Informatik B

Vorlesung 23

MIDlets

Rückblick

- J2ME
  - Konfiguration
  - Profil
  - Optionale Pakete
- MIDlet
  - MIDlet-Suite
  - Lebenszyklus
  - Erste UI-Komponenten
  - **`Command`**

Beispiel: Einkaufsliste

- Es soll eine Einkaufsliste für das Handy implementiert werden
- Wird das Programm gestartet, werden diverse Artikel angezeigt, die ausgewählt sein können oder nicht

Beispiel: Einkaufsliste

- Die Einträge können aus- oder abgewählt werden, je nachdem was eingekauft werden soll
- Ist ein Artikel nicht in der Liste vorhanden, kann er hinzugefügt werden

Beispiel: Einkaufsliste

- Es soll ein Menü vorhanden sein, mit dem man
  - Alle Artikel abwählen kann
  - Artikel wieder löschen kann
  - Einen Artikel hinzufügen kann

User-Interface

- In der CLDC sind keine UI-Elemente vorhanden
- Alle UI-Elemente sind im MIDP definiert
- Ein **`Display`**-Objekt kann ausschließlich **`Displayable`**-Objekte darstellen
  - **`TextBox`**: Eine Fläche zur Texteingabe mit einer begrenzten Anzahl Zeichen
  - **`Alert`**: Informiert den User über Fehler, Warnings oder andere Infos
  - **`List`**: Enthält eine Menge von auswählbaren Elementen
  - **`Form`**: Enthält verschiedene Item-Objekte

User-Interface

- Es gibt folgende **`Item`**-Objekte, die einer Form zugeordnet werden können:
  - **`ChoiceGroup`**: Gruppe auswählbarer Elemente
  - **`CustomItem`**: Abstrakte Klasse, um eigene Items zu implementieren
  - **`DateField`**: Editierbares **`Item`**, um Datum und Zeit anzuzeigen
  - **`Gauge`**: Anzeige in Form einer ProgressBar
  - **`ImageItem`**: Zeigt ein Image-Objekt an
  - **`Spacer`**: Platzhalter
  - **`StringItem`**: Label und textueller Inhalt (nicht vom User editierbar)
  - **`TextField`**: Editierbares Textfeld
- Beispiel: **`item1`**

User-Interface

- Um für die Einkaufsliste ein User-Interface zu gestalten bietet sich ein **`Displayable`**-Objekt der Klasse **`List`** an
- Beim Erstellen der **`List`** kann angegeben werden, ob
  - immer nur ein Eintrag ausgewählt sein darf (**`Choice.EXCLUSIVE`**)
  - beliebig viele Einträge ausgewählt sein dürfen (**`Choice.MULTIPLE`**)
- Ein **`List`**-Objekt enthält einzelne Einträge bestehend aus einem Text und einem optionalen Bild die mit **`append(String s, Image i)`** hinzugefügt werden können
- Beispiel: **`einkaufsliste1`**

UI-Kommandos

- Damit ein User ein MIDlet beenden kann, muss ihm ein Kommando an die Hand gegeben werden
- Dazu wird ein entsprechendes Command-Objekt erzeugt
  `Command exCom = new Command("Exit", Command.EXIT, 1);`
- Das **`Command`**-Objekt wird an das **`Displayable`** gehängt:
  `aList.addCommand(exCom);`
- Zusätzlich wird das **`Displayable`** mit einem **`CommandListener`** versehen:

```
aList.setCommandListener(new CommandListener() {
 public void commandAction(Command c, Displayable d) {
 }
});
```

- Beispiel: **`einkaufsliste2`**

UI-Kommandos

- Es soll möglich sein einen Artikel hinzuzufügen
- Dazu muss
  - über ein Kommando ein Eingabefeld geöffnet werden
  - das Eingabefeld mit Kommandos versehen werden, um die Aktion abzubrechen oder um die Aktion zu beenden
  - der eingegebene Artikel muss in die Liste passend (lexikografisch sortiert) eingefügt werden

