Listing 1: src/exceptions/UnreachableLocationException.java

```java
package exceptions;

public class UnreachableLocationException extends TransportException {

    public UnreachableLocationException(final String msg) {
        super(msg);
    }

    @Override
    public String toString() {
        return "Unreachable Location Exception: " + msg;
    }
}
```

Listing 2: src/exceptions/TransportException.java

```java
package exceptions;

public class TransportException extends Exception {

    protected String msg;

    public TransportException(final String msg) {
        super(msg);
        this.msg = msg;
    }

    @Override
    public String toString() {
        return "Transport Exception: " + msg;
    }
}
```

Listing 3: src/exceptions/InvalidCargoException.java

```java
package exceptions;

public class InvalidCargoException extends CargoException {

    public InvalidCargoException(final String msg) {
        super(msg);
    }

    @Override
    public String toString() {
        return "Invalid Cargo Exception: " + msg;
    }
}
```

Listing 4: src/exceptions/OverloadedException.java

```java
package exceptions;

public class OverloadedException extends CargoException {

    public OverloadedException(final String msg) {
        super(msg);
    }

    @Override
    public String toString() {
        return "Overloaded Exception: " + msg;
    }
}
```

Listing 5: src/exceptions/CargoException.java

```java
package exceptions;

public class CargoException extends TransportException {

    public CargoException(final String msg) {
        super(msg);
    }

    @Override
    public String toString() {
        return "Cargo Exception: " + msg;
    }
}
```

Listing 6: src/transport/CargoType.java

```java
package transport;

public enum CargoType {
    LIQUID, SOLID
}
```

Listing 7: src/transport/test/TransportTest.java

```java
package transport.test;

import static transport.Country.Austria;
import static transport.Country.France;
import static transport.Country.GreatBritain;
import static transport.Country.USA;

import exceptions.InvalidCargoException;
import exceptions.OverloadedException;
import exceptions.TransportException;
import inout.Out;
import transport.Cargo;
import transport.CargoType;
import transport.Location;
import transport.transporter.CargoPlane;
import transport.transporter.ContainerTruck;
import transport.transporter.TankTruck;
import transport.transporter.Transporter;

public class TransportTest {

  public static void main(String[] args) {

    Location linz = new Location("Linz", 0, 0, Austria);
    Location paris = new Location("Paris", 300, 400, France);
    Location la = new Location("LA", 12000, 1000, USA);
    Location london = new Location("London", 2000, -100, GreatBritain);
    Location ny = new Location("NY", 8000, 0, USA);

    Transporter plane = new CargoPlane("plane", 1000, 20000, london);
    Transporter tankTruck = new TankTruck("tank", 80, 10000, linz);
    Transporter containerTruck = new ContainerTruck("truck", 600, 8000, linz);

    Cargo solid_15 = new Cargo(CargoType.SOLID, "solid_15", 15000);
    Cargo solid_5 = new Cargo(CargoType.SOLID, "solid_5", 5000);
    Cargo liquid_15 = new Cargo(CargoType.LIQUID, "liquid_15", 15000);
    Cargo liquid_5 = new Cargo(CargoType.LIQUID, "liquid_5", 5000);

    double cost = 0;
```

```
40
        // plane

        try {
          cost += plane.goTo(paris);
45        Out.println("Plane flight to Paris ok: " + plane.toString());

          plane.load(solid_15);
          Out.println("Loaded solid_15 on plane ok: " + plane.toString());

50        cost += plane.goTo(ny);
          Out.println("Plane flight to NY ok: " + plane.toString());

          plane.unload();
          Out.println("Plane unload ok: " + plane.toString());
55      } catch (TransportException e) {
          Out.println("++ERROR++: Unexpected exception: " + e.toString());
        }

        Out.println("Cost for plane is: " + cost);
60      Out.println();

        try {
          plane.load(liquid_15);
          Out.println("++ERROR++: InvalidCargoException expected: " + plane);
65      } catch (InvalidCargoException e) {
          Out.println("Expected exception is: " + e.toString());
        } catch (TransportException e) {
          Out.println("++ERROR++: Unexpected exception: " + e.toString());
        }
70
        // Tank truck

        Out.println();

75      cost = 0;
        try {

          cost += tankTruck.goTo(paris);
          Out.println("Tank truck go to Paris ok: " + tankTruck.toString());
80
          tankTruck.load(liquid_5);
          Out.println("Tank truck loading liquid 5000 ok: " + tankTruck.toString());

          cost += tankTruck.goTo(linz);
85        Out.println("Tank truck go to Linz ok: " + tankTruck.toString());

          tankTruck.unload();
          Out.println("Tank truck unloading ok: " + tankTruck.toString());

90      } catch (TransportException e) {
          Out.println("++ERROR++: Unexpected exception: " + e.toString());
        }

        Out.println("Cost for tank truck is: " + cost);
95      Out.println();

        try {
          tankTruck.load(solid_5);
          Out.println("++ERROR++: InvalidCargoException expected: " + tankTruck);
100     } catch (InvalidCargoException e) {
          Out.println("Expected exception is: " + e.toString());
        } catch (TransportException e) {
          Out.println("++ERROR++: Unexpected exception: " + e.toString());
```

```
    }

    try {
      tankTruck.load(liquid_15);
      Out.println("++ERROR++: OverloadedException expected: " + tankTruck);
    } catch (OverloadedException e) {
      Out.println("Expected exception is: " + e.toString());
    } catch (TransportException e) {
      Out.println("++ERROR++: Unexpected exception: " + e.toString());
    }

    // Container truck

    Out.println();
    cost = 0;
    try {

      cost += containerTruck.goTo(paris);
      Out.println("Container truck go to Paris ok: " + containerTruck.toString());

      containerTruck.load(solid_5);
      Out.println("Container truck loading solid 5000 ok: " +
          containerTruck.toString());

      cost += containerTruck.goTo(linz);
      Out.println("Container truck go to Linz ok: " + containerTruck.toString());

      containerTruck.unload();
      Out.println("Container truck unloading ok: " + containerTruck.toString());

    } catch (TransportException e) {
      Out.println("++ERROR++: Unexpected exception: " + e.toString());
    }

    Out.println("Cost for container truck is: " + cost);
    Out.println();

    try {
      containerTruck.load(liquid_5);
      Out.println("++ERROR++: InvalidCargoException expected: " + containerTruck);
    } catch (InvalidCargoException e) {
      Out.println("Expected exception is: " + e.toString());
    } catch (TransportException e) {
      Out.println("++ERROR++: Unexpected exception: " + e.toString());
    }

    try {
      containerTruck.load(solid_15);
      Out.println("++ERROR++: OverloadedException expected: " + containerTruck);
    } catch (OverloadedException e) {
      Out.println("Expected exception is: " + e.toString());
    } catch (TransportException e) {
      Out.println("++ERROR++: Unexpected exception: " + e.toString());
    }

  }
}
```

