Lösung zu Aufgabe 1: Schleifen (8 Punkte)

Folgendes einfache Java-Programm verwendet zunächst die `while`-Schleife zur Ausgabe aller beim Programmaufruf mitgegebenen Programmparameter, dann die do-Schleife und anschließend beide Varianten der `for`-Schleife.
Bei der do-Schleife muss darauf geachtet werden, dass die Ausführung des Schleifenrumpfes verhindert wird, wenn überhaupt keine Argumente beim Programmaufruf mitgegeben werden. Dies wird durch Voranstellen der `if`-Anweisung erreicht.

```
public class Schleifen {
  public static void main(String[] args) {
    int i;
    //Mit while-Schleife

    i = 0;
    while (i < args.length) {
      System.out.println(args[i]);
      i = i + 1;
    }


    //Mit do-Schleife

    i = 0;
    if (args.length > 0)
      do {
        System.out.println(args[i]);
        i = i + 1;
      } while (i < args.length);


    // Mit erster Variante der for-Schleife

    for (i = 0; i < args.length; i = i + 1)
      System.out.println(args[i]);


    // Mit zweiter Variante der for-Schleife

    for (String arg : args)
      System.out.println(arg);
  }
}
```

Lösung zu Aufgabe 2: Sichtbarkeit (11 Punkte)

Gegeben ist der folgende Programmausschnitt:

```
package a;
public class A {
   public int i;
   int j;
   protected int k;
   private int l;
   public int m() {...}
   int n() {...}
   protected int o() {...}
   private int p() {...}
}

package a;
public class B {...}

package a.a;
import a.A;
public class C {...}

package a.a;
import a.A;
public class D extends A {...}
```

a) Auf einer per `A a` deklarierten Variable sind abhängig davon, innerhalb welcher Klasse die Deklaration steht, die folgenden Elemente zugreifbar:

- a ist in Klasse A selbst deklariert:
  alle
- a ist in Klasse B deklariert:
  a.i, a.j, a.m(), a.n(), auch a.k und a.o(), da `protected default access` einschließt.
- a ist in Klasse C deklariert:
  a.i und a.m()
- a ist in Klasse D deklariert:
  a.i, a.m()

  Dieses Ergebnis ist nicht selbstverständlich, daher hier die ausführliche Erläuterung:
  Die Sichtbarkeitsstufe 'protected' schließt die Sichtbarkeitsstufe 'package' ein, also die, welche immer dann besteht, wenn es keinen Sichtbarkeitsmodifikator gibt.
  Das heißt also, daß ein Element einer Klasse A mit dem Modifikator 'protected' zunächst einmal überall dort sichtbar ist, wo es auch ohne Modifikator sichtbar wäre, also in A und in anderen Klassen, die sich im selben Package wie A befinden.
  Zusätzlich sind solche Elemente aber unter bestimmten Bedingungen auch aus Subklassen von A heraus zugreifbar, egal, in welchem Package diese liegen:

'A protected member or constructor of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object.'

Für die Aufgabe heißt das:

Wenn in der Klasse D eine Variable a vom Typ A deklariert ist, dann sind über a die als 'protected' deklarierten Elemente von A *nicht* zugreifbar, denn die Klasse D - die Klasse, in der a deklariert ist - ist nicht an der Implementierung von A - der Klasse, die dem Deklarationstyp von a entspricht - beteiligt.

Hat die in D deklarierte Variable a hingegen den Deklarationstyp D, so sind über a auch die als 'protected' deklarierten Elemente von A zugreifbar, denn die Klasse D - die Klasse, in der a deklariert ist - ist natürlich an der Implementierung von D - der Klasse, die dem Deklarationstyp von a entspricht - beteiligt.

In der Realität hat man es aber meistens mit einem viel einfacheren Fall zu tun: Man will aus einer Subklasse heraus *direkt* auf als 'protected' deklarierte Elemente der Superklasse zugreifen, also etwa so:

```
class D extends A {
    void aMethod() {
        k = 42;  // Zugriff auf das ererbte Attribut k
        System.out.println(o());  // Zugriff auf die ererbte Methode o()
    }
}
```

Wenn man sich verdeutlicht, daß die beiden Zugriffe in Wirklichkeit sehr wohl über eine Objektreferenz erfolgen, nämlich über 'this', und daß 'this' hier natürlich den Deklarationstyp D hat, dann wird klar, daß beide Zugriffe erlaubt sind.

b) Wenn man in der Klasse D eine Variable vom Typ `D a` deklariert, kann man auf a.i, a.k, a.m() und a.o()
zugreifen.

