Inhaltsverzeichnis

1 Überblick Orangegurt

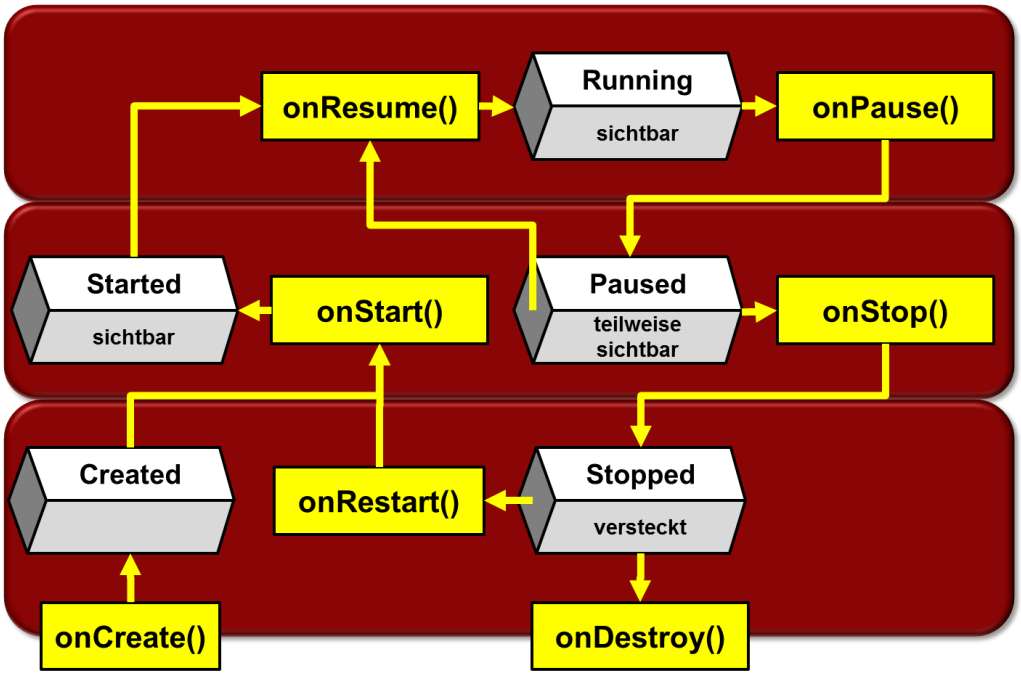
Wenn ihr den Gelbgurt (also Stufe 1) des Android™ Kurses absolviert habt, dann seid ihr in der Lage die Entwicklungsumgebung auf Basis von Eclipse aufzusetzen, ihr kennt die wesentlichen Layouts und könnt Views platzieren. Weiterhin sind euch die grundlegenden Strukturen und Files eines Android Projektes bekannt. Ihr könnt also einfache Apps realisieren, welche im Wesentlichen aus einer Activity bestehen und basierend auf Useraktivitäten Berechnungen durchführen und Ergebnisse anzeigen können. Auf dieser Basis wird die zweite Stufe des Kurses aufsetzen, der Orangegurt.
Solltet ihr die ersten Inhalte verpasst haben, so könnt ihr dies jederzeit nachholen. Sämtliche Informationen zum Gelbgurt und Orangegurt findet ihr im Netz unter www.codeconcert.de.

Die Inhalte des Orangegurtes sind darauf ausgelegt, mehrere Activities zu erzeugen und diese zu einem in sich stimmigen Programm zu kombinieren. Hierbei müsst ihr lernen, wie der sogenannte Life Cycle der Activities aufgebaut ist. Weiterhin werden wir uns um das Thema Threading kümmern, damit unsere Apps keine nervigen Screen Freeze Situationen provozieren (also Situationen, bei denen die App für kurze Zeit scheinbar nicht mehr reagiert). Danach sehen wir uns noch ein paar weitere grundlegende Themen wie Fragments und den Settings Dialog an.

Dieses Dokument wird sich primär um das Thema Life Cycle, Intents (Benachrichtigungen innerhalb Android) und in diesem Zusammenhang den und dem Interface „Parcelable“, Intent Filter kümmern. Themen wie BroadcastReceiver und Services werden wir in einer späteren Gürtelfarbe kennen lernen.

2 Life Cycle einer Activity

Jede Activity durchläuft in ihrem „Leben“ einen Life Cycle „Lebenszyklus“, welcher im Wesentlichen der unten dargestellten Grafik entspricht.



Wichtig zu verstehen ist, dass das Management dieses Life Cycles durch das Android System durchgeführt wird. Insofern müssen wir als Programmierer auch darüber informiert werden, was gerade passiert. Dies läuft über sogenannte Callback Methoden also Methoden, welche durch das Android System aufgerufen werden, welche wir als Programmierer wiederum anpassen (überschreiben) können.

Die wichtigsten Callback Methoden im Überblick:

Methode:	**Wird aufgerufen wenn:**	**Was soll passieren:**
onCreate()	Die Activity initial erzeugt wird. Dies kann entweder durch den Start der ganzen App sein (also wenn die MainActivity gestartet wird), oder wenn in einer Activity eine zweite (Sub)Activity gestartet wird.	Wir fügen hier im Regelfall Layoutkomponenten ein – sofern wir diese dynamisch erzeugen wollen.
onStart()	Sobald die Activity erzeugt wurde, wird sofort danach onStart() aufgerufen. Sie rückt damit in den Fokus des sichtbaren Bereiches. Sollte sie vorher „versteckt“ gewesen sein, so würde die Reaktivierung der Activity ebenfalls über diese Methode laufen.	Üblicherweise benötigen wir diese Methode nicht für eigene Implementierungen.
onResume()	Wenn die Activity beginnen soll, mit dem User zu interagieren – also wenn Usereingaben möglich sein sollen, dann wird onResume() aufgerufen.	Hier sollten alle inhaltlichen Updates auf dem Bildschirm erfolgen. Bspw. sollten Infotexte mit dem aktuellen Wert belegt werden. Auch müssen alle Ressourcen allokiert werden, welche in der onPause Methode freigegeben wurden.
onPause()	Sobald die Activity nicht mehr für die Usereingabe aktiv sein soll. Dies ist entweder, wenn eine andere Activity in den Vordergrund rückt, oder wenn die aktuelle Activity komplett geschlossen werden soll.	Alle CPU intensiven Aktivitäten sollten nun unterbrochen werden, da nun eine andere Activity die CPU Leistung benötigt. Vor allem Grafikanimationen sollten beendet werden. Es wird allerdings empfohlen, den Code an dieser Stelle möglichst performant zu gestalten, da die Folgeactivity nicht starten kann, bevor onPause() beendet wurde.
onStop()	Wenn die Activity nicht mehr sichtbar ist, wird diese Methode aufgerufen.	Alle Aktionen, welche aufgrund der Performance nicht in onPause aufgenommen wurden, müssen hier ausgeführt werden. Insbesondere Zwischenergebnisse sollten hier abgespeichert (persistiert) werden.
onDestroy ()	Kurz bevor die Activity komplett entfernt wird, erfolgt der Aufruf dieser Methode, was zugleich der letzte Call ist. Dies muss nicht zwingend bedeuten, dass die Activity nicht mehr neu erzeugt wird – es kann sein, dass das System die Activity nur beendet, um Platz zu sparen. Dies passiert auch, wenn bspw. der Bildschirm gedreht wird und das Layout von vertikal auf horizontal wechselt! Wichtig ist noch, dass in Sonderfällen das System die Activity einfach „killt“, also ohne den onDestroy() Aufruf beendet.	Diese Methode muss im Regelfall nicht von uns implementiert werden, da wir üblicherweise in der onPause() und onStop() Methode alle notwendigen Schritte unternommen haben, unsere Ressourcen freizugeben. Einzige Ausnahme bilden Threads, welche im Hintergund etwas bewerkstelligen sollen (bspw. Datendownload).

Bei der Überschreibung dieser Methoden solltet ihr immer darauf achten, dass die Methode der Superklasse aufgerufen wird, da hier mitunter wichtige interne Einstellungen vorgenommen werden. Weiterhin gilt zu verstehen, dass alle Ressourcen, welche in einer Methode freigegeben werden, in dem entsprechenden Pendant allokiert werden müssen (bspw. onStart() <-> onStop()). Bis auf „onCreate()“ haben die Methoden keinen Parameter.

2.1 Der Einstieg in eine Activity

Da es die wichtigste Methode ist, sehen wir uns die onCreate Methode nochmal kurz näher an. Diese wird im Eclipse Projekt bereits automatisch hinterlegt. Hier werden die Layoutinformationen spezifiziert. Aus diesem Grund hat Eclipse auch bereits den ContentView gesetzt – und zwar mit den Layoutinformationen unserer MainActivity:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  // meine dynamischen Layout Konfigurationen
}
```

Zuerst erkenn wir am „@Override“, dass wir eine existierende Methode überschreiben. Danach erfolgt die Signatur der Methode mit den vorgesehenen Parametern. Für onCreate wird ein Bundle übergeben, welches den gespeicherten Stauts der Instanz beinhaltet. Details hierzu findet ihr weiter Unten beim Kapitel „Speichern des Activity-Status“. In der ersten Zeile wird immer die Supermethode aufgerufen, indem die gleichen Parameter einfach weitergereicht werden. Danach wird das Layout geladen, welches den Bildschirm aufbaut. Alle weiteren Codezeilen müsst ihr selbst formulieren. Wenn ihr bspw. dynamisch in einer Tabelle in eurem Layout Zeilen hinzufügen müsst, ist dies hier ein guter Zeitpunkt.

2.2 Speichern des Activity-Status

Wie oben beschrieben, kann Android eine derzeit nicht aktiv genutzte Activity jederzeit auflösen – womit sämtliche Klassenstatuswerte (also die Klassenvariablenwerte) nicht mehr existent sind. Sollten wir nun wieder die Activity zurückholen wollen, so werden alle Klassenvariablen zurückgesetzt. Was wir also benötigen ist eine Möglichkeit, den Status einer Activity in solch einer Situation zwischen zu speichern. Android bietet hier eine Möglichkeit, dies zu tun.
Bevor Android eine Activity in den „Destroyed“ Status versetzt (also letztendlich löscht), wird die Methode „onSaveInstanceState()“ aufgerufen. Diese Methode ist einzig dafür da, Werte in einem Bundle abzulegen. Dabei werden die Statuswerte der einzelnen Views (sofern sie eine eindeutige ID besitzen) ebenfalls im Bundle abgelegt. Folgender Code soll dies verdeutlichen:

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
  super.onSaveInstanceState(savedInstanceState);
  savedInstanceState.putInt(INST_KEY_SCORE, iScoreValue);
}
```

In dem Beispiel wird die onSaveInstanceState Methode überschrieben. Als Parameter ist das Bundle vorgesehen, welches als Datenspeicher fungiert. Um den Status der Views zu sichern, muss die überschriebene Methode der Superklasse auch aufgerufen werden – die Werte werden somit im savedInstanceState Bundle geschrieben. In dieses Bundle können nun zusätzlich individuelle Name/Value Pairs abgelegt werden. In unserem Beispiel wurde der Wert einer Klassenvariable „iScoreValue“ in das Bundle gelegt. Als Name Wert wurde eine Konstante genutzt, welche natürlich am Anfang unserer MainActivity Klasse definiert werden muss und im Rahmen dieser Acitvity eindeutig sein muss. Wenn nun die Activity gelöscht wird, ist das Bundle noch weiterhin existent.

Wenn nun die Activity wieder neu erzeugt („reanimiert“) wird, so ruft Android zuerst die Methode onRestoreInstanceState() auf. Diese holt die Informationen wieder in das Bundle zurück und übergibt es der „onCreate“ Methode, welche es als Parameter erhält. Folgender Code zeigt nun beispielhaft, wie man an die Daten wieder herankommt:

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

```
  if (savedInstanceState != null) {
    iScoreValue = savedInstanceState.getInt(INST_KEY_SCORE);
  } else {
    iScoreValue = 0;
  }
}
```

Im Bundle „savedInstanceState“ liegt nun der zwischengespeicherte Wert unseres Scores – jedoch nur, wenn onCreate nach einem „Destroy“ aufgerufen wird. Ansonsten ist das Objekt null. Aus diesem Grunde prüfen wir erst, ob das Objekt existiert und wenn ja, dann übernehmen wir unseren Score Wert aus dem Bundle mittels des vorher definierten Keys (INST_KEY_SCORE). Sollte das Objekt null sein, so können wir den initialen Wert für iScoreValue übernehmen – in unserem Fall die 0.

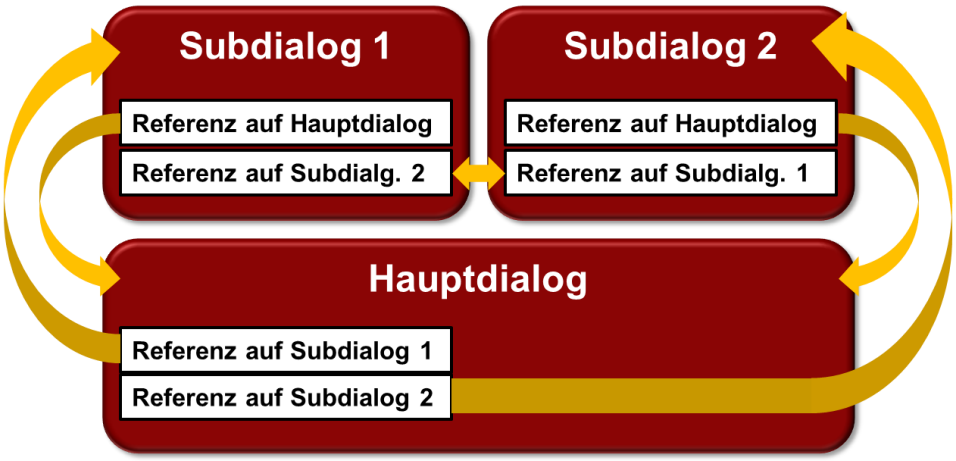
Weitere Details zum Instance State findet ihr hier:
http://developer.android.com/guide/components/activities.html#SavingActivityState

Wichtig zu verstehen ist, dass beim Layoutwechsel (bspw. wenn das Gerät von vertikaler in horizontale Richtung geneigt wird), die Activity komplett „zerstört“ wird und neu aufgebaut wird. Insofern ist der Layoutwechsel ein guter Test, ob wir unseren Status sauber speichern und wieder zurückspielen.