Listing 8: src/transport/Cargo.java

```
package transport;

public class Cargo {
    private final CargoType type;
    private final int weight;
```

```
    private final String description;

    public Cargo(final CargoType type, final String description, final int weight) {
        this.type = type;
        this.weight = weight;
        this.description = description;
    }

    public CargoType getType() {
        return type;
    }

    public int getWeight() {
        return weight;
    }

    @Override
    public String toString() {
        return String.format("%1$s with a weight of %2$d and description %3s",
            this.type, this.weight,
                this.description);
    }

}
```

Listing 9: src/transport/Country.java

```
package transport;

/**
 * Enumeration for a set of countries which are serviced by the transport company.
 */
public enum Country {

  Austria, Germany, France, Italy, Belgium, GreatBritain, Canada, USA;

  /**
   * Tests if the two countries are connected overland.
   * @param a the first country
   * @param b the second country
   * @return true if the two countries are connected overland
   */
  public static boolean landConnected(Country a, Country b) {
    return europe(a) && europe(b) || america(a) && america(b) || a == GreatBritain &&
        b == GreatBritain;
  }

  /**
   * Tests if the country belongs to continental Europe (not to Britain)
   * @param e the country to be tested
   * @return true if the country belongs to continental Europe
   */
  private static boolean europe(Country e) {
    return e == Austria || e == Germany || e == France || e == Italy || e == Belgium;
  }

  /**
   * Tests if the country belongs to the America continent.
   * @param e the country to be tested
   * @return true if the country belongs to the America continent
   */
  private static boolean america(Country e) {
    return e == Canada || e == USA;
  }
```

```
}
```

Listing 10: src/transport/transporter/CargoPlane.java

```
package transport.transporter;

import exceptions.InvalidCargoException;
import exceptions.TransportException;
import transport.Cargo;
import transport.CargoType;
import transport.Location;

public class CargoPlane extends Transporter {
    protected static final double startingCosts = 100;
    protected static final double landingCosts = 100;

    public CargoPlane(final String description, final double costPerKM, final int
        maximumWeight,
            final Location currentLocation) {
        super(description, costPerKM, maximumWeight, currentLocation);

    }

    /**
     * Sets the new destination and calculates the cost based on distance and
     * costPerKm + static starting and landing costs
     *
     * @param destination
     * @return
     */
    @Override()
    public double goTo(final Location destination) {
        final double costForTransport = this.costPerKM *
            this.currentLocation.getDistance(destination) + startingCosts
                + landingCosts;
        this.currentLocation = destination;
        return costForTransport;
    }

    /**
     * Loads a cargo. Only allows one cargo with a maximum weight and it the type is
     * not liquid
     *
     * @param c
     * @throws Exception
     */
    @Override()
    public void load(final Cargo c) throws TransportException {
        if (c.getType() == CargoType.LIQUID) {
            throw new InvalidCargoException("Liquid Cargo not allowed on a cargo
                plane");
        }
        super.load(c);

    }

}
```

Listing 11: src/transport/transporter/TankTruck.java

```
package transport.transporter;

import exceptions.InvalidCargoException;
import exceptions.TransportException;
import exceptions.UnreachableLocationException;
```

```java
import transport.Cargo;
import transport.CargoType;
import transport.Location;

public class TankTruck extends Transporter {

    public TankTruck(final String description, final double costPerKM, final int
        maximumWeight,
            final Location currentLocation) {
        super(description, costPerKM, maximumWeight, currentLocation);

    }

    /**
     * Sets the new destination and calculates the cost based on distance and
     * costPerKm
     *
     * @param destination
     * @return
     * @throws UnreachableLocationException
     */
    @Override()
    public double goTo(final Location destination) throws UnreachableLocationException
        {
        if (!this.currentLocation.reachableOverland(destination)) {
            throw new UnreachableLocationException("Container truck cannot reach this
                location");
        }
        final double costForTransport = this.costPerKM *
            this.currentLocation.getDistance(destination);
        this.currentLocation = destination;
        return costForTransport;
    }

    /**
     * Loads a cargo. Only allows one cargo with a maximum weight and it the type is
     * not solid
     *
     * @param c
     * @throws Exception
     */
    @Override()
    public void load(final Cargo c) throws TransportException {
        if (c.getType() == CargoType.SOLID) {
            throw new InvalidCargoException("Solid Cargo not allowed on a tank truck");
        }
        super.load(c);

    }

}
```

Listing 12: src/transport/transporter/Transporter.java

```java
package transport.transporter;

import exceptions.OverloadedException;
import exceptions.TransportException;
import exceptions.UnreachableLocationException;
import transport.Cargo;
import transport.Location;

public class Transporter {
    protected String description;
    protected int maximumWeight;
```

```
    protected double costPerKM;
    protected Location currentLocation;
    protected Cargo cargo;

    public Transporter(final String description, final double costPerKM, final int
        maximumWeight,
            final Location currentLocation) {
        this.description = description;
        this.maximumWeight = maximumWeight;
        this.costPerKM = costPerKM;
        this.currentLocation = currentLocation;
    }

    /**
     * Sets the new destination and calculates the cost based on distance and
     * costPerKm
     *
     * @param destination
     * @return
     * @throws UnreachableLocationException
     */
    public double goTo(final Location destination) throws UnreachableLocationException
        {
        final double costForTransport = this.costPerKM *
            this.currentLocation.getDistance(destination);
        this.currentLocation = destination;
        return costForTransport;
    }

    /**
     * Loads a cargo. Only allows one cargo with a maximum weight and certain types
     * based on class
     *
     * @param c
     * @throws Exception
     */
    public void load(final Cargo c) throws TransportException {
        if (c.getWeight() > this.maximumWeight || this.cargo != null) {
            throw new OverloadedException("Loading this cargo is not allowed");
        }
        this.cargo = c;
    }

    /**
     * Unloads a cargo by setting cargo to null and returning the cargo
     *
     * @return
     */
    public Cargo unload() {
        final Cargo cargoToUnload = this.cargo;
        this.cargo = null;
        return cargoToUnload;
    }

    @Override()
    public String toString() {
        String msg = "Description: " + this.description + " | Max. weight: " +
            this.maximumWeight + " | Cost/KM: "
                + this.costPerKM + " | Current Loc: " +
                    this.currentLocation.toString();
        if (this.cargo != null) {
            msg += " | Cargo: " + this.cargo.toString();
        }
        return msg;
```

```
    }

}
```

Listing 13: src/transport/transporter/ContainerTruck.java

```
1  package transport.transporter;

   import exceptions.InvalidCargoException;
   import exceptions.TransportException;
5  import exceptions.UnreachableLocationException;
   import transport.Cargo;
   import transport.CargoType;
   import transport.Location;

10 public class ContainerTruck extends Transporter {

       public ContainerTruck(final String description, final double costPerKM, final int
           maximumWeight,
               final Location currentLocation) {
           super(description, costPerKM, maximumWeight, currentLocation);
15
       }

       /**
        * Sets the new destination and calculates the cost based on distance and
20      * costPerKm
        *
        * @param destination
        * @return
        * @throws UnreachableLocationException
25      */
       @Override()
       public double goTo(final Location destination) throws UnreachableLocationException
           {
           if (!this.currentLocation.reachableOverland(destination)) {
               throw new UnreachableLocationException("Container truck cannot reach this
                   location");
30         }
           final double costForTransport = this.costPerKM *
               this.currentLocation.getDistance(destination);
           this.currentLocation = destination;
           return costForTransport;
       }
35
       /**
        * Loads a cargo. Only allows one cargo with a maximum weight and it the type is
        * not liquid
        *
40      * @param c
        * @throws Exception
        */
       @Override()
       public void load(final Cargo c) throws TransportException {
45         if (c.getType() == CargoType.LIQUID) {
               throw new InvalidCargoException("Liquid Cargo not allowed on a container
                   truck");
           }
           super.load(c);

50     }

}
```

Listing 14: src/transport/Location.java

```
1  package transport;
2
3  /**
4   * Represents a location that can be reached by a transporter. As a
5   * simplification, a location is specified via its x-coordinate and y-coordinate
6   * and the continent.<br>
7   * A location object is immutable, i.e. it is not possible to change the data
8   * fields after the construction of the object.
9   */
10 public class Location {
11   /** Name of the location. */
12   private final String name;
13   /** x-coordinate of the location. */
14   private final int xCoord;
15   /** y-coordinate of the location. */
16   private final int yCoord;
17   /** the country of this location. */
18   private final Country country;
19
20   /**
21    * Creates a new location and initializes all fields.
22    *
23    * @param name
24    *             The name of the location.
25    * @param xCoord
26    *             The x-coordinate of the location.
27    * @param yCoord
28    *             The y-coordinate of the location.
29    * @param continent
30    *             The country of this location.
31    */
32   public Location(String name, int xCoord, int yCoord, Country country) {
33     super();
34     this.name = name;
35     this.xCoord = xCoord;
36     this.yCoord = yCoord;
37     this.country = country;
38   }
39
40   /**
41    * Checks if this location is on the same continent as the
42    * <code>other</code> location.
43    *
44    * @param other
45    *             The other location.
46    * @return true if the continent name is equal, false otherwise.
47    */
48   public boolean reachableOverland(Location other) {
49     return Country.landConnected(this.country, other.country);
50   }
51
52   /**
53    * Computes the distance between this location and the <code>other</code>
54    * location.
55    *
56    * @param other
57    *             The other location.
58    * @return the distance between the locations.
59    */
60   public double getDistance(Location other) {
61     double dx = xCoord - other.xCoord;
62     double dy = yCoord - other.yCoord;
63     return Math.sqrt(dx * dx + dy * dy);
```

```
  }

  /**
   * Returns the name of the location.
   *
   * @return The name.
   */
  public String getName() {
    return name;
  }

  /**
   * Returns the continent name of the location.
   *
   * @return The continent name.
   */
  public Country getCountry() {
    return country;
  }

  @Override
  public String toString() {
    return String.format("%1$s in %2$s at (%3$d/%4$d)", name, country, xCoord, yCoord);
  }
}
```