Lösung zu Aufgabe 3: Zuweisungen (10 Punkte)

a) Unveränderliche blocklokale Variablen und Attribute müssen entweder an der Deklarationsstelle oder in den zugehörigen Blöcken, Methoden bzw. Konstruktoren initialisiert werden.
Nach der Initialisierung darf keine weitere Zuweisung mehr stattfinden.

```
class C {
  final int xwert, ywert = 9;
  C(){
    xwert = 7;  // KORREKT: Initialisierung im Konstruktor
                // und nicht an Deklarationsstelle
    mult();
  }
  void mult(){
    final int zaehler1 = 10, zaehler2 = -10;
    zaehler2 = 0;  // UNZULAESSIG: zaehler2 ist unveraenderlich
    ywert = 27; // UNZULAESSIG: ywert ist unveraenderlich
    ...
  }
}
```

b)
```
Object o = new String[6]; // ZULAESSIG //
/* Begründung:
   Jeder Feldtyp ist ein Subtyp von Object.
   Eine Variable vom Typ Object kann also sowohl (Referenzen auf)
   Objekte beliebiger Klassentypen speichern, als auch (Referenzen
   auf) Felder. */

Tier[] tier  = new Eisbaer[6];/* ZULAESSIG //
/* Begründung:
Ein Feldtyp  S[ ] ist genau dann ein Subtyp eines anderen Feldtyps T[ ],
wenn S ein Subtyp von T ist.\\
In diesem Fall ist Eisbaer Subtyp von Tier. */

Eisbaer[] baer  = new Tier[6];  // UNZULAESSIG //
/* Begründung:
Tier ist kein Subtyp von Eisbaer, daher ist diese Zuweisung
unzulässig. */
```

Lösung zu Aufgabe 4: objektorientierte Denkweise (10 Punkte)

a) Das Attribut `motorLaeuft` darf nicht static sein, sonst wäre es ein gemeinsames Attribut **aller** Motorräder, d.h. die Motorräder wären alle gleichzeitig entweder an oder aus, was man natürlich nicht haben will. Und damit darf auch die Methode `start()` nicht static sein, denn eine statische Methode wird **nicht auf einem Exemplar** aufgerufen (hat keinen impliziten Parameter), sie kennt überhaupt keine Exemplare.

b)
```
public class Motorrad {

    private boolean motorLaeuft;

    public void start() {
       motorLaeuft = true;
}

public static void main(String[] args) {
    Motorrad m = new Motorrad();
    m.start();
}
}
```

Erläuterung:
Einer der Grundgedanken objektorientierter Programmierung ist es, Daten und die auf ihnen möglichen Operationen in einer Struktur zusammenzufassen, einem Objekt. Man kann dem Objekt Nachrichten senden (Methoden aufrufen) und dann tut es etwas, typischerweise verändert es seinen Zustand, welcher durch seine Attribute repräsentiert wird.
Wenn jemand auf den Starterknopf seines Motorrads drückt, sendet er durch den Knopfdruck eine Nachricht an ein ganz bestimmtes Motorrad und dieses eine Motorrad reagiert darauf. Dem entspricht in einer ganz primitiven Simulation der Aufruf einer Methode

```
start()
```

auf einem Motorrad-Exemplar, welches daraufhin ein boolsches Attribut `motorLaeuft` von false auf true setzt.
Kurz gesagt:

Wenn man ein Motorrad starten will, braucht man auch ein Motorrad, also ein **Exemplar**. Und damit sieht der Code für das Motorrad sinnvollerweise etwa so wie oben vorgestellt aus.

Lösung zu Aufgabe 5: Interfaces, Streams, generische Typen (8 Punkte)

```
class WillNurLesen {

   /* Definieren Sie eine Variable eingabe zur Eingabe von
   Strings */

   ReadStream<String> eingabe = new IOStream<String>();

   String lese(){
      /* Befehl zum Lesen */

     return eingabe.read();

   }
}

 class WillNurSchreiben {

    /* Definieren Sie eine Variable ausgabe zur Ausgabe von
    Strings */

    WriteStream<String> ausgabe = new IOStream<String>();

    void schreibe(String st){
       /* Befehl für die Ausgabe */
        /* bitte hier Text ergänzen */

      ausgabe.write(st);

    }
 }
```

Lösungen zu Aufgabe 6: Objektströme (10 Punkte)

a) Um bei der Ausgabe mehrfach referenzierte Objekte zu erkennen und Zirkularitäten auflösen zu können, vergeben Objektströme für jedes Objekt eine Nummer und verwalten die Zuordung der ausgegebenen Objekte zu deren Nummern in einer Tabelle. Steht beim Durchlauf durch das Objektgeflecht ein weiteres Objekt *obj* zur Ausgabe an, wird zunächst anhand der Tabelle geprüft, ob *obj* bereits ausgegeben wurde. Ist das der Fall, wird nur die zugehörige Objektnummer herausgeschrieben. Andernfalls erhält *obj* eine Nummer, wird in die Tabelle eingetragen und dann mit seinen Attributwerten in den Ausgabestrom geschrieben.
Auf diese Weise werden alle im Geflecht erreichbaren Objekte behandelt und *der Reihe nach* ausgegeben; man spricht deshalb auch vom *Serialisieren* von Objektgeflechten.

b) Java sieht einen Mechanismus vor, mit dem deklariert werden muss, ob eine Klasse serialisierbar sein soll.
Eine Klasse ist genau dann serialisierbar, wenn sie ein Subtyp des Schnittstellentyps
`java.io.Serializable`
ist.
D.h. serialisierbare Klassen müssen entweder `Serializable` „implementieren“ oder einen Supertyp besitzen, der bereits Subtyp von `Serializable` ist; die Eigenschaft, serialisierbar zu sein, wird also vom Supertyp an den Subtyp weitergegeben.
Da der Schnittstellentyp `Serializable` weder Attribute noch Methoden besitzt, braucht eine Klasse, die `Serializable` implementieren soll, diesen Typ nur zusätzlich in die implements-Liste aufzunehmen.
Bei dem Versuch, ein Objekt einer nicht serialisierbaren Klasse auszugeben, wird eine `NotSerializableException` ausgelöst.

Aufgabe 7: Überschreiben und Überladen (12 Punkte)

Gegeben ist das folgende Programm. Welche Ausgabe liefert es und warum?

```
public class TierTest {
   public static void main(String[] args) {
      Tier t1   = new Tier();
      Tier t2   = new Vogel();
      Fisch f   = new Karpfen();
      Vogel v1  = new Vogel();
      Vogel v2  = new Spatz();
      Huhn h    = new Huhn();
      Karpfen k = new Karpfen();

      Super sup1 = new Super();
      Super sup2 = new Sub();

      sup1.m(h, v2);
      sup2.m(v1, k);
      sup1.m(t1, t2);
      sup1.m(v1, k);
      sup1.m(v2, f);
      sup2.m(v1, f);
   }
}

class Super {
   public void m(Tier t1, Tier t2) {
      System.out.println("1");
   }

   public void m(Tier t, Fisch f) {
      System.out.println("2");
   }

   public void m(Fisch f, Tier t) {
       System.out.println("5");
   }
}

class Sub extends Super {
  public void m(Tier t1, Fisch t2) {
     System.out.println("3");
  }

  public void m(Vogel v, Fisch f) {
      System.out.println("4");
  }
}

class Tier {
}
```

```
class Fisch extends Tier {
}

class Vogel extends Tier {
}

class Huhn extends Vogel {
}

class Spatz extends Vogel {
}

class Karpfen extends Fisch {
}
```

Lösung zu Aufgabe 7: **(12 Punkte)**

`sup1.m(h, v2);` hier wird 1 ausgegeben, denn:
h und v2 sind zur Compilezeit als Huhn und Vogel typisiert, der Methodenaufruf wird an die speziellste (passendste) Methode gebunden. Hier passt nur die Methode mit Ausgabe 1.

`sup2.m(v1, k);` hier wird 3 ausgegeben, denn:
sup2 hat zur Compilezeit den Typ Super, bekannt sind nur die Signaturen der Methoden mit Ausgabe 1, 2 und 5, deswegen wird insbesondere nicht 4 ausgegeben.
Zur Laufzeit wird dann die Methode mit Ausgabe 2 durch die mit Ausgabe 3 überladen.

`sup1.m(t1, t2);` hier wird 1 ausgegeben, denn:
t1 und t2 sind zur Compilezeit als Tier typisiert. Der Aufruf wird also an die Methode mit Ausgabe 1 gebunden.

`sup1.m(v1, k);` hier wird 2 ausgegeben, denn:
v1 und k sind zur Compilezeit als Vogel und Karpfen typisiert, der Aufruf wird also an die speziellste Methode gebunden: Hier ist die Methode mit Ausgabe 2 spezieller als die mit Ausgabe 1.

`sup1.m(v2, f);` hier wird 2 ausgegeben, denn:
v2 und f sind zur Compilezeit als Vogel und Fisch typisiert, der Aufruf wird also an die speziellste Methode gebunden: Hier ist die Methode mit Ausgabe 2 spezieller als die mit Ausgabe 1.