UI-Kommandos

- Ein **`TextField`** muss in eine Form gepackt werden
- Die **`Form`** muss mit Kommandos versehen werden
- Die **`Form`** wird sichtbar gemacht
- Ein **`List`**-Objekt kann nicht sortieren, daher muss das Ergebnis „von Hand“ einsortiert werden
- Beispiel: **`einkaufsliste3`**

UI-Kommandos

- Der User soll die Möglichkeit haben Einträge in der Liste zu löschen
- Dazu soll ihm eine neue **`List`** präsentiert werden, in der er seine Auswahl tätigen kann
- Eine neue List ist notwendig, da die vom User getroffene Auswahl nicht verändert werden darf
- Auch hier müssen die entsprechenden Kommandos erstellt werden
- Zusätzlich soll ein Ticker (Lauftext) integriert werden, der dem User zusätzliche Informationen gibt
  `aList.setTicker(new Ticker(„Text"));`
- Der Text wird dann im oberen Bereich des Displays als Lauftext angezeigt
- Beispiel: **`einkaufsliste4`**

Datenspeicherung

- Ein MIDlet hat nicht die Möglichkeit Daten in eine Datei zu speichern
- Die MIDP-Spezifikation fordert, dass die Hardware über nichtflüchtigen Speicher von mindestens 8kB verfügen muss
- Die Art des Speichers ist nicht vorgegeben
- Das *Record-Management-System* (*RMS*) stellt eine geräteunabhängige API zur Verfügung
- Es sind lesende und schreibende Zugriffe möglich

Datenspeicherung

- Das RMS arbeitet wie eine einfache Satzorientierte Datenbank
- Die Datenbank wird als Record-Store bezeichnet
- Ein Record-Store wird durch ein MIDlet angelegt und erhält einen Namen
- Der Record-Store wird der MIDlet-Suite zugeordnet, wobei alle MIDlets der Suite gleichberechtigten Zugriff haben
- Eine Suite kann mehrere Record-Stores besitzen, wobei jeder innerhalb der Suite einen eindeutigen Namen haben muss
- Es gibt keine harte Obergrenze für die Anzahl der Stores, aber je Suite müssen 5 Stores unterstützt werden (sofern genug Speicher verfügbar ist)

Datenspeicherung

- Jeder Datensatz in einem Store hat eine eindeutige ID (Primärschlüssel)
- Über die ID kann direkt auf einen Datensatz zugegriffen werden
- Das RMS garantiert, dass jeder API-Aufruf atomar ist
- Transaktionen (ähnlich zu Synchronisation bei Threads) werden nicht unterstützt
- Der Programmierer muss selbst für die Datenkonsistenz sorge tragen, insbesondere bei nebenläufiger Programmierung

Datenspeicherung

- Das RMS-API liegt im Paket **`javax.microedition.rms`**
- Die zentrale Klasse ist **`RecordStore`**
- Mit ihr ist es möglich
  - Einen Record-Store zu erzeugen, öffnen, schließen, löschen
  - Informationen über alle sichtbaren Stores zu erhalten
  - Datensätze zu lesen, ändern, löschen
  - Suchen von Datensätzen

RecordStore

- Öffnen eines Record-Store:
  ```
  RecordStore store =
  RecordStore.openRecordStore("Name", false);
  ```
- Der Wert **`false`** führt dazu, dass bei nicht vorhandenem Store eine Exception ausgelöst wird, **`true`** bewirkt, dass bei nicht Vorhandensein ein Store angelegt wird
- Schließen eines Record-Store:
  ```
  store.closeRecordStore();
  ```
- Löschen eines Record-Store:
  ```
  RecordStore.deleteRecordStore(„Name");
  ```
- Die AMS löscht bei Deinstallation einer MIDlet-Suite automatisch alle verbundenen Record-Stores

Daten einfügen

- In einen offenen Record-Store können Daten eingefügt werden:
```
byte[] record = “Hallo“.getBytes();
store.addRecord(record, 0, record.length);
```
- Es kann nur der Inhalt von **`byte`**-Arrays gespeichert werden
- Die Speicherung folgt in einem bisher noch nicht belegten Speicherplatz
- Die ID des Schlüssels liefert die Methode als **`int`** zurück

Daten einfügen

- **`addRecord`** akzeptiert nur **`byte`**-Arrays
- Objekte und Werte elementarer Datentypen müssen entsprechend umgewandelt werden
- Es existiert aber keine Serialisierung
- Die Umwandlung kann mithilfe der Klasse **`DataOutputStream`** vorgenommen werden
- Die Klasse bietet folgende Methoden:
  - **`writeBoolean(boolean b)`**
  - **`writeInt(int i)`**
  - **`writeLong(long l)`**
  - **`writeUTF(String s)`**
  - **`writeChar(int c)`**
  - **`…`**

Daten einfügen

- Ein **`DataOutputStream`** erwartet einen Strom in den er schreiben kann
- Dies kann ein Strom einer Netzwerkverbindung sein, oder aber ein **`ByteArrayOutputStream`**:

```
ByteArrayOutputStream baos = new
ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
```

- Werden nun Daten in den **`DataOutputStream`** geschrieben (**`writeUTF`**, **`writeInt`**, …) werden die Daten in ein **`byte`**-Array umgewandelt
- Dies kann zur Vereinfachung in einer Klasse zusammengefasst werden

Daten einfügen

```
public class Test {
    private String name;
    private int zahl;

    // Konstruktoren, …

    public byte[] getBytes() {
          ByteArrayOutputStream baos = new
                ByteArrayOutputStream();
          DataOutputStream dos = new
                DataOutputStream(baos);

          dos.writeUTF(name == null? "": name);
          dos.writeInt(zahl);

          dos.close();
          return baos.toByteArray();
    }
}
```

Daten einfügen

- Durch das Schließen des **`DataOutputStream`**-Objektes wird auch der **`ByteArrayOutputStream`** geschlossen
- Ein **`Test`**-Objekt kann auf diese Weise sehr einfach in einen Record-Store geschrieben werden:

```
Test t = new Test(“Name“, 42);
byte[] rec = t.getBytes();
store.addRecord(rec, 0, rec.length);
```

Daten lesen

- Aus einem offenen Record-Store können Daten gelesen werden:
  `byte[] record = store.getRecord(id);`
- Da die Methode ein **`byte`**-Array zurückliefert bietet sich auch hier die Verwendung eines **`DataInputStream`**-Objektes an, das folgende Methoden bietet:
  - **`readBoolean(boolean b)`**
  - **`readInt(int i)`**
  - **`readLong(long l)`**
  - **`readUTF(String s)`**
  - **`readChar(int c)`**
  - **…**

Daten lesen

- Analog zur Umwandlung in ein **`byte`**-Array können die Daten zurückgewonnen werden:

```
public void setBytes(byte[] record) {
    ByteArrayInputStream bais = new
          ByteArrayInputStream();
    DataInputStream dis = new
          DataInputStream(bais);

    name = dis.readUTF();
    zahl = dis.readInt();
    dis.close();
}
```

- Die read-Operationen in **`setBytes`** müssen in der gleichen Reihenfolge wie die write-Anweisungen in der Methode **`getBytes`** erfolgen

Daten manipulieren

- Ein Datensatz kann überschrieben werden:
  `store.setRecord(id, record, 0, record.length);`
- Ein Datensatz kann gelöscht werden:
  `store.deleteRecord(id);`
- Wird ein Datensatz gelöscht, kann kein Datensatz mit dem gleichen Schlüssel geschrieben werden!!!

Speichern der Einkaufsliste

- Soll die Einkaufsliste gespeichert werden, muss der Name eines Artikels gespeichert werden
- Zusätzlich muss gespeichert werden, ob der Artikel ausgewählt war oder nicht
- Die **`List`** selbst kann nicht gespeichert werden, da sie nicht in ein **`byte`**-Array umgewandelt werden kann
- Daher muss eine Hilfsklasse implementiert werden
- Des Weiteren muss darauf geachtet werden, dass Artikel aus der Liste gelöscht werden können
- Um das Programm einfach zu halten werden dabei entstehende leere Records mit einem *Dummy*-Element gefüllt
- Beispiel: **`einkaufsliste5, einkaufsliste_final`**

Game API

- Die Game API ist im Package **`javax.microedition.lcdui.game`** enthalten und besteht lediglich aus fünf Klassen:
  - **`GameCanvas`**:
    - Enthält einen Puffer in dem alle Rendering-Operationen stattfinden, erst wenn das Bild fertig berechnet ist wird es auf dem Device angezeigt (*Doublebuffering*)
    - Zusätzlich kann man ermitteln, welche Tasten gedrückt wurden
  - **`Layer`**: Abstrakte Klasse
  - **`Sprite`**: Ein (animiertes) Bild
  - **`TiledLayer`**: Eine Art Textur
  - **`LayerManager`**: Verwaltet einzelne Layer und setzt einen sichtbaren Bereich

GameCanvas

- Mit der Methode **`getKeyStates()`** kann abgefragt werden, welche Taste des Geräts gedrückt wurde

```
int keyStates = getKeyStates();
if ((keyStates & GameCanvas.LEFT_PRESSED) != 0)
    // do something for left-key
```

- In den Graphischen Kontext kann gezeichnet werden:

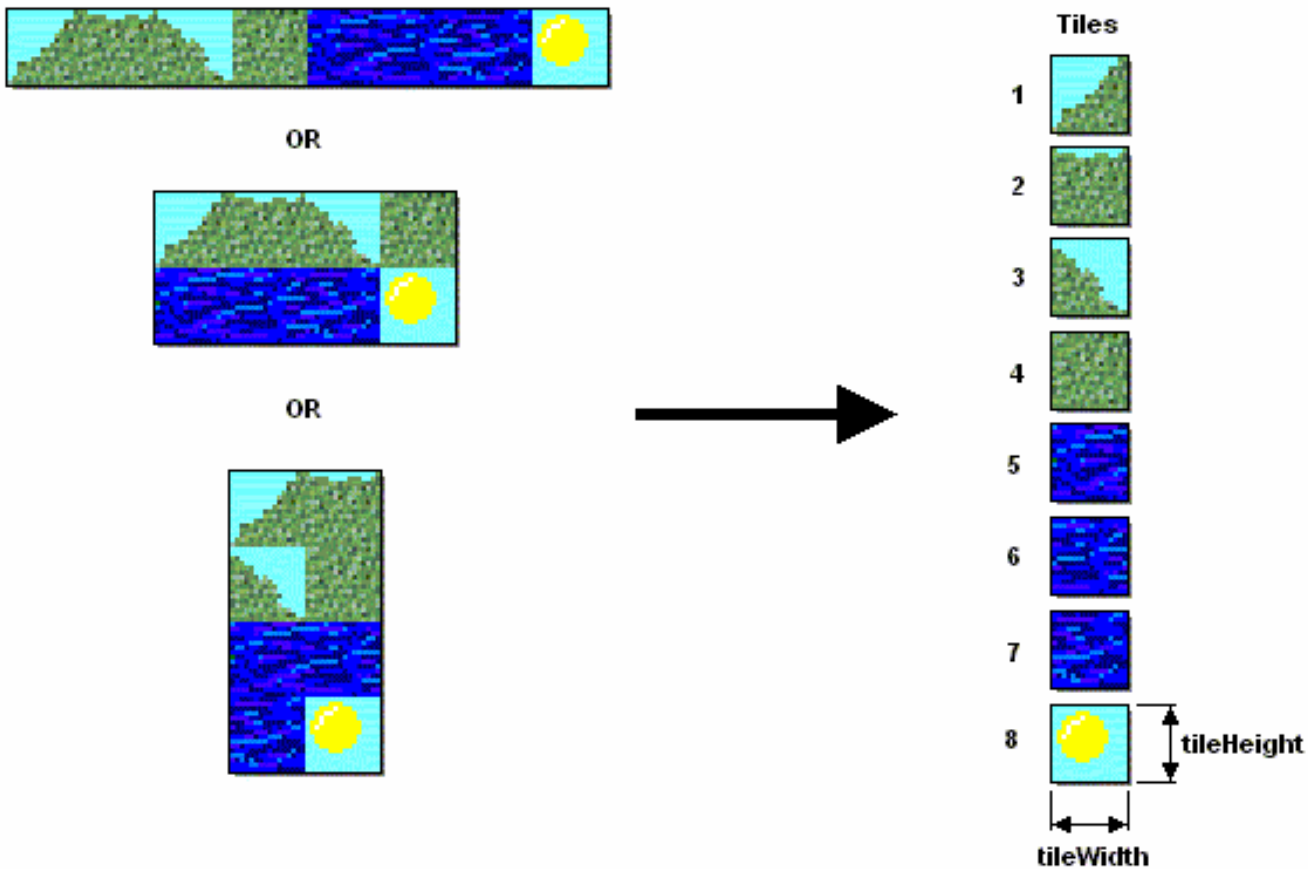
```
Graphics g = getGraphics();
g.setColor(0xffffff);
g.fillRect(0, 0, width, height);
flushGraphics();
```

- Das Bild ist erst sichtbar, wenn **`flushGraphics`** aufgerufen wird
- Beispiel: **`gameapi1`**

`TiledLayer`

- Einen Hintergrund kann man (unter anderem) mit einem **`TiledLayer`**-Objekt erzeugen
- Dazu braucht man zunächst eine Ausgangsgrafik
- Diese Grafik wird in Einzelteile zerlegt
- Dazu wird beim Konstruktor die Breite und Höhe in Anzahl Einzelbilder, die Grafik selbst und die Größe der Einzelbilder mitgegeben
- Dann wird zur Hilfe ein **`int`**-Array definiert, welches die Lage der Einzelbilder zueinander angibt

TiledLayer




```
tiledLayer = new TiledLayer(10, 10, image, 32, 35);
```

Quelle: sun.com

TiledLayer

- Im Folgenden muss die Position eines jeden Teiles angegeben werden:

```
int[] map = { 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
              0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
              0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
              0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
              0,  0,  0,  0,  8,  0,  0,  0,  0,  0,
              0,  0,  0,  0,  1,  2,  3,  0,  0,  0,
              0,  0,  0,  1,  4,  4,  4,  3,  0,  0,
              0,  0,  1,  4,  4,  4,  4,  4,  3,  0,
              0,  1,  4,  4,  4,  4,  4,  4,  4,  3,
              5,  6,  7,  7,  6,  5,  6,  5,  6,  7};
for (int i = 0; i < map.length; i++) {
     int column = i % 10;
     int row = (i - column) / 10;
     tiledLayer.setCell(column, row, map[i]);
}
```

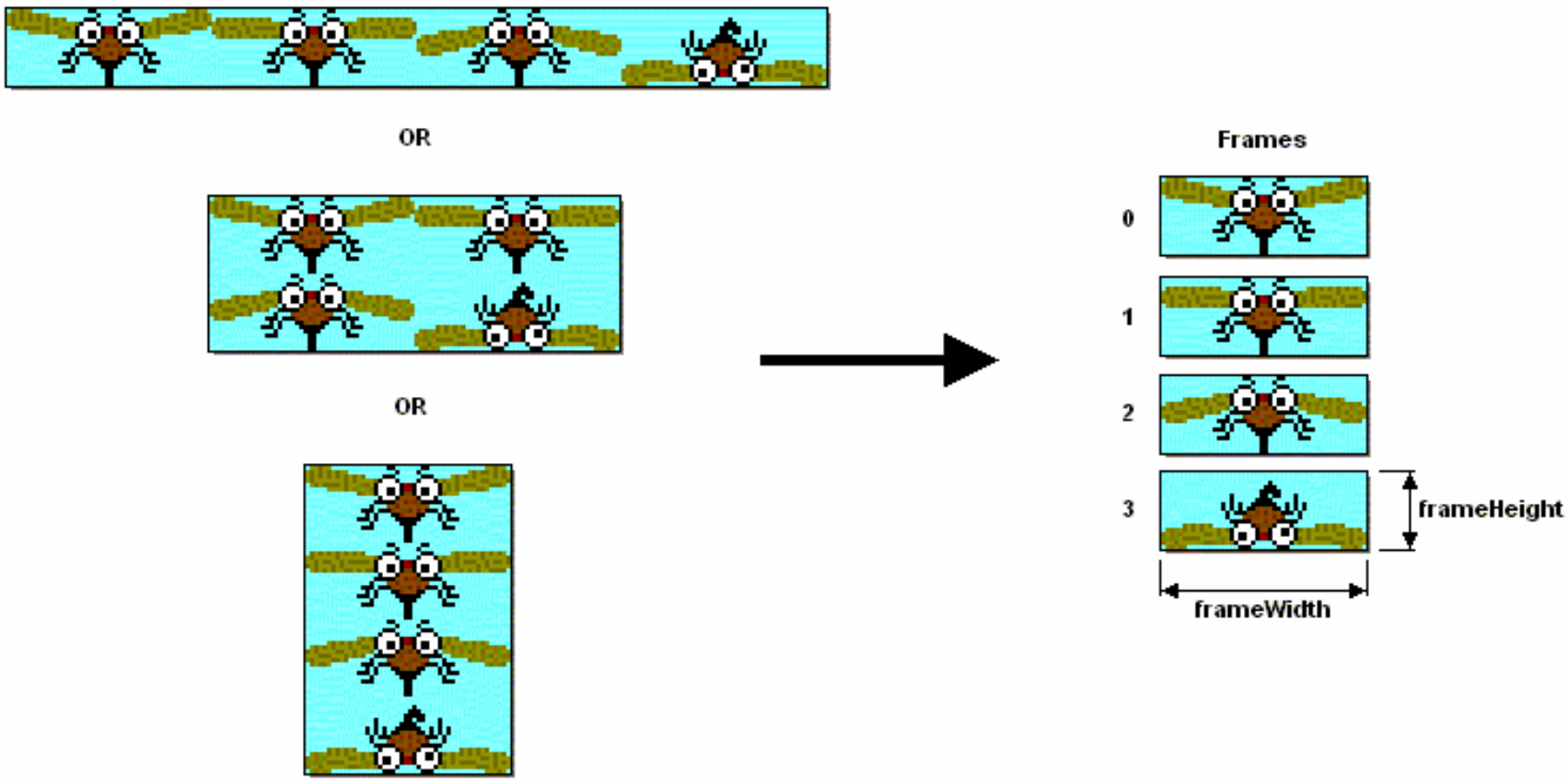
`TiledLayer`

- Ein **`TiledLayer`** kann mit der Methode **`paint(Graphics g)`** gezeichnet werden
- Es zeichnet sich dann in den graphischen Kontext **`g`**
- Am besten wird dies in die **`paint`** Methode der Unterklasse des **`GameCanvas`** integriert
- Mit der Methode **`setCell(int col, int row, int tileIndex)`** können zur Laufzeit einzelne Grafiken ausgetauscht werden
- Beispiel: **`gameapi2`**

`Sprite`

- Ein Sprite dient zur Darstellung eines animierten grafischen Objektes
- Ein Sprite verfügt über eine Collision-Detection
- Auch hier wird eine Grafik definiert, in der verschiedene Teile vorhanden sind
- Wie beim **`TiledLayer`** wird die Grafik in Einzelbilder zerlegt
- Die Einzelbilder können in einer **`paint`**-Methode gezeichnet werden
- Dabei kann man sich entscheiden welches Bild dargestellt werden soll

Sprite



Quelle: sun.com

`Sprite`

- Zu beachten ist, dass ein Sprite (wie die anderen gezeichneten Objekte auch) nicht gelöscht wird
- Daher ist der Bereich eines **`Sprite`**-Objektes immer neu zu zeichnen, wenn das **`Sprite`** wegbewegt wurde
- Die Position eines **`Sprite`** kann sehr einfach gesetzt werden:
  `sprite.setPosition(mX, mY);`
- Beispiel: **`gameapi3`**

Die letzte Folie …

Viel Erfolg bei der Klausur und den kommenden Prüfungen

Auf Wiedersehen!!!!