TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

# Software Entwicklung 1

Annette Bieniusa / Peter Zeller

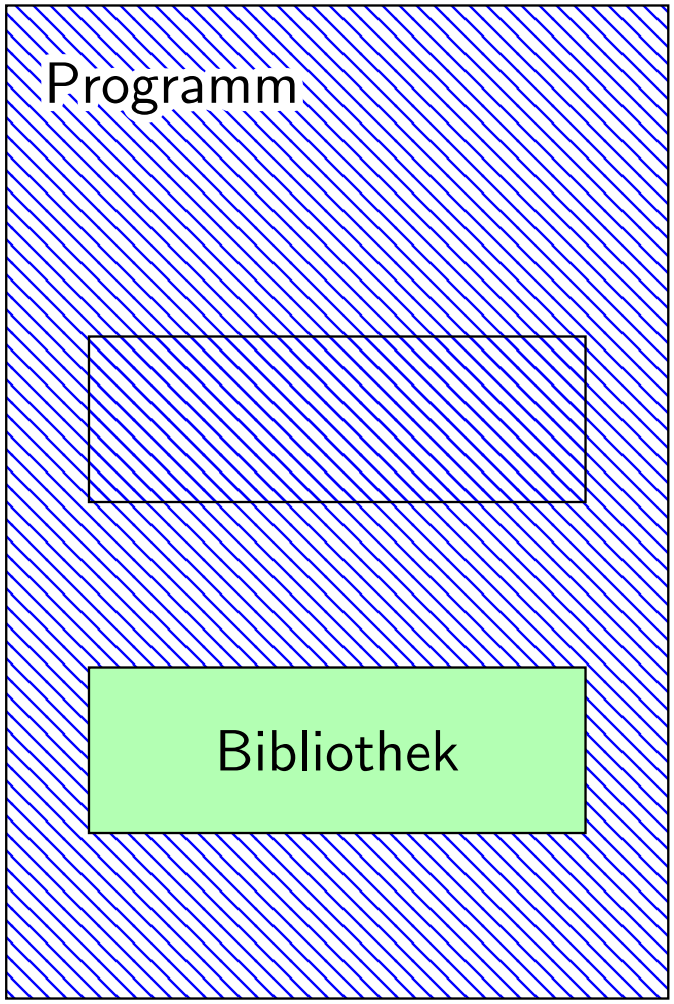
AG Softech
FB Informatik
TU Kaiserslautern

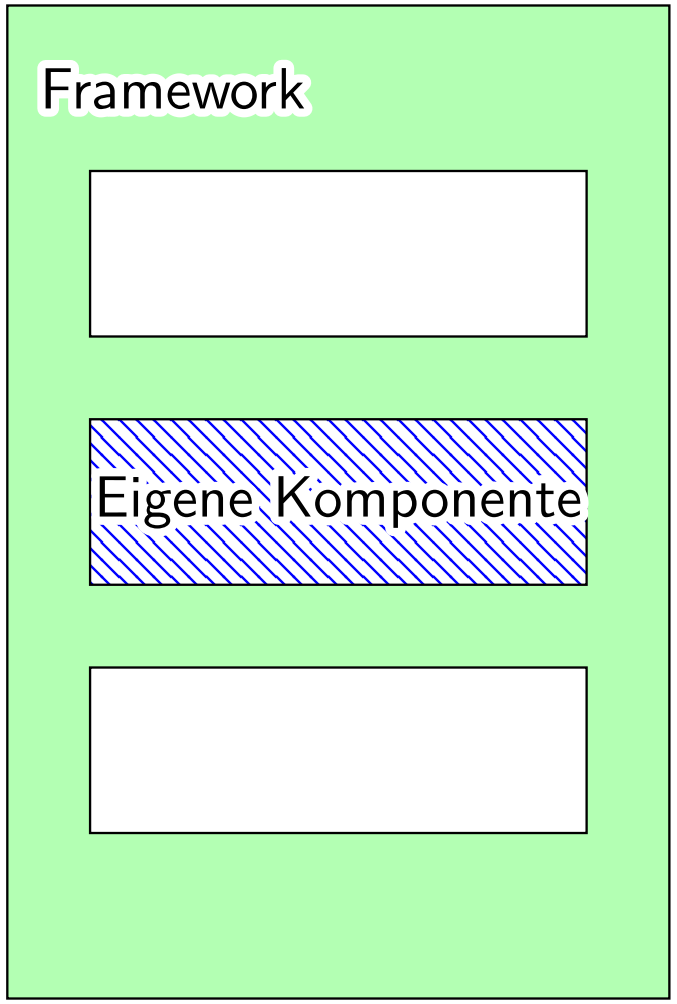
# Das Android Framework zur Entwicklung von Apps

# Frameworks

- Ein Framework ist ein erweiterbares und anpassbares System bestehend aus Modulen oder Klassen, welches eine grundlegende Funktionalität sowie Mechanismen für die Kommunikation bereitstellt.
- Typischerweise spezifisch für eine Aufgabe oder Domäne
  - Hier: Apps für Android Tablets und Smartphones
- Wichtig zur Wiederverwendung von Software
  - Hier: Grundfunktionalität, die von vielen Apps benötigt wird
  - Benutzeroberfläche
  - Persistenz
  - Kommunikation mit anderen Apps
  - ...
- Die Kontrolle liegt oft beim Framework. Das Framework ruft Code auf, der von Benutzern des Frameworks geschrieben wurde.

# Bibliotheken – Frameworks



# Android Grundlagen

Wir behandeln hier:

- Entwicklungsumgebung und grundlegender Aufbau einer App
- Implementierung von Benutzeroberflächen
- Ereignisse und das Beobachter-Muster
- Interaktion mit anderen Anwendungen

*Ziel dieses Kapitel ist es, praktische Anwendungen der gelernten Sprachkonstrukte zu zeigen. Android-Entwicklung dient hier nur als Beispiel und ist nicht klausurrelevant.*

# Eine Anwendung besteht aus mehr als nur Java-Dateien:

`build.gradle`: Konfiguration von

- Verwendete Versionen
- Verwendete Bibliotheken
- Details zum Bauen der Anwendung

`src/test/`: Testfälle, die unabhängig von Android sind

`src/androidTest/`: Testfälle, die abhängig von Android sind

`src/main/AndroidManifest.xml`: Konfiguration der Anwendung

- Bereitgestellte Komponenten (insbesondere Aktivitäten)
- Name der Main-Aktivität
- Name der Anwendung, verwendetes Icon
- Benötigte Rechte und benötigte Features (API Level)

`src/main/java`: Java-Code für die Anwendung

`src/res/`: Resourcen, zum Beispiel:

- Grafiken
- Layouts
- String-Konstanten (Unterstützung für verschiedene Sprachen)

# Android Komponenten

**Activity** Repräsentieren einzelne Bildschirm-Seite mit Benutzer-Oberfläche

**Service** Arbeitet im Hintergrund, keine Benutzer-Oberfläche.
Benutzt für länger dauernde Operationen
Beispiele: Musik im Hintergrund abspielen, Daten aus dem Internet laden