`sup2.m(v1, f);` hier wird 3 ausgegeben, denn:
sup2 hat zur Compilezeit den Typ Super, bekannt sind nur die Signaturen der Methoden mit Ausgabe 1, 2 und 5, deswegen wird insbesondere nicht 4 ausgegeben.
Zur Laufzeit wird dann die Methode mit Ausgabe 2 durch die mit Ausgabe 3 überladen.

Man beachte:
Die Deklarationstypen der aktuellen Parameter müssen Subtypen der Typen der formalen

Parameter sein. Dann und nur dann kommt eine Methode überhaupt in Betracht.
Gibt es keine Methode, deren Deklaration dieser Bedingung entspricht, kommt es zu einem Compilerfehler.
Gibt es nur eine, wird ein Aufruf dieser Methode in den Bytecode geschrieben.
Gibt es mehrere, muss die *passendste* gewählt werden.
Gibt es eine, die passender ist als die anderen in Betracht kommenden, wird ein Aufruf dieser Methode in den Bytecode geschrieben.
Gibt es mehrere gleichermaßen passende, kommt es zu einem Compilerfehler ('ambigious').

Nun haben wir Bytecode. Der Methodenaufruf aus dem Quellcode wurde an einen Methodenaufruf im Bytecode 'gebunden'. Die Auflösung der Überladung erfolgte durch den Compiler, also statisch und damit zwangsläufig nur anhand der Deklarationstypen der Beteiligten, denn nur diese Typen kennt der Compiler.

Zur Laufzeit geht es jetzt aber noch mal weiter. Nun spielt der tatsächliche Typ des Objekts eine Rolle, auf dem die Methode aufgerufen wurde (also nicht der Deklarationstyp der Variablen, auf der aufgerufen wurde) und falls die im Bytecode stehende Methode in diesem Typ überschrieben wurde, wird die überschreibende Methode ausgeführt (dynamische Methodenwahl).

Lösung zu Aufgabe 8: abstrakte Klassen (10 Punkte)

- Was gilt für die Implementierung abstrakter Klassen?

  `Antwort:` Die Implementierung abstrakter Klassen muss nicht vollständig sein; d.h. in abstrakten Klassen dürfen Methoden ohne Rumpf deklariert werden. Derartige Methoden werden abstrakte Methoden genannt.

- Was können abstrakte Klassen an Subklassen vererben?

  `Antwort:` deklarierte Attribute und Methoden (auch wenn sie abstrakt sind) sowie vorhandene Methodenimplementierungen

- Lassen sich abstrakte Klassen instanziieren?

  `Antwort:` Es nicht möglich, Instanzen von abstrakten Klassen zu erzeugen.

- Wozu können abstrakte Klassen verwendet werden?

  `Antwort:`Abstrakte Klassen können verwendet werden, um gemeinsame Implementierungsteile zukünftiger Subtypen an einer Stelle zusammenzufassen und damit die Programme kleiner und pflegeleichter zu machen.

- Java verzichtet auf Mehrfachvererbung. Was macht man in Java, wenn eine Klasse mehrere Supertypen besitzen soll?

  `Antwort:` Besitzt eine Klasse mehrere Supertypen, muss der Programmierer entscheiden, welcher Supertyp als Klasse implementiert werden soll und welche Supertypen als Schnittstelle zu realisieren sind. Der Programmierer kann dabei ausnutzen, daß eine Klasse in Java mehrere Interfaces implementieren kann.

Lösung zu Aufgabe 9: Parallelität (12 Punkte)

Am Lehrgebiet Programmiersysteme benutzen Daniela und Ursula denselben Drucker. Jede druckt eine Datei fünfmal auf dem Drucker aus. Die Namen der zu druckenden Dateien werden der `main`-Methode der Klasse `Test` als Parameter übergeben. Das einmalige Drucken einer Datei sei ein *Druckjob*. Jede Benutzerin initiiert also fünf Druckjobs.

```
import java.io.*;

class Drucker {
  void druckeDatei(String dateiname) {
    try {
      BufferedReader in = new BufferedReader(new FileReader(dateiname));
      String line = in.readLine();
      while (line != null) {
        // Zeile line auf dem Drucker ausgeben
        ...
        line = in.readLine();
      }
    }
    catch (Exception e) {
      System.out.println("Eine Ausnahme ist aufgetreten.");
    }
  }
}

class Benutzer extends Thread {
  Drucker drucker;
  String dateiname;
  int anzahl;

  Benutzer(Drucker drucker, String dateiname, int anzahl) {
    this.drucker = drucker;
    this.dateiname = dateiname;
    this.anzahl = anzahl;
  }

  public void run() {
    for (int i=0; i<anzahl; i++) {
      drucker.druckeDatei(dateiname);
    }
  }
}

class Test {
  public static void main (String[] argv) {
    if (argv.length >= 2) {
      Drucker d = new Drucker();
      Benutzer daniela  = new Benutzer(d, argv[0], 5);
      Benutzer ursula = new Benutzer(d, argv[1], 5);
      daniela.start();
      ursula.start();
    }
    else
      System.out.println("Bitte zwei Dateinamen als Argumente uebergeben!");
  }
}
```

Bei dem angegebenen Programm kann es passieren, dass ein Druckjob von Ursula mit einem Druckjob von Daniela vermischt auf dem Drucker ausgegeben wird. Synchronisieren Sie die beiden Threads für Daniela und Ursula nun so, dass zunächst alle Druckjobs der einen Benutzerin gedruckt werden und erst im Anschluss daran die Druckjobs der anderen Benutzerin. Begründen Sie, warum Ihr Vorschlag das Problem löst.

Erste Programmänderung:

```
synchronized void druckeDatei(String dateiname) {
```

Zweite Programmänderung: Die Schleife in der `run`-Methode wird durch den Monitor des Druckers überwacht. Dies kann durch einen synchronized-Block realisiert werden:

```
public void run() {
   synchronized(drucker) {
     for (int i = 0; i < anzahl; i++) {
       drucker.druckeDatei(dateiname);
     }
   }
 }
```

Begründung:
Durch die erste Programmänderung sperrt ein Thread den Monitor des `Drucker`-Objekts, bevor er mit dem Drucken eines seiner Druckjobs beginnt. Der Monitor bleibt während des gesamten Druckvorgangs für diesen Druckjob gesperrt. Der andere Thread muss aber zum Drucken seines Druckjobs ebenfalls den Monitor des `Drucker`-Objekts betreten. Deswegen kann er erst dann mit dem Drucken seines nächsten Druckjobs beginnen, wenn der Monitor wieder freigegeben ist, also nachdem der erste Thread seinen Druckjob vollständig gedruckt hat. Deshalb können sich Druckjobs nicht mehr vermischen.
Durch die zweite Programmänderung sichert sich der Benutzer exklusiven Zugang zum Drucker für alle seine Druckjobs. Denn während er die Schleife ausführt, in der die Druckjobs abgesetzt werden, ist der Monitor des Druckers gesperrt, so dass kein anderer Benutzer den Rumpf der Methode `druckeDatei` ausführen kann.
Im Rahmen des gegebenen Programms kann man auf die Synchronisierung der Methode `druckeDatei` verzichten, da sich die beiden Benutzerinnen Daniela und Ursula an das Protokoll halten, den Monitor des Druckers vor dem Initiieren ihrer Druckjobs zu sperren.Wenn es aber noch andere Benutzer geben könnte, die den Drucker benutzen, muss die Methode `druckeDatei` synchronisiert werden, da die anderen Benutzer dieses Protokoll verletzen könnten.

Lösung 10: Beobachter (12 Punkte)

zu a.)

```
public interface SchaltzustandListener extends EventListener {
  public void rot(SchaltzustandEvent e);
  public void rotgelb(SchaltzustandEvent e);
  public void gruen(SchaltzustandEvent e);
  public void gelb(SchaltzustandEvent e);
}
```

zu b.)

```
public class Ampelsteuerung {

  // Die Klasse Ampelsteuerung erzeugt ein Ampel- und ein Ampelzustandsobjekt,
  // platziert beide Objekte geeignet in einen Frame und
  // registriert das Ampelobjekt als Schaltzustands-Listener beim Ampelzustandsobjekt.

  public static void main(String[] args) {
    Ampelzustand ampelzustand = new Ampelzustand();
    Ampel ampel = new Ampel();

    // Ampel als Schaltzustands-Listener beim Ampelzustandsobjekt registrieren
    ampelzustand.addSchaltzustandListener(ampel);

    // Hauptfenster erzeugen, sowie Groesses, Position und
    // Layout festlegen.
    Frame f = new Frame();
    BorderLayout borderLayout1 = new BorderLayout();
    f.setSize (310,450);
    f.setLocation (100,100);
    f.setLayout(borderLayout1);

    // Ampel- und Ampelzustandsobjekt zum Frame hinzufuegen
    f.add(ampel, BorderLayout.CENTER);
    f.add(ampelzustand, BorderLayout.SOUTH);

    // Beobachter fuer Fensterereignisse einfuegen
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing (WindowEvent e) {
        // Programm beenden
        System.exit (0);
      }
    });
    // Hauptfenster sichtbar machen.
    f.setVisible (true);
  }
}
```