufgabe

- Shape.java
- Circle.java
- WhiteBoard.java

1 Shape.java

```
import java.awt.Graphics;

public abstract class Shape {

        int x,y,width,height;

        public Shape(int x, int y, int width, int height) {
                this.x = x;
                this.y = y;
                this.width = width;
                this.height = height;
        }
        public void move(int x, int y) {
                this.x = x;
                this.y = y;
        }
        public int getX(){ return x;}
        public int getY(){ return y; }
        public int disX(int x) { return x-this.x; }
        public int disY(int y) { return y-this.y; }

        public abstract void draw(Graphics g);
        public abstract boolean isInside(int x, int y);
}//Shape
```

2 Circle.java

```
import java.awt.Graphics;

public class Circle extends Shape {

        public Circle (){ this(30,20,100); }

        public Circle (int x, int y, int diameter){
                super(x,y,diameter,diameter);
        }

        public boolean isInside(int x, int y){
                double radius = height/2;
                double centerX = this.x + radius;
                double centerY = this.y + radius;
                double distanceSquared =
                 (centerX-x)*(centerX-x) + (centerY-y)*(centerY-y);
                return ( Math.sqrt(distanceSquared) <= radius);
        }

        public void draw(Graphics g){
                g.drawOval(x,y,height,width);
        }

}// Circle
```


3 WhiteBoard.java

```
import java.awt.Graphics;
import java.awt.Frame;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.awt.event.MouseAdapter;

public class WhiteBoard extends Frame{

    Shape shapes[] = new Shape[100];
    Shape selected = null;
    int xdiff, ydiff;

    public WhiteBoard (){
        super("WhiteBoard");
        setSize(400,300);
        setVisible(true);

                                              //- Fortsetzung folgt -
```


3 WhiteBoard.java (2)

```
this.addMouseMotionListener(
        new MouseMotionAdapter(){
                public void mouseDragged(MouseEvent e) {
                        if (selected != null) {
                                selected.move (  e.getX()-xdiff,
                                                 e.getY()-ydiff );
                                repaint();
                        }
                }
        });

                                                  //- Fortsetzung folgt -
```


3 WhiteBoard.java (3)

```
this.addMouseListener(
    new MouseAdapter(){
          public void mousePressed(MouseEvent e) {
                selected = null;
                if (e.getButton()>1) { // delete all selected
                      for(int i=0; i<shapes.length; i++)
                            if (shapes[i] != null &&
                                shapes[i].isInside(e.getX(),e.getY()))
                                  shapes[i] = null;
                } else {
                      for(int i=0; i<shapes.length; i++)
                            if (shapes[i] != null &&
                              shapes[i].isInside(e.getX(),e.getY())) {
                                selected = shapes[i]
                                // how far inside is (x,y) located
                                xdiff = selected.disX(e.getX());
                                ydiff = selected.disY(e.getY());
                                break;
                            }
                }
                repaint();
          }
});
}                                                 //- Fortsetzung folgt -
```

3 WhiteBoard.java (4)

```
        public void addShape(Shape shape){
                // wie in Aufgabe 1
        }

        public void addShape(String classname, String xstr, String ystr) {
                // wie in Aufgabe 1
        }

        public void paint(Graphics g){
                // wie in Aufgabe 1
        }

        public static void main(String [] args){
                // wie in Aufgabe 1
        }
}// WhiteBoard
```


L.1 Fehlerbehandlung

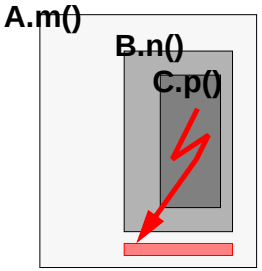
- Programm beenden (`System.exit()`)
  - meist eine schlechte Idee
- Ausgabe einer Fehlermeldung
  - hilft nicht den Fehler zu überwinden
- spezieller Rückgabewert kennzeichnet Fehler
  - Konstruktoren haben keinen Rückgabewert
  - Was ist wenn die Methode den Wertebereich des Rückgabewerts bereits voll ausnutzt?
- Aufruf einer benutzerdefinierten Fehlerroutine
  - unschön
  - Was muss diese Methode tun?
- Lösung: Ausnahmebehandlung (Exceptions)!

L Ausgewählte Kapitel des Java Laufzeitsystems

- Fehlerbehandlung (Exceptions)
- Ein-/Ausgabesystem (Streams)
- Threads (Teil 1)

1 Was ist Ausnahmebehandlung?

- Weiterreichen des Programmflusses vom Fehlerursprung zur Fehlerbehandlung



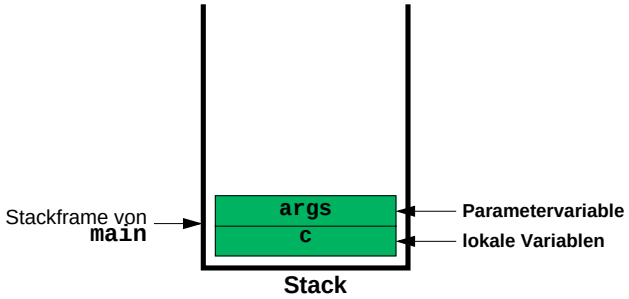
- Verantwortlichkeit:
  - Autor eines Codestücks kann den Fehler erkennen, weiß jedoch nicht wie er behandelt werden soll.
  - Benutzer des Codes weiß was zu tun ist, hat jedoch nicht die Möglichkeit den Fehler zu erkennen.

2 Was passiert bei einem Methodenaufruf?

```
class Customer {
 void createAccount
              (Bank bank) {
  Account account =
              new Account();
  bank.newAccount(account, 5);
```

```
class Bank {
  void newAccount
        (Account a, int i) {
      int counter = 0;
    ...
  }
```

```
class Main {
  public static void main(String
args[]) {
       Customer c = new Customer();
       c.createAccount(new Bank());
```

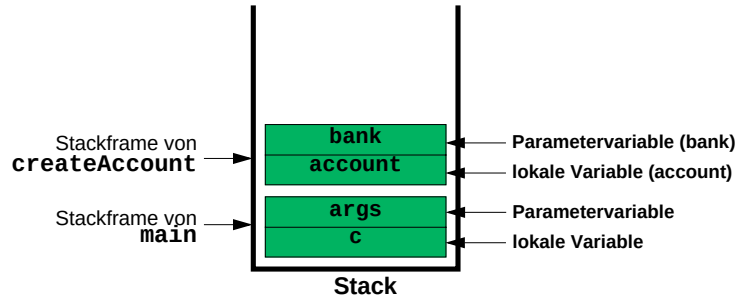


2 Was passiert bei einem Methodenaufruf? (2)

```
class Customer {
  void createAccount(Bank
bank) {
    Account account = new
Account();
    bank.newAccount(account,
```

```
class Bank {
  void newAccount(Account a, int
i) {
      int counter = 0;
    ...
  }
```

```
class Main {
  public static void main(String
args[]) {
       Customer c = new Customer();
       c.createAccount(new Bank());
```

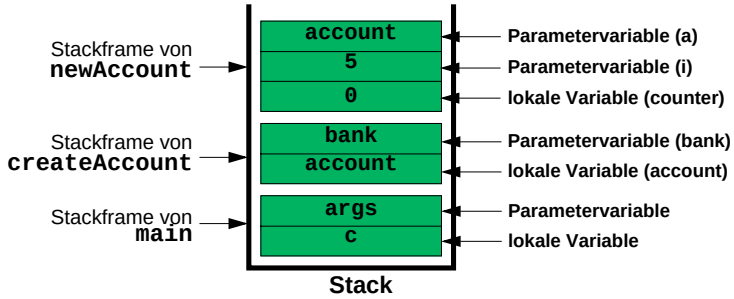


2 Was passiert bei einem Methodenaufruf? (3)

```
class Customer {
  void createAccount(Bank
bank) {
    Account account = new
Account();
    bank.newAccount(account,
```

```
class Bank {
  void newAccount(Account a, int
i) {
      int counter = 0;
    ...
  }
```

```
class Main {
  public static void main(String
args[]) {
       Customer c = new Customer();
       c.createAccount(new Bank());
```

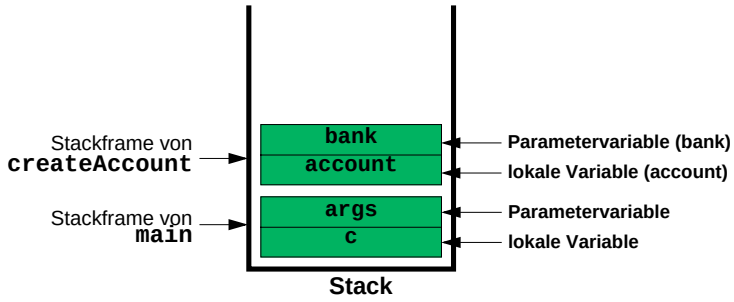


2 Was passiert bei einem Methodenaufruf? (4)

```
class Customer {
  void createAccount(Bank
bank) {
    Account account = new
Account();
    bank.newAccount(account,
```

```
class Bank {
  void newAccount(Account a, int
i) {
      int counter = 0;
    ...
  }
```

```
class Main {
  public static void main(String
args[]) {
       Customer c = new Customer();
       c.createAccount(new Bank());
```

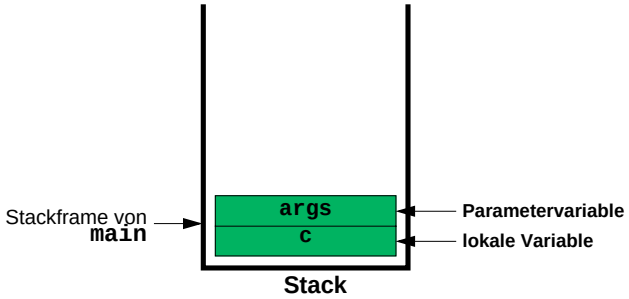


2 Was passiert bei einem Methodenaufruf? (5)

```
class Customer {
  void createAccount(Bank
bank) {
    Account account = new
Account();
    bank.newAccount(account,
```

```
class Bank {
  void newAccount(Account a, int
i) {
      int counter = 0;
    ...
  }
```

```
class Main {
  public static void main(String
args[]) {
      Customer c = new Customer();
      c.createAccount(new Bank());
```



3 Try, Throw und Catch-Anweisung

```
try {
    ...
    if (...) throw new MyException();
    ...
} catch(MyException e) {
    // exception handler
    ...
}
```

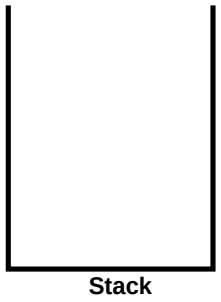
- throw wird benutzt um eine *Ausnahme* (*Exception*) zu werfen
- ein **`catch`**-Block muss direkt nach dem **`try`**-Block folgen
- es kann mehr als nur einen **`catch`**-Block geben
  - der passende catch-Block wird nach der Programmreihenfolge gesucht
- ein Methode muss nicht alle Exception fangen
  - nicht gefangene Exception werden automatisch an die aufrufende Methode weitergereicht

2 Was passiert bei einem Methodenaufruf? (6)

```
class Customer {
  void createAccount(Bank
bank) {
    Account account = new
Account();
    bank.newAccount(account,
```

```
class Bank {
  void newAccount(Account a, int
i) {
      int counter = 0;
    ...
  }
```

```
class Main {
  public static void main(String
args[]) {
      Customer c = new Customer();
      c.createAccount(new Bank());
```



4 Finally-Anweisung

- der **`finally`**-Block wird immer beim Verlassen eines **`try`**-Blockes ausgeführt
  - kann zum Aufräumen benutzt werden bei (un-)behandelten Ausnahmen

```
try {
    ...
} catch(...) {
    ...    // Fehlerbehandlung
} finally {
    ...    // Ressourcen freigeben
}
```

- ein **`finally`**-Block kann auch ohne **`catch`**-Block benutzt werden:

```
try {
    ...
    if (...) return;
    ...
} finally { ... }
```

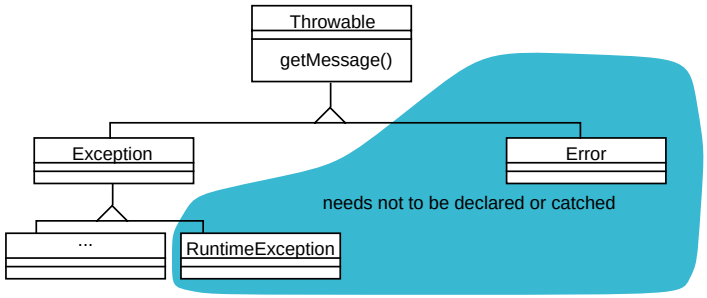
5 throws-Anweisung

- unbehandelte Exceptions müssen bei der Methode angeben werden:

```
class Test {
    void m() throws MyException {
        ...
        if (...) throw new MyException();
        ...
    }
}
```


6 Fehlerklassen

- alle Exceptions sind von **`Throwable`** abgeleitet
- Ausnahmen die beinahe überall auftreten können sind:
  - **`Error`**: Linkerfehler, Fehler im Format von Klassendateien, out of memory,..
  - **`RuntimeException`**: array index, null pointer, illegal cast, ...
- Ausnahmen von Anwendungen sind von **`java.lang.Exception`** abgeleitet



7 Ausnahmen und Vererbung: Fehlerbehandlung

- Ausnahmebehandlung von Unterklassen durch mehrere catch-Blöcke
- Bemerkung: Die Oberklasse behandelt alle Unterklassen, die Oberklasse sollte daher immer am Ende “gefangen” werden:

```
class MathException {}
class ZeroDivideException extends MathException {}
class InvalidArgException extends MathException {}
try {
    ...
} catch(ZeroDivideException e) {
    ...
} catch(InvalidArgException e) {
    ...
} catch(MathException e) {
    ...
}
```


8 Beispiel

```
class TestException extends Exception {
  public TestException(String s) {super(s);}
}


public class Test {

  public void hello() throws TestException {
    if (...) throw new TestException("...an error
description...");
  }

  public void testIt() {
    try {
      hello();
      ...
    }
    catch (TestException t) {
      System.out.println("Exception raised:" + t.getMessage());
    }
    finally {
      // clean up
    }
  }
```

9 Ausnahmen und Vererbung: Throwing

- Können überschriebene Methoden andere Exceptions werfen, als die ursprüngliche Methode?
- Grundsatz:
  - Unterklassen können überall dort verwendet werden wo die Oberklasse erwartet wird.
  - Unterklassen sind “besser” als Oberklassen.
- das bedeutet:
  - Unterklassen dürfen keine zusätzlichen Exception werfen.
  - Unterklassen können Unterklassen von den deklarierten Ausnahmen der Oberklasse werfen.
  - Unterklassen dürfen keine Oberklassen von den ursprünglich deklarierten Ausnahmen werfen.

9 Beispiel

```
class E1 extends Exception {}
class E2 extends Exception {}
class E3 extends E2 {}
class A {
    void m() throws E2 {}
}
class B extends A {
    void m() throws ??? {}
}
```

- ??? =

9 Beispiel (2)

```
class E1 extends Exception {}
class E2 extends Exception {}
class E3 extends E2 {}
class A {
    void m() throws E2 {}
}
class B extends A {
    void m() throws ??? {}
}
```

- ??? =

Falsch sind:	Richtig sind:
`E1`	`E2`
`Exception`	`E3`
`...`	keine

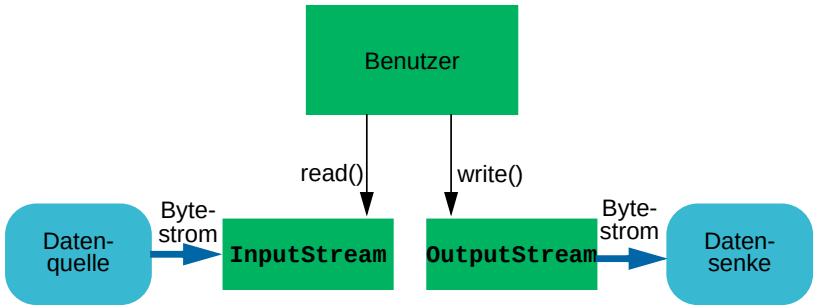
10 Zusammenfassung

- werfen von Ausnahmen: `throw new MyException("...");`
- Programm-Block für den die Ausnahmebehandlung gilt: `try { ... }`
- Ausnahmebehandlung:
  ```
  try {
  ... throw new MyException("..."); ...
  } catch(MyException e) { ... }
  ```
- Zusätzlich: **`finally`**-Block
- Ausnahmen müssen von **`Throwable`** abgeleitet sein.
- Ausnahmen von Anwendungen sollten von **`Exception`** abgeleitet werden.
- Ausnahmen müssen bei der Methodendeklaration angegeben werden: `void mymethod() throws ...`

L.2 Das Java Ein-/Ausgabesystem

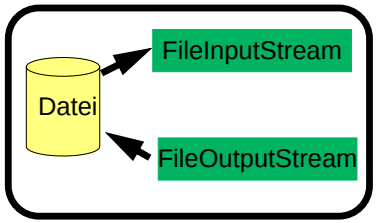
- Grundlegendes Konzept: Ströme (Streams)
  - Byteströme (InputStream/OutputStream)
  - Zeichenströme (Reader/Writer)

1 Byteströme



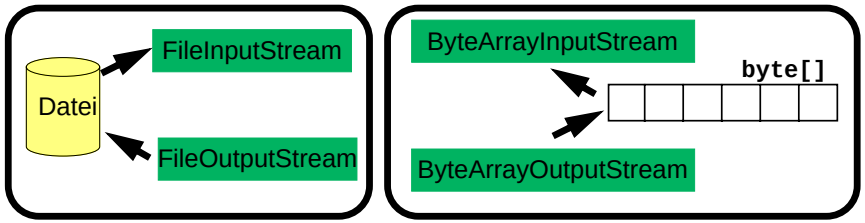
2 Spezialisierungen von Strömen

- Wo kommen die Daten her, wo gehen sie hin?



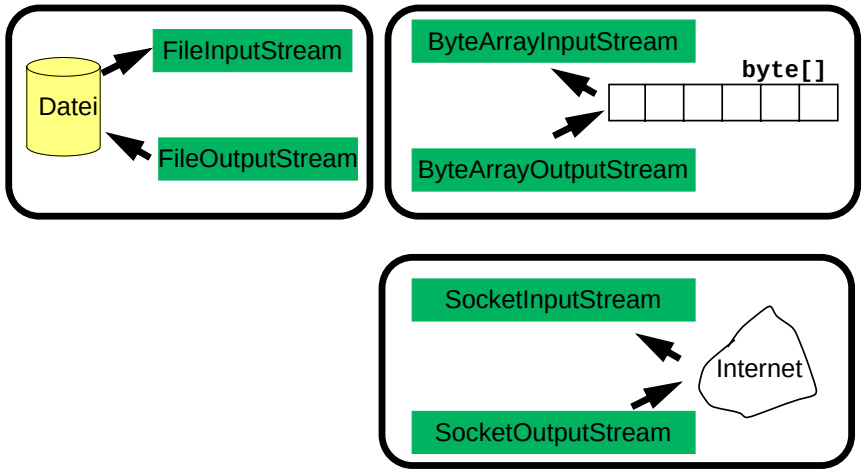
2 Spezialisierungen von Strömen (2)

- Wo kommen die Daten her, wo gehen sie hin?

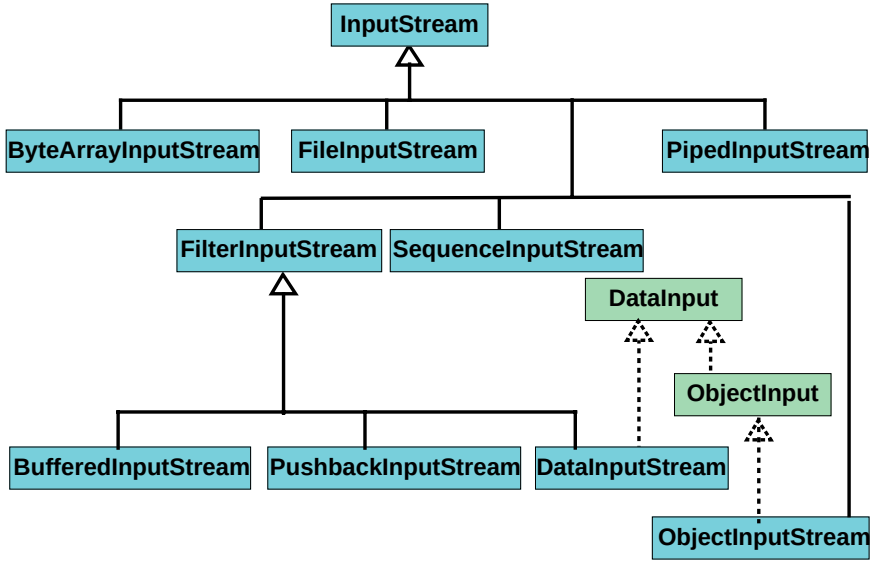


2 Spezialisierungen von Strömen (3)

- Wo kommen die Daten her, wo gehen sie hin?

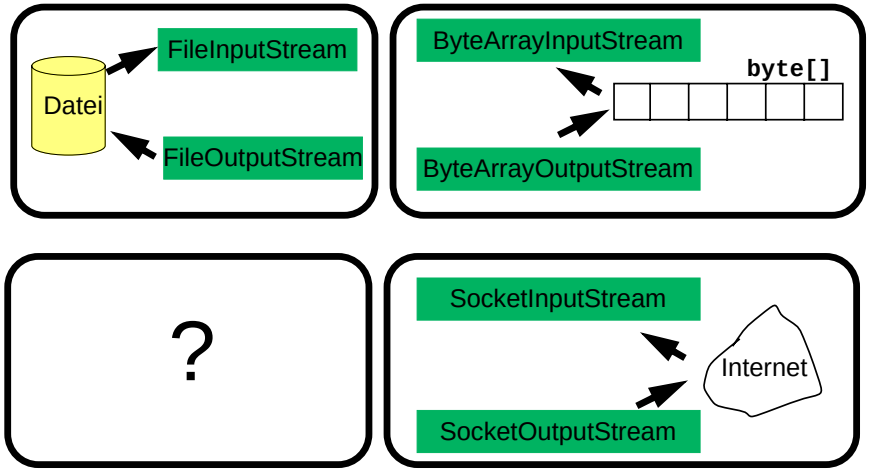


3 Klassendiagram der Eingabeströme



2 Spezialisierungen von Strömen (4)

- Wo kommen die Daten her, wo gehen sie hin?



4 FileInputStream, FileOutputStream

- Aus einer Datei lesen:

```
import java.io.*;

public class InTest {
  public static void main (String argv[]) throws IOException {
    FileInputStream f = new FileInputStream ("/tmp/test");
    byte buf[] = new byte[4];
    f.read(buf);
  }
}
```

4 FileInputStream, FileOutputStream (2)

- In eine Datei schreiben:

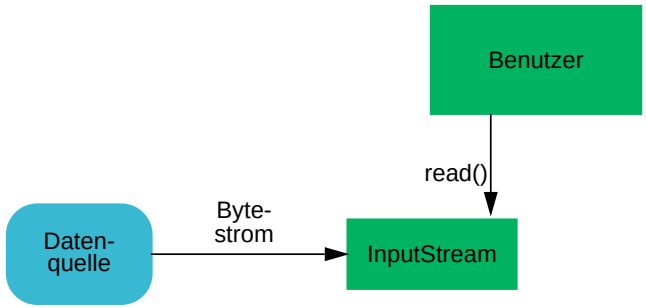
```
import java.io.*;

public class OutTest {
  public static void main (String argv[]) throws IOException {
    FileOutputStream f = new FileOutputStream ("/tmp/test");
    byte buf[] = new byte[4];
    for (byte i=0; i < buf.length; i++) buf[i]=i;
    f.write(buf);
  }
}
```

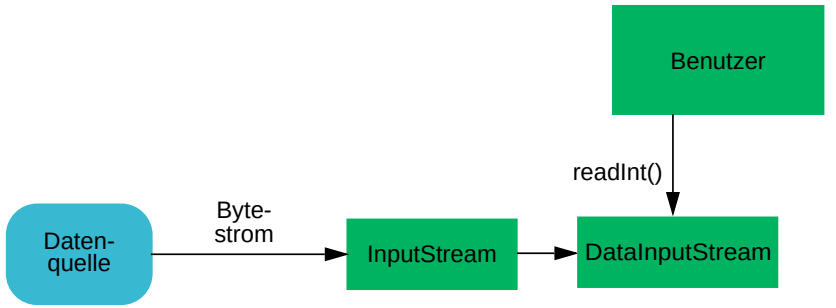

5 Kombinieren von Strömen

- Aus einfachen Strömen “komfortable” Ströme generieren
- Der komfortable Strom umhüllt den einfachen Strom
- ➜ Decorator Design-Pattern

5 Kombinieren von Strömen (2)



5 Kombinieren von Strömen (3)



6 DataInputStream

- **`InputStream`** ist relativ unkomfortabel
- **`DataInputStream`** wird verwendet um eine *binäre Darstellung* der Daten zu lesen (int, float,...)
- Ein **`DataInputStream`** kann aus jedem **`InputStream`** erzeugt werden:

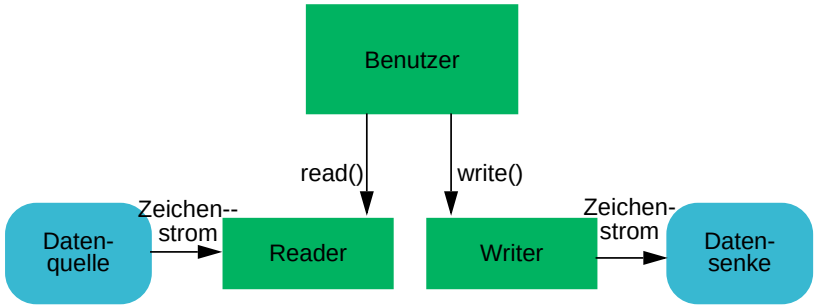
```
InputStream in = new FileInputStream ("/tmp/test");
DataInputStream dataIn = new DataInputStream(in);
float f = dataIn.readFloat();
```

- **`readLine()`** kann verwendet werden um ganze Zeilen zu lesen:

```
for(;;) {
    String s = dataIn.readLine();
    System.out.println(s);
}
```


7 Reader/Writer

- Zeichenströme zur Ein- und Ausgabe (**`Reader, Writer`**)



- Zeichenströme enthalten Unicodezeichen (16 bit)

8 Reader

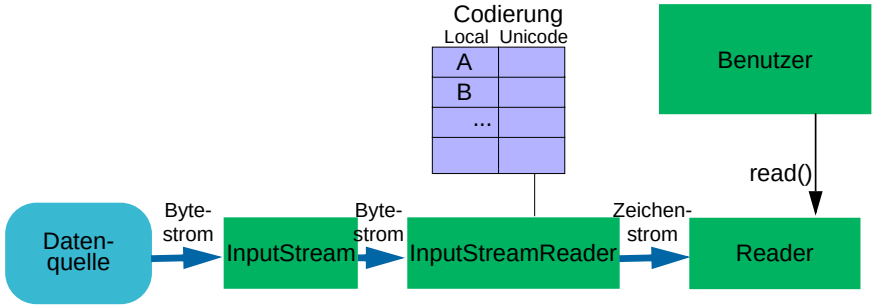
- wichtige Methoden:
  - **`int read()`**
    liest ein Zeichen und gibt es als **`int`** zurück
  - **`int read(char buf[])`**
    liest Zeichen in ein Array. Liefert die Anzahl der gelesenen Zeichen zurück oder -1 falls ein Fehler aufgetreten ist
  - **`int read(char buf[], int offset, int len)`**
    liest **`len`** Zeichen in den Puffer **`buf`**, beginnend ab **`offset`**
  - **`long skip(long n)`**
    überspringt **`n`** Zeichen
  - **`void close()`**
    schließt den Strom

8 FileReader

- Wird verwendet um aus einer Datei zu lesen
- Konstruktoren:
  - **`FileReader(String fileName)`**
  - **`FileReader(File file)`**
  - **`FileReader(FileDescriptor fd)`**
- Keine weitere Methoden (nur die von **`InputStreamReader`** geerbten)
- Was ist ein **`InputStreamReader`**?

9 Byte- und Zeichenströme

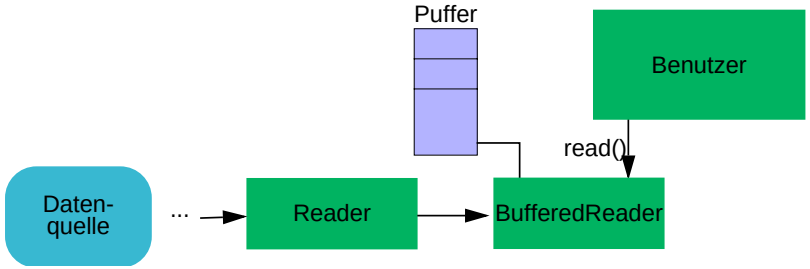
- Umwandeln von Byteströmen in Zeichenströme mit Hilfe einer Codierung



- einige Codierungen: "Basic Latin", "Greek", "Arabic", "Gurmukhi"

10 Gepufferte Ein-/Ausgabe

- Lesen/Schreiben von einzelnen Zeichen kann teuer sein.
- Umrechnung der Zeichenkodierung kann teuer sein.
- Falls möglich **`BufferedReader`**, **`BufferedWriter`** verwenden.
- **`BufferedReader`** kann aus jedem anderen Reader erzeugt werden.
- Wichtige Methoden von **`BufferedWriter`**: **`void flush()`**:
  Leert den Puffer - schreibt den Puffer zum ungepufferten Writer:



10 Gepufferte Ein-/Ausgabe (2)

- **`BufferedReader`** kann ganze Zeilen lesen: **`String readLine()`**

```
BufferedReader in = new BufferedReader(new FileReader("test.txt"));
String line = in.readLine();


BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String line = in.readLine();
```


11 PrintWriter

- Kann von jedem **`OutputStream`** oder **`Writer`** erzeugt werden.
- **`println(String s)`**: schreibt den String und das/die EOL Zeichen.
- Beispiel: Datei einlesen und auf der Standardausgabe ausgeben:

```
import java.io.*;

public class CopyStream {
    public static void main(String a[]) throws Exception {
        BufferedReader in = new BufferedReader(
                            new FileReader("test.txt"));
        PrintWriter out = new PrintWriter(System.out);
        for(String line; (line = in.readLine())!=null;) {
            out.println(line);
        }
        out.close();
    }
}
```

12 FileWriter

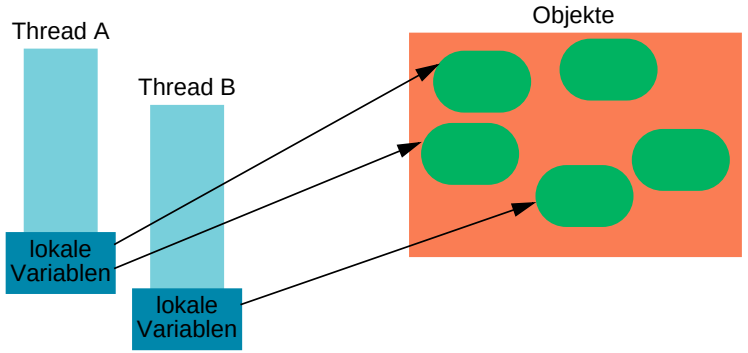
- `FileWriter` wird verwendet um Zeichen in eine Datei zu schreiben.
- Nachdem das Schreiben beendet ist sollte `close()` aufgerufen werden!

L.3 Threads

- Referenz:
  - D. Lea. *Concurrent Programming in Java - Design Principles and Patterns*. The Java Series. Addison-Wesley 1997.

1 Was ist ein Thread?

- Aktivitätsträger mit:
  - eigenem Instruktionszähler
  - eigenen Registern
  - eigenem Stack
- Alle Threads laufen im gleichen Adressbereich



2 Vorteile / Nachteile

- Vorteile:
  - ausführen paralleler Algorithmen auf einem Multiprozessorrechner
  - durch das Warten auf langsame Geräte (z.B. Netzwerk, Benutzer) wird nicht das gesamte Programm blockiert
- Nachteile:
  - komplexe Semantik
  - Fehlersuche sehr schwierig
  - John Ousterhout: *Why Threads Are A Bad Idea (for most purposes)*.

3 Thread Erzeugung (1)

1. Eine Unterklasse von **`java.lang.Thread`** erstellen.
2. Dabei die **`run()`**-Methode überschreiben.
3. Eine Instanz der Klasse erzeugen.
4. An dieser Instanz die Methode **`start()`** aufrufen.

- Beispiel:

```
class Test extends Thread {
    public void run() {
        System.out.println("Test");
    }
}

Test test = new Test();
test.start();
```


3 Thread Erzeugung (2)

1. Das Interface **`java.lang.Runnable`** implementieren.
   Dabei muss eine **`run()`**-Methode implementiert werden.
2. Ein Objekt instantiieren, welches das Interface **`Runnable`** implementiert.
3. Eine neue Instanz von **`Thread`** erzeugen, dem Konstruktor dabei das **`Runnable`**-Objekt mitgeben.
4. Am neuen Thread-Objekt die **`start()`**-Methode aufrufen.

- Beispiel:

```
class Test implements Runnable {
    public void run() {
        System.out.println("Test");
    }
}

Test test = new Test();
Thread thread = new Thread(test);
thread.start();
```


4 Die Methode `sleep`

- Ein Thread hat die Methode **`sleep(long n)`** um für **`n`** Millisekunden zu “schlafen”.
- Der Thread kann jedoch verdrängt worden sein nachdem er aus dem **`sleep()`** zurückkehrt.

5 Die Methode `join`

- Ein Thread kann auf die Beendigung eines anderen Threads warten:

```
workerThread = new Thread(worker);
...
workerThread.join();
worker.result();
```

6 Multithreading Probleme

```
public class Test implements Runnable {
  public int a=0;
  public void run() {
    for(int i=0; i<100000; i++) {
      a = a + 1;
    }
  }

  public static void main(String[] args) {
    Test value = new Test();
    Thread t1 = new Thread(value);
    Thread t2 = new Thread(value);
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch(Exception e) {
      System.out.println("Exception");
    }
    System.out.println("Expected a=200000 but a="+value.a);
  }
}
```

Was ist das Ergebnis dieses Programmes?

zwei Threads erzeugen, mit demselben `Runnable` Objekt

beide Threads starten

auf Beendigung der beiden Threads warten

6 Multithreading Probleme (2)

- Ergebnis einiger Durchläufe: 173274, 137807, 150683
- Was passiert, wenn **`a = a + 1`** ausgeführt wird?

```
LOAD a into Register
ADD 1 to Register
STORE Register into a
```

- mögliche Verzahnung wenn zwei Threads beteiligt sind (initial a=0):
  - T1-load: a=0, Reg1=0
  - T2-load: a=0, Reg2=0
  - T1-add: a=0, Reg1=1
  - T1-store: a=1, Reg1=1
  - T2-add: a=1, Reg2=1
  - T2-store: a=1, Reg2=1
- Die drei Operationen müssen *atomar* ausgeführt werden!

7 Das Schlüsselwort `synchronized`

- Jedes Objekt kann als Sperre verwendet werden.
- Um eine Sperren anzufordern und freizugeben wird ein **`synchronized`** Konstrukt verwendet.
- Methoden oder Blöcke können als **`synchronized`** deklariert werden:

```
class Test {
    public synchronized void m() { ... }
    public void n() { ...
              synchronized(this) {
                  ...
              }
    }
}
```

- ein Thread kann eine Sperre mehrfach halten (rekursive Sperre)
- verbessertes Beispiel: **`synchronized(this) { a = a + 1; }`**

8 Wann soll `synchronized` verwendet werden?

- **`synchronized`** ist nicht notwendig:
  - wenn Code immer nur von einem Thread ausgeführt wird (single-threaded context)
  - für einfache get-Methoden (siehe Ausnahmen unten)
- **`synchronized`** sollte verwendet werden:
  - wenn Daten geschrieben wird
  - wenn mit dem Objekt Berechnungen durchgeführt werden (auch wenn der Zustand *nur gelesen* wird)
  - für get-Methoden, die **`long`** oder **`double`** Typen zurückliefern
  - für einfache get-Methoden, die blockieren sollen wenn eine Zustandsveränderung durchgeführt wird

9 Synchronisationsvariablen (Condition Variables)

- Thread muss warten bis eine Bedingung wahr wird.
- zwei Möglichkeiten:
  - aktiv (polling)
  - passive (condition variables)
- Jedes Objekt kann als Synchronisationsvariable verwendet werden.
- Die Klasse **`Object`** enthält Methoden um ein Objekt als Synchronisationsvariable zu verwenden.
  - **`wait`**: auf ein Ereignis warten

```
while(! condition) { wait(); }
```

  - **`notify`**: Zustand wurde verändert, die Bedingung könnte wahr sein, einen anderen Thread benachrichtigen
  - **`notifyAll`**: alle wartenden Threads aufwecken (teuer)

10 Warten und Sperren

- **`wait`** kann nur ausgeführt werden, wenn der aufrufende Thread eine Sperre an dem Objekt hält.
- **`wait`** gibt die Sperre frei bevor der Thread blockiert wird (atomar)
- beim Deblockieren wird die Sperre wieder atomar angefordert

11 Condition Variables - Beispiel

- PV-System: Bedingung: *count > 0*

```
class Semaphore {
    private int count;
    public Semaphore(int count) { this.count = count; }
    public synchronized void P() throws InterruptedException{
        while (count <= 0) {
            wait();
        }
        count--;
    }
    public synchronized void V() {
        count++;
        notify();
    }
}
```


11 Condition Variables - Beispiel (2)

- Bestellsystem: ein Thread akzeptiert Kundenabfragen (**`SecretaryThread`**) ein anderer Thread bearbeitet sie (**`WorkerThread`**)
- Secretary:

```
class SecretaryThread implements Runnable {
   public void run() {
      for(;;) {
         Customer customer = customerLine.nextCustomer();
         WorkerThread worker = classify(customer);
         worker.insertCustomer(customer);
      }
   }
}

interface WorkerThread {
   public void insertCustomer(Customer c);
}
```

11 Condition Variables - Beispiel (3)

- Worker:

```
class SpecificWorker implements Runnable, WorkerThread {
   public void run() {
      for(;;) {
         while(queue.empty())  // race condition!!
            synchronized (this) { wait(); }
         Customer customer = queue.next();
         // do something nice with customer
         // ...
      }
   }
   public void insertCustomer(Customer c) {
      queue.insert(c);
      this.notify();
   }
}
```


M Hinweise zur 3. Aufgabe: StringTokenizer

- Schneidet Strings in Tokens
- Definiert in: **`java.util`**
- Beispiel:

```
String str = "Hello this is a test"
StringTokenizer tokenizer = new StringTokenizer(str);

// Token einlesen bis zum Ende des Strings
while(tokenizer.hasMoreTokens()) {
        System.out.println(tokenizer.nextToken());
}
```