**Content Provider** Verwaltet Daten, kann mit anderen Apps geteilt werden.
Beispiel: Kontakt-Daten im Adressbuch

**Broadcast Receiver** Reagiert auf System-weite Nachrichten.
Beispiele: Bildschirm wurde ausgeschaltet, ein Bild wurde aufgenommen

# Intents: Kommunikation zwischen Komponenten

Intents sind Nachrichten, die zwischen Komponenten verschickt werden.
Ein Intent besteht aus:

- Einer allgemeinen Aktion (`action`)
  Zum Beispiel: `ACTION_DIAL`
- Daten, die zu der Aktion gehören (`data`)
  Zum Beispiel: `tel:123`

Weitere optionale Attribute:

- Kategorie (`category`)
- MIME type (`type`)
- Empfänger-Komponente (`component`)
- Zusätzliche Daten (`extras`)

# Intents: Explizit und Implizit

**Explicit Intent** Empfänger-Komponente wird explizit angegeben.

**Implicit Intent** System findet geeigneten Empfänger basierend auf Eigenschaften des Intents.

# Starten einer neuen Aktivität

Das Starten einer neuen Aktivität ist ein Beispiel für einen expliziten Intent. Es wird die Klasse der Aktivität angegeben, die gestartet werden soll.

```
Intent intent = new Intent(this, OtherActivity.class);
intent.putExtra("Produkt", "Schokolade");
intent.putExtra("Anzahl", 5);
startActivity(intent);
```

# Deklaration der Haupt-Activity

Die Hauptaktivität ist der Activity, die beim Starten der Anwendung aktiviert wird.
Sie legt fest, was nach dem Starten der Anwendung angezeigt wird.

Beim Starten der Anwendung wird ein impliziter Intent mit Aktion `MAIN` verschickt.
In der AndroidManifest.xml wird festgelegt, welche Aktivität darauf reagiert:

```
...
<activity android:name="de.unikl.se1.MainActivity"
          android:label="@string/example_label">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.LAUNCHER"/>
  </intent-filter>
</activity>
...
```

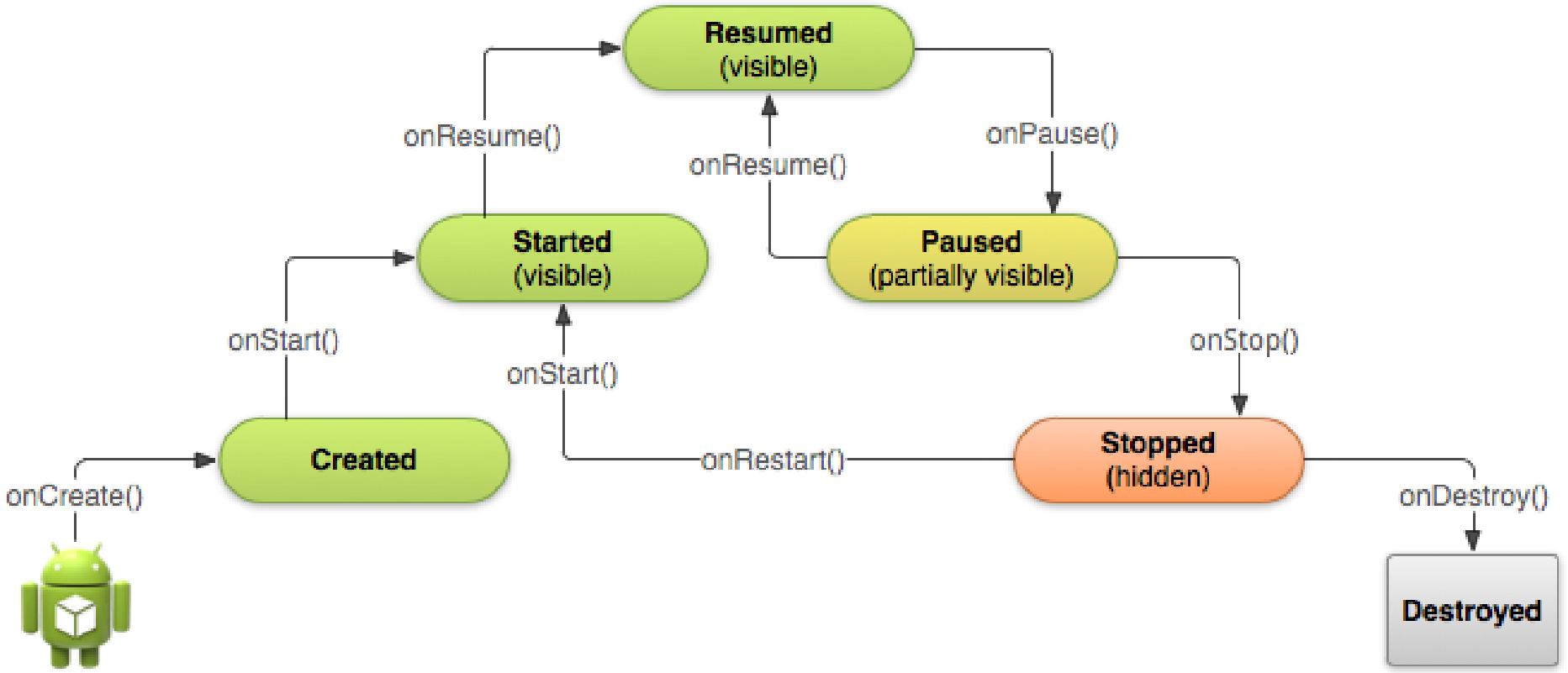
# Typische Main-Aktivität

```
import android.app.Activity;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      // View laden
      setContentView(R.layout.activity_main);
      // View initialisieren ...
      // Zustand wiederherstellen ...
  }

  @Override
  protected void onPause() {
      super.onPause();
      // Zustand sichern ...
  }
}
```

# Main-Aktivität

- Unterklasse einer Aktivität des Frameworks (hier: `Activity`)
- Überschreiben von Callback-Methoden:
  - `onCreate` und `onPause` werden vom Framework aufgerufen
  - Dokumentation gibt an, welche Methode wann aufgerufen wird
  - Keine eigene `main`-Prozedur
- In `onCreate` wird in der Regel die Benutzer-Oberfläche initialisiert

# Lebenzyklus einer Aktivität I




Quelle: http://developer.android.com/training/basics/activity-lifecycle/starting.html

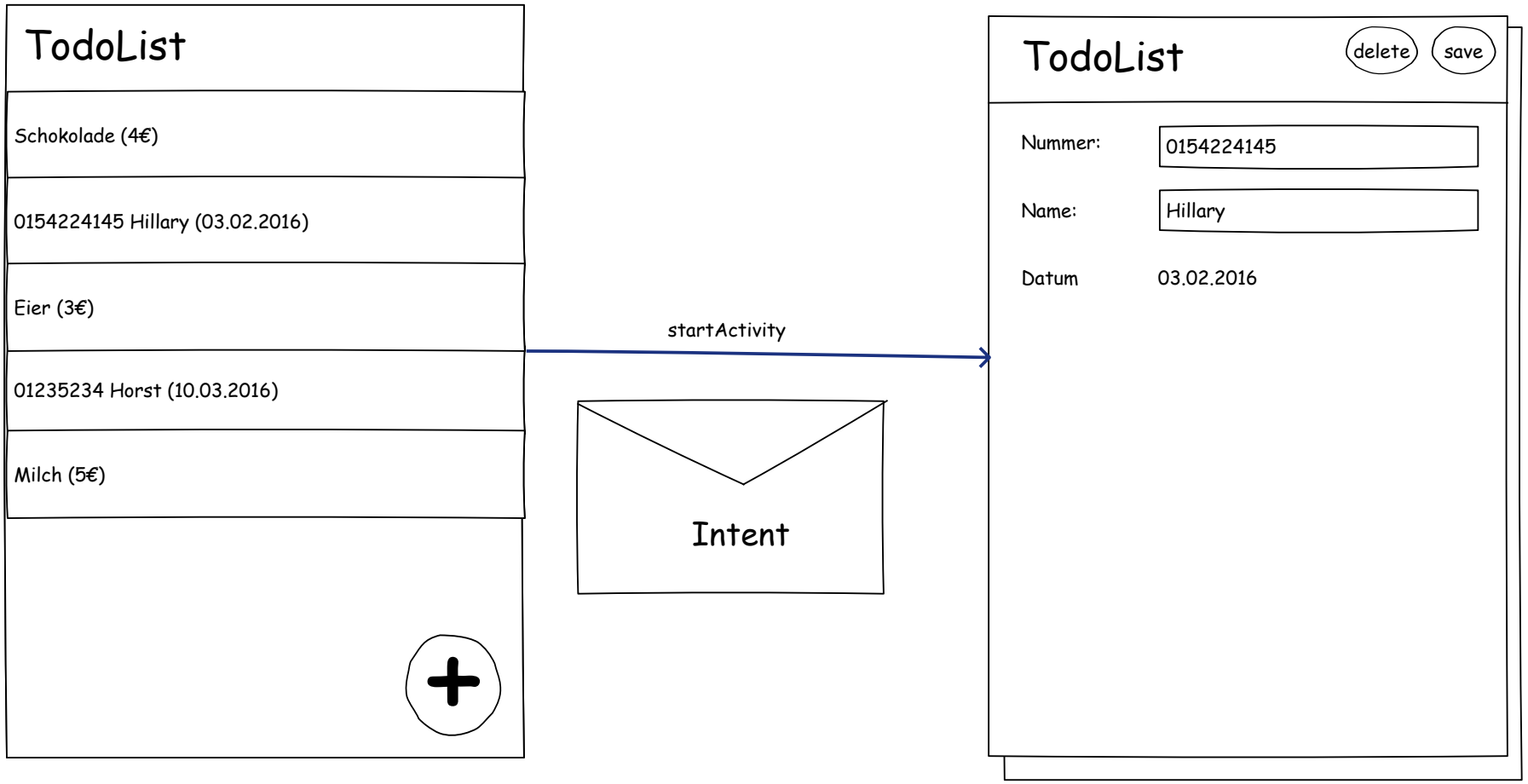
# Lebenzyklus einer Aktivität II

- *Resumed*: Aktivität ist im Vordergrund, Benutzer kann damit interagieren
- *Paused*: Andere Aktivität ist im Vordergrund, aber die Aktivität kann noch teilweise sichtbar sein. Der Benutzer kann nicht mit der Aktivität interagieren und es wird kein Code der Aktivität ausgeführt.
- *Stopped*: Aktivität ist komplett im Hintergrund. Es wird kein Code ausgeführt, aber Zustand von Attributen bleibt noch erhalten.
- *Created/Started*: Zwischenzustände

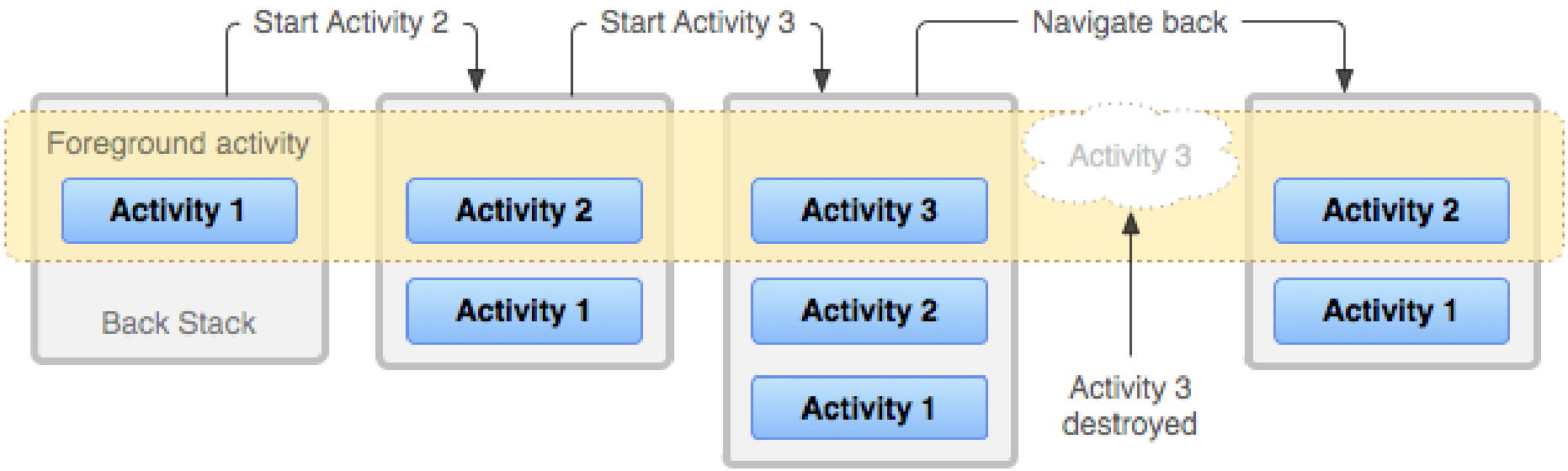
# Wechsel zwischen Aktivitäten

`MainActivity`

`PhoneTaskEditActivity`



# Der Aktivitäts-Stapel




Quelle: http://developer.android.com/guide/components/tasks-and-back-stack.html

- Neue Aktivitäten werden über bestehenden gestartet
- Alte Aktivität bleibt im Hintergrund (Zustand `Stopped`)
- Zurück-Taste beendet die aktuelle Aktivität und geht zurück zur darunterliegenden (Verhalten lässt sich anpassen)

