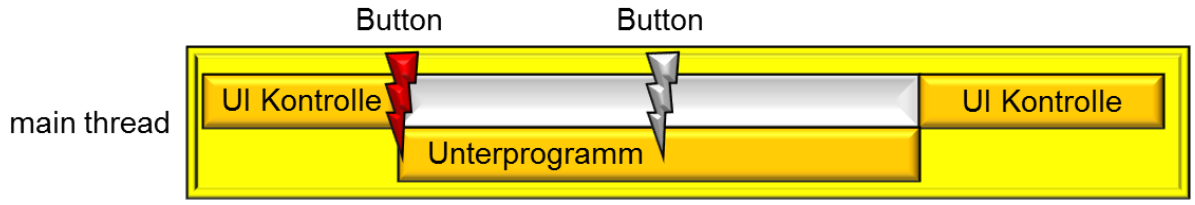
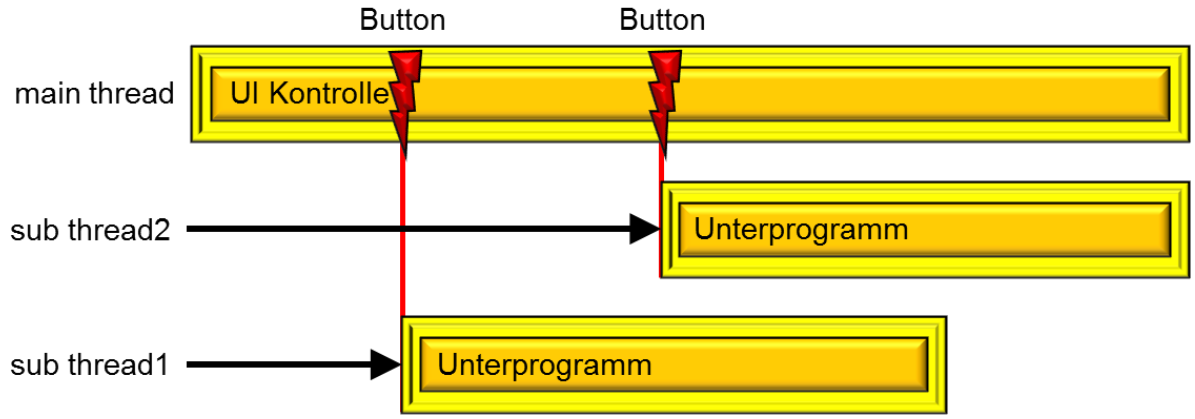
Inhaltsverzeichnis

1 Threading

Bei Android Applikationen wird für das UI (User Interface) ein eigener Therad gestartet, welcher sich um das Handling der einzelnen sichtbaren Elemente kümmert. Wenn wir bspw. einen Button anklicken, dann ruft dieser Thread bspw. die onClicked Methode auf, wo wir Code hinterlegen, der dem Klickbefehl eine sinnvolle Funktion zuweist. Dieser Thread kann auch auf dem zugrundeliegenden Linux System identifiziert werden und wird der „main thread“ oder auch „UI thread“ genannt.
Das Problem ist nun, dass wenn dieser Button einen aufwändigeren Prozess startet, dann würde die Userinteraktion für diesen Zeitraum unterbrochen werden. Im folgenden Bild ist der UI thread aufgezeigt:



Wie wir in der Grafik sehen, kümmert sich der UI thread um die UI Kontrolle. Sobald ein Button angeklickt wird, startet dieser ein von uns geschriebenes Unterprogramm. Würde dieses nur sehr kurz sein, so hätten wir kein Problem. Wenn es aber, wie in dieser Grafik dargestellt, relativ lange läuft, dann könnte sich der UI thread nicht mehr um die UI Kontrolle kümmern – ein eventueller Klick auf einen Button wäre somit wirkungslos. Wir haben den Eindruck, dass unsere App nicht mehr reagiert (sog. Screen Freeze). Nach ca. 5 Sekunden würde darüber hinaus Android den "application not responding" (ANR) Dialog anzeigen. Dies ist auf jeden Fall zu vermeiden. Die Standardlösung für solche Situationen ist ein zweiter Thread. Dies ist ein parallel laufender Prozess, welcher sich um unser Unterprogramm kümmert, so dass der UI thread nach wie vor sich um die UI Kontrolle kümmern kann:



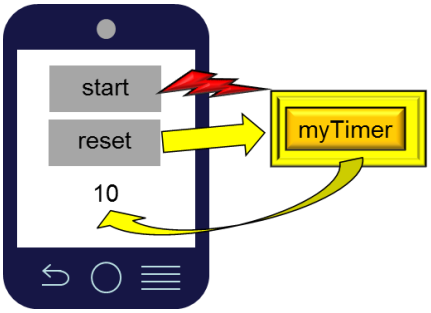
Wie ihr seht, startet der Button nun einen neuen Task, in dem das Unterprogramm läuft. Die UI Kontrolle kann somit ungehindert weitergehen und ein erneuter Klick auf den Button wird problemlos verarbeitet.

In diesem Dokument werden wir uns nun um die Möglichkeiten kümmern, solche sub threads zu erzeugen und zu nutzen. Der erfahrene Java Entwickler wird jetzt dazu neigen einfach einen neuen Thread zu erzeugen – doch in Android würden wir hier auf ein zweites Problem stoßen. Wenn wir – und das ist sehr oft der Fall – aus dem erzeugten Thread wieder auf die UI zugreifen wollen, dann würden wir eine Forderung von Android verletzten und zwar, dass kein anderer Thread auf den UI Thread zugreifen darf. Android lässt sich also ungerne ins Handwerk pfuschen. Doch die Android Entwickler haben hierfür nun Möglichkeiten vorgesehen, dieses Problem nun doch zu lösen. Die erste Lösung startet zwar einen unabhängigen Thread, hängt aber sämtliche Manipulationen an der UI an den UI Thread. Die zweite Lösung nutzt einen Pool von Background Threads der UI, welche für unsere Belange genutzt werden. Wir werden uns beide Möglichkeiten ansehen. Die Möglichkeiten von „Services“ werden wir in einer späteren Gürtelfarbe kennenlernen.

Wer im Netz nach diesem Thema sucht, findet unter dem folgenden Link einen guten Startpunkt:

http://developer.android.com/guide/components/processes-and-threads.html

2 Umzusetzende Funktionalität



Wir werden folgende Situation umsetzen. Wir haben eine Activity, welche aus zwei Buttons und einem TextView besteht. Wenn wir den Start Button klicken, dann soll ein Timer von 10 rückwärts bis 0 zählen und den aktuellen Zählwert in dem TextView der Activity anzeigen. Wenn wir vorher den Reset Button drücken, dann wird der Timer wieder auf 10 gesetzt. Wenn wir den Start Button klicken solange der Timer läuft, passiert nichts – der Startbutton ist während der Timer läuft also blockiert. Folgende Funktionen sollen damit getestet werden.

- Starten eines Threads
- Zugriff vom Thread auf die Activity
- Zugriff von der Activity auf den Thread

Da wir für alle Lösungen das gleiche Layout verwenden, hier unser **activity_main.xml** File:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:gravity="center"
  tools:context=".MainActivity">

    <Button
      android:id="@+id/bttn_start"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="bttnStartClicked"
      android:text="@string/button_start"/>
    <Button
      android:id="@+id/bttn_reset"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="bttnResetClicked"
      android:text="@string/button_pause"/>

    <TextView
      android:id="@+id/text_counter"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/empty_value"/>

</LinearLayout>
```

Wichtig für uns sind lediglich, dass wir später die beiden Methoden „bttnResetClicked“ und „bttnStartClicked“ in unserer MainActivity einfügen.

Die Stringwerte könnt ihr nach belieben setzen.

3 Lösung mit eigenem Thread

3.1 Die einfache Thread Lösung

Wir versuchen zuerst die einfache Thread Lösung, welche den wesentlichen Teil richtig umsetzt, aber an einigen Stellen an seine Grenzen stößt. Sehen wir uns erst mal folgenden Code an:

```
public class MainActivity extends Activity {
    private int iCounter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }

    public void bttnStartClicked(View myView) {
        new Thread(new Runnable() {
            public void run() {
                iCounter = 10;
                while (iCounter >= 0) {
                    getTextView().post(new Runnable() {
                        public void run() {
                            getTextView().setText(
                            String.valueOf(iCounter));
                        }
                    });
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    iCounter--;
                }
            }
        }).start();
    }

    private TextView getTextView() {
        return (TextView) findViewById(R.id.text_counter);
    }
}
```

Das eigentlich spannende ist die Methode „bttnStartClicked“. Hier erzeugen wir unseren Arbeitsthread, der den Timer laufen lassen soll. In den Thread packen wir eine innere Klasse, welche die Arbeit verrichten soll. Damit sie über den Thread gesteuert werden kann, muss sie das Interface „Runnable“ implementieren – oder als innere Klasse eben vom Typ Runnable sein. Diese muss mindestens die Methode „run“ aufweisen:

```
new Thread(new Runnable() {
    public void run() {
```

Hier setzen wir unseren Counter auf 10 und lassen eine Schleife laufen, welche solange läuft, solange die Zählvariable iCount größer oder gleich 0 ist:

```
iCounter = 10;
while (iCounter >= 0) {
  // hier kommt die eigentliche Thread Arbeit rein
    iCounter--;
}
```

Nun implementieren wir das Innere der Schleife. Zuerst kümmern wir uns darum, dass die TextView unserer Activity unseren Zählwert anzeigt. Hierzu müssen wir eine neue Methode unserer Views nutzen, die post Methode. Diese platziert in die Messagequeue der UI Kontrolle einen Befehl, der ausgeführt wird, sobald die UI Kontrolle dazu Zeit hat. Damit dieser Befehl von der Messagequeue richtig verarbeitet werden kann muss es ebenfalls das Interface Runnable implementieren und entsprechend die run Methode aufweisen. Wichtig hier zu verstehen ist, dass die Ausführung unseres Updates des TextViews nicht sofort, sondern erst verzögert stattfindet. Das ist für den Moment zwar noch nicht so tragisch, in dem nächsten Kapitel wird uns dieses Verhalten aber nochmal beschäftigen.

```
getTextView().post(new Runnable() {
    public void run() {
        getTextView().setText(String.valueOf(iCounter));
    }
});
```

Die Tatsache, dass wir unsere Threadrelevanten Klassen als innere Klassen realisieren ermöglicht es uns, direkt auf die einzelnen Objekte, Methoden und Variablen unserer Klasse zuzugreifen. Das ist zwar praktisch, es wird sich aber herausstellen, dass die inneren Klassen auch seine Nachteile aufweisen. Doch zuerst sehen wir uns noch die sleep Funktion an, welche unseren Timer im Sekundentakt laufen lässt. Thread.sleep(1000) erzeugt also ein Pause von 1000 Milisekunden und muss in einen Try/Catch Block:

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

In einem realen Projekt müsste die Fehlerbehandlung etwas ausgefeilter sein – aber für uns soll das mal in dieser Form reichen. Am Schluss müssen wir unseren neu erstellten Thread noch mit der Methode .start() starten, was den Thread dazu bringt in unserer inneren Klasse die Methode „run“ aufzurufen.

```
}).start();
```

Wenn wir nun unser Programm starten sehen wir, dass der Timer mit dem Start Button tatsächlich startet und der Zähler sauber runterläuft. Da wir den Reset Button noch nicht mit einer Funktion belegt haben, werden wir ihn jetzt auch noch nicht betätigen. Zu beobachten ist aber auch, dass wenn wir den Start Button zweimal hintereinander drücken, wir scheinbar zwei Timer am Laufen haben, welche beide unseren TextView neu belegen. Wir haben also zwei parallele Threads erzeugt, die erst wieder terminieren, wenn die While-Schleife in unserer inneren Klasse abgelaufen ist. Dieses Verhalten müssen wir irgendwie unterbinden.

Wir benötigen also irgendeine Möglichkeit, von der Activity aus auf unsere innere Klasse zuzugreifen. Das Problem ist, dass wir keine public Methoden einbauen können, da wir ja lediglich unser Interface „Runnable“ als Klassendefinition verwendet haben und dieses ja nur die „run“ Methode kennt. Gleiches gilt für die Notwendigkeit schreibend auf unsere innere Klasse zuzugreifen um den Timer zurückzusetzen. Wie bereits erwähnt, stößt unser Konzept der inneren Klassen an seine Grenzen. Im nächsten Kapitel werden wir dieses Problem umgehen.

3.2 Thread Lösung mit eigener Klasse

Wie wir gesehen haben, müssen wir uns etwas überlegen, damit wir aus der Activity heraus auf unseren Arbeitsthread zugreifen können. Dies funktioniert dann, wenn wir auch tatsächlich eine eigene Klasse erzeugen, welche das Interface „Runnable“ implementiert. Hier haben wir nun die komplette Freiheit unsere inneren Variablen und Methoden so zu designen, wie wir es benötigen. Wir sehen uns einen beispielhaften Code für solch eine Implementierung. Wir erzeugen also eine neue Klasse in dem Package unseres Projektes und nennen diese „MyTimer“:

```
public class MyTimer implements Runnable{
    private int iCounter = 0;
    private int iStartValue = 10;
    private boolean bIamBusy = false;
    private TextView myTextView;

    @Override
    public void run() {
        bIamBusy = true;
        resetMyTimer();
        while (iCounter > 0) {
            iCounter--;
            myTextView.post(new Runnable() {
                public void run() {
                    myTextView.setText(String.valueOf(iCounter));
                }
            });
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        bIamBusy = false;
    }

    public MyTimer(int iValue, TextView myTextViewToSet) {
        iStartValue = iValue;
        myTextView = myTextViewToSet;
    }

    public boolean isBusy() {
        return bIamBusy;
    }

    public void resetMyTimer() {
        iCounter = iStartValue + 1;
    }
}
```

Wie ihr seht, hat sich das eigentliche Konzept nicht geändert – wir haben lediglich ein paar kosmetische Eingriffe durchgeführt. Sehen wir uns den Code im Einzelnen an. Zuerst brauchen wir die Klasse, welche das Interface „Runnable“ implementiert – weshalb wir weiter unten auch die Methode „run“ brauchen:

```
public class MyTimer implements Runnable{
```

Als nächstes brauchen wir ein paar Klassenvariablen. iCounter wird die Zählvariable sein, welche auch in der Activity angezeigt wird. iStartValue ist eine Variable, mit deren Hilfe wir den Zähler flexibel auf einen Wert setzen können. bIamBusy soll anzeigen, ob der Thread gerade läuft oder nicht. myTextView ist eine

Referenz auf die TextView, da wir diese ja updaten wollen. Wir erinnern uns, dass wir in der vorausgegangen Lösung eine innere Klasse verwendet haben und dort direkt auf unseren TextView Zugriff hatten. Hier funktioniert das eben nicht mehr.

```
private int iCounter = 0;
private int iStartValue = 10;
private boolean bIamBusy = false;
private TextView myTextView;
```

Soweit dürfte das keine Probleme bereiten. In der run Methode wird sich jedoch einiges ändern. Zuerst setzen wir unser bIamBusy Flag auf true, damit wir später erkennen können, dass unser Thread gerade läuft Weiterhin setzen wir unseren Timer mit einer eigenen Methode zurück. Wenn wir das alles erledigt haben, können wir die Schleife starten:

```
@Override
public void run() {
    bIamBusy = true;
    resetMyTimer();
    while (iCounter > 0) {
```

Nun zur ersten Änderung – wir haben im Obigen Beispiel die Erhöhung unseres Zählers an das Ende der run Methode gelegt, was ja korrekt funktioniert hat. Das Problem ist, dass wenn wir unseren Zähler zurücksetzen, dies aller Wahrscheinlichkeit nach dann sein wird, wenn der Thread gerade auf .sleep läuft, da er in dieser Zeile nahezu die gesamte Verarbeitungszeit verbringt. Wenn aber in genau dieser Zeit der Zähler wieder auf 10 gesetzt wird, ist der nächste Schritt die Reduktion des Zählers auf 9 und danach wird die Schleife wieder neu gestartet. Wir würden die 10 also nie sehen.
Insofern müssen wir die Zählerreduktion vor den sleep Befehl setzen. Jetzt haben wir aber Oben gelernt, dass wir das Update des TextView Elements über eine Messagequeue erledigt wird und wir somit nicht wirklich sagen können, ob der Update sofort, oder mit einer gewissen Verzögerung vonstattengeht. Es könnte also sein, dass wir zwar zuerst das setText aufrufen und danach den Zