Informatik B

Vorlesung 22

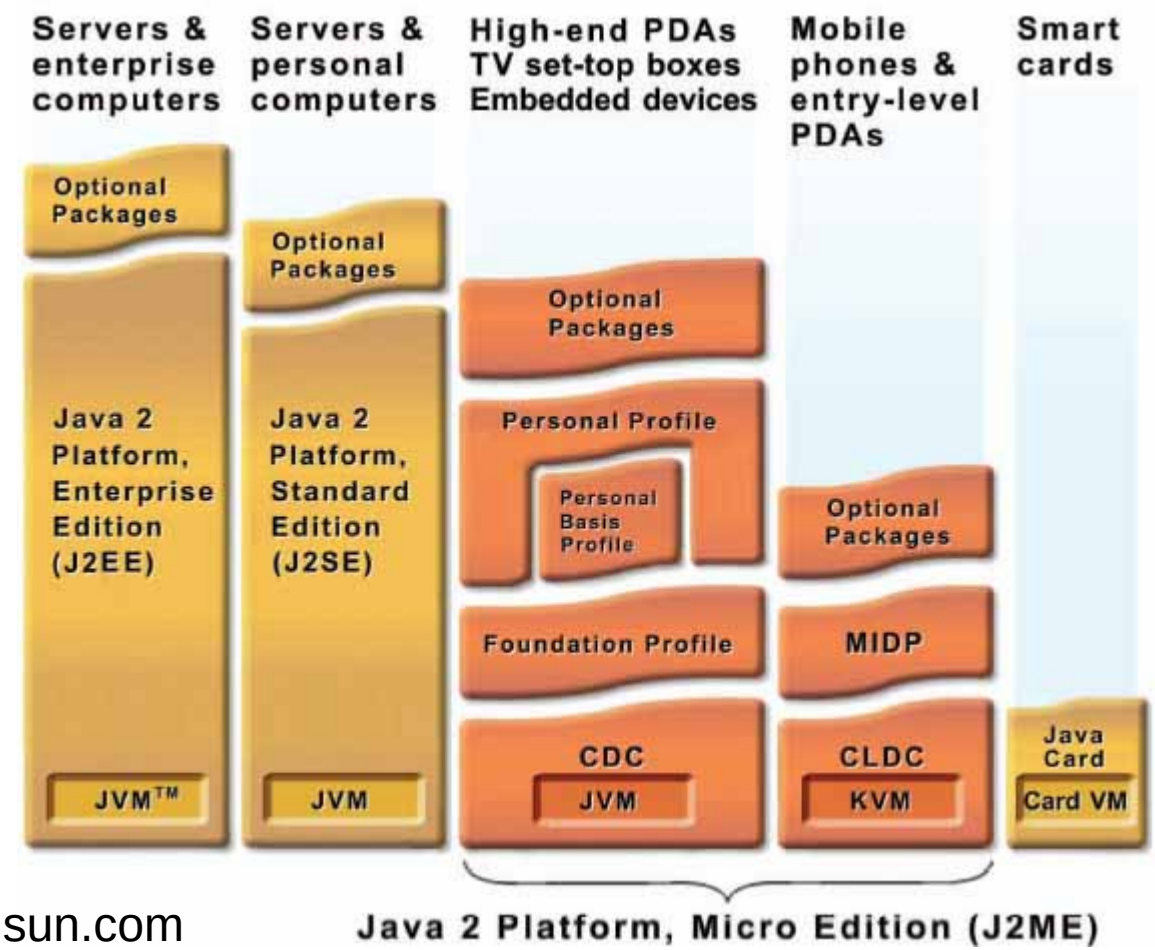
Evaluation, J2ME

Evaluation

J2ME

- J2ME ist die Abkürzung von Java2 Micro Edition
- Das Einsatzgebiet von J2ME ist in den Bereichen:
  - Set-Top-Box
  - Videotelefone
  - PDA
  - Smartphone
  - Pager
  - Mobiltelefon

Einordnung J2ME



Quelle: sun.com

J2ME

- Die unterstützten Endgeräte lassen sich in zwei Gruppen unterteilen:
  - Gemeinsam genutzte stationäre Geräte
  - Persönliche, mobile, vernetzte Geräte
- J2ME ist:
  - Objektorientiert
  - Portabel
  - Sicher (Sandboxkonzept)

J2ME vs. Native

- Viele mobile Endgeräte lassen sich auch mittels plattformspezifischer Bibliotheken in C++ implementieren
- J2ME deckt meist nur einen Teil der Funktionalität des Endgerätes ab
- J2ME-Programme sind meist nicht so effizient, wie nativ Compilierte Programme
- J2ME wird von den meisten Geräteherstellern unterstützt
- J2ME ist offen für (herstellerspezifische) Erweiterungen
- J2ME ist robuster und weniger fehleranfällig als C++
- Java-Entwickler können sich in J2ME schnell einarbeiten

J2ME

- J2ME ist die am schwierigsten zu durchschauende Edition
  - Vorhandene Erfahrungen mit J2SE können nicht komplett übernommen werden
  - J2ME und J2SE haben in Teilen unterschiedliche Philosophien
  - J2ME bietet keine komplett einheitliche Spezifikation

J2ME Architektur

- J2ME soll ein breites Feld von Geräten abdecken
- Ein einheitliches Softwareprodukt ist so gut wie nicht erreichbar
- J2ME definiert daher verschiedene Spezifikationen
- Die Spezifikationen sind für verschiedene Endgeräte ausgelegt
- J2ME ist in drei Schichten aufgebaut:
  - Konfiguration
  - Profile
  - Optionale Pakete

Konfiguration

- Eine Konfiguration definiert den Mindeststandard an Plattformfunktionalität
- Durch eine Konfiguration wird eine Geräteklasse definiert
- Die Hersteller müssen den vollen Umfang der Konfiguration implementieren
- Eine Konfiguration kann eine Teilmenge von J2ME sein

Konfiguration

- Eine Konfiguration definiert:
  - Die unterstützten Sprachmittel von Java
  - Den Funktionsumfang der JVM
  - Die nutzbaren Klassenbibliotheken
- Es gibt zur Zeit 2 Konfigurationen:
  - Connected, Limited Device Configuration (CLDC), z.B. für Handys
  - Connected Device Configuration (CDC), z.B. für Set-Top-Boxen

Hardwareanforderungen

CDC

- mit grafischer Oberfläche
- Speicher: 2-16 MB
  - min. 512kB ROM
  - min. 256kB RAM
- Komplette Implementierung der JVM
- Grosse Bandbreite für die Datenkommunikation
- TCP/IP (verschiedene Verbindungstypen)
- permanente Verbindung

CLDC

- Minimale grafische Oberfläche
- Speicher: 160-512kB
- min. 128 kB ROM für JavaVM und CLDC-Libraries
- min. 32 kB for Java runtime and objects
- Beschränkte VM, z.B: KVM
- Geringe Bandbreite für die Datenkommunikation
- normalerweise kein TCP/IP
- keine permanente Verbindung
- Ausgelegt für Geräte mit geringem Stromverbrauch

Profile

- Ein Profil erweitert eine Konfiguration um spezielle Leistungsmerkmale
- Ein Profil umfasst eine Klassenbibliothek, in der die Leistungsmerkmale implementiert sind
- Im Gegensatz zu einer Konfiguration müssen nicht alle Vorgaben implementiert werden
- Ein wichtiges Profil ist das *Mobile Information Device Profile* (*MIDP*)
- MIDP liegt in der aktuellen Version 2.0 vor

MIDP

- MIDP ist das Mobile Information Device Profile
- Es gibt die Versionen MIDP1.0 und MIDP2.0
- Beide Versionen kommen mit der CLDC1.0 und CLDC 1.1 als Unterbau zurecht
- Die Klassenbibliothek ist als Ergänzung zu CLDC zu verstehen
- Anwendungen für das MIDP heißen MIDlets
- Ein MIDlet stammt von der abstrakten Klasse **`javax.microedition.midlet.MIDlet`** ab

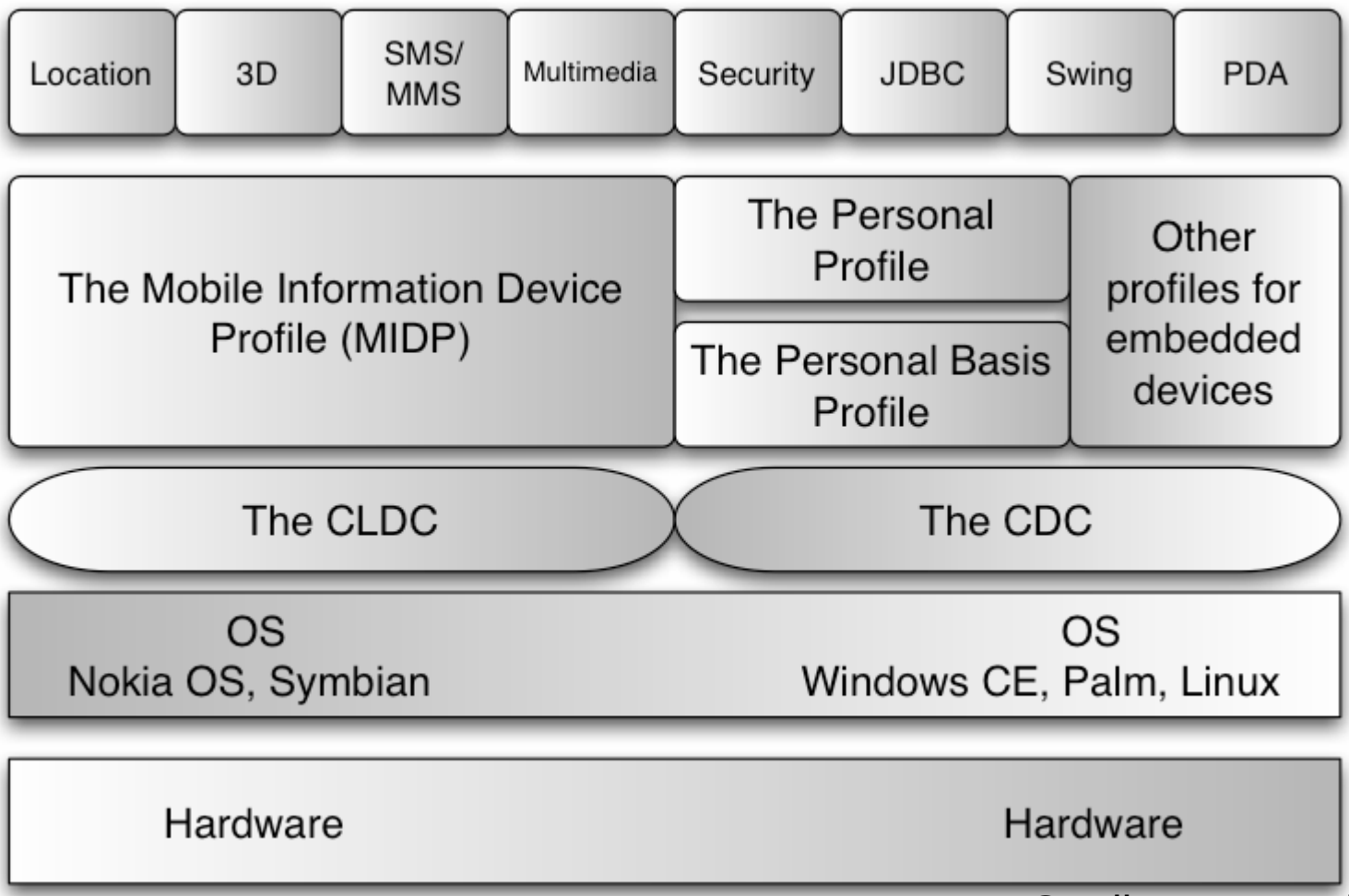
MIDP

- Das API bietet:
  - Steuerung des Applikationszyklus
  - Bedienoberflächen
  - 2D-Spiele
  - Medienverarbeitung
  - Kommunikation
  - Verwaltung persistenter Daten