Listing 15: src/inout/Window.java

```
package inout;

import java.awt.Color;
import java.awt.Component;
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.image.BufferedImage;

/**
 * Diese Klasse erlaubt die einfache Ausgabe von graphischen Objekten Punkt,
 * Linie, Kreis und Rechteck in einem statischen Fenster. Dazu stellt es eine
 * Reihe von statischen Methoden zur Verf?gung:
 *
 * Mit Methode open wird die Klasse initialisiert und ein Fenster erzeugt und
 * ge?ffnet.
 * <p>
 * Methoden drawPoint, drawLine, drawCircle und drawRectangle erlauben das
 * Zeichnen von Punkten, Strecken, Kreisen bzw. Rechtecken. Weiters gibt es die
 * Methoden mit einem zus?tzlichen Parameter color, mit der die Farbe zum
 * Zeichnen spezifiziert werden kann.
 * <p>
 * Methoden fillCircle und fillRectangle erlauben das Zeichnen von Kreisen bzw.
 * Rechtecken gef?llt mit einer bestimmten Farbe.
 * <p>
 * Die Klasse hat eine Reihe von Restriktionen, die beachtet werden sollten. Das
 * Fenster hat eine fixe Gr??e (standardm??ig 800 mal 600 Pixel) und erlaubt
 * kein Scrollen. Das Fenster sollte auch nicht vergr??ert werden.
 *
 * @author Herbert Praehofer
 * @date
 */
public class Window {
```

```
  // Breite und H?he

  /** Konstante f?r Standardbreite */
  public static final int DEFAULT_WIDTH = 800;

  /** Konstante f?r Standardh?he */
  public static final int DEFAULT_HEIGHT = 600;

  /** Variable, welche die Breite des Fensters definiert. */
  public static int width;

  /** Variable, welche die Hoehe des Fensters definiert. */
  public static int height;

  // open

  /**
   * Initialisiert das Ausgabefenster und ?ffnet es.
   */
  public static void open() {
    open(DEFAULT_WIDTH, DEFAULT_HEIGHT);
  }

  /**
   * Initialisiert das Ausgabefenster und ?ffnet es.
   *
   * @param w
   *            Die Breite f?r das
   * @param w
   *            Die Hoehe f?r das Fenster
   */
  public static void open(int w, int h) {
    width = w;
    height = h;
    windowO = new Frame("WindowO");
    contentPane = new WindowOPanel();
    windowO.add(contentPane);
    image = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
    image.getGraphics().fillRect(0, 0, w, h);
    windowO.setSize(w + 12, h + headerHeight + 12);
    windowO.addWindowListener(new Window.WindowClosingAdapter(true));
    windowO.setVisible(true);
  }

  /** L?scht den Inhalt des Fensters */
  public static void clear() {
    image.getGraphics().fillRect(0, 0, width, height);
    contentPane.repaint();
  }

  // Methoden zum Zeichnen.

  /** Zeichnet einen Punkt bei der angebenen Position (x, y). */
  public static void drawPoint(int x, int y) {
    Graphics g = image.getGraphics();
    g.setColor(Color.black);
    g.fillRect(x - 1, y - 1, 3, 3);
    contentPane.repaint();
  }

  /** Zeichnet eine Linie von Position (x1, y1) zu Position (x2, y2). */
  public static void drawLine(int x1, int y1, int x2, int y2) {
    Graphics g = image.getGraphics();
```

```
    g.setColor(Color.black);
    g.drawLine(x1, y1, x2, y2);
    contentPane.repaint();
  }

  /** Zeichnet ein Rechteck bei Position (x, y) mit Breite w und H?he h. */
  public static void drawRectangle(int x, int y, int w, int h) {
    Graphics g = image.getGraphics();
    g.setColor(Color.black);
    g.drawRect(x, y, w, h);
    contentPane.repaint();
  }

  /** Zeichnet einen Kreis mit Mittelpunkt (x, y) und Radius r. */
  public static void drawCircle(int x, int y, int r) {
    Graphics g = image.getGraphics();
    g.setColor(Color.black);
    g.drawOval(x - r, y - r, 2 * r, 2 * r);
    contentPane.repaint();
  }

  /** Gibt den Text text auf Position x/y aus */
  public static void drawText(String text, int x, int y) {
    Graphics g = image.getGraphics();
    g.setColor(Color.black);
    g.drawString(text, x, y);
    contentPane.repaint();
  }

  /** Zeichnet einen Punkt bei der angebenen Position (x, y) mit Farbe color. */
  public static void drawPoint(int x, int y, Color color) {
    Graphics g = image.getGraphics();
    g.setColor(color);
    g.fillRect(x - 1, y - 1, 3, 3);
    contentPane.repaint();
  }

  /**
   * Zeichnet eine Linie von Position (x1, y1) zu Position (x2, y2) mit Farbe
   * color.
   */
  public static void drawLine(int x1, int y1, int x2, int y2, Color color) {
    Graphics g = image.getGraphics();
    g.setColor(new Color(color.getRed(), color.getGreen(), color.getBlue()));
    g.drawLine(x1, y1, x2, y2);
    contentPane.repaint();
  }

  /**
   * Zeichnet ein Rechteck bei Position (x, y) mit Breite w und H?he h mit
   * Farbe color.
   */
  public static void drawRectangle(int x, int y, int w, int h, Color color) {
    Graphics g = image.getGraphics();
    g.setColor(color);
    g.drawRect(x, y, w, h);
    contentPane.repaint();
  }

  /**
   * Zeichnet einen Kreis Zeichnet einen Kreis mit Mittelpunkt (x, y) und
   * Radius r mit Farbe color.
   */
  public static void drawCircle(int x, int y, int r, Color color) {
```

```
    Graphics g = image.getGraphics();
    g.setColor(new Color(color.getRed(), color.getGreen(), color.getBlue()));
    g.drawOval(x - r, y - r, 2 * r, 2 * r);
    contentPane.repaint();
  }

  /** Gibt den Text text auf Position x/y in Farbe color aus */
  public static void drawText(String text, int x, int y, Color color) {
    Graphics g = image.getGraphics();
    g.setColor(color);
    g.drawString(text, x, y);
    contentPane.repaint();
  }

  /**
   * Zeichnet ein gef?lltes Rechteck bei Position (x, y) mit Breite w und H?he
   * h mit Farbe color.
   */
  public static void fillRectangle(int x, int y, int w, int h, Color color) {
    Graphics g = image.getGraphics();
    g.setColor(color);
    g.fillRect(x, y, w, h);
    contentPane.repaint();
  }

  /**
   * Zeichnet einen gef?llten Kreis mit Mittelpunkt (x, y) und Radius r mit
   * Farbe color.
   */
  public static void fillCircle(int x, int y, int r, Color color) {
    Graphics g = image.getGraphics();
    g.setColor(color);
    g.fillOval(x - r, y - r, 2 * r, 2 * r);
    contentPane.repaint();
  }

  private static java.awt.Point p = null;

  /**
   * Wartet auf einen Mouseclick im Fenster und liefert die Position als
   * Ergebnis. Blockiert solange der Mouseclick nicht erfolgt ist.
   *
   * @return die Position des Mouseclicks
   */
  public static java.awt.Point getMouseClick() {

    contentPane.addMouseListener(new MouseAdapter() {
      /**
       * Invoked when the mouse has been clicked on a component.
       */
      @Override
      public void mouseClicked(MouseEvent e) {
        p = e.getPoint();
        synchronized (contentPane) {
          contentPane.notifyAll();
        }
      }
    });

    // blockiere solange Mouseclick nicht erfolgte
    synchronized (contentPane) {
      try {
        contentPane.wait();
      } catch (InterruptedException e1) {
```

```
230           e1.printStackTrace();
            }
          }
          return p;
        }

        // Privater Bereich
        // ---------------------------------------------------------------------

        /** Main-Frame */
        private static Frame windowO;

        /**
         * Panel mit Inhalt. Ist von Klasse WindowOPanel, welches das BufferedImage
         * image zeichnet
         */
        private static WindowOPanel contentPane;

        /**
         * BufferedImage, in welches gezeichnet wird und welches dann in contentPane
         * ausgegeben wird
         */
        private static Image image;

        /** Variable, die die Hoehe des Headers des Frame definiert */
        private static int headerHeight = 24;

        /** Innere Klasse zum Schlie?en des Frame */
        static class WindowClosingAdapter extends WindowAdapter {

          private boolean exitSystem;

          /**
           * Erzeugt einen WindowClosingAdapter zum Schliessen des Fensters. Ist
           * exitSystem true, wird das komplette Programm beendet.
           */
          public WindowClosingAdapter(boolean exitSystem) {
            this.exitSystem = exitSystem;
          }

          /**
           * Erzeugt einen WindowClosingAdapter zum Schliessen des Fensters. Das
           * Programm wird nicht beendet.
           */
          public WindowClosingAdapter() {
            this(true);
          }

          /** Schlie?t das Fenster und terminiert die Anwendung */
          @Override
          public void windowClosing(WindowEvent event) {
            event.getWindow().setVisible(false);
            event.getWindow().dispose();
            if (exitSystem) {
              System.exit(0);
            }
          }
        }

        /**
         * Klasse f?r die contentPane im WindowO Frame. Es wird paint von Component
         * ?berschrieben und das BufferedImage image gezeichnet.
         */
        static class WindowOPanel extends Component {
```

```
    /**
     *
     */
    private static final long serialVersionUID = 1146096508264896197L;

    @Override
    public void paint(Graphics g) {
      g.drawImage(image, 0, 0, null);
    }
  }
}
```

