Mobile Informationssysteme II

Peter Becker
Hochschule Bonn-Rhein-Sieg
Fachbereich Informatik
peter.becker@h-brs.de

Vorlesung Wintersemester 2010/11

Allgemeines zur Vorlesung

- Homepage zur Veranstaltung:

  `http://www2.inf.h-brs.de/~pbecke2m/mobis2/`

- Die Folien zur Vorlesung (Skript) stehen auf der Homepage vor der Vorlesung zur Verfügung.

- Format: PDF, einseitig

- Folien sind knapp, hoher Selbstlernanteil

Übungen

- Beginn: 5. Oktober
- Raum C 175
- Programmieraufgaben, Literaturstudium, etc.
- Bearbeitungszeit: abhängig von den Aufgaben, i.d.R. ein bis zwei Wochen
- Sie bekommen Zugriff auf das Labor Wissens- und Informationsmanagement. ☞ `ux-2e00.inf.fh-bonn-rhein-sieg.de`

Lernziele

- Erfahrungen mit der Erfassung und Verarbeitung von Sensordaten erlangen,
- Techniken, die Android zur Ausführung von Aktivitäten im Hintergrund anbietet, einsetzen können,
- Aufbau und Funktionsweise von Location-Based Services kennenlernen,
- Eigenschaften von Adhoc-Netzen kennen und mobile Geräte in solche Netze einbinden können,
- selbständig Programme für mobile Geräte entwickeln können und

- Aspekte des modernen Softwaredesigns kennenlernen und anwenden können.

Modulzuordnung und Studienleistung

- Modulgruppe: Wahlpflicht (WP)
  5. Semester
- 2V + 3Ü, 5 Credits
- Prüfung im ersten oder im zweiten Prüfungszeitraum
- Prüfungsform: mündlich
- Prüfung über Inhalte der Vorlesung und Übung

Inhalt

1. Erfassung und Verarbeitung von Sensordaten
2. Hintergrundverarbeitung in Android: Services und Notifications
3. Weiteres zur Netzwerkprogrammieurng, Adhoc-Netze und Bluetooth
4. Standortbezogene Dienste (Location-Based Services)
5. 2D/3D-Grafikprogrammierung
6. Einbindung von native Code

Literatur

Reto Meier
Professional Android 2 Application Development
John Wiley & Sons
2010

Arno Becker, Marcus Pant
Android 2: Grundlagen und Programmierung
Dpunkt Verlag, 2010

Heiko Mosemann
Android. Anwendungen für das Handy-Betriebssystem erfolgreich programmieren
Hanser Fachbuch, 2009

S. Conder, L. Darcey
Android. Wireless Application Development
Addison-Wesley, 2010

1. Erfassung und Verarbeitung von Sensordaten

Lernziele:

- Typische in mobilen Geräten enthaltene Sensorarten kennen,
- Daten von solchen Sensoren empfangen können,
- Einstellmöglichkeiten für Sensoren kennen und
- empfangene Sensordaten verarbeiten können.

Sensor

Ein *Sensor* ist ein technisches Bauteil, das bestimmte physikalische (oder chemische Eigenschaften) seiner Umgebung qualitativ oder als Messgröße quantitativ erfassen kann.

- Physikalische (oder chemische) Eigenschaften werden in ein elektrisches Signal umgewandelt.
- Abtastung des elektrischen Signals und
- Erzeugung eines Datenstroms der Messwerte.
- in der Regel numerische Messwerte

Passive und aktive Sensoren

- Ein *passiver Sensor* benötigt keine Hilfsenergie; auf Basis des technischen Prinzips wird ein elektrisches Signal erzeugt.
  Beispiel: dynamisches Mikrofon

- Bei einem *aktiven Sensor* wird Hilfsenergie benötigt. Die Messung erfolgt, indem physikalische Eigenschaften eine Veränderung der Parameter von passiven Bauteilen bewirken.
  Beispiel: Widerstandsthermometer

Arten von Sensoren in mobilen Geräten

- Beschleunigungssensor
- Lagesensor (Gyroskop)
- Lichtsensor
- Entfernungsmesser (Sonar)
- Kompass
- Thermometer

- Manometer, Barometer
- Lokalisierung (GPS)
- prinzipiell auch die Kamera

Anbindung weiterer Sensoren durch Ad-hoc Netzwerke (z.B. Bluetooth) möglich.

Aktoren

Gegenstück zu Sensoren, zur Erzeugung von physikalischen Effekten

In mobilen Geräten:

- Bildschirmanzeige, Views
- Lautsprecher
- Vibrator

Sensor-Manager

Klasse `android.hardware.SensorManager`

- Schnittstelle zu den Sensoren
- Eine Referenz auf den Sensor-Manager erhält man durch:

```
SensorManager sensorManager =
    (SensorManager)getSystemService(Context.SENSOR_SERVICE);
```

- API nicht stabil, unbedingt Online-Dokumentation heranziehen

Die Klasse `Sensor`

Klasse `android.hardware.Sensor`

- definiert Konstanten für Sensorarten
- verschiedene öffentliche Methoden für Informationen über einen Sensor: Messbereich, Auflösung, Stromverbrauch, etc.

  `public float getMaximumRange ()`
  maximum range of the sensor in the sensor's unit.

  `public String getName ()`
  name string of the sensor.

  `public float getPower ()`
  the power in mA used by this sensor while in use.

`public float getResolution ()`
resolution of the sensor in the sensor's unit.

`public int getType ()`
generic type of this sensor.

`public String getVendor ()`
vendor string of this sensor.

`public int getVersion ()`
version of the sensor's module.

Arten von Sensoren

definiert in `android.hardware.Sensor`:

- `TYPE_ALL`

  alle Sensoren

- `TYPE_ACCELEROMETER`

  Beschleunigungssensor in drei Achsen, Einheit m/s^2

- `TYPE_GYROSCOPE`

  Gyroskop

- `TYPE_LIGHT`

  Lichtsensor, einfallendes Licht in Lux

- `TYPE_MAGNETIC_FIELD`

  Erdmagnetfeld in drei Achsen, Einheit sind Mikrotesla

- `TYPE_ORIENTATION`

  Orientierung des Gerätes in Grad entlang drei Achsen; deprecated

- `TYPE_PRESSURE`

  Drucksensor

- `TYPE_PROXIMITY`

  Entfernungsmessung in Meter

- `TYPE_TEMPERATURE`

  Temperaturmessung, Celsius

Ermittlung aller Sensoren eines Typs: In der Klasse SensorManager Methode:

- `List<Sensor> getSensorList(int type)`

Standardsensor eines Typs ermitteln:

- `Sensor getDefaultSensor(int type)`

Sensoreigenschaften

- Genauigkeit

  Wie hoch ist der mittlere Messfehler? `SENSOR_STATUS...`

- Änderungsrate

  Wie häufig können Messwerte aktualisiert werden? `SENSOR_DELAY...`

Vordefinierte Konstanten als Werte für beide Eigenschaften

Genauigkeit

Konstanten in der Klasse `SensorManager`:

- `SENSOR_STATUS_ACCURACY_HIGH`

  Sensor arbeitet mit höchster Genauigkeit.

- `SENSOR_STATUS_ACCURACY_MEDIUM`

  Sensor arbeitet mit mittlerer Genauigkeit, eine Kalibrierung kann die Messgenauigkeit verbessern.

- `SENSOR_STATUS_ACCURACY_LOW`

  Sensor arbeitet mit geringer Genauigkeit, eine Kalibrierung erscheint notwendig.

- `SENSOR_STATUS_UNRELIABLE`

  Sensor liefert unzuverlässige Werte, eine Kalibrierung ist notwendig oder es können keine Sensordaten gelesen werden.

Änderungsrate

Konstanten in der Klasse `SensorManager`:

- `SENSOR_DELAY_FASTEST`

  höchstmögliche Änderungsrate

- `SENSOR_DELAY_GAME`

  Änderungsrate geeignet für Spiele

- `SENSOR_DELAY_NORMAL`

  Standard-Änderungsrate

- `SENSOR_DELAY_UI`

  Änderungsrate geeignet für Benutzerschnittstelle

Sensordaten empfangen (1)

- via Schnittstelle `android.hardware.SensorEventListener`
- Achtung: `SensorListener` ist deprecated!
- `void onSensorChanged (SensorEvent event)`

  Neue Messdaten liegen für einen Sensor vor. Sensor und Werte werden durch ein Objekt der Klasse `android.hardware.SensorEvent` beschrieben.

- `void onAccuracyChanged (Sensor sensor, int accuracy)`

  Die Genauigkeit für den Sensor `sensor` hat sich gändert. `accuracy` ist der neue Genauigkeitswert.

Sensordaten empfangen (2)

Registrierungsmethoden für den `SensorEventListener` in der Klasse `SensorManager`:

- `public boolean registerListener(SensorEventListener listener, Sensor sensor, int rate)`

  `listener` erhält von `sensor` Daten mit Änderungsrate `rate`

  Liefert `true`, wenn der Sensor unterstützt und verfügbar ist.

- `public boolean registerListener (SensorEventListener listener, Sensor sensor, int rate, Handler handler)`

  wie oben, aber die `SensorEvents` werden über `handler` verteilt

- `public void unregisterListener (SensorEventListener listener)`

  `listener` von allen Sensoren abmelden

- `public void unregisterListener (SensorListener listener, Sensor sensor)`

  `listener` von `sensor` abmelden

Sensordaten

Klasse android.hardware.SensorEvent mit `public` Instanzvariablen statt Getter-Methoden:

- `public Sensor sensor`

  Der Sensor, der das Ereigenis ausgelöst hat.

- `public long timestamp`

  Zeit (in Nanosekunden?), zu der das Ereignis eintrat.

- `public final float[] values`

  Werte für die Sensoren, hängen vom Sensortyp ab.

- `public int accuracy`

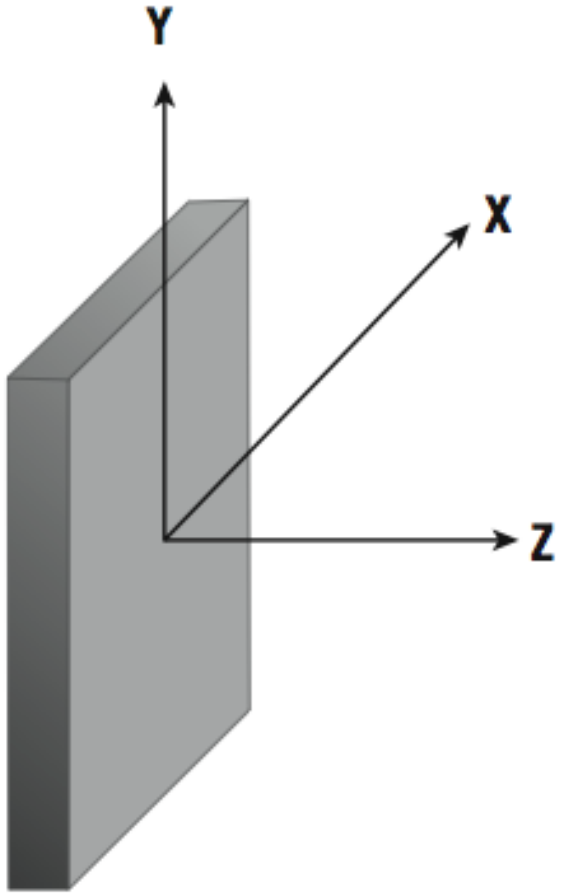
  Genauigkeit

Beschleunigungssensor

- Beschleunigung entspricht der Änderung der Geschwindigkeit pro Zeiteinheit. Erste Ableitung der Funktion $v(t)$.
- Beschleunigungsmesser können nicht zwischen Beschleunigung durch Gravitation oder durch Bewegung unterscheiden. Konsequenz: Beschleunigung von $-9.8 m/s^2$ im Ruhezustand in Richtung der y-Achse der Zeichnung (vertikal).
- Konstante für Erdbeschleunigung:
  `SensorManager.STANDARD_GRAVITY`
- Messung der Beschleunigung erfolgt für drei Achsen.

Ausgehend von Gerät flach auf dem Tisch, Bildschirm nach oben (unterschiedlich zur Zeichnung):

- vertikal: runter (+) / hoch (–)
  x-Achse, `DATA_X=0`
- longitudinal: näher (+) / weiter (–)
  y-Achse, `DATA_Y=1`
- lateral: links (+) / rechts (–)
  z-Achse, `DATA_Z=2`



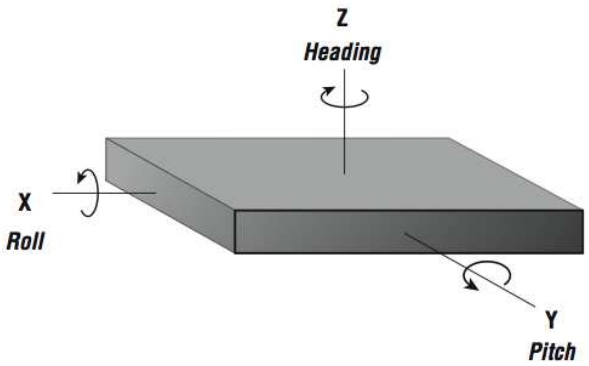
Zugriff auf die Werte im Feld `values` nach Konvention:

- `values[0]: vertikal`
- `values[1]: longitudinal`
- `values[2]: lateral`

Man beachte: Die Konstanten `DATA_X`, `DATA_Y` und `DATA_Z` sind als deprecated gekennzeichnet.

Lagesensor (Orientation)

- Roll, beschreibt ein Kippen des Gerätes nach links (+) oder rechts (–)
- Pitch, beschreibt ein Kippen des Gerätes nach vorne (+) oder hinten (–)
- Heading, Bearing, Rotation um die z-Achse der Zeichnung,
  Heading alleine stellt einen Kompass dar.



Kombination aus Kompass (Heading) und Beschleunigungssensor um Pitch und Roll zu ermitteln.

Achsen sind in Geräten wiederum anders als in der Zeichnung:

- `values[0]: heading`
- `values[1]: pitch`
- `values[2]: roll`

Virtueller Sensor

- Der Orientierungssensor ist in den meisten Geräten ein sogenannter *virtueller Sensor*.
- Die physikalischen Werte (hier Winkel) werden nicht direkt gemessen, sondern aus anderen Sensorwerten durch ein Modell abgeleitet.
- Die Lage wird z.B. aus den Messwerten des Beschleunigungssensors und des Sensors für das Erdmagnetfeld ermittelt.
- Android stellt die Methoden zur Berechnung der Lage öffentlich zur Verfügung.

Lageermittlung in Android

- Die Repräsentation der Lage eines mobilen Gerätes erfolgt mit Hilfe einer sogenannten *Drehmatrix* bzw. *Rotationsmarix*.
- Solch eine Matrix beschreibt eine Drehung in einem euklidischen Raum, hier im $\mathbb{R}^3$.
- Android stellt in der Klasse SensorManager eine Methode bereit, um eine Rotationsmatrix zu berechnen, die die aktuelle Lage des Gerätes beschreibt.
- Eingabe: Messwerte des Beschleunigungs- und des Erdmagnetfeldsensors

- Mit Hilfe dieser Rotationsmatrix kann über eine weitere Methode die Lage des Gerätes, beschrieben durch Winkel (Heading, Pitch, Roll), ermittelt werden.

Drehung/Rotation

Eine *Drehung* ist eine lineare Abbildung, die längentreu, winkeltreu und orientierungserhaltend ist.

- Kongruenzabbildung: Form und Größe werden nicht verändert
- Affine Abbildung: Kollinearität, Parallelität und Längenverhältnisse bleiben erhalten

Drehmatrix

Im $\mathbb{R}^2$ wird eine Drehung um den Ursprung mit Winkel α durch eine Drehmatrix beschrieben:

$$\mathrm{R}(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

Drehmatrizen haben eine Reihe von interessanten Eigenschaften (allgemein, nicht nur im $\mathbb{R}^2$)

Eigenschaften von Drehmatrixzen

- Die Spalten einer Drehmatrix bilden ein Orthogonalsystem, d.h., sie stehen paarweise senkrecht aufeinander.
- Zwei Vektoren x und y sind orthogonal, wenn

$$< x, y >= \sum_{i=1}^{n} x_i \cdot y_i = 0$$

  gilt.
- Die Länge der Spaltenvektoren ist gleich 1, dies entspricht der Längentreue.

- Insgesamt bilden die Spalten damit ein Orthonormalsystem.
- $\det \mathrm{R}(\alpha) = 1$, dies entspricht der Orientierungserhaltung.
- Für die Inverse einer Drehmatrix gilt:

$$\mathrm{R}^{-1}(\alpha) = (\mathrm{R}(\alpha))^{\mathsf{T}}$$

Damit im $\mathbb{R}^2$:

$$\mathrm{R}^{-1}(\alpha) = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$$

Rotationsmatrizen in 3D

- Während in 2D eine Drehung durch einen Winkel beschrieben wird, benötigen wir für 3D drei Winkel, da wir insgesamt drei Paare von Achsen haben.

- Diese Winkel werden auch als Eulersche Winkel bezeichnet.

- Wir können uns die Lage eines Körpers vorstellen, als drei nacheinander ausgeführte Drehungen um verschiedene Achsen.

- Jede Achsendrehung hat ihre eigene Rotationsmatrix.

- Drehung um die x-Achse um den Winkel γ:

$$R_x(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix}$$

- Drehung um die y-Achse um den Winkel β:

$$R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$$

- Drehung um die z-Achse um den Winkel α:

$$\mathrm{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Durch Multiplikation der Matrizen entsteht die Gesamtrotationsmatrix R, z.B.

$$\mathrm{R} = \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{pmatrix}$$

Ermittlung der Drehwinkel

- Hat man eine Rotationsmatrix R, kann man aus den r_{ij} die entsprechenden Winkel ermitteln.
- Die Auflösung kann auf verschiedene Weise erfolgen, man sollte auf numerische Stabilität achten.
- Beispiel:

$$\begin{aligned} \beta &= \arcsin(-\mathrm{r}_{31}) \\ \alpha &= \arcsin(\frac{\mathrm{r}_{21}}{\cos\beta}) \\ \gamma &= \arcsin(\frac{\mathrm{r}_{32}}{\cos\beta}) \end{aligned}$$

- Es sind aber auch andere Formeln möglich, siehe Literatur.

Ermittlung der Rotationsmatrix

- Lage des Koordinatensystems: x-Achse nach Osten, y-Achse nach Norden, z-Achse zum Himmel
- Rotationsmatrix R ergibt sich aus:

$$R \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix}$$

und

$$I \cdot R \cdot \begin{pmatrix} m_x \\ m_y \\ m_z \end{pmatrix} = \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix}$$

- Hierbei sind a_x, a_y, a_z die Beschleunigungsmesswerte in Richtung der drei Achsen und m_x, m_y, m_z die Messwerte für das Erdmagnetfeld.

- I ist eine simple Rotationsmatrix, die die Messwerte des magnetischen Feldes in das gleiche Koordinatensystem überführt, wie das zur Beschleunigungsmessung.

Android-Methoden

- `public static boolean getRotationMatrix (float[] R, float[] I, float[] gravity, float[] geomagnetic)` Muss zuerst aufgerufen werden. R wird vorher allokiert (als Feld) und die Werte werden dort zeilenweise abgelegt.

  `gravity` und `geomagnetic` sind die Felder mit Messwerten von den Sensoren.

- `public static float[] getOrientation (float[] R, float[] values)`

  Zur Ermittlung der Winkel (im Bogenmaß), das Feld `values` wird gefüllt.

2. Hintergrundverarbeitung in Android: Services und Notifications

Übersicht:

- In Mobis 1: Threads; hier genauerer Blick auf Services
- Verschiedene Nutzungsszenarien für Services
- Kommunikation mit Services
- Möglichkeiten für Benachrichtigungen an den Nutzer aus dem Hintergrund
- Remote-Services

Service

- *Services* sind Teil des Komponentenmodells von Android.
- Sie kapseln Programmlogik, die unabhängig von einer Benutzerschnittstelle ausgeführt werden kann.
- Sie haben ihr eigenes Lebenszyklusmodell und werden anders behandelt als Activities.
- Sie können Schnittstellen für den Zugriff durch andere Komponenten anbieten.

Vorteile von Services

- automatischer Restart
- höhere Priorität als nicht sichtbare Activities
- Priorität kann heraufgesetzt werden auf die sichtbarer Activities

Arten von Services

- Local Service

  Service, der im Prozess der Anwendung läuft, die ihn gestartet hat.

- Remote Service

  Ein Service der in einem eigenen Prozess läuft.

  Vorteil: Stabilität
  Nachteil: Erhöhte Kommunikationskosten, Interprozesskommunikation (IPC) notwendig, typischerweise in Form von Remote Procedure Calls (RPC)

Wir betrachten zunächst nur Local Services.

Deklaration von Services im Manifest (1)

- Service muss als Komponente im Manifest deklariert werden, auf gleicher Ebene wie Activities.
- Für Local Services genügt die Angabe des Namens der Klasse, die den Service implementiert.

```
<service android:name=".MeinTollerService"/>
```

Deklaration von Services im Manifest (2)

- Für eine Remote Service muss man zusätzlich einen Prozessnamen angeben:

```
<service android:name=".MeinTollerService"
         android:process=":MeinServiceProzess"/>
```

  Prozessname beginnt mit Großbuchstabe:

  – Nur die startende Anwendung kann der Service verwenden,
  – Service läuft unter der User-ID der startenden Anwendung

  Prozessname beginnt mit Kleinbuchstabe:

  – Jede Anwendung kann auf Service zugreifen

Services und Threads

- Ein Service sorgt nicht automatisch für Parallelität, seine Lebenszyklusmethoden laufen im Main Thread
- Konsequenz: auch für langlaufende Teile im Service benötigen wir Threads.
- Service hat höhere Priorität, wird nach Möglichkeit nicht unterbrochen und wenn doch u.U. wieder automatisch gestartet.
- Threads dagegen hängen am Prozess und sterben mit dem Prozess.

Szenario 1: Services als Dienstleistung ohne Kommunikation

- Operationen: Starten und Stoppen des Service
- Der Service arbeitet autonom im Hintergrund.
- Ansonsten keine Kommunikation mit dem Service notwendig

Die Klasse Service

Zur Implementierung eines Service müssen wir von der Klasse `android.app.Service` ableiten.

Die wichtige Methoden in diesem Szenario:

- `public void onCreate ()`

  Wird aufgerufen, wenn der Service (nicht das repräsentierende Objekt) erzeugt wird.

- `public int onStartCommand (Intent intent, int flags, int startId)`

  seit Android 2.0: Wird jedesmal aufgerufen, eine andere Komponente den Service starten will.

`intent`: Der Intent, der den Start auslöste, kann zusätzliche Daten beinhalten

`flags`: Flags, die vom Laufzeitsystem gesetzt werden und genauer über den Start informieren

`startId`: Eine ID, die vom Laufzeitsystem vergeben wird.

Der Rückgabewert informiert das Laufzeitsystem über die "Betriebsart" des Service.

- `public void onStart (Intent intent, int startId)`

  vor Android 2.0

- `public abstract IBinder onBind (Intent intent)`

  In diesem Szenario ohne Bedeutung, muss als abstrakte Methoden aber implementiert werden.

- `public void onDestroy ()`

  Wird aufgerufen, wenn der Service beendet wird, weil z.B. alle Klienten den Service nicht mehr benötigen

- `stopSelf()`

  Damit der Service sich selbst anhalten kann.

Alle Methoden werden implizit durch die Laufzeitumgebung aufgerufen, kein expliziter Aufruf!

Betriebsart

Konstanten für die Rückgabe in `onStartCommand()`:

- `START_STICKY`: Für Services, die explizit gestartet und gestoppt werden sollen.

  Automatischer Restart nach Unterbrechung durch das Laufzeitsystem

- `START_NOT_STICKY`: Für Services, die sich nach ausgeführter Arbeit selbst beenden (`stopSelf()`).

  Kein automatischer Restart durch das Laufzeitsystem

- `START_REDELIVER_INTENT`: Für Services, bei denen garantiert werden soll, dass sie ihre Arbeit beenden können

Automatischer Restart bei Unterbrechung vor Aufruf von `stopSelf()`

Flags:

- `START_FLAG_REDELIVER`: Unterbrechung durch das Laufzeitsystem bevor `stopSelf()` aufgerufen wurde

  Intent wird wieder mit übergeben

- `START_FLAG_RETRY`: Unterbrechung durch das Laufzeitsystem bei der Betriebsart `START_STICKY`, Intent wird nicht mit übergeben

Szenario 1: Lebenszyklus

- `onCreate()`: Wird beim Erzeugen des Service aufgerufen und sollte für Initialisierungen verwendet werden.
- `onStartCommand()`: Wenn eine Komponente den Service startet, mehrfacher Aufruf möglich
- `onDestroy()`: Wird aufgerufen, wenn der Service beendet wird.

Szenario 1: Starten und Stoppen

Hierfür stehen Methoden im `Context` zur Verfügung, Beispiel Activity:

- `public abstract ComponentName startService (Intent service)`

  vgl. `startActivity()`, Startet einen Service mit Hilfe eines Intents (explizit oder implizit)

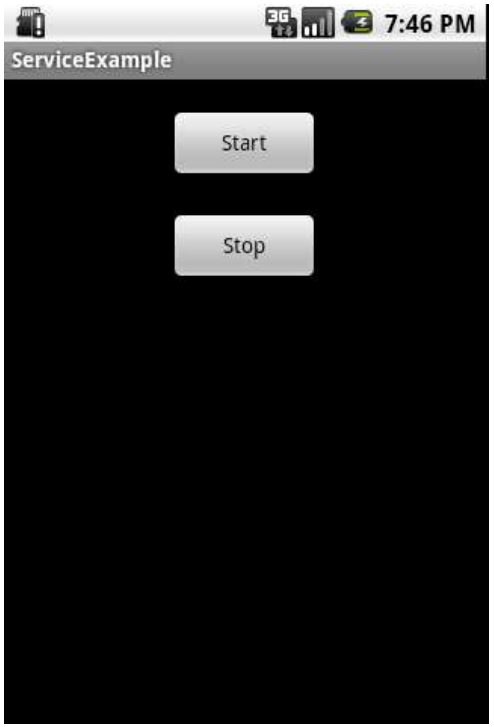
  Liefert `null`, wenn der Service nicht gestartet werden konnte.

- `public abstract boolean stopService (Intent service)`

  Hält den Service an, auch wenn noch andere Komponenten den Service benutzen.

Szenario 1: Beispiel

Beispiel 2.1. *Schreiben in Log alle 5 Sekunden*



Szenario 2: Service mit Callback an andere Komponente

- Wenn im Service ein Ereignis eintritt, über das die Klienten informiert werden sollen, kann man einen Broadcast Intent verschicken.
- Broadcast Intents sind auch selbstdefinierbar und können mit zusätzlichen Daten versehen werden.
- Für die Behandlung eines Intents, müssen wir einen Broadcast Receiver implementieren, hier typischerweise einen dynamischen Broadcast Receiver.

Wiederholung: Dynamischer Broadcast Receiver

☞ siehe Mobile Informationsysteme I

Hier aber:

- Verschicken der Broadcast Intents auf Anwendungsebene
- Definition eigener Broadcast Intents

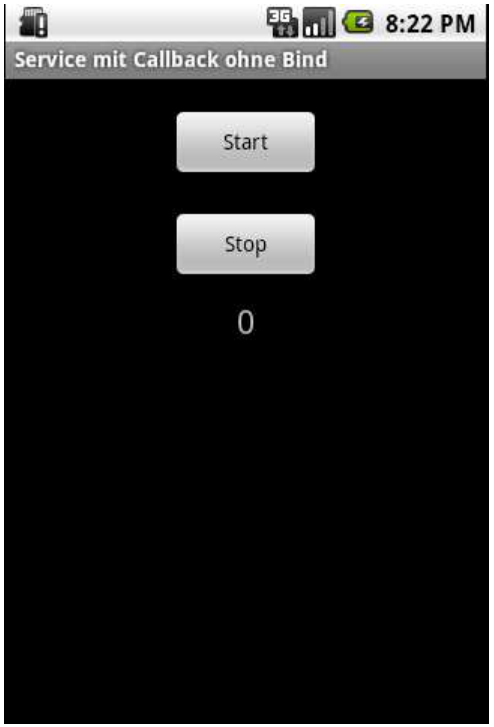
Szenario 2: Anwendungsstruktur

- In der Service-Klasse wird eine öffentliche String-Konstante als Intent-Bezeichner definiert.
- In der Service-Routine der Service-Klasse wird ein Broadcast Intent versendet.
- In der Client-Klasse wird ein Broadcast Receiver (als nested class) definiert und es wird eine Instanz dieser Klasse erzeugt.
- In der Client-Klasse wird der Broadcast Receiver registriert. Hierbei wird der öffentliche Intent-Bezeichner aus der Service-Klasse verwendet.

- Wir starten und stoppen den Service wie bisher üblich.
- Die gewünschten Reaktion auf eintreffende Broadcast Intents definieren wir in `onReceive()` des Broadcast Receivers.
- Da der Broadcast Receiver in der Activity liegt und im Main-Thread läuft, können wir von `onReceive()` aus die GUI-Elemente manipulieren.
- Die Methode `onReceive()` definiert den Lebenszyklus eines Broadcast Receivers. Hier dürfen natürlich auch keine langdauernden Berechnungen stattfinden.

Szenario 2: Beispiel

Beispiel 2.2. *Schreiben in Log alle 5 Sekunden; zusätzlich wird ein Broadcast Intent mit einem Zählerwert versendet, der in einer View angezeigt wird.*



Szenario 3: Service mit erweiterter Schnittstelle

- Ein Service bietet zusätzliche Methoden an, über die sich die Klienten über den Zustand des Service informieren können
- oder weitere Dienstleistungen des Service nutzen können.
- Die zusätzlichen Methoden und Dienstleistungen können mit Hilfe eines sogenannten `Binder`s genutzt werden.
- Auswirkungen auf den Lebenszyklus: Methode `onBind()` des Service ist vernüftig zu implementieren, Rückgabe einer `IBinder`-Instanz.

Szenario 3: Binder

- Eine `Binder`-Instanz stellt zusätzliche Methoden für den (eventuellen remote) Zugriff auf einen Service bereit.
- Klasse `Binder` im Paket `android.os`
- Schnittstelle `IBinder` im Paket `android.os`
- `Binder` implementiert `IBinder`.
- Typisches Vorgehen: Innerhalb der Service-Klasse wird eine öffentliche Binder-Klasse abgeleitet von `Binder` definiert.

- Weiterhin wird in der Service-Klasse eine Instanz der Binder-Klasse angelegt und gespeichert, z.B. in einer privaten Klassenvariablen.

- Die Methode `onBind()` liefert dann die Binder-Instanz als Ergebnis.

Szenario 3: Lebenszyklus

Services mit Binder haben einen anderen Lebenszyklus:

- `onCreate()`: Wie in Szenario 1
- `onBind()`: Wird aufgerufen, wenn sich eine Methode mit dem Service verbunden hat und liefert typischerweise die Binder-Instanz als Ergebnis.
- `onUnbind()`: Wird aufgerufen, wenn alle Komponenten die Verbindung mit dem Service beendet haben. Wird in der Regel nicht überschrieben.
- `onDestroy()`: Wie in Szenario 1

- Anders als in Szenario 1 kann hier ein Service erst dann angehalten werden, wenn keine Komponente mehr eine Verbindung zum Service hat.

- Auch ein `stopService()` hält in diesem Fall den Service nicht an.

Szenario 3: Verbinden und Trennen

- Verbinden und Trennen statt Starten und Stoppen
- Verbinden: Mit Hilfe der Methode

```
public boolean bindService (Intent service,
                            ServiceConnection conn,
                            int flags)
```

  Verbinden (und evtl. Erzeugen) mit Hilfe eines Intents

  Zur Verwaltung der Verbindung wird ein Objekt verwendet, das `ServiceConnection` implementiert.

  Automatisches Starten mit Hilfe des Flags `Context.BIND_AUTO_CREATE`

- Trennen: Mit Hilfe der Methode

```
public abstract void unbindService (ServiceConnection conn)
```

Szenario 3: `ServiceConnection`

Schnittstelle im Paket `android.component`; enthält zwei Methoden:

- `public void onServiceConnected (ComponentName name, IBinder service)`

  Aufruf erfolgt als Folge von `bindService()`;

  `service` ist hier die Binder-Instanz, die der Service in `onBind()` als Ergebnis lieferte.

  typischerweise erfolgt Downcast auf konkreten Typ (öffentliche innere Klasse des Service)

- `public void onServiceDisconnected (ComponentName name)`

  wenn die Verbindung zum Service beendet wurde, z.B. durch ein `unbindService()`

Szenario 3: Anwendungsstruktur

- Schnittstelle für die Methoden definieren, die der Service (via Binder) anbieten soll
- in Service-Klasse: öffentliche innere Binder Klasse, die die Schnittstelle implementiert
- in Service-Klasse: Binder-Instanz wird einmalig angelegt und gespeichert
- in Service-Klasse: `onBind()` liefert Binder-Instanz
- in Klient-Klasse: Implementierung von `ServiceConnection`

- in Klient-Klasse: Instanziierung einer `ServiceConnection`
- in Klient-Klasse: Aufruf von `bindService()` und `unbindService()`
- in Implementierung von `ServiceConnection`: Downcast von `IBinder` auf öffentlichen Binder-Typ, Speicherung des Binders
- in Klient-Klasse: Zugriff auf die Dienste des Service über das Binder-Objekt

Szenario 3: Beispiel

Beispiel 2.3. *Service mit folgenden Diensten:*

- `int generateRandomInt()`

  neue Zufallszahl wird generiert

- `int getLastRandomInt()`

  liefert die zuletzt erzeugte Zufallszahl

- `void setRange(int low, int high)`

  Intervall für die Generierung

Remote Services

- bisher: Alle Services waren lokal, liefen im gleichen Prozess wie die nutzenden Komponenten.
- jetzt: Services in eigenem Prozess. Damit betreten wir den Bereich der verteilten Systeme.
- insbesondere: Interprozesskommunikation (IPC) für Datenaustausch mit Service notwendig,
- aber auch eine Reihe weiterer allgemeiner Techniken verteilter Systeme: IDL, RPC, Marshalling, Asynchronität
- Android bietet kein Remote Method Invocation (RMI) an!

IDL

- Interface Definition Language, also Schnittstellenbeschreibungssprache
- beschreibt remote angebotene Dienste, reine Spezifikationssprache
- Definition u.a. von:
  - Methoden mit Ein- und Ausgabeverhalten
  - Datenstrukturen
  - Asynchronität
- eigene Syntax, eigene Datentypen, aber angelehnt an Implementierungssprachen

- IDL-Compiler zur automatischen Erzeugung von Quelltexten für Hilfsklassen oder -prozeduren, z.B. sogenannte `Stub`-Klassen

- Beispiel für C (Sprache) und RPC (Verteilte Umgebung): `rpcgen`

- Bindung an unterschiedliche Sprachen möglich, z.B. bei CORBA

AIDL

- AIDL steht für Android Interface Definition Language
- Dateiendung: `.aidl`
- automatische Generierung von Java Klassen mit weiteren inneren Klassen
  - für den remote Aufruf von Diensten eines Service und
  - der Übertragung der Parameterdaten und des Ergebnisses (Marshalling)

Marshalling

- Bezeichnet das Umwandeln von einfachen oder strukturierten Daten in ein Format, das die Übermittlung der Daten an andere Prozesse ermöglicht.
- Auf der Empfängerseite werden die Daten wieder in ihre originäre Form gebracht (Unmarshalling).
- Problem: Sender und Empfängerseite nutzen u.U. unterschiedliche Repräsentationsformate.
- Ursachen: unterschiedliche Rechnerarchitekturen, unterschiedliche Sprachstandards

- Dies muss von den entsprechenden IDL-Compilern berücksichtigt werden. Basis hierzu sind neutrale Datenrepräsentationsformate.

- Beispiel C in Verbindung mit `rpcgen`: External Data Representation (XDR)

Stub

- Als *Stub* bezeichnet man Quelltext, der für anderen Quelltext, typischerweise auf einem anderen Rechner oder Speicherbereich steht.
- vgl. Entwurfsmuster Proxy
- Beispiel: Eine Methode, die in einem anderen Prozess verfügbar ist, wird lokal durch einen Stub zur Verfügung gestellt.
- Der Stub übernimmt Marshalling der Methodenparameter, Kommunikation mit dem anderen Prozess und Unmarshalling des Ergebnisses.
- Der Stub-Quelltext für den Client wird dabei automatisch vom IDL-Compiler erzeugt und kann i.d.R. direkt so genutzt werden.

- Die meisten IDL-Compiler erzeugen auch einen sogenannten `Server-Stub`. Dies ist ein Quelltext-Rahmen, in den der Entwickler nur noch die Implementierung des eigentlichen Server-Dienstes einsetzen muss.

AIDL in der Praxis

- Anlegen einer Textdatei mit der Endung .aidl im `src`-Verzeichnis
- In der ersten Zeile muss das zugehörige Package deklariert werden. Diese Package-Deklaration wird in den generierten Quelltext übernommen.
- Im allereinfachsten Fall sieht die AIDL-Definition wie eine übliche Schnittstellendefinition aus.

- Beispiel:

```
package mobis2.beispiel24;

interface IRandomService {
    int generateRandomInt();
    int getLastRandomInt();
    void setRange(int low, int high);
}
```

- Der im SDK enthaltene AIDL-Compiler erzeugt daraus automatisch im gen-Verseichnis eine Datei `IRandomService.java`.

Was enthält der generierte Quelltext?

In `IRandomService.java` werden definiert:

- Die Schnittstelle `IRandomService`, mit den drei definierten Methoden, erweitert um eine Exception-Deklaration.
- Die öffentliche abstrakte innere Klasse `IRandomService.Stub`, abgeleitet von `Binder`, mit den abstrakten Methoden aus `IRandomService`.

  Der Typ `IRandomService.Stub` wird später im Service genutzt, um die konkrete Binder-Klasse zu implementieren, dient also als Rahmen für die Serviceimplementierung.

- Die private innere Klasse `IRandomService.Stub.Proxy`, implementiert (konkret) `IRandomService`, führt dazu Marshalling, Kommunikation mit Service und Unmarshalling aus.

Serviceimplementierung

- Wir definieren zum Typ `IRandomService.Stub` eine konkrete Instanz, z.B. in Form einer anonymen Klassendefinition.

  Hierbei müssen wir die Methoden des Service implementieren.

- Die wichtigste Methode ist jetzt `onBind()`.

- Dort geben wir einfach unsere `IRandomService.Stub` als Ergebnis zurück.

Manifest

- Service muss im Manifest deklariert werden.
- Verwendung des Attributs `android:process`, um einen eigenen Prozess für den Service zu erzwingen.
- Definition eines Intent-Filters zwingend erforderlich, da kein expliziter Aufruf möglich.

Servicenutzung

Prinzipiell wie bei einem lokal Service mit Verbindung (Szenario 3), mit den folgenden Änderungen:

- Wir definieren eine private Variable vom Typ `IRandomService`.
- In der Methode `onServiceConnected()`: Statt eines Downcasts nutzen wir die Methode `asInterface()` der Klasse `IRandomService.Stub`.
- Diese Methode liefert (für uns verdeckt) auf der Client Seite eine Instanz der Proxy-Klasse `IRandomService.Stub.Proxy`. Diese implementiert aber `IRandomService`.

- Beispiel:

```
...
private IRandomService service;
...
public void onServiceConnected(ComponentName name, IBinder binder) {
    service = IRandomService.Stub.asInterface(binder);
}
```

- Jetzt können wir an anderer Stelle mit Hilfe des Proxy-Objektes `service` die Methoden des Remote Service aufrufen, fast so, als wären es lokale Methoden:

```
int value;
try {
    value = service.generateRandomInt();
}
catch (RemoteException e) { ... }
```

Szenario 4: Remote Service mit Verbindung und einfachen Daten (Zusammenfassung)

1. AIDL-Schnittstelle deklarieren,
2. Serviceklasse implementieren, dabei Stub-Klasse um Methoden aus AIDL-Schnittstelle erweitern,
3. Service in Manifest eintragen,
4. Service mit `bindService()` starten und in `onServiceConnected()` auf Schnittstelle (Proxy-Objekt) des Service zugreifen und
5. Methoden der AIDL-Schnittstelle über das Proxy-Objekt aufrufen.

Szenario 4: Beispiel

Beispiel 2.4. *Dienste wie bei Szenario 3, diesmal aber als Remote Service in eigenem Prozess.*