# Design von Benutzer-Oberflächen

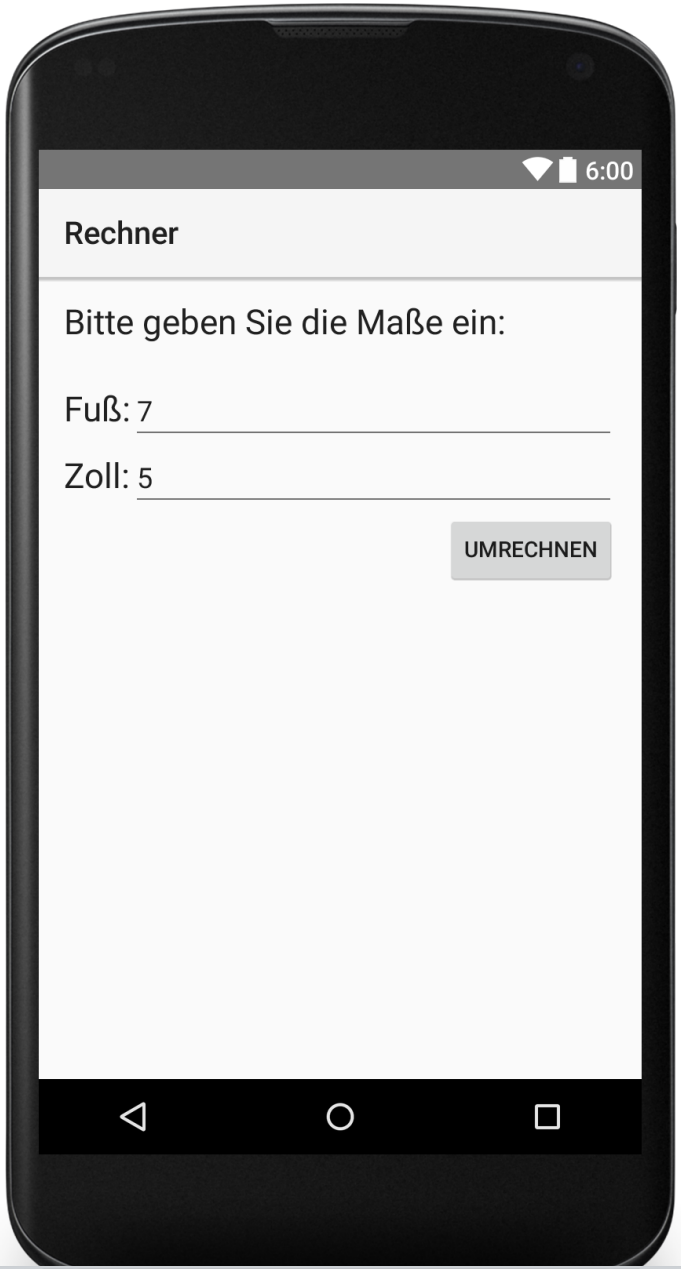
Android bietet mehrere Möglichkeiten, Benutzer-Oberflächen zu bauen:

- Java-Code (nicht empfohlen)
- XML
- Designer-Tool

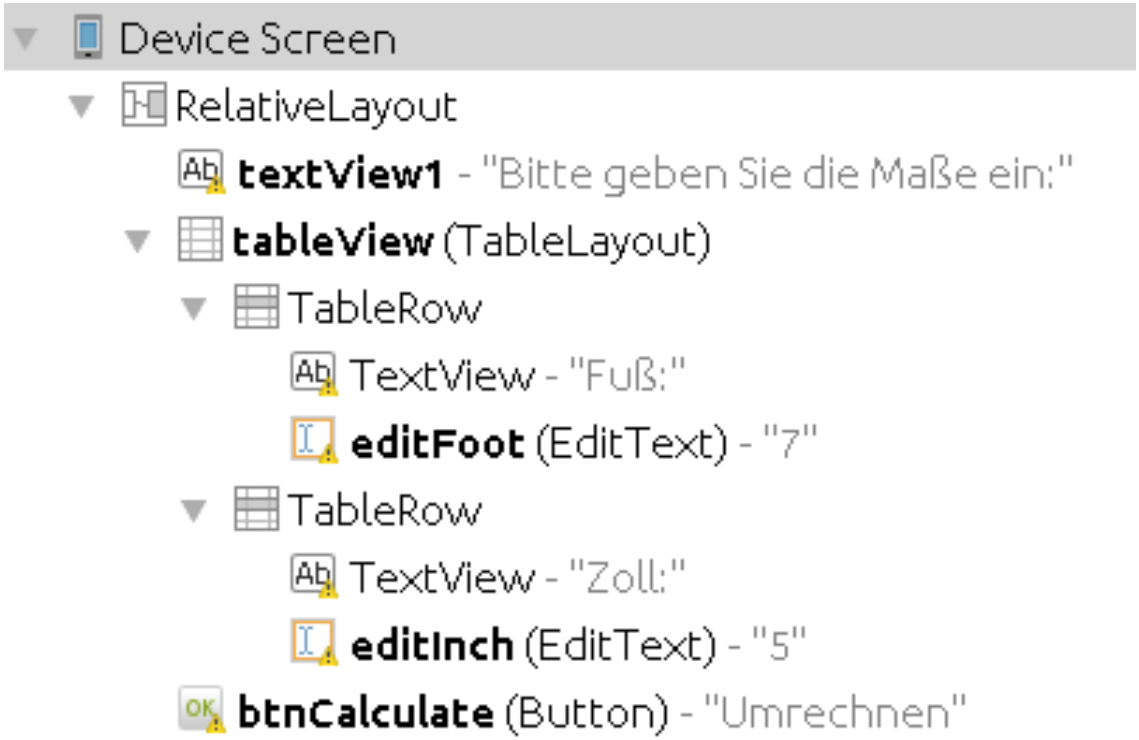
# Modellierung von Benutzer-Oberflächen

- Eine Benutzeroberfläche soll durch Objekte repräsentiert werden.
- Sie besteht unter anderem aus:
  - Text-Feldern
  - Eingabe-Feldern
  - Buttons
  - Tabellen und Listen
- In einer Tabelle und Liste können wieder andere Elemente verschachtelt vorkommen
- Es muss möglich sein, die Elemente auf dem Bildschirm anzuordnen
- Die Anordnung soll auf Bildschirmen mit verschiedener Größe funktionieren