Optionale Pakete

- Optionale Klassenpakete gehen über die Funktionalität eines Profils hinaus
- Diese Pakete werden im *Java Community Process Program* (*JCP*) definiert
- JCP ist ein Zusammenschluss von Einzelpersonen, Herstellern und Organisationen
- Optionale Pakete müssen nicht implementiert sein
- Daraus folgt: Durch die Nutzung optionaler Pakete sinkt die Portabilität einer Applikation

J2ME



Quelle: Javaworld.com

KVM (Kilobyte Virtual Machine)

- Die KVM soll die essentiellen Eigenschaften der Programmiersprache unterstützen
- Sie muss mit 160 bis 512kB Speicher auskommen
- Die KVM stellt eine stark eingeschränkte JVM dar
- Daher sind **`.class`** Dateien nicht ohne weiteres portierbar
- Ein spezieller Compiler existiert nicht

KVM (Kilobyte Virtual Machine)

- In der CLDC 1.0 gibt es folgende Einschränkungen:
  - Keine Fließkomma-Zahlen
  - Keine native Methoden
  - Kein Reflection (=> keine Serialisierung)
  - Threads werden unterstützt, aber keine Threadgruppen oder Daemon-Threads
  - Der Garbage-Collector führt keine *finalization* durch (=> Objekt hat auch keine **`finalize()`** Methode)
  - Eingeschränkte Exceptions

Bytecode-Verifikation

- Da der Compiler der J2SE verwendet wird, aber die KVM starken Einschränkungen unterliegt, muss der Bytecode vor Ausführung getestet werden
- In der KVM stehen für den Bytecode-Verifikator lediglich 10kB zur Verfügung
- Daher müssen ausgiebige Tests nach dem Compilieren durch einen Präverifikator durchgeführt werden
  1. Der Präverifikator überprüft die **`.class`** Datei und erzeugt eine **`.class`** Datei, die sich zur Laufzeit einfach testen lässt; die Präverifikation geschieht in der Entwicklungsumgebung
  2. Der Laufzeitverifikator erhält eine präverifizierte **`.class`** Datei und überprüft den Bytecode vor der Ausführung
- Der Laufzeitverifikator ist wichtig, da er sicherstellt, dass nur auf gültigen Speicher zugegriffen wird

MIDlet-Suite

- Die Programme werden über eine MIDlet-Suite (entspricht einem JAR-File) auf das mobile Endgerät gebracht
- Eine MIDlet-Suite besteht aus einem oder mehreren MIDlets
- Dazu werden die präverifizierten **`.class`**-Dateien (ggf. weitere Ressourcen) in ein JAR-File gepackt
- Im JAR-File muss ein Manifest enthalten sein, in dem steht, welches Profil verwendet wird

MIDlet-Suite

- Beispiel einer Manifest-Datei:

```
MIDlet-Name: My MIDlet-Suite
MIDlet-Version: 1.0
MIDlet-Vendor: UniOS
MicroEdition-Profile: MIDP-2.0
MicroEdition-Configuration: CLDC-1.0
MIDlet-1: Hallo Welt MIDlet,
          /icons/myicon.png,
          package.HelloWorld
```

Signierung

- Mit MIDP2.0 wurden Trusted MIDlet-Suites eingeführt
- Manche Aktionen (Netzwerkaufbau,…) sind nur signierten MIDlet-Suites vorbehalten
- Dazu wird das JadTool verwendet
- Das Tool ist im WTK (Wireless Toolkit) enthalten
- Bei der Signierung ist es teilweise notwendig, eine Signatur des Herstellers des mobilen Gerätes einzubinden
- Dies ist möglich, indem ein WTK des entsprechenden Herstellers verwendet wird

Entwickeln mit Eclipse

- Eclipse kann einem viele Arbeitsschritte abnehmen
- Dazu gibt es das EclipseME-Plugin: http://www.eclipseme.org
- Dafür ist die Installation eines WTK notwendig
  - Sun: http://developers.sun.com/mobility/index.html
  - Nokia: http://www.forum.nokia.com/
- Das WTK sollte zuerst installiert werden, danach wird EclipseME installiert

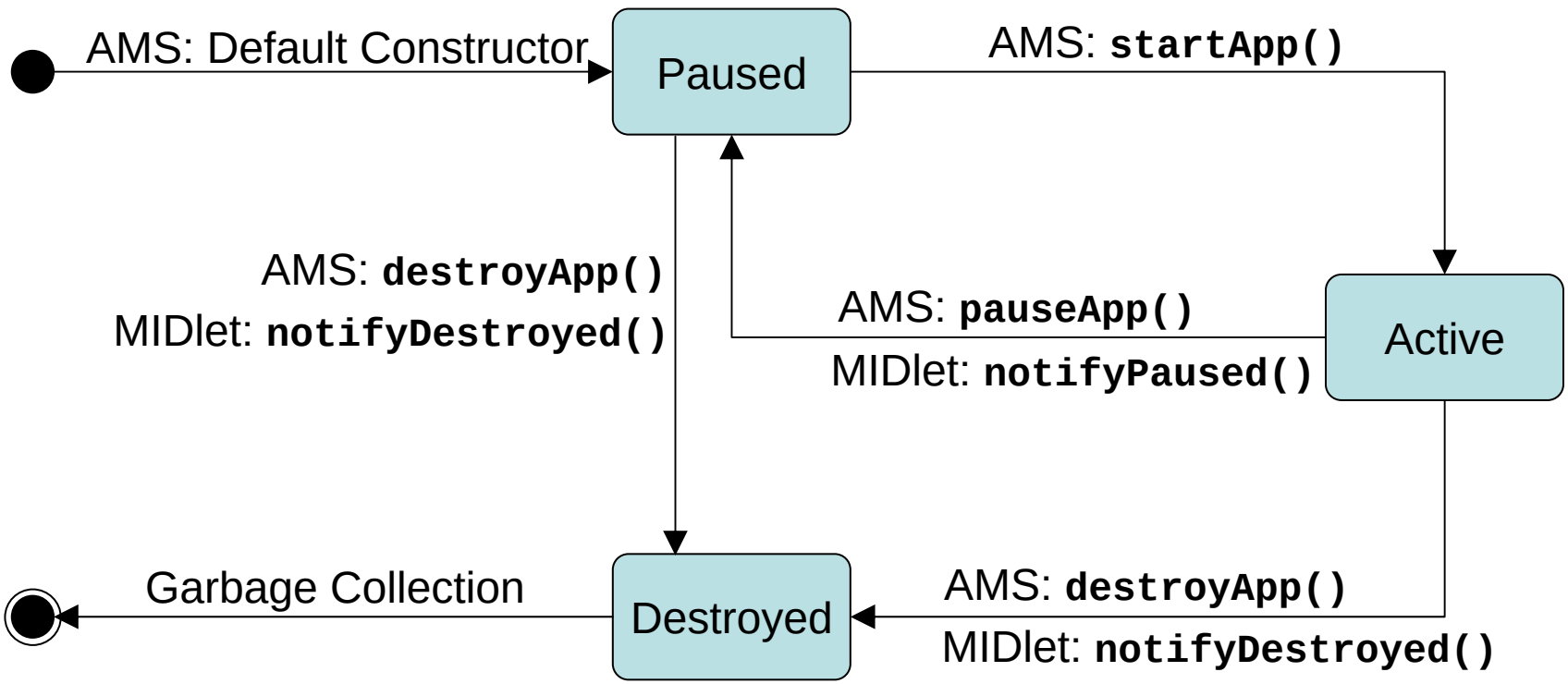
Ein erstes MIDlet

- Ein MIDlet stammt von der abstrakten Klasse **`javax.microedition.midlet.MIDlet`** ab
- Implementiert werden müssen die Methoden:
  - **`destroyApp(boolean arg0)`**
  - **`void pauseApp()`**
  - **`void startApp()`**
- Beispiel: **`helloworld1`**

Zyklen eines MIDlet

- *Paused*:
  - Das MIDlet ist initialisiert, hat aber keine Möglichkeit Ausgaben auf das Display zu schreiben
  - In diesem Zustand sollte das MIDlet keine rechenintensiven Dinge durchführen
  - Das MIDlet sollte Ressourcen (Netzverbindungen) freigeben
- *Active*:
  - Das MIDlet läuft im Vordergrund und kann auf das Display Ausgaben tätigen
- *Destroyed*:
  - Das MIDlet wird beendet und es können ggf. noch Ressourcen freiggeben werden

Zyklen eines MIDlet



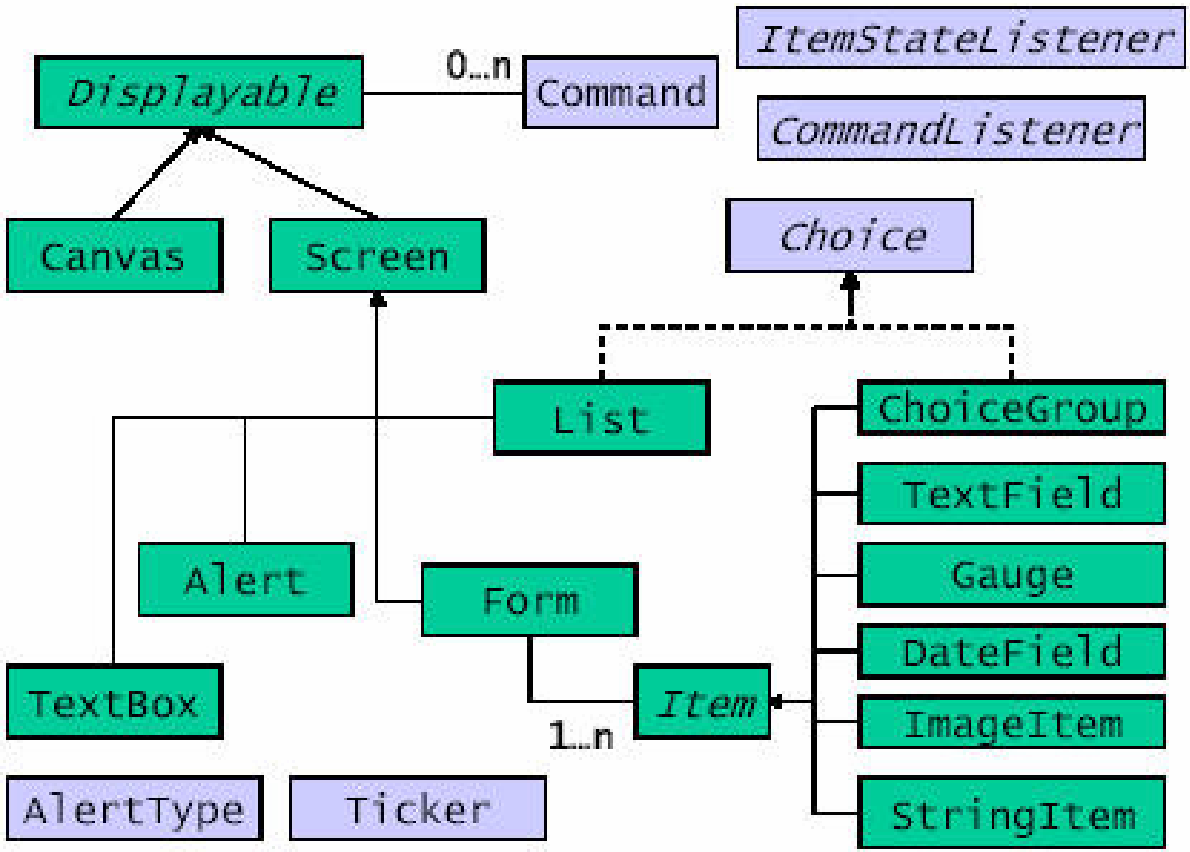
AMS = Application Management Software

Quelle: Java2ME, dpunkt Verlag

Displayable

- Um Informationen auf dem Handy darzustellen, muss ähnlich wie bei der J2SE GUI Programmierung eine Oberfläche erstellt werden
- Die Basisklasse von UI-Komponenten ist **`Displayable`**
- Die Klassen unterteilen sich in High- und Lowlevel-Klassen
- Highlevel-Komponenten erben von der Klasse **`Screen`**
- Lowlevel-Komponenten erben von der Klasse **`Canvas`**

Displayable



`Displayable`

- Die statische Methode **`Display.getDisplay(MIDlet m)`** liefert ein **`Display`**-Objekt für eine MIDlet-Instanz
- Soll ein **`Displayable`**-Objekt sichtbar gemacht werden geschieht dies über die Instanz-Methode **`setCurrent(Displayable d)`** der Klasse **`Display`**
- **`setCurrent`** ersetzt das vorher gesetzte **`Displayable`**, das neue muss aber nicht sofort angezeigt werden
- Vor der Anzeige eines **`Displayable`** kann noch ein Alert-Fenster eingeblendet werden
- Beispiel: **`helloworld2`**, **`helloworld3`**

Programmstruktur

- UI-Komponenten können jederzeit erzeugt werden
- Sollen UI-Komponenten direkt nach dem Starten sichtbar sein, sollten diese im Konstruktor angelegt werden
- Zeitaufwändige Initialisierungen sollten nicht in der Methode **`startApp()`** durchgeführt werden
- Während das MIDlet läuft kann es sein, dass der Bildschirm freigegeben werden muss (z.B. gehen Anrufe immer vor)
- Beim reaktivieren des MIDlets wird **`startApp()`** aufgerufen, daher sollte **`startApp()`** zwischen erstem und nachfolgendem Aufruf unterscheiden können, damit beim wiederholten mal keine Ressourcen doppelt geholt/erzeugt werden
- Beispiel: **`helloworld4`**

Kommandos

- Wie bei Swing oder AWT gibt es auch bei MIDlets die Möglichkeit auf Aktionen zu reagieren
- Dazu dienen Instanzen der Klasse **`Command`**
- Kommandos beinhalten ausschließlich die Informationen zur Darstellung in der Bedienoberfläche
- Ein **`Command`** Objekt erhält
  - Ein Label (Text)
  - Die Kommandoart
  - Eine Priorität

Kommando

- Es gibt acht Kommandotypen:
  - **`Command.BACK`**: zum vorherigen Dialog
  - **`Command.OK`**: Positive Antwort
  - **`Command.CANCEL`**: Negative Antwort
  - **`Command.EXIT`**: Beenden
  - **`Command.HELP`**: Hilfe anfordern
  - **`Command.STOP`**: Vorgang unterbrechen
  - **`Command.ITEM`**: Allg. Item-Kommando
  - **`Command.SCREEN`**: Allg. Screen-Kommando

Kommandos

- Kommandos lassen sich mit **`addCommand(Command c)`** einem **`Displayable`** zuordnen
- Zusätzlich kann einem **`Displayable`** mit **`setCommandListener()`** ein **`CommandListener`**-Objekt zuordnen
- Ein **`Displayable`** kann nur einen Listener erhalten
- Daher wird innerhalb des Listeners unterschieden, welches Kommando ausgewählt wurde

Kommandos

- Der Listener **`CommandListener`** enthält nur die Methode **`commandAction(Command c, Displayable d)`**
- Anhand des übergebenen **`Command`**-Objektes muss innerhalb der Methode entschieden werden, welche Aktion ausgeführt werden soll
- **`d`** ist das ereignisauslösende **`Displayable`**-Objekt
- Beispiel: **`helloworld5`**

Zusammenfassung

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

Ausblick

- J2ME
  - Entwickeln von UI
  - Abspeichern/Laden von Daten
  - Beispiel: Einkaufsliste
  - Game API