Cheat Sheet Java-OO (2)

Serialisierung/Deserialisierung

Dient dem lesen und schreiben von Objekten. Um ein Objekt in eine Datei zu muss es das Interface `Serializable` implementieren.
Beispiel Code Serializerung/Deserialisierung:

```
public void serialize(final Serializable object, final String
     filename) {
  FileOutputStream fs = null;
  ObjectOutputStream os = null;
  try {
    fs = new FileOutputStream(new File(filename));
    os = new ObjectOutputStream(fs);
    os.writeObject(object);
    os.close();
    fs.close();
  } catch (final IOException e) {
    e.printStackTrace();
  }
}

public Serializable load(final String filename) {
  FileInputStream fi = null;
  ObjectInputStream oi = null;
  Serializable object = null;
  try {
    fi = new FileInputStream(filename);
    oi = new ObjectInputStream(fi);
    object = oi.readObject();
    oi.close();
    fi.close();
  } catch (final IOException e) {
    e.printStackTrace();
  }
  return (object == null) ? new Object() : object;
}
```

Das serialisierte Objekt ist in den entsprechenden Typ zu casten. Grundsätzlich wird der gesamte Objektgraph gespeichert. Instanzvariablen werden dabei transient gespeichert. Falls die Superklasse nicht serialisierbar ist, wird einfach deren Default-Konstruktor ausgeführt.

Datei-Operationen

Datein lasse sich mit Hilfe von `File` und `FileWriter` schreiben.

```
public void write(final String content, final String path) {
  BufferedWriter buffer = null;
  try {
    buffer = new BufferedWriter(new FileWriter(new File(path))
        );
    buffer.write(content);
  } catch (final IOException e) {
    e.printStackTrace();
  } finally {
    // Close the buffer
    try {
      if (buffer != null) {
        buffer.close();
      }
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Das Lesen funktioniert über `File` und `FileReader`:

```
public String read(final String path) {
  BufferedReader reader = null;
  try {
    final File file = new File(path);
    reader = new BufferedReader(new FileReader(file));
    final StringBuffer stringBuffer = new StringBuffer();
    String line = reader.readLine();
    while (line != null) {
      stringBuffer.append(line);
      line = reader.readLine();
    }
  } catch (final FileNotFoundException e) {
    e.printStackTrace();
  } catch (final IOException e) {
    e.printStackTrace();
  } finally {
    // Close the buffer
    try {
      if (reader != null) {
        reader.close();
      }
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
  return buffer.toString();
}
```

`File` repräsentiert die Datei auf einem Datenträger. Nicht den Inhalt der Datei. Bietet Methoden wie `mkdir, isDirectory, delete, ....`

Threads

Ein Thread ist ein separater Ausführungsstrang: Jeder Thread hat seinen eigenen Aufruf-Stack → Ein `Thread`-Objekt repräsentiert einen solchen Ausführungsstrang.

Ein `Thread` braucht eien Job der as `Runnable` Interface implementiert.

Zustände von Threads

`NEW` instantiert, `start()` wurde aber noch nicht aufgerufen

`RUNNABLE` Die `start()` Methode wurde aufgerufen, ein neuer Stack wurde erzeugt, der Thread wartet darauf vom Thread-Scheduler ausgeführt zu werden.

`RUNNING` Der Thread läuft

`BLOCKED` Der Thread ist vorrüber nicht lauffähig (wartet auf Ergebnisse, schläft, wartet auf ein gesperrtes Objekt)

Wissenswertes

- Threads können einen Namen haben (`myThread.setName()`)!
- Die Arbeitsweise des Thread-Schedulers ist nicht vorhersebar bzw. garantiert
- Die Methode `Thread.sleep()` zwingt einen Thread den Zustand `RUNNING` aufzugeben. Anderen Threads dadurch eine Chance ausgeführt zu werden.
- Threads können Problemen verursachen, wenn zwei oder mehrere Threads Zugriff auf das gleiche Objekt haben (siehe Locks)
- `synchronized` schützt davor, dass mehrere Threads gleichzeitig den selben Block durchlaufen.
- Um Objekte theadsicher zu machen, müssen Anweisungen zu atomaren Prozessen gemacht werden.
- Selbst wenn ein Objekt mehrere synchronisierte Methoden hat, gibt es nur einen Schlüssel. Solange sich irgendein Thread in einer synchronisierten Methode in diesem Objekt befindet, kann kein Thread in irgendeine andere synchronisierte Mehtode desselben Objekts eintreten.

Locks

Anweisungen, die nur von einem Thread gleichzeitig durchlaufen werden dürfen beginnen mit `lock()` und enden mit `unlock()`.

```
public class ReentrantLockedExample implements Runnable {
  private final Lock lock = new ReentrantLock();

  @Override
  public void run() {
    // uncritical code ...
    lock.lock();
    // critical actions
    lock.unlock();
    // sum more uncritical code
  }
}
```

`ReentrantLock` bietet nützliche Methoden wie `isLocked(), getQueueLength(), tryLock(), ...`
Mit der Implementierung `ReentrantReadWriteLock` kann Code erstellt werden, der es erlaubt, dass mehrere Threads in einen gemeinsamen Abschnitt eintreten können, der zum Lesen deklariert wurde. Der Block zum Schreiben ist dann gelockt.
Bei Eintritt in einen Block der zum Schreiben deklariert wurde, werden alle anderen Blöcke (lesen/schreiben) für andere Threads gelockt.

Wait/Notifiy

Jedes Objekt besitzt die Methoden `wait()` und `notify()`. Ein Thread der über das Monitor-Objekt verfügt, kann diese Methoden aufrufen und sich so in einen Wartezustand versetzen oder einen anderen Thread aufwecken. Beispiele

```
public class Waiter implements Runnable {
  private Object monitor;

  public Waiter(Object monitor) {
    this.monitor = monitor;
  }

  @Override
  public void run() {
    while (true) {
      synchronized (monitor) {
        System.out.println("warte bis geweckt ...");
        try {
          monitor.wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        System.out.println("... wurde geweckt");
      }
    }
  }
}

public class Notifier implements Runnable {
  private Object monitor;

  public Notifier(Object monitor) {
    this.monitor = monitor;
  }

  @Override
  public void run() {
    while (true) {
      System.out.println("kleine pause ...");
      try {
        Thread.sleep(3000);
        System.out.println("weiter gehts ...");
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      synchronized (monitor) {
        // Monitor benachrichtigen:
        monitor.notify();
      }
    }
  }
}
```

ExecutorService

Der ExecutorService erlaut es Thread-Pools anzulegen. Ein und derselbe Thread kann dann mehrmals gestartet werden (Methode: `execute()`)

```
public void usePool() {
  final ExecutorService executor = Executors.
      newCachedThreadPool();
  final Runnable1 r1 = new MyRunnable1();
  final Runnable2 r2 = new MyRunnable2();

  executor.execute(r1);
  executor.execute(r2);

  try {
    Thread.sleep(500);
    System.out.println("Sleeping ...");
    Thread.sleep(500);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }

  executor.execute(r1);
  executor.execute(r2);

  executor.shutdown();
}
```

Mit dem `Callable<T>`-Interface kann dem Aufrufer ein Rückgabewert zurückgegeben werden.

```
public void useCallable() {
  final ExecutorService executorService = Executors.
      newCachedThreadPool();
  final Collection<Callable<Integer>> tasks =
    new ArrayList<Callable<Integer>>();

  tasks.add(new MyCallable(21));
  tasks.add(new MyCallable(35));

  List<Future<Integer>> result = null;
  try {
    result = executorService.invokeAll(tasks);
  } catch (InterruptedException e1) {
    e1.printStackTrace();
  }

  // Die Ergebnisse stehen in Future:
  for (Future<Integer> future : result) {
    try {
      System.out.println(future.get());
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }
  }
  executorService.shutdown();
}
```

Timed Tasks

Die Implementierung von zeitgesteuerten Abläufen nimmt uns zum Teil die Java-Bibliothek ab, die dazu die Klassen

- `java.util.Timer`
- `java.util.TimerTask`

anbietet. Sie helfen bei der zeitgesteuerten Ausführung. Ein `TimerTask` ist eine Klasse, die uns `Runnable` implementieren lässt und Operationen umfasst, die zu einem Zeitpunkt oder in einer beliebigen Wiederholung ausgeführt werden sollen

```
public class Task extends TimerTask {
  @Override
  public void run() {
    // do something
  }
}
```

Verwendung:

```
public void doTimedStuff() {
  final Timer timer = new Timer();

  // Start in 2 Sekunden
  timer.schedule(new Task("Task 1"), 2000);

  // Start in einer Sekunde dann Ablauf alle 5 Sekunden
  timer.schedule(new Task("Task 2"), 1000, 5000);

  // Start am ... (Datum)
  final Calendar c = Calendar.getInstance();
  c.add(Calendar.SECOND, 15); // Zeitpunkt setzen
  Date d = c.getTime();
  timer.schedule(new Task("Task 3"), d);
}
```

Collections

Interface	Interface	Beschreibung
List	ArrayList, LinkedList, Vector, Stack	Sammlung gleichartiger Objekte
Set	HashSet, LinkedHashSet	Menge von Objekten (d.h. keine Doppelten)
Map	Attributes, HashMap, Hashtable, IdentityHashMap, RenderingHints, WeakHashMap	Key-Value-Speicher
SortedSet	TreeSet	Geordnete Menge von Objekten
SortedMap	TreeMap	Geordnete Key-Value-Paare

- Seit Java 6 sind die Collections typsicher und werden als `List<T>` deklariert
- Vector ist im Gegensatz zu ArrayList synchronisiert

Generics

Verwendungsbeispiele

```
public class TierWaage {
  public static <T extends Tier> void printGewicht(T t) {
    System.out.println(t.getGewicht());
  }
}
public class TierComparator implements Comparator<Tier> {
  @Override
  public int compare(Tier o1, Tier o2) {
    return o1.getGewicht() - o2.getGewicht();
  }
}
public class Regal<T extends Ware> {
  final List<T> waren = new ArrayList<T>();

  public void add(T ware) {
    waren.add(ware);
  }

  public List<T> getWaren() {
    return waren;
  }
}
```

GUI

Es gibt verschiedene Gui-Implementationen (AWT, Swing, SWT). Für die GUI-Komponeneten gilt allgemein: Komponenten werden der Elternkomponente mit
`parent.getContentPane().add(newComponent)` hinzugefügt.

Beispiel Fenster

```
public class Buttons {
  private JFrame frame;
  public void show() {
  frame = new JFrame("WindowTitle");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  final JButton button1 = new JButton("Button 1");
  final JButton button2 = new JButton("Button 2");
  final JLabel label = new JLabel();
  label.setSize(300, 50);
  button1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      label.setText("Button 1 wurde angeklickt");
    }
  });
  button2.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      label.setText("Button 2 wurde angeklickt");
    }
  });
  Container pane = frame.getContentPane();
  pane.setLayout(new BorderLayout());
  pane.add(label, BorderLayout.CENTER);
  pane.add(button1, BorderLayout.NORTH);
  pane.add(button2, BorderLayout.SOUTH);
  frame.setSize(300, 100);
  frame.setVisible(true)
}
```

Beispiel Menu

```
public void createMenuBar(final Frame frame) {

  // Menüzeile (JMenuBar) erzeugen und in das Fenster (JFrame)
      einfügen
  final JMenuBar bar = new JMenuBar();
  frame.setJMenuBar(bar);

  // Menü (JMenu) erzeugen und in die Menüzeile (JMenuBar)
      einfügen
  final JMenu dateiMenu = new JMenu("Datei");
  bar.add(dateiMenu);

  // Menüeinträge (JMenuItem) erzeugen und dem Menü (JMenu) "
      Datei" hinzufügen
  final JMenuItem oeffnenItem = new JMenuItem("Öffnen");
  oeffnenItem.addActionListener(new OpenActionListener());
  dateiMenu.add(oeffnenItem);

  final JMenuItem beendenItem = new JMenuItem("Beenden");
  beendenItem.addActionListener(new ExitActionListener());
  dateiMenu.add(beendenItem);
}
```

LayoutManager

LayoutManager ermöglichen das Platzieren der Inhaltselemente:

`BorderLayout`	Besteht aus fünf Bereichen: North, South (ganze Breite), East, West und Center, die einzeln gefüllt werden können.(Default für `JFrame`)
`FlowLayout`	Platziert alle Elemente nebeneinander und bricht um, wenn der Platz nicht ausreicht (Default für `JPanel`)

Implentieren von Listenern

- Das entsprechende Element (z.B. `JPanel`) den entsprechenden Listener implementieren lassen.
- Durch eine anonyme innere Klasse

Revision: 1.0, Datum: 26. Juni 2013