Listing 16: src/inout/Out.java

```
package inout;

import java.io.FileOutputStream;
import java.io.PrintStream;

/**
 * Simple output to the console and to files.
 * <p>
 * This class allows printing formatted data either to the console or to a file.
 * It is intended to be used in an introductory programming course when classes,
 * packages and exceptions are unknown at the beginning. To use it, simply copy
 * Out.class into the current directory.
 * </p>
 * <p>
 * All output goes to the current output file, which is initially the console.
 * Opening a file with open() makes it the new current output file. Closing a
 * file with close() switches back to the previous output file.
 * </p>
 */

public class Out {

  private static PrintStream out;
  private static PrintStream[] stack;
  private static int sp;
  private static boolean done;

  /**
   * Return true if the previous Out operation was successful, otherwise
   * return false.
   */
  public static boolean done() {
    return done && !out.checkError();
  }

  /** Print the boolean value b either as "true" or "false". */
  public static void print(boolean b) {
    out.print(b);
  }

  /** Print the character value c. */
  public static void print(char s) {
    out.print(s);
  }

  /** Print the integer value i. */
  public static void print(int i) {
    out.print(i);
  }

  /** Print the long value l. */
```

```
  public static void print(long l) {
    out.print(l);
  }

  /** Print the float value f. */
  public static void print(float f) {
    out.print(f);
  }

  /** Print the double value d. */
  public static void print(double d) {
    out.print(d);
  }

  /** Print the character array a. */
  public static void print(char[] a) {
    out.print(a);
  }

  /** Print the String s. */
  public static void print(String s) {
    out.print(s);
  }

  /** Print the Object o as resulting from String.valueOf(o). */
  public static void print(Object o) {
    out.print(o);
  }

  /**
   * Terminate the current line by writing a line separator string. On windows
   * this is the character sequence '\r' and '\n'
   */
  public static void println() {
    out.println();
  }

  /** Print the boolean value b and terminate the line. */
  public static void println(boolean b) {
    out.println(b);
  }

  /** Print the character value c and terminate the line. */
  public static void println(char s) {
    out.println(s);
  }

  /** Print the integer value i and terminate the line. */
  public static void println(int i) {
    out.println(i);
  }

  /** Print the long value l and terminate the line. */
  public static void println(long l) {
    out.println(l);
  }

  /** Print the float value f and terminate the line. */
  public static void println(float f) {
    out.println(f);
  }

  /** Print the double value d and terminate the line. */
  public static void println(double d) {
```

```
    out.println(d);
  }

  /** Print the character array a and terminate the line. */
  public static void println(char[] a) {
    out.println(a);
  }

  /** Print the String s and terminate the line. */
  public static void println(String s) {
    out.println(s);
  }

  /**
   * Print the Object o as resulting from String.valueOf(o) and terminate the
   * line.
   */
  public static void println(Object o) {
    out.println(o);
  }

  /**
   * Open the file with the name fn as the current output file. All subsequent
   * output goes to this file until it is closed. The old output file will be
   * restored when the new output file is closed.
   */
  public static void open(String fn) {
    try {
      PrintStream s = new PrintStream(new FileOutputStream(fn));
      stack[sp++] = out;
      out = s;
    } catch (Exception e) {
      done = false;
    }
  }

  /**
   * Close the current output file. The previous output file is restored and
   * becomes the current output file.
   */
  public static void close() {
    out.flush();
    out.close();
    if (sp > 0) {
      out = stack[--sp];
    }
  }

  static { // initializer
    done = true;
    out = System.out;
    stack = new PrintStream[8];
    sp = 0;
  }

}
```