⇒ *Problem ähnlich zur Modellierung von Text-Dokumenten auf Übungsblatt 8.*



# Layout-Baum





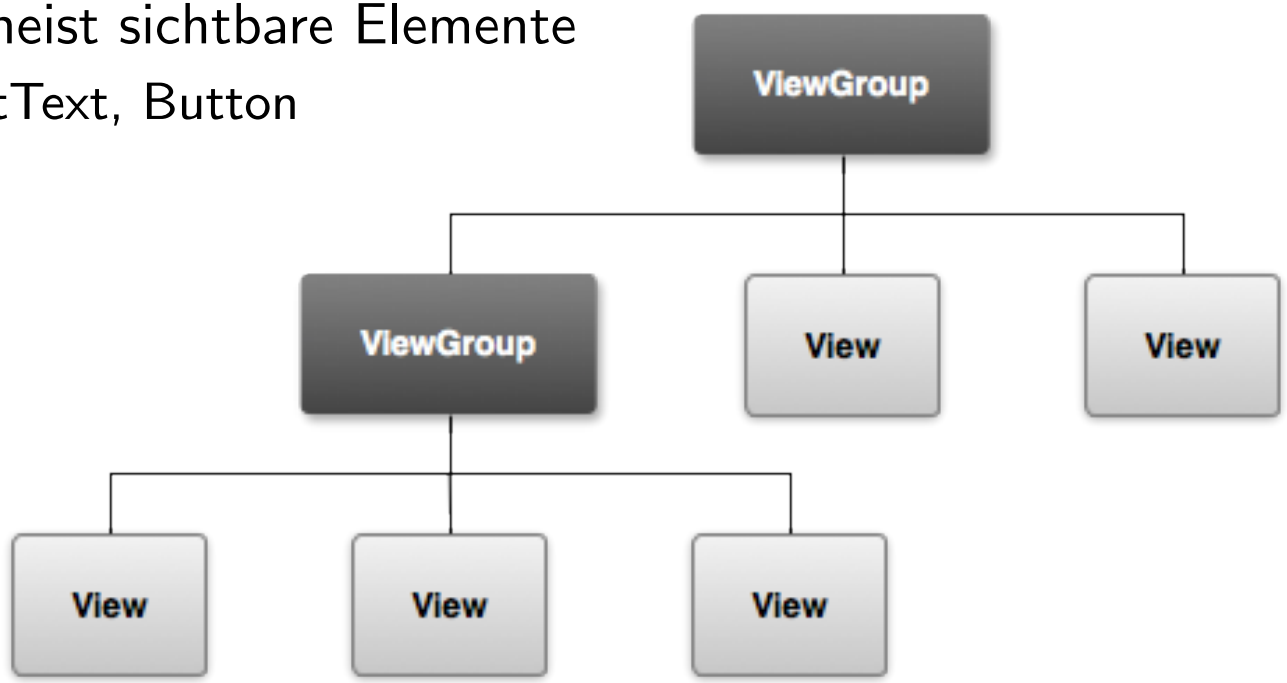
Elemente können eine `id` haben,
zum Beispiel `editFoot`, `editInch`.

# Layout in XML (Übersicht)

```
<RelativeLayout ...>
  <TextView ... />
  <TableLayout ... >
    <TableRow ... >
      <TextView ... />
      <EditText ... />
    </TableRow>
    <TableRow ... >
      <TextView ... />
      <EditText ... />
    </TableRow>
  </TableLayout>
  <Button ... />
</RelativeLayout>
```

# Baum-Struktur im Detail

- Sichten bilden eine Baum-Struktur
- Elemente im Baum sind Subklassen der Klasse `View`
- Innere Knoten sind Subklassen der Klasse `ViewGroup`
  - Bsp.: RelativeLayout, TableLayout, TableRow
- Blätter im Baum sind meist sichtbare Elemente
  - Bsp.: TextView, EditText, Button




Quelle: http://developer.android.com/guide/topics/ui/overview.html

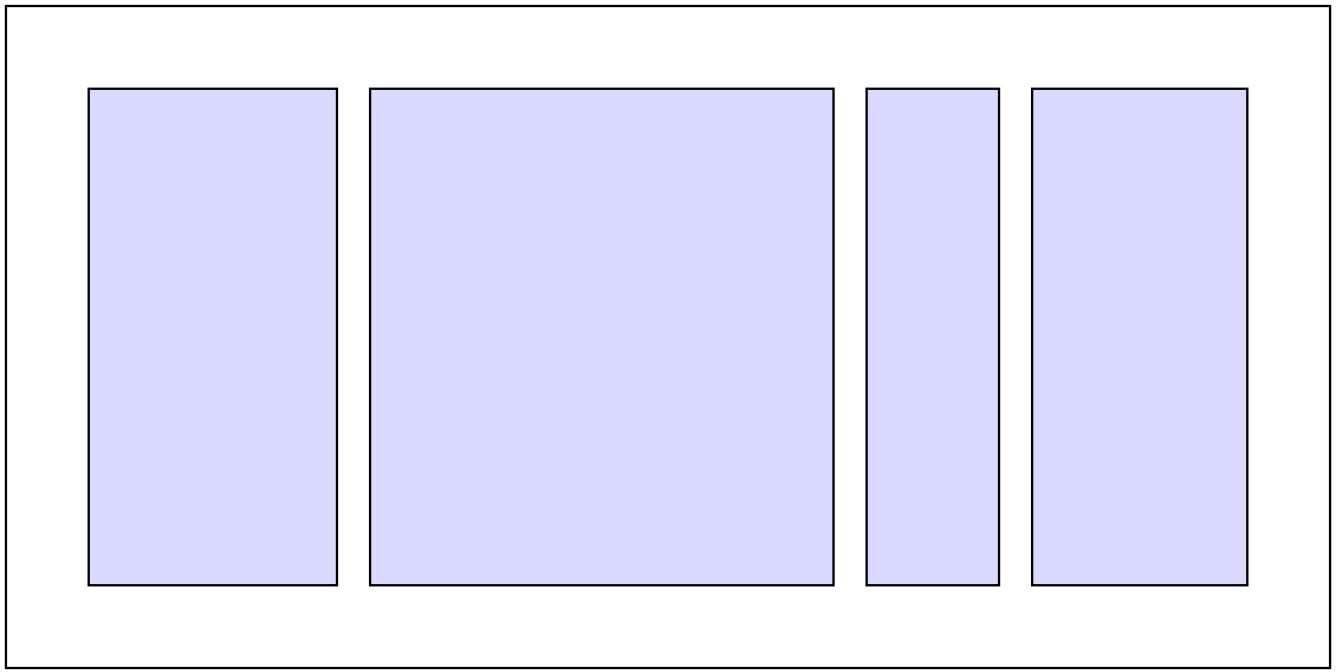
# Layouts

Problem: Eine App soll auf Geräten mit unterschiedlicher Auflösung gut aussehen. Deshalb ist es nicht sinnvoll mit einem absoluten Koordinatensystem zu arbeiten.

Lösung: Layouts, welche Elemente automatisch anordnen

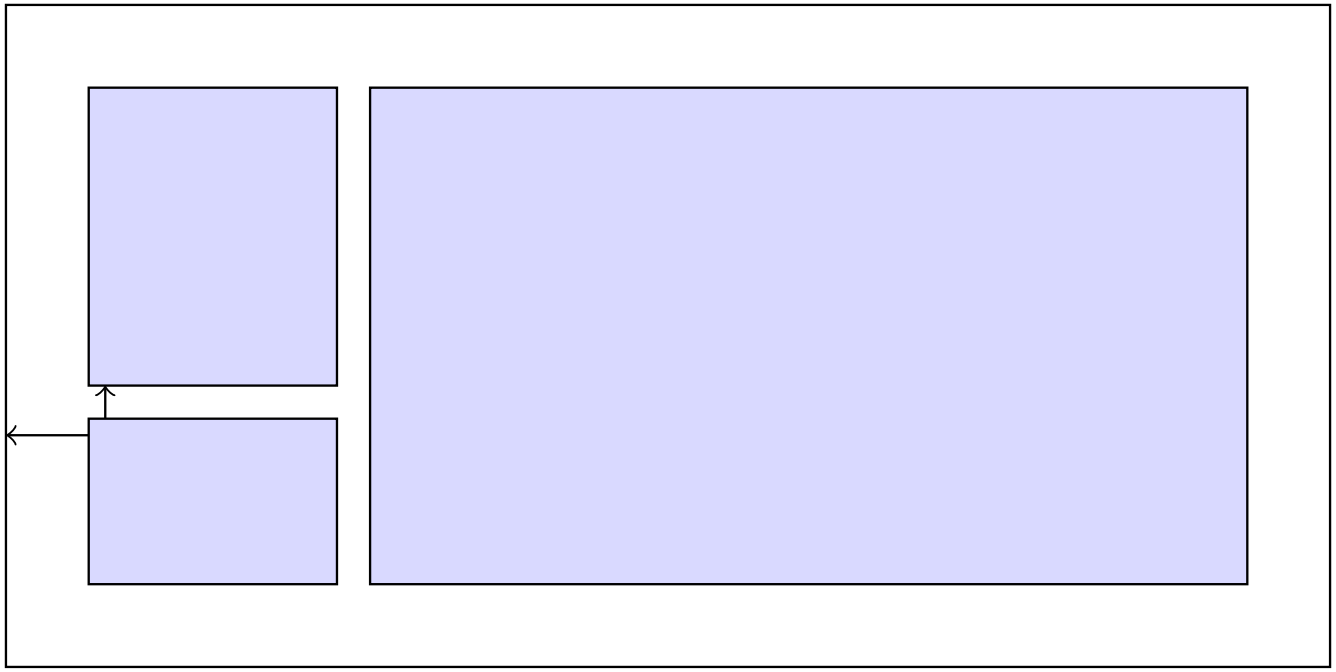
- `LinearLayout` ordnet alle Kinder in einer Richtung an (entweder horizontal oder vertikal)
- `RelativeLayout` kann Kinder relativ zu anderen Kindern und relativ zum Rand anordnen
- `TableLayout` ordnet Kinder nach Zeilen und Spalten an
- ...

# LinearLayout



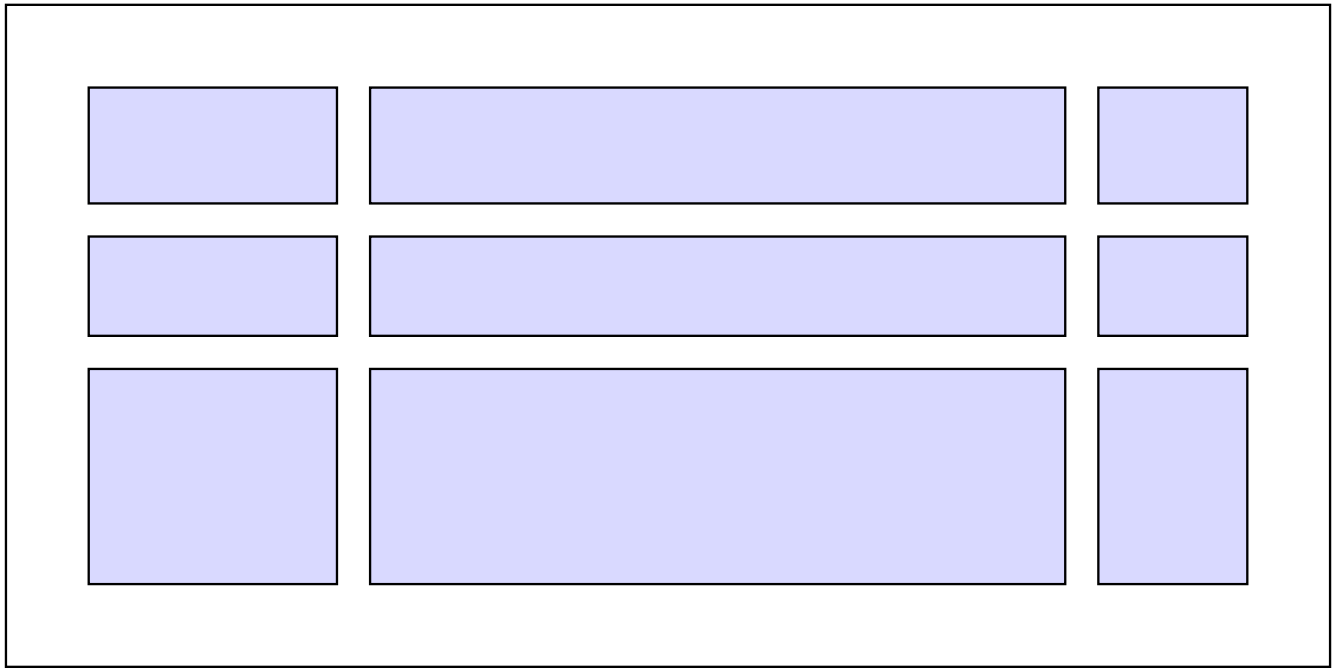
- Anordnung von Kindern nebeneinander (horizontal) oder untereinander (vertikal).
- Feste und relative Größen möglich.

# RelativeLayout



- Position von Kindern relativ zu anderen Kindern oder zum Rand.
- Beispiel: Unter Kind A und am linken Rand.

# TableLayout



- Anordnung von Kindern in Tabelle

# Verbindung zum Code: Die R-Klasse

Frage: Wie greift man vom Java-Code aus auf Elemente zu, die nicht in Java erstellt wurden?

- `aapt` Tool generiert automatisch eine Klasse `R` mit Konstanten
- Für jede `xml`-Datei im Ordner `res/layout` wird eine Konstante `R.layout.name` mit einer eindeutigen Nummer erstellt
- `R.drawable.name` für Grafiken im Ordner drawable
- `R.string.name` für String-Konstanten
- Für jedes Element mit `android:id`-Attribut wird eine Konstante `R.id.name` erstellt.

Die Konstanten sind jeweils eindeutige `int`-Werte und können im Java-Code mit `R.category.name` und in XML mit `@category/name` benutzt werden.

# Verbindung Layout zum Code (Erstellen)

In der `onCreate`-Methode einer Aktivität wird das Layout mit `setContentView` geladen:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    int layoutResID = R.layout.activity_main;
    setContentView(layoutResID);

    ...
}
```

# Verbindung Layout zum Code (Lesen und Ändern) I

```
int editFootId = R.id.editFoot;
View editFootView = findViewById(editFootId);
EditText tFoot = (EditText) editFootView;
Editable tFootText = tFoot.getText();
String tFootString = tFootText.toString();
double foot = Double.parseDouble(tFootString);
```

- Für jedes Element mit `android:id`-Attribut wird eine Konstante `R.id.name` erstellt.
- Methode `findViewById` liefert View mit gegebener `id` (Methode geerbt von `Activity`)
- View-Objekte bieten Methoden zum Lesen und Ändern

# Verbindung Layout zum Code (Lesen und Ändern) II

```
private void calculate() {
    EditText tFoot = (EditText) findViewById(R.id.editFoot);
    EditText tInch = (EditText) findViewById(R.id.editInch);
    TextView tv = (TextView) findViewById(R.id.textView1);
    try {
        double foot = Double.parseDouble(
                        tFoot.getText().toString());
        double inch = Double.parseDouble(
                        tInch.getText().toString());
        double cm = inch * 2.54 + foot * 30.48;
        tv.setText("Das sind " + Math.round(cm) + "cm.");
    } catch (NumberFormatException e) {
        tv.setText("Konnte Eingabe nicht leesen.");
    }
}
```

# Verbindung Layout zum Code (Ereignisse) I

- Aktionen von Benutzern sind Ereignisse.
- Wenn man an einem Ereignis interessiert ist, kann man einen `Listener` setzen.
- Eine Methode des `Listener` wird aufgerufen, wenn ein Ereignis stattfindet.

```
class OnCalculateClick implements View.OnClickListener {
    @Override
    public void onClick(View v) {
        calculate();
    }
}
Button btn = (Button) findViewById(R.id.btnCalculate);
btn.setOnClickListener(new OnCalculateClick());
```

# Verbindung Layout zum Code (Ereignisse) II

Oft verwendet man anonyme Klassen, da die Klasse nur an einer Stelle verwendet wird:

```
Button btn = (Button) findViewById(R.id.btnCalculate);
btn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        calculate();
    }
});
```

Das Registrieren von `Listenern` passiert in der Regel in `onCreate`.

# Listener auf Framework-Seite (vereinfacht)

```
public class View {
  public interface OnClickListener {
      void onClick(View v);
  }

  private OnClickListener mOnClickListener = null;

  public void setOnClickListener(OnClickListener li) {
    mOnClickListener = li;
  }

  public boolean performClick() {
    if (mOnClickListener != null) {
      mOnClickListener.onClick(this);
      return true;
    }
    return false;
  }
```

# Beobachter-Muster I

Das gezeigt Prinzip von `Listener`n entspricht dem Beobachter-Muster (*observer pattern*). Das Beobachter-Muster besteht im Allgemeinen aus

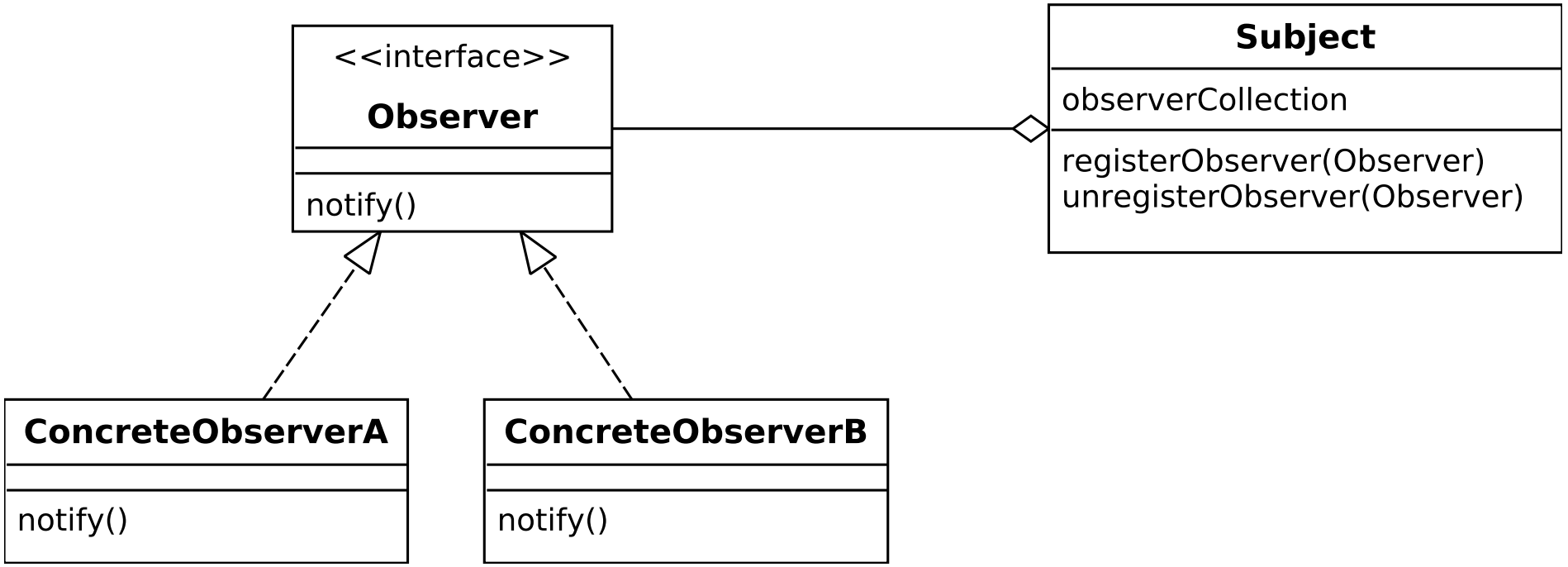
- einem Subjekt (hier: der Button)
- einem oder mehreren Beobachtern (hier: der `OnClickListener`)

Bei Ereignissen oder Änderungen am Subjekt werden die Beobachter informiert, welche mit dem Subjekt verbunden sind.

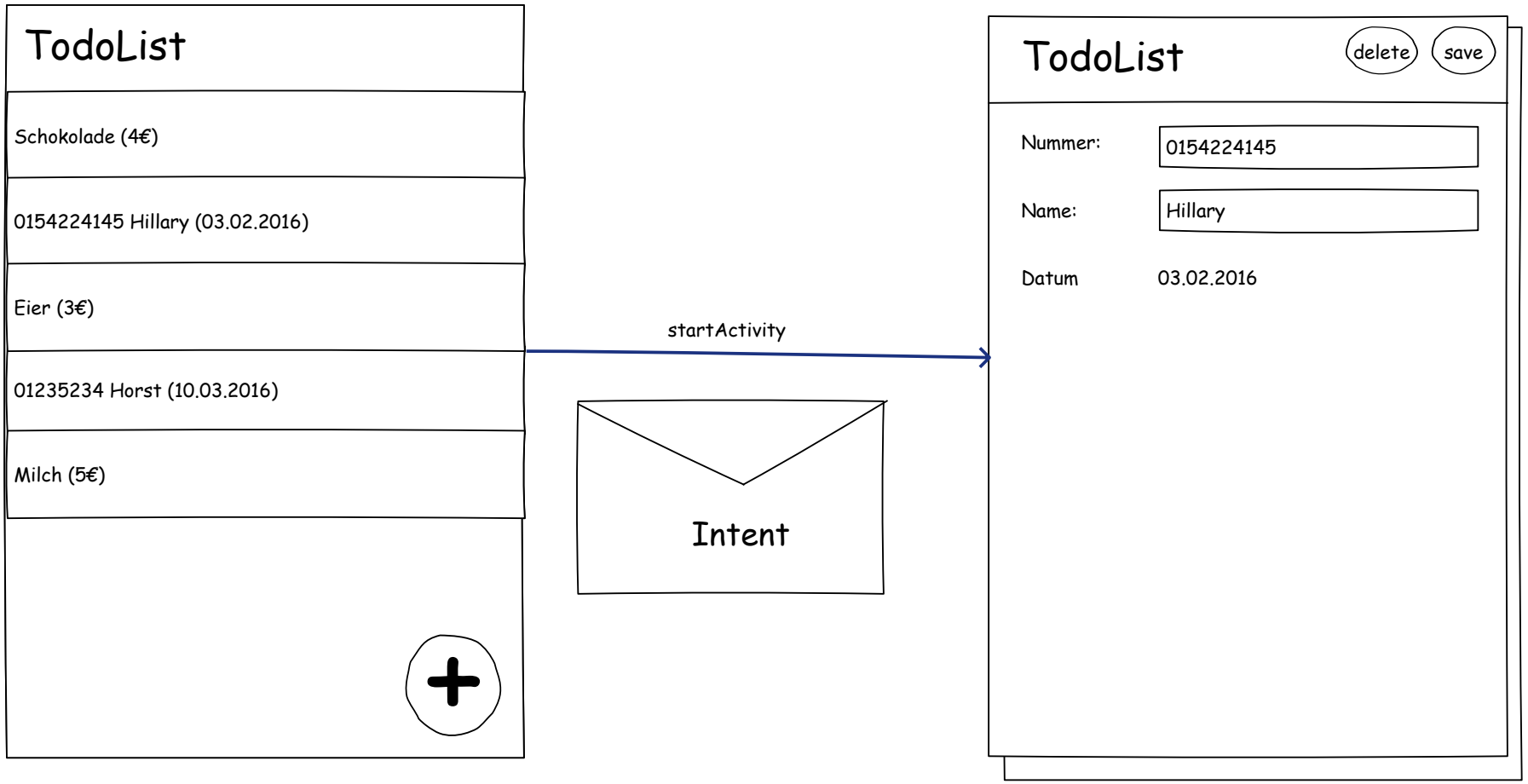
Der Code im Subjekt ist unabhängig von konkreten Beobachtern. Daher wird ein Interface verwendet, welches später in verschiedenen Varianten implementiert werden kann.

Meistens wird eine Liste von Beobachtern unterstützt, in Android-Views nur ein einzelner.

# Beobachter-Muster II



# Beispiel: TODO-App



# Architektur: Trennung von Model und View I

Bei der Entwicklung von Anwendungen mit einer grafischen Benutzeroberfläche wird der Code oft in zwei Teile aufgeteilt:

**Model** Repräsentiert die zugrundelegende Anwendung und den aktuellen Zustand. Es bietet eine Schnittstelle zur Kern-Funktionalität der Anwendung.
Das Model ist unabhängig von möglichen Views.

**View** Zeigt das Model (oder Teile davon) auf dem Bildschirm an.
Verantwortlich für Interaktion mit Benutzer.

Es gibt viele speziellere Ausprägungen dieses allgemeinen Prinzips:

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-Viewmodel (MVVM)

# Architektur: Trennung von Model und View II

- Darstellung kann angepasst werden, ohne dass Funktionalität der Anwendung betroffen ist.
- Implementierungsdetails im Model können leicht ausgetauscht werden, solange die Schnittstelle zur View gleich bleibt.
  Beispiel: Persistenz ändern von JSON-Datei zu SQLite.
- Benutzer-Oberflächen sind schwer automatisch zu testen.
  Trennung macht es möglich, die Grundfunktionalität der Anwendung zu testen, ohne Benutzeroberfläche zu beachten.
- Gleiches Model kann für verschiedene Benutzeroberflächen wiederverwendet werden.
- Designer können an Views arbeiten, während Coder am Model arbeiten.

# Trennung von Model und View in Android

- Activity verwaltet die View, verarbeitet Ereignisse
- View wird definiert durch Layout in XML
  und optional einer Java-Klasse für Zugriff auf die View und Weiterleiten von Ereignissen an Controller
- Model besteht aus einfachen Java Klassen
  (Alternative: Model ist ein ContentProvider)

# Umsetzung im Todolist Beispiel

Siehe Demo + Code in `androidTodoList.zip`.

# Verwenden von externen Diensten mit Intents

Intents können verwendet werden, um mit anderen Apps zu kommunizieren.
Beispiel: Telefon-Nummer aus den Kontakten auswählen:

```
private static final int CONTACT_PICKER_RESULT = 1;

@Override
public void onChooseContactClick() {
  Intent intent = new Intent(
                  Intent.ACTION_PICK,
                  CommonDataKinds.Phone.CONTENT_URI);
  startActivityForResult(intent, CONTACT_PICKER_RESULT);
}
```

Es wird eine neue Aktivität mit impliziten Intent gestartet.
Android wählt eine passende App aus, welche die Aktion `ACTION_PICK` mit gegebenen URI unterstützt. Bei mehreren passenden Apps kann Benutzer entscheiden.

# Ergebnis von Aktivitäten I

Eine Aktivität, die mit `startActivityForResult` gestartet wird, kann Daten an die aufrufende Aktivität zurücksenden.

Der Empfang des Ergebnisses passiert in Callback-Methode `onActivityResult(int requestCode, int resultCode, Intent data)`.

**requestCode** Wert, der beim Starten der Aktivität angegeben wurde.
→ Zuordnung von Aufruf zu Antwort.

**resultCode** Gibt an, wie Aufruf beendet wurde.
Zum Beispiel `RESULT_OK` oder `RESULT_CANCELED`

**data** Die Ergebnis-Daten.

# Ergebnis von Aktivitäten II

```
@Override
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == CONTACT_PICKER_RESULT) {
        if (resultCode == Activity.RESULT_OK) {
            // Daten aus Antwort-Intent lesen:
            Uri contentUri = data.getData();
            // Daten von Uri lesen:
            loadFromContentUri(contentUri);
        }
    }
}
```

# Ergebnis von Aktivitäten III

```
private void loadFromContentUri(Uri contentUri) {
  ContentResolver r = getContentResolver();
  try (Cursor c = r.query(contentUri, null, null, null, null)) {
    if (c.moveToFirst()) {
      String name = c.getString(
                          c.getColumnIndex(DISPLAY_NAME_PRIMARY));
      String number = c.getString(c.getColumnIndex(NUMBER));
      view.setName(name);
      view.setNumber(number);
    }
  }
}
```

# Rechte-Management

Im Allgemeinen muss eine Anwendung erst Rechte einfordern, um auf die Kontakte zugreifen zu können.

Beim Auswählen des Kontakts wird der Anwendung ein temporäres Recht gegeben, Daten des ausgewählten Kontakts zu lesen.

Für allgemeine Zugriffe auf die Kontakte wird das Recht `READ_CONTACTS` benötigt.

Benötigte Rechte werden in der `AndroidManifest.xml` festgelegt und vom Benutzer beim Installieren blind akzeptiert.

Seit Android 6 (Marshmallow) kann eine App auch zur Laufzeit neue Rechte anfordern, also erst dann wenn sie auch benötigt werden.

# Zusammenfassung

- Framework-Begriff
- Interaktion von Komponenten durch Nachrichten
- Entwicklung von Benutzeroberflächen mit Layouts
- Beobachter-Muster und Ereignisse
- Trennung von Model und View
- Android Grundlagen