3 Intents

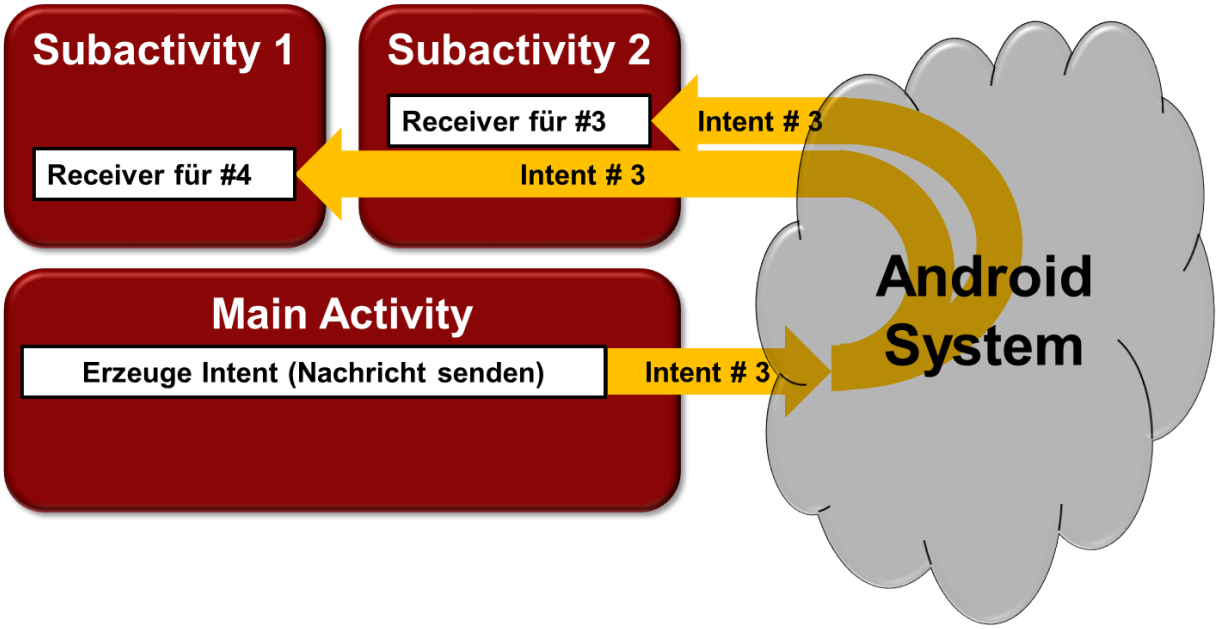
3.1 Loosely Coupled Architecture

Bevor wir uns das Thema Intents ansehen, müssen wir zuerst das Bewusstsein für die Notwendigkeit von dem Architekturkonzept der losen Koppelung schaffen. Für viele, die ihre ersten Programmierschritte im Desktopbereich (bspw. mit Swing) gemacht haben, ist dieser Gedanke erst mal nicht zwingend naheliegend. Die folgende Grafik zeigt ein Konzept der engen Koppelung, wie es bei Desktopanwendungen häufig anzutreffen ist.



Die einzelnen Dialogelemente haben eine Referenz auf die anderen beteiligten Dialogelemente. Der Informationsaustausch zwischen den Dialogelementen kann also sehr bequem durchgeführt werden. Man muss lediglich in den einzelnen Dialogklassen Getter- und Settermethoden veröffentlichen und kann mit diesen Datenwerte transportieren.
Das Problem bei dieser Methodik ist, dass die Kommunikationsstruktur sehr statisch ist und keinerlei Interaktionen von außen zulässt, da der Entwickler der Klassen diese Kommunikationsstruktur mit den entsprechenden Getter- und Settermethoden bestimmt. Auch ist die Wartbarkeit im Sinne von Austauschbarkeit von einzelnen Elementen eher schwierig. Man könnte sich zwar dem Java Konstrukt der Interfaces bedienen, was jedoch bei der Programmierung schnell umständlich werden kann – vor allem lehrt die Erfahrung, dass der Durschnittsentwickler seine Kommunikationsstrukturen nicht als Interfacedefinition hinterlegt, sondern seine Klassen direkt einbindet.
Noch viel wichtiger jedoch ist es, eine einheitliche Kommunikationsstrategie zu verfolgen. Bei mobilen Geräten gibt es im Gegensatz zu Desktopanwendungen viel mehr Informationsquellen, welche für potentielle

Apps sinnvoll nutzbar sind. So gibt es bspw. neben den Datenströmen zwischen meinen Programmkomponenten noch Daten aus den einzelnen Sensoren des Devices (Beschleunigungssensor, Kompass etc.). Weiterhin haben wir die Kamera(s). Hinzu kommt, dass eine App vielleicht informiert werden möchte, ob das Telefon klingelt, ob eine SMS eingetroffen ist usw. All diese Informationen sollen zentral verwaltet werden, so dass der Aufwand, einzelne Informationsquellen anzuzapfen, so klein wie möglich gehalten werden kann. Sehen wir uns das Kommunikationskonzept von Android mal etwas näher an:



Wie man sieht, sendet die Main Activity einen Intent an das Android System, welche diesen an alle anderen Komponenten weiterreicht. Wenn eine Activity (in unserem Beispiel Subactivity 2) einen Receiver für diese Activity hat, dann kann sie diesen Intent auch empfangen. Insofern können auch mehrere Objekte einen Intent empfangen und ggf. verwerten.

Grundsätzlich sind Intents also als Informationsträger zu sehen, welche im Andorid System eine Aktivität auslösen können bzw. sollen.

Der Vorteil dieser Archtektur ist es nun, dass zum einen das Android System die Oberhand über die einzelnen Activities behält, da die Kommunikation mit den Activities zentral verläuft. Weiterhin können Informationen, welche von einer Activity ausgehen von mehreren Activities verwertet werden (bspw. die Information, dass der Akku jetzt den Geist aufgeben wird könnte für mehrere Activities der Anlass sein, seine Daten zu speichern). Weiterhin können hier einzelne Elemente ohne großes Risiko durch aktuellere Versionen ausgetauscht werden, sofern die Intentverarbeitung unangetastet bleibt.

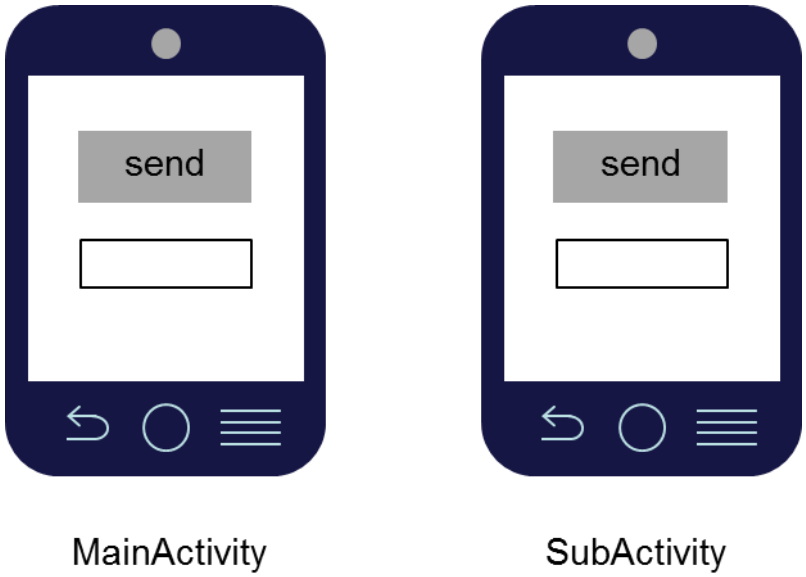
3.2 Der Intent zum Starten einer Activity ohne Rückantwort

Intents sind Informationsträger und können zwei wesentliche Informationskategorien tragen:

- Was soll getan werden?
- Was ist passiert!

Intents, welche vom Android System kommen, wie bspw. niedriger Akkustand, eingehender Anruf, zeigen im Regelfall an, das etwas passiert ist. Eventuelle Intent Empfänger können diese Information aufgreifen und verarbeiten.

Intents, welche von eurer App erzeugt werden sind normalerweise dazu da, dass sie eine konkrete Aktivität triggern sollen. Der einfachste Fall ist, eine neue Activity zu erzeugen. Sehen wir uns dies im folgenden Beispiel etwas näher an. Wir erzeugen zwei Activities – eine MainActivity und eine SubActivity:



Beide versehen wir mit einem Sendebutton und einem EditText Element. Hier die Layoutdefinition der MainActivity:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/mainRelativeLayout"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="#0000ff">
  <Button android:id="@+id/sendButtonMain"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="10dp"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:onClick="myClickMethod"
    android:text="@string/sendButtonText" />
  <EditText android:id="@+id/editTextMain"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_below="@id/sendButtonMain"
    android:hint="@string/editTextHint"/>
</RelativeLayout>
```

Vergesst nicht, die Strings „editTextHint“ und „sendButtonText“ zu definieren. Die SubActivity sieht genauso aus, nur dass wir die IDs dergestalt ändern, dass „main“ mit „sub“ ersetzt wird (also subRelativeLayout, sendButtonSub und editTextSub – und Achtung, die ID sendButtonMain gibt es auch im Attribut „layout_below“). Um die beiden Dialoge besser unterscheiden zu können wählen wir für die MainActivity die Hintergrundfarbe „#0000ff“, also Blau und für die SubActivity „#ff0000“, also Rot.

Wir wollen nun folgende Funktionalität realisieren:

- Bei Klick auf den „Send Button“ der MainActivity soll die SubActivity erscheinen
- In der SubActivity soll der Text, welcher in der MainAcivity in das EditText Feld eingetragen wurde, ebenfalls in dem EditText Feld auftauchen.

Wir haben im Layout die Methode „myClickMethod“ als Aktion beim Klick des Buttons hinterlegt. Also müssen wir auch diese Methode realisieren:

```
public void myClickMethod(View view) {
  Intent intent = new Intent(this, SubActivity.class);
  EditText editText = (EditText) findViewById(R.id.editTextMain);
  String message = editText.getText().toString();
  intent.putExtra(EXTRA_MESSAGE_DEF, message);
  startActivity(intent);
}
```

Gehen wir die einzelnen Codeelemente durch. Zuerst wird ein Intent erzeugt. Wir werden im Anschluss noch über die verschiedenen Konstruktoren blicken, für unser erstes Beispiel werden wir uns auf den hier dargestellten konzentrieren. Er verlangt zwei Parameter – den Context (Informationen über die Applikationsumgebung, also im Wesentlichen dem Android System) und der expliziten Klasse, welche den Intent empfangen soll (also in unserem Fall die SubActivity, welche erzeugt werden soll).

```
Intent intent = new Intent(this, SubActivity.class);
```

Danach lessen wir den Inhalt des EditText Feldes aus und speichern den Inhalt in einem String:

```
EditText editText = (EditText) findViewById(R.id.editTextMain);
String message = editText.getText().toString();
```

Nun wird in den Intent ein “Extra” hineingelegt, was nichts anderes ist, als eine Sammlung von „Name/Value Pairs“, also Werten, welche mit einem eindeutigen Namen versehen werden und unter diesem Namen später wieder gefunden werden können. Intern ist dieses Konstrukt als „Bundle“ realisiert.

```
intent.putExtra(EXTRA_MESSAGE_DEF, message);
```

Der eindeutige wird üblicherweise als Konstante in der Activity angelegt und sollte global eindeutig sein. Insofern müsst ihr noch in der MainActivity eine Klassenkonstante festlegen:

```
public static String EXTRA_MESSAGE_DEF =
 "com.example.course.MY_MESSAGE";
```

Zum Schluss wird dem System mitgeteilt, dass eine Activity gestartet werden soll, wobei für diese Aktion das System den Intent erhält – was bedeutet, dass auch an das Android System mit Intents beeinflusst wird.

```
startActivity(intent);
```

Was jetzt noch benötigt wird, ist das Auslesen der Intentinformationen und das Eintragen in das EditText Feld in der SubActivity. Hierzu suchen wir die „onCreate“ Methode in der SubActivity und ergänzen am unteren Ende sie wie folgt:

```
Intent intent = getIntent();
String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE_DEF);
EditText editText = (EditText) findViewById(R.id.editTextSub);
editText.setText("Sub received: " + message);
```

Hier holen wir uns erst mal den Intent, welcher die Activity gestartet hat:

```
Intent intent = getIntent();
```

Mit Hilfe dieses Objektes können wir nun auf die Extras zugreifen, indem wir mit Hilfe des eindeutigen Namens den vorher eingetragenen Wert holen:

```
String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE_DEF);
```

Der Rest sollte klar sein – es wird das EditText Element gesucht und dort der Message Wert eingetragen. Zur Verdeutlichung hängen wir vor den String noch den Text „Sub received“, damit wir sehen, dass der Wert tatsächlich durch diese Methode gesetzt wird.

```
EditText editText = (EditText) findViewById(R.id.editTextSub);
editText.setText("Sub received: " + message);
```

Wenn die App nun im Emulator gestartet wird, so sehen wir die MainActivity (wir erkennen sie am blauen Hintergrund). Wir tragen einen beliebigen Text in die EditText View ein und klicken auf „Send“. Nun wechselt der Hintergrund auf Rot – wir haben also die SubActivity gestartet. In dem EditText Feld finden wir nun wieder den eingegebenen Text in Großbuchstaben wieder – er wurde also mit dem Intent gesendet.

3.3 Der Intent zum Starten einer Activity mit Rückantwort

Jetzt wollen wir den Code dergestalt erweitern, dass der EditText Wert aus der SubActivity wieder zurück in die Wenn wir nun von der SubActivity eine Nachricht zurück zur aufrufenden Activity senden wollen, dann müssen wir hier noch etwas ergänzen. Die im vorausgegangenen Kapitel beschriebene Vorgehensweise ist (erstmal) eine Einbahnstraße. Für solche Situationen haben die Android Entwickler eine zweite Methode zum Activity Start vorgesehen. Wir ändern unseren Code in der MainActivity wie folgt:

```
public void myClickMethod(View view) {
  Intent intent = new Intent(this, SubActivity.class);
  EditText editText = (EditText) findViewById(R.id.editTextMain);
  String message = editText.getText().toString();
  intent.putExtra(EXTRA_MESSAGE_DEF, message);
  startActivityForResult(intent, ACTIVITY_RESULT_REQUEST_SUB);
}
```

Wie ihr seht, wurde lediglich der Befehl zum Starten einer Activity durch einen neuen ersetzt:

```
startActivityForResult(intent, ACTIVITY_RESULT_REQUEST_SUB);
```

Diese Methode benötigt noch zusätzlich einen int Parameter (größer gleich 0), welcher einen Code erwartet, der bei der Rückmeldung von der SubActivity wieder eingesetzt wird. Für’s Erste packen wir hier einfach eine int Konstante rein, die ähnlich wie unsere Message ID als Klassenkonstante deklariert wird:

```
public static int ACTIVITY_RESULT_REQUEST_SUB = 1;
```

Nun müssen wir in der SubActivity Klasse noch die Methode implementieren, welche beim Klick auf den Button aufgerufen wird. Hier wird wiederum ein Intent erzeugt, welcher für die Nachrichtenübermittlung von der SubActivity zur MainActivity verwendet wird:

```
public void myClickMethod(View view) {
  Intent intent = new Intent();
  EditText editText = (EditText) findViewById(R.id.editTextSub);
  String message = editText.getText().toString();
  intent.putExtra(EXTRA_MESSAGE_REPLY_DEF, message);
  setResult(RESULT_OK, intent);
  finish();
}
```

Ein wesentlicher Unterschied ist, dass wir im Intent keinerlei Parameter setzen. Dies liegt daran, dass die Klasse im Kontext der MainActivity aufgerufen wurde und von vorneherein darauf ausgelegt ist, eine Nachricht zurückzuschicken. Insofern reicht hier zur Intenterzeugung lediglich folgende Zeile:

```
Intent intent = new Intent();
```

Danach wird wieder aus dem EditText View der Inhalt ermittelt und in den Intent unter einer eindeutigen Bezeichnung eingelagert:

```
EditText editText = (EditText) findViewById(R.id.editTextSub);
String message = editText.getText().toString();
intent.putExtra(EXTRA_MESSAGE_REPLY_DEF, message);
```

Auch hier wieder daran denken, dass die Namensdefinition des Extras als Klassenkonstante deklariert werden muss:

```
public static String EXTRA_MESSAGE_REPLY_DEF =
 "com.example.course.MY_MESSAGE_REPLY";
```

Wirklich neu sind nun die letzten beiden Zeilen der Methode. Zuerst wird das Ergebnis gesetzt, welches einmal einen Resultcode beinhaltet und anschließend noch den intent:

```
setResult(RESULT_OK, intent);
```

Danach wird die Activity beendet:

```
finish();
```

Nun wird automatisch der Intent an die aufrufende Activity gesendet. Hier wiederum muss eine eigene Methode erstellt werden, welche diesen empfängt. Folgender Code ist somit in der MainActivity vorzusehen:

```
protected void onActivityResult(int requestCode, int resultCode,
  Intent data) {
  EditText editText = (EditText) findViewById(R.id.editTextMain);
  String message =
    data.getStringExtra(SubActivity.EXTRA_MESSAGE_REPLY_DEF);
  editText.setText("Main received: " + message);
}
```

Diese Methode ist eine sogenannte „Callback Methode“ und ist bereits abstrakt in der Activity Klasse vorhanden. Callback heißt, dass das System selbstständig diese Methode aufruft und somit vorab wissen muss, wie die Methodensignatur aussieht. Wir müssen nun lediglich die Methode mit einem sinnvollen Inhalt über-

schreiben. Als erstes fallen die Parameter auf. „requestCode“ ist die Nummer, welche beim Start der SubActivity hinterlegt wurde. Danach folgt der resultCode, der in der SubActivity durch die Methode „setResult“ festgelegt wurde. In „data“ findet sich lediglich der Intent wieder, in dem ja unsere „message“ eingepackt wurde. Diese muss nun wie gehabt extrahiert werden und entsprechend weiterverarbeitet werden. Wir haben hier wieder zur besseren Unterscheidung der beiden Aktivitäten dem gesendeten String den Text „Main received:“ vorangesetellt

Nun können wir die Activity ausprobieren. Wir starten die App und sehen den blauen Bildschirm. Hier tragen wir den String „ab“ ein. Durch „Send“ wird die SubActivity (roter Bildschirm) gestartet und im Fenster steht „Sub received: ab“. Wir ändern nun den String in „abc“ und senden erneut. Die MainActivity (blauer Bildschirm) zeigt unseren String nun als „Main received: abc“ an. Unsere TestApp für Nachrichtenaustausch zwischen den Activities funktioniert.

3.4 Das Bundle

Wie wir gesehen haben, ist das Bundle Objekt ein Transportmittel für Informationen. Da bei Programmiersprachen Informationen in Variablen mit Datentypen organisiert sind, können Bundles auch verschiedene Datentypen speichern – String und int haben wir ja schon gesehen. Sehen wir uns mal eine umfangreichere Liste an und sehen, was mit dem Bundle noch alles geht. Grundsätzlich gilt zu sagen, dass es eine „put“ Methode gibt, welche Werte als Name/Value Pair (oder Key/Value) in dem Bundle ablegt. Mit einer „get“ Methode liest man den Wert anhand des Keys wieder aus. Weiterhin gibt es noch eine weitere „get“ Methode, bei der ein Defaultwert gesetzt wird, wenn der Key nicht gefunden wird. Die Methoden beinhalten im Namen jeweils den Datenwert. Beispiele für boolean:

- putBoolean(key, value)
- getBoolean(key)
- getBoolean(key, defaultValue)

Hier eine Übersicht:

Kategorie:	**Bemerkung**	**Datentyp:**	**Methoden:**		
			put	**get**	**get (def.)**
Einfache Datentypen	Einfache Datentypen sind die Grundbausteine aller Informationen im Rechner. Für alle einfachen Datentypen (ganze Zahlen, Gleitkommazahlen, boolschen Werten und Character Zeichen) existierend die drei grundtypen von Methoden (put, get ohne Defaultwert und get mit Defaultwert).	boolean, byte, char, double, float, int, long, short	Ja	Ja	Ja
Zeichenketten	Für Zeichenketten bietet das Bundle zwei wesentliche Klassen an – String und CharSequence. Nachdem CharSequence „lediglich“ ein Interface ist, welches auch vom String implementiert wird, ist es nur eine verallgemeinerte Version – welche im Zweifelsfall breiter nutzbar ist.	CharSequence, String	Ja	Ja	Ja
Arrays	Für alle oben genannten Datentypen gibt es auch eine Arrayimplementierung. Bspw. für int gibt es putIntArray(key, value) um ein Array von int Werten abzulegen.	boolean, byte, char, double, float, int, long, short, CharSequence, String	Ja	Ja	Nein
Serializable	Mit dieser Methode werden serialisierbare Objekte in ein Bundle eingetragen. Die meisten Objekte sind serialisierbar. Der Mechanismus wurde in Java eingeführt, um Objekte in einem Datenstrom (Neztwerk oder File) zu versenden. Details zum Interface	Alle serialisierbaren Objekte	Ja	Ja	Nein

Kategorie:	Bemerkung	Datentyp:	Methoden:		
			put	get	get (def.)
	Serializable können im Netz gefunden werden. Grundätzlich ist das mit Vorsicht zu genießen, da die Serialisierung die App inperformant werden lässt.				
Parcelable	Ein Parcelable Objekt ist ein schlankes Konstrukt, um strukturierte Datensammlungen einfach in einem Bundle zu platzieren und somit eine „lightweight“ Alternative zu serializable. Auf das Parcelable wird weiter unten nochmal eingegangen. Für Parcelable Objekte existiert auch eine Arrayimplementierung.	Parcelable	Ja	Ja	Nein
ArrayList	Da das Arraykonstrukt für manche Anwendungen zu unflexibel ist, wurde die ArrayList als Bundle Element eingeführt. Sie existiert zwar nicht für alle Datentypen, aber einige wichtige sind vorhanden. Achtung – da ArrayList in Java nur Objekte akzeptiert ist die ArrayList für Ganzzahlenwerte vom Typ Integer (also Objekt) und nicht int.	CharSequence, Integer, Parcelable, String	Ja	Ja	Nein

Bei der Wahl des Datentyps für Bundle Elemente sollte immer die Performance berücksichtigen. Folgende Tabelle soll einen Anhaltspunkt für die Wahl des Datentyps darstellen:

Fragestellung	Antwort	Aktion:
Habe ich wenige Daten, welche ich durch einfache Datentypen darstellen kann?	Ja	Verwende einfache Datentypen
	Nein	Nächste Frage
Habe ich wenige Textdaten, welche ich nur anzeigen muss?	Ja	Verwende Strings
	Nein	Nächste Frage
Habe ich wenige Textdaten, welche ich nicht nur anzeigen muss, sondern eventuell auch in Files schreiben muss, oder als CharSequence weiterverarbeiten muss?	Ja	Verwende CharSequence
	Nein	Nächste Frage
Habe ich mehrere Datenwerte gleichen Datentyps, bei denen ich einfach die Anzahl feststellen kann (idealerweise Anzahl immer gleich)?	Ja	Verwende Array. Fragestellung ob einfache Datentypen, CharSequence oder String siehe Oben.
	Nein	Nächste Frage
Habe ich mehrere Datenwerte gleichen Datentyps, bei denen ich nicht einfach die Anzahl feststellen kann (Anzahl sehr flexibel), wobei der Datentyp ganze Zahlen oder Text beinhaltet?	Ja	Verwende ArrayList. Fragestellung ob CharSequence oder String siehe Oben.
	Nein	Nächste Frage
Habe ich kompliziertere Datenstrukturen, welche ich in einem von mir definierten Objekt ablegen kann?	Ja	Verwende Parcelable – indem das Interface Parcelable in mein Objekt implementiert wird
	Nein	Nächste Frage
Kann ich die Daten in einem Objekt lagern?	Ja	Verwende ein serializable Object
	Nein	Suche nach Alternativen (bspw. Filesystem oder Datenbank)

3.5 Das Interface „Parcelable“

Wie oben bereits angemekrt, ist das Parcelable Interface geschaffen worden, um eine performantere Alternative zum serialisieren von Objekten zu schaffen. Sehen wir uns das Interface mal genauer an. Im Wesentlichen ist Parcelable lediglich ein Interface, welches einen effizienten Zugriff auf ein „Parcel“ Objekt erlaubt. Details zum Parcel Objekt können aus der API Beschreibung von Android ersehen werden.

Wir wollen für dieses Beispiel eine einfache Implementierung realisieren, welche einen int, einen double, zwei Strings und ein String Array beinhaltet. Hierfür erzeugen wir folgende Klasse:

```
public class MyParcel implements Parcelable {
  private int myIntData;
  private double myDoubleData;
  private String myStringDataA;
  private String myStringDataB;
  private String[] myStringArrayData;

  public static final Parcelable.Creator<MyParcel> CREATOR =
           new Parcelable.Creator<MyParcel>() {
    public MyParcel createFromParcel(Parcel in) {
      return new MyParcel(in);
    }
    public MyParcel[] newArray(int size) {
      return new MyParcel[size];
    }
  };

  public MyParcel() {
  }

  private MyParcel(Parcel in) {
    myStringArrayData = new String[2];
    myIntData = in.readInt();
    myDoubleData = in.readDouble();
    myStringDataA = in.readString();
    myStringDataB = in.readString();
    in.readStringArray(myStringArrayData);
  }

  @Override
  public void writeToParcel(Parcel dest, int flags)
  {
    dest.writeInt(myIntData);
    dest.writeDouble(myDoubleData);
    dest.writeString(myStringDataA);
    dest.writeString(myStringDataB);
    dest.writeStringArray(myStringArrayData);
  }

  @Override
  public int describeContents() {
    return 0;
  }
}
```

Für die privaten Objektvariablen (myIntData, myDoubleData, myStringData und myStringArrayData) müsst ihr noch Getter- und Settermethoden deklarieren – das muss ich hier nicht vormachen. Sehen wir uns mal den Code etwas detaillierter an – an der ein- oder anderen Stelle könnt ihr vielleicht verstehen, warum einige

Blogger das Konzept des Parcelable Interfaces nicht wirklich als „geglückt“ einstufen. Aber wenn man sich daran mal gewöhnt hat, ist es durchaus machbar.
Beginnen wir mit dem CREATOR. Dies ist ein Objekt, welches dem System ermöglicht auf Basis der im Parcel Objekt abgelegten Daten das von uns erzeugte Objekt wieder zu erzeugen:

```
public static final Parcelable.Creator<MyParcel> CREATOR =
        new Parcelable.Creator<MyParcel>() {
  public MyParcel createFromParcel(Parcel in) {
    return new MyParcel(in);
  }
  public MyParcel[] newArray(int size) {
    return new MyParcel[size];
  }
};
```

Es handelt sich hier um ein Interface (Parcelable.Creator). Hier werden lediglich zwei Methoden benötigt – einmal createFromParcel, welches unser Objekt neu erzeugt und newArray, welches ein Array von unserem Objekt mit der vorgegebenen Größe erzeugt. Hier ist lediglich zu beachten, dass unser Klassenname korrekt eingetragen ist (er taucht hier 6x auf – wir hatten unsere Klasse ja „MyParcel“ getauft.

Danach habe ich einen Standardkonstruktor eingefügt, stellvertretend für alle anderen denkbaren Konstruktoren. Man kann bspw. einen Konstruktor mit Parametern schreiben, welcher alle Instanzvariablen belegt.

```
public MyParcel() {
}
```

Der nächste Konstruktor ist ausschließlich für das CREATOR Objekt, welches ja einen Konstruktor für unsere Klasse in der Methode „createFromParcel“ enthält. Aus diesem Grunde wird er auch als „private“ geflagged. Hier werden sämtliche Werte aus dem Parcel ermittelt und in unserer Klasse abgelegt:

```
private MyParcel(Parcel in) {
  myStringArrayData = new String[2];
  myIntData = in.readInt();
  myDoubleData = in.readDouble();
  myStringDataA = in.readString();
  myStringDataB = in.readString();
  in.readStringArray(myStringArrayData);
}
```

Was hier passiert ist, dass die Werte sequenziell im Parcel eingetragen werden. Die Reihenfolge ist also wichtig. Wir werden das Thema Reihenfolge gleich nochmal ansprechen. Nun kommt der Teil, in dem die Werte in das Parcel reingeschrieben werden. Hier muss die gleiche Reihenfolge eingehalten werden, wie im Konstruktor!

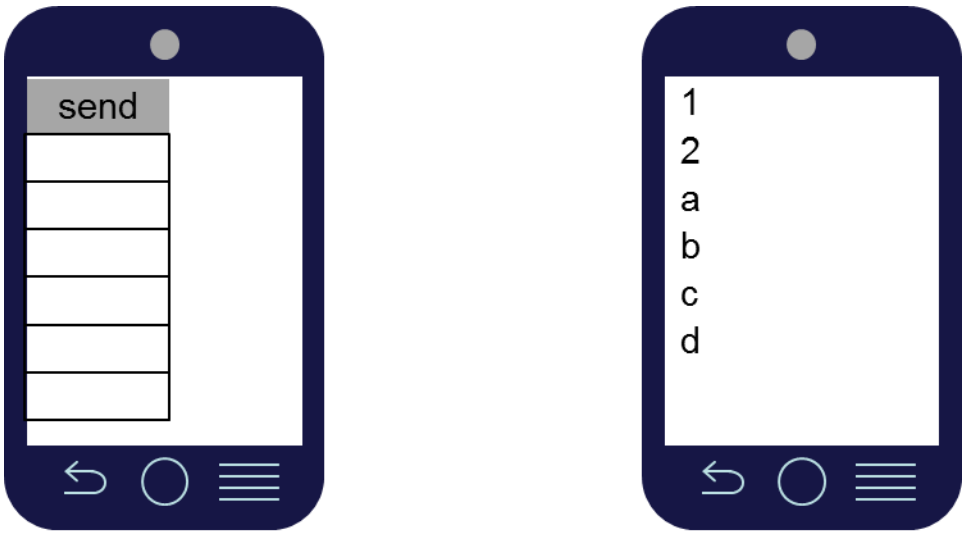
```
@Override
public void writeToParcel(Parcel dest, int flags)
{
  dest.writeInt(myIntData);
  dest.writeDouble(myDoubleData);
  dest.writeString(myStringDataA);
  dest.writeString(myStringDataB);
  dest.writeStringArray(myStringArrayData);
}
```

Bei den Parametern ist in dieser Methode lediglich der erste wichtig, da dies das Parcel ist, in das die Werte eingetragen werden. Die Flags sind für uns momentan unwichtig, da sie nur dann benötigt werden, wenn das gesamte Parcel standardmäßig ein Rückgabewert einer Methode ist.

Die letzte Methode ist lediglich dazu da, eine Bitmaske zu erzeugen, um etwaige zusätzliche Details zu liefern – wir setzen den Wert stets auf 0:

```
@Override
public int describeContents() {
  return 0;
}
```

Soweit zu einer Klasse, welche das Parcelable Interface implementiert. Um die Funktion nun testen zu können, bauen wir uns eine App, welche zwei Acitivties aufweist. Die MainActivity beinhaltet einen Send Button und sechs EditText Views. Die SubActivity lediglich sechs TextViews:



MainActivity SubActivity

Zuerst ein paar Details über die Main Activity. Das erste EditText View Feld wird im Layout als android:inputType="number" und das zweite Feld als android:inputType="numberDecimal" gesetzt, so dass lediglich Ziffern, bzw. für das zweite Feld Ziffern mit Kommastelle eingegeben können. Die anderen Felder können mit android:inputType="textNoSuggestions" geflagged werden, damit die Eingabe einfacher ist.
Folgende Namenskonventionen sollen gelten – damit wir uns später im Code nicht verirren:

Viewpos:	**View Art:**	**ViewID:**	**Beschreibung:**
0	Button	sendButtonMain	Einfacher Button, welcher beim Klick die Methode myClickMethod aufruft.
1	EditText	editIntMain	Eingabefeld für eine ganze Zahl (inputType="number")
2	EditText	editDoubleMain	Eingabefeld für eine ganze Zahl (inputType="numberDecimal")
3	EditText	editStringAMain	Eingabefeld für beliebigen Text
4	EditText	editStringBMain	Eingabefeld für beliebigen Text
5	EditText	editStringArray0Main	Eingabefeld für beliebigen Text
6	EditText	editStringArray1Main	Eingabefeld für beliebigen Text

Wie ihr seht, geht es hier darum Datenwerte zu erzeugen. Ziel wird es sein, diese Datenwerte in unser Objekt der Klasse „MyParcel“ zu schreiben und mittels einem Intent an die SubActivity zu schicken. Hier haben wir

nun lediglich die Notwendigkeit die Daten anzuzeigen. Insofern ist die SubActivity auch relativ simpel aufgebaut:

Viewpos:	View Art:	ViewID:	Beschreibung:
0	TextView	showIntSub	Textanzeige
1	EditText	showDoubleSub	Textanzeige
2	EditText	showStringASub	Textanzeige
3	EditText	showStringBSub	Textanzeige
4	EditText	showStringArray0Sub	Textanzeige
5	EditText	showStringArray1Sub	Textanzeige

Das Layout könnt ihr selbst wählen – ich habe ein simples vertikal ausgerichtetes LinearLayout gewählt. Sehen wir uns nun die beiden wichtigen Stellen im Code an – und zwar zuerst die Stelle, an der wir die Daten aus dem MainActivity in MyParcel schreiben und mit Hilfe des Intents an die SubActivity schicken:

```
public void myClickMethod(View view) {
  double myDoubleData = Double.parseDouble(((EditText)
     findViewById(R.id.editDoubleMain)).getText().toString());
  int myIntData = Integer.parseInt(((EditText)
     findViewById(R.id.editIntMain)).getText().toString());
  String myStringDataA = ((EditText)
     findViewById(R.id.editStringAMain)).getText().toString();
  String myStringDataB = ((EditText)
     findViewById(R.id.editStringBMain)).getText().toString();
  String[] myStringArrayData = new String[2];
  myStringArrayData[0] = ((EditText)
     findViewById(R.id.editStringArray0Main)).getText().toString();
  myStringArrayData[1] = ((EditText)
     findViewById(R.id.editStringArray1Main)).getText().toString();

  Intent intent = new Intent(this, SubActivity.class);
  MyParcel myParcel = new MyParcel();
  myParcel.setMyDouble(myDoubleData);
  myParcel.setMyInt(myIntData);
  myParcel.setMyString(myStringDataA);
  myParcel.setMyString(myStringDataB);
  myParcel.setMyStringArray(myStringArrayData);
  intent.putExtra(EXTRA_MESSAGE_DEF, myParcel);
  startActivity(intent);
}
```

Der erste Schritt ist, die einzelnen Werte in lokale Variablen zu schreiben. Dabei müssen wir bei dem Ganzzahlenwert (int) und dem Gleitkommawert (double) den String aus unserem EditText Feld parsen. Da die EditText View den Text nur als „Editable“ liefert, müssen wir hieraus erstmal einen String machen. Auf ein eventuelles Errorhandling verzichten wir hier, da Errorhandling nicht im Fokus dieses Kapitels steht.
Die restlichen Werte können ganz normal ausgelesen werden und in die Stringvariablen, bzw. das Stringarray eingetragen werden.

Danach wird der Intent für die Erzeugung unserer SubActivity und das MyParcel Objekt erzeugt. Hier verwenden wir unseren Standardkonstruktor – wenn ihr einen anderen Konstruktor geschrieben habt, dann verwendet diesen. Lediglich der Konstruktor mit dem „Parcel“ Argument dürft ihr hier nicht verwenden, der ist für den CREATOR bestimmt.

```
  Intent intent = new Intent(this, SubActivity.class);
  MyParcel myParcel = new MyParcel();
```

Nun können mit den Settermethoden welche ihr geschrieben habt die Werte in myParcel eingetragen werden. Solltet ihr andere Namen für die Setter verwendet haben, dann müsst ihr entsprechend eure Namen hier eintragen. Am Schluss wird myParcel in den Intent gepackt und die SubActivity gestartet.

Soviel zur MainActivity. Widmen wir uns nun der SubActivity. Hier gilt es, in der onCreate Methode die einzelnen Werte aus dem Intent rauszuholen und in die entsprechenden TextView Elemente zu platzieren:

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_sub_activity);

  Intent intent = getIntent();
  MyParcel myParcel = (MyParcel)
        intent.getParcelableExtra(MainActivity.EXTRA_MESSAGE_DEF);
  ((TextView)findViewById(R.id.showDoubleSub)).setText(
        String.valueOf(myParcel.getMyDouble()));
  ((TextView) findViewById(R.id.showIntSub)).setText(
        String.valueOf(myParcel.getMyInt()));
  ((TextView) findViewById(R.id.showStringASub)).setText(
        myParcel.getMyStringA());
  ((TextView) findViewById(R.id.showStringBSub)).setText(
        myParcel.getMyStringB());
  ((TextView) findViewById(R.id.showStringArray0Sub)).setText(
        myParcel.getMyStringArray()[0]);
  ((TextView) findViewById(R.id.showStringArray1Sub)).setText(
        myParcel.getMyStringArray()[1]);
}
```

Wir holen uns also die einzelnen Viewelemente und setzen den Text entsprechend unserer Parcelwerte. Wenn wir nun die App starten, dann sehen wir, dass wir in der MainActivity die Werte eintragen können und nach dem Senden diese wieder in unserer SubActivity erscheinen.

Nun wollen wir noch die oben gemachte Aussage prüfen, dass die Reihenfolge im Konstruktor und in der Methode writeToParcel identisch sein muss. Tauschen wir also im Konstruktor die beiden Variablen myStringDataA und myStringDataB aus:

```
  private MyParcel(Parcel in) {
    myStringArrayData = new String[2];
    myIntData = in.readInt();
    myDoubleData = in.readDouble();
    myStringDataB = in.readString();
    myStringDataA = in.readString();
    in.readStringArray(myStringArrayData);
  }
```

Wenn wir nun die App neu starten sehen wir, dass die mittleren beiden Werte vertauscht wurden. Dem nicht genug – auch wenn wir die Reihenfolge von Zuweisungen Variablen unterschiedlichen Datentyps vertauschen, dann kommt mitunter kompletter Blödsinn raus. Also bitte immer daran denken – immer die Reihenfolge einhalten!

Fazit: Wenn ihr eigene Objekte in euren Apps habt welche an andere Activities gesendet werden müssen, dann könnt ihr in euren Objekten das Parcelable Interface implementieren und somit relativ bequem die Objekte an andere Activities senden – sofern ihr die oben genannten Schritte einhaltet. Im oben genannten Beispiel wurde „nur“ ein Objekt zur Demonstration der Kommunikation erzeugt – in euren Apps habt ihr eventuell Spielinformationen in Objekten gelagert, welche in verschiedenen Activities zu bewerten sind.

3.6 Eigenschaften eines Intents

Nun haben wir uns ein wenig mit Intents auseinandergesetzt und die ersten Berührungsängste verloren. Intents stellen eine der wichtigsten Elemente in der Android Programmierung dar – insofern liegt der Gedanke nahe, dass sie noch etwas mehr können, als „nur" Activities starten. Fangen wir also mal mit den wesentlichen Eingenschaftsmerkmalen an. Weitere Details zu Intents findet ihr unter folgenden Webseiten:

http://developer.android.com/reference/android/content/Intent.html
http://developer.android.com/guide/components/intents-filters.html

Aus diesen Seiten stammen auch die meisten Informationen in den folgenden Kapiteln.

3.6.1 ComponentName

Dieser optionale Parameter macht aus einem Intent einen sogenannten „expliziten Intent" – also ein Intent, der für exakt eine Komponente bestimmt ist. Alle anderen Komponenten würden diesen Intent ignorieren. Wir haben unsere Intents bis jetzt immer explizit formuliert, da wir ja eine ganz bestimmte Activity starten wollten. Wir erinnern uns an den Intent Konstruktor der vorausgegangenen Beispiele:

```
Intent intent = new Intent(this, SubActivity.class);
```

Wir teilen dem Intent mit dem zweiten Parameter mit, dass die Activity mit dem Namen „SubActivity" (im gleichen Package in der die aktuelle Klasse liegt) den Intent verarbeiten soll, und keiner sonst. Sollten wir eine Activity starten wollen, welche in einem anderen Package liegt, so müssen wir dieses explizit angeben:

```
Intent intent = new Intent(this, com.yourExample.YourSubActivity.class);
```

Der Wert kann auch über eigene Settermethoden (setComponent, setClass oder setClassName) nachträglich gesetzt werden.
Sollte ein Intent die Eigenschaft der Component nicht gesetzt haben, so liegt ein impliziter Intent vor – das System versucht somit aufgrund der anderen (unten beschriebenen) Eigenschaften einen geeigneten Empfänger für den Intent zu finden.

3.6.2 Action

Die Action definiert entweder, was den Intent ausgelöst hat („broadcast intents") oder was zu tun ist. Ein Beispiel für eine Action, welche die durchzuführende Aktion beschreibt ist ACTION_MAIN, welche einen Startpunkt für eine Activity erzeugt – sie wird immer genutzt, wenn wir eine App starten. Ein weiteres Beispiel währe ACTION_EDIT, welche einen schreibenden Zugriff auf bestimmte Daten erzeugt. Unter folgendem Link könnt ihr weitere vordefinierte Werte für Actions sehen.

http://developer.android.com/reference/android/content/Intent.html#Constants

Es ist allerdings auch möglich, eigene Action Werte zu definieren. Um Verwechslungsmöglichkeiten auszuschließen sollte der volle Packagename von euch in den String eingebaut werden, bspw. „com.myExample.myProject.DO_SOMETHING".

3.6.3 Data

Diese Eigenschaft beinhaltet eine URI (Uniform Resource Identifier), also eine eindeutige Adresse einer Ressource und den MIME Type (Multipurpose Internet Mail Extensions Type), also eine Klassifizierung des Datentyps einer Ressource. Es ist grundsätzlich anzumerken, dass die Daten natürlich zur Action passen müssen. Wenn bspw. die Action keine Daten benötigt, so bleibt diese Eigenschaft leer.

Mögliche Inhalte der URI sind, wenn wir bspw. eine Internetadresse anzeigen wollen, so beinhaltet die URI eine http Adresse. Bei einem Textfile, welches zu öffnen und ändern ist, so beinhaltet die URI die Adresse des Dokuments etc.
Mögliche Inhalte des MIME Types sind nicht Android spezifisch, sondern global gültig. Trotzdem habe ich hier mal ein paar grundlegende MIME Typen aufgelistet:

MIME Type:	Inhalt:
/	Alle Formate
image/*	Alle Image Formate
image/jpeg	Jpeg codierte Bilder
image/jpg	Jpeg codierte Bilder
image/png	Png codierte Bilder
image/bmp	Bitmaps
audio/*	Alle Audio Formate
audio/mpeg	MP3 codierte Audioformate
audio/mid	Midi Daten
audio/wav	Wav codierte Audioformate
video/*	Alle Videoformate
video/mpeg	MPEG codierte Videoformate
video/3gpp	3GPP codierte Videoformate
text/*	Alle Textformate
text/plain	Unformatierte Texte
text/csv	Kommaseparierte Textdateien
text/html	HTML Dateien
text/enriched	Text mit Formatierungen (kursiv, fett etc.)

Die Daten werden mit folgenden Methoden gesetzt:
Nur die URI: setData()
Nur der MIME Typ: setType()
Beides: setDataAndType()

3.6.4 Category

Die Category gibt noch weitere (allgemeinere) Informationen über die Komponente, welche den Intent verarbeiten soll. Wenn bspw. die Category CATEGORY_PREFERENCE gesetzt wurde, dann muss die verarbeitende Activity ein Preference Panel sein. Die einzelnen Categories können unter folgendem Link gefunden werden:

http://developer.android.com/reference/android/content/Intent.html#Constants

Es können pro Intent mehrere Categories gesetzt werden – insofern lautet die Methode nicht „set“ sondern: addCategory()

3.6.5 Extras

Hier werden die individuellen Daten gespeichert. Weiter Oben haben wir gesehen, wie wir Daten in die Extras reinschreiben und auch wieder herausholen. Darüber hinaus gibt es noch Actions, welche standardisierte Extras Daten erwarten. Beispielsweise hat die Action „ACTION_HEADSET_PLUG“ noch die Extra Werte state, name und microphone. ACTION_TIMEZONE_CHANGED liefert noch den Extra Wert „time-zone“. Details hierzu findet ihr hier:

http://developer.android.com/reference/android/content/Intent.html#Constants

3.6.6 Flags

Den letzten Block bilden die Flags. Hier können noch weitere Verhaltenseinstellungen – meist im Zusammenhang mit der gewählten Action gesetzt werden. Für unsere ersten Apps werden wir diese Flags noch nicht brauchen, wer die existierenden Flags durchgehen möchte, findet sie hier:

http://developer.android.com/reference/android/content/Intent.html#setFlags(int)

3.7 Der Intent Filter

Wie wir gelernt haben, definiert mir die Existenz einer Komponente einen Intent als „explizit“. Fehlt diese Information sprechen wir von einem Impliziten Intent. Für solche (und nur solche) sieht Android eine Möglichkeit vor, die für die Activity (und später auch Services, bzw. Broadcast Receiver) interessanten Intents herauszufiltern. Nur die Activities, welche nicht durch ein Filterkriterium aussortiert werden, gelangen zur Activity vor. Die Filter werden über die folgenden (Kombinationen) von Werten spezifiziert:

- Action
- Category
- Data

Einen Intent Filter haben wir bereits gesehen und zwar zum Starten unserer MainActivity. Da Intent Filter festlegen, ob eine Activity gestartet wird, finden wir die Filter auch im Manifest File und nicht im Code (Ausnahmen bilden dynamisch geladenen Broadcast Recevier – doch die werden wir erst in einem späteren Stadium besprechen). Sehen wir uns unser Manifest File der letzten App mal diesbezüglich an – dort finden wir einen Bereich, der die MainActivity genauer beschreibt:

```
<activity
  android:name="com.example.course.MainActivity"
  android:label="@string/app_name" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Wie wir sehen, wird die MainActivity gestartet, wen nein Intent mit der Action “MAIN”, also Erzeugung eines Entry Points und der Category „LAUNCHER“, also Anzeige im „Top Level“ als neue App eintrifft. Zu beachten ist hier, dass die Java Stringkonstante android.content.Intent.ACTION_MAIN im XML File als android.intent.action.MAIN zu finden ist. Gleiches gilt für die Category, in der der Java Präfix „CATEGORY_“ in XML als eigenes Verzeichnis gewertet wird „category.“.

Prinzipiell ist die Nutzung des Intent Filters relativ geradlinig. Deshalb möchte an dieser Stelle lediglich auf den URI Filter eingehen, da wir hier im Zweifelsfall am ehesten dran schrauben werden und der noch ein paar Zusatzinfos benötigt. Wenn wir uns (basierend auf dem Beispiel aus http://developer.android.com/guide/components/intents-filters.html) folgende URL ansehen:

content://com.example.project:200/folder/subfolder/etc

und davon ausgehen, dass sich an dieser Stelle ein Video befindet, dann wird der Intent Filter wie folgt darauf angesetzt:

```
<intent-filter>
  <data android:mimeType="video/mpeg"
    android:scheme="content"
    android:host="com.example.project"
    android:port="200"
    android:path="folder/subfolder/etc"
  />
</intent-filter>
```

Die einzelnen Elemente sind zwar optional, wobei sie trotzdem in einer sinnvollen Kombination auftreten müssen. Hier noch der Hinweis, dass ich beliebig viele „data“ Elemente innerhalb dem Intent Filter eintragen kann, wobei beim ersten „Treffer“ die nachfolgenden nicht weiter beachtet werden müssen.

Die Frage ist nun, was passiert mit meinem Intent, wenn mehrere Activities den Intent verarbeiten könnten. In solch einem Fall kann der Intent als ACTION_CHOOSER gesetzt werden – oder alternativ kann der Intent mittels Intent.chreateChooser() erzeugt werden. Dies würde für den Orangegurt etwas zu weit führen – wir werden uns später darum kümmern.

3.8 Browserstart via Intent

Als letztes Beispiel wollen wir ein Browserfenster öffnen und dort eine vorgegebene URL anzeigen lassen. Hierfür erzeugen wir einen Intent, welcher als Action den Wert ACTION_VIEW beinhaltet (also Anzeigen der vorgegebenen Daten):

```
Intent intent = new Intent(Intent.ACTION_VIEW);
```

Danach übergeben wir dem Intent die notwendigen Daten – für uns ist das lediglich eine Internetadresse, deren Inhalt wir anzeigen lassen wollen:

```
intent.setData(Uri.parse("http://www.codeconcert.de/"));
```

Am Schluss teilen wir dem Empfänger mit, das seine Activity durch den Intent gestartet werden, soll – wir wollen schließlich ein Browserfenster öffnen, was nichts anderes ist, als eben eine Activity:

```
startActivity(intent);
```

Wenn ihr diesen Code einfach in die Methode legt, welche einem Button zugeordnet ist, könnt ihr die Funktion prüfen.

4 Lizenz

5 Haftung

Ich übernehme keinerlei Haftung für die Richtigkeit der hier gemachten Angaben. Sollten Fehler in dem Dokument enthalten sein, würde ich mich über eine kurze Info unter maik.aicher@gmx.net freuen.