Listing 17: src/inout/In.java

```
package inout;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
```

```
import java.util.LinkedList;

/**
 * Simple input from the keyboard or from a file.
 * <p>
 * This class allows reading formatted data either from the keyboard or from a
 * file. It is intended to be used in an introductory programming course when
 * classes, packages and exceptions are unknown at the beginning. To use it,
 * simply copy In.class into the source file directory.
 * </p>
 *
 * <p>
 * All input comes from the current input file, which is initially the keyboard.
 * Opening a file with open() makes it the new current input file. Closing a
 * file with close() switches back to the previous input file.
 * </p>
 *
 * <p>
 * When reading from the keyboard, reading blocks until the user has entered a
 * sequence of characters terminated by the return key. All methods read from
 * this input buffer (including the terminating '\r' and '\n') until the buffer
 * is fully consumed. When a method tries to read beyond the end of the buffer,
 * it blocks again waiting for the next buffer.
 * </p>
 *
 * <p>
 * End of file detection: When reading from the keyboard, eof can be signaled as
 * ctrl-Z at the beginning of a new line. When reading from a file, eof occurs
 * when an attempt is made to read beyond the end of the file. In either case
 * In.done() returns false if the requested data could not be read because of
 * eof.
 * </p>
 */
public class In {

  /**
   * End of file indicator returned by read() or peek() when no more
   * characters can be read.
   */
  public static final char eof = '\uffff';

  private static final int empty = '\ufffe';

  private static final char eofChar = '\u0005'; // ctrl E
  private static InputStream in;
  private static LinkedList<InputStream> inputStack;
  private static LinkedList<Character> bufferStack;
  private static boolean done; // true if recent operation was successful
  private static char buf; // last read character
  private static char[] LS; // line separator (eol)

  private static char charAfterWhiteSpace() {
    char c;
    do {
      c = read();
    } while (done && c <= '␣');
    return c;
  }

  private static String readDigits() {
    StringBuffer b = new StringBuffer();
    char c = charAfterWhiteSpace();
    if (done && c == '-') {
      b.append(c);
```

```
      c = read();
    }
    while (done && Character.isDigit(c)) {
      b.append(c);
      c = read();
    }
    buf = c;
    return b.toString();
  }

  private static String readFloatDigits() {
    StringBuffer b = new StringBuffer();
    char c = charAfterWhiteSpace();
    if (done && (c == '+' || c == '-')) {
      b.append(c);
      c = read();
    }
    while (done && Character.isDigit(c)) {
      b.append(c);
      c = read();
    }
    if (done && (c == '.')) {
      b.append(c);
      c = read();
      while (done && Character.isDigit(c)) {
        b.append(c);
        c = read();
      }
    }
    if (done && (c == 'e' || c == 'E')) {
      b.append(c);
      c = read();
      if (done && (c == '+' || c == '-')) {
        b.append(c);
        c = read();
      }
      while (done && Character.isDigit(c)) {
        b.append(c);
        c = read();
      }
    }
    buf = c;
    return b.toString();
  }

  /**
   * Read a raw character (byte). If an attempt is made to read beyond the end
   * of the file, eof is returned and done() yields false. Otherwise the read
   * byte is in the range 0..255.
   */
  public static char read() {
    char c;
    if (buf != empty) {
      c = buf;
      if (buf != eof) {
        buf = empty;
      }
    } else {
      try {
        c = (char) in.read();
      } catch (IOException e) {
        done = false;
        c = eof;
        buf = eof;
```

```
135       }
        }
        if (c == eofChar && inputStack.size() == 0) {
          c = eof;
          buf = eof;
140     }
        done = c != eof;
        return c;
      }

145   /**
       * Current available raw characters. In case of an error 0 is returned and
       * done() yields false.
       */
      public static int available() {
150     int avail;

        try {
          avail = in.available();
        } catch (IOException exc) {
155       avail = 0;
          done = false;
        }

        return avail;
160   }

      /**
       * Read a character, but skip white spaces (byte). If an attempt is made to
       * read beyond the end of the file, eof is returned and done() yields false.
165    * Otherwise the read byte is in the range 0..255.
       */
      public static char readChar() {
        return charAfterWhiteSpace();
      }
170
      /**
       * Read a boolean value. This method skips white space and tries to read an
       * identifier. If its value is "true" the method returns true otherwise
       * false. If the identifier is neither "true" nor "false" done() yields
175    * false.
       */
      public static boolean readBoolean() {
        String s = readIdentifier();
        done = true;
180     if (s.equals("true")) {
          return true;
        }
        done = s.equals("false");
        return false;
185   }

      /**
       * Read an identifier. This method skips white space and tries to read an
       * identifier starting with a letter and continuing with letters or digits.
190    * If a token of this structure could be read, it is returned otherwise the
       * empty string is returned and done() yields false.
       */
      public static String readIdentifier() {
        StringBuffer b = new StringBuffer();
195     char c = charAfterWhiteSpace();
        if (done && Character.isLetter(c)) {
          b.append(c);
          c = read();
```

```
      while (done && (Character.isLetter(c) || Character.isDigit(c))) {
        b.append(c);
        c = read();
      }
    }
    buf = c;
    done = b.length() > 0;
    return b.toString();
  }

  /**
   * Read a word. This method skips white space and tries to read a word
   * consisting of all characters up to the next white space or to the end of
   * the file. If a token of this structure could be read, it is returned
   * otherwise an empty string is returned and done() yields false.
   */
  public static String readWord() {
    StringBuffer b = new StringBuffer();
    char c = charAfterWhiteSpace();
    while (done && c > ' ') {
      b.append(c);
      c = read();
    }
    buf = c;
    done = b.length() > 0;
    return b.toString();
  }

  /**
   * Read a line of text. This method reads the rest of the current line
   * (including eol) and returns it (excluding eol). A line may be empty.
   */
  public static String readLine() {
    StringBuffer b = new StringBuffer();
    char c = read();
    while (done && c != LS[0]) {
      b.append(c);
      c = read();
    }

    int i = 0;
    while (c == LS[i]) {
      ++i;
      if (i >= LS.length) {
        break;
      }
      c = read();
    }

    if (i < LS.length) {
      buf = c;
    } else {
      buf = empty;
    }
    if (b.length() > 0) {
      done = true;
    }
    return b.toString();
  }

  /**
   * Read the whole file. This method reads from the current position to the
   * end of the file and returns its text in a single large string. done()
   * yields always true.
```

```
   */
  public static String readFile() {
    StringBuffer b = new StringBuffer();
    char c = charAfterWhiteSpace();
    while (done) {
      b.append(c);
      c = read();
    }
    buf = eof;
    done = true;
    return b.toString();
  }

  /**
   * Read a quote-delimited string. This method skips white space and tries to
   * read a string in the form "...". It can be used to read pieces of text
   * that contain white space.
   */
  public static String readString() {
    StringBuffer b = new StringBuffer();
    char c = charAfterWhiteSpace();
    if (done && c == '"') {
      c = read();
      while (done && c != '"') {
        b.append(c);
        c = read();
      }
      if (c == '"') {
        c = read();
        done = true;
      } else {
        done = false;
      }
    } else {
      done = false;
    }
    buf = c;
    return b.toString();
  }

  /**
   * Read an integer. This method skips white space and tries to read an
   * integer. If the text does not contain an integer or if the number is too
   * big, the value 0 is returned and the subsequent call of done() yields
   * false. An integer is a sequence of digits, possibly preceded by '-'.
   */
  public static int readInt() {
    String s = readDigits();
    try {
      done = true;
      return Integer.parseInt(s);
    } catch (Exception e) {
      done = false;
      return 0;
    }
  }

  /**
   * Read a long integer. This method skips white space and tries to read a
   * long integer. If the text does not contain a number or if the number is
   * too big, the value 0 is returned and the subsequent call of done() yields
   * false. A long integer is a sequence of digits, possibly preceded by '-'.
   */
  public static long readLong() {
```

```
    String s = readDigits();
    try {
      done = true;
      return Long.parseLong(s);
    } catch (Exception e) {
      done = false;
      return 0;
    }
  }

  /**
   * Read a float value. This method skips white space and tries to read a
   * float value. If the text does not contain a float value or if the number
   * is not well-formed, the value 0f is returned and the subsequent call of
   * done() yields false. An float value is as specified in the Java language
   * description. It may be preceded by a '+' or a '-'.
   */
  public static float readFloat() {
    String s = readFloatDigits();
    try {
      done = true;
      return Float.parseFloat(s);
    } catch (Exception e) {
      done = false;
      return 0f;
    }
  }

  /**
   * Read a double value. This method skips white space and tries to read a
   * double value. If the text does not contain a double value or if the
   * number is not well-formed, the value 0.0 is returned and the subsequent
   * call of done() yields false. An double value is as specified in the Java
   * language description. It may be preceded by a '+' or a '-'.
   */
  public static double readDouble() {
    String s = readFloatDigits();
    try {
      done = true;
      return Double.parseDouble(s);
    } catch (Exception e) {
      done = false;
      return 0.0;
    }
  }

  /**
   * Peek at the next character. This method skips white space and returns the
   * next character without removing it from the input stream. It can be used
   * to find out, what token comes next in the input stream.
   */
  public static char peek() {
    char c = charAfterWhiteSpace();
    buf = c;
    return c;
  }

  /**
   * Open a text file for reading The text file with the name fn is opened as
   * the new current input file. When it is closed again, the previous input
   * file is restored.
   */
  public static void open(String fn) {
    try {
```

```
      InputStream s = new FileInputStream(fn);
      bufferStack.add(new Character(buf));
      inputStack.add(in);
      in = s;
      done = true;
    } catch (FileNotFoundException e) {
      done = false;
    }
    buf = empty;
  }

  /**
   * Close the current input file. The current input file is closed and the
   * previous input file is restored. Closing the keyboard input has no effect
   * but causes done() to yield false.
   */
  public static void close() {
    try {
      if (inputStack.size() > 0) {
        in.close();
        in = inputStack.removeLast();
        buf = bufferStack.removeLast().charValue();
        done = true;
      } else {
        done = false;
        buf = empty;
      }
    } catch (IOException e) {
      done = false;
      buf = empty;
    }
  }

  /**
   * Check if the previous operation was successful. This method returns true
   * if the previous read operation was able to read a token of the requested
   * structure. It can also be called after open() and close() to check if
   * these operations were successful. If done() is called before any other
   * operation it yields true.
   */
  public static boolean done() {
    return done;
  }

  static { // initializer
    done = true;
    in = System.in;
    buf = empty;
    inputStack = new LinkedList<InputStream>();
    bufferStack = new LinkedList<Character>();
    LS = System.getProperty("line.separator").toCharArray();
    if (LS == null || LS.length == 0) {
      LS = new char[] { '\n' };
    }
  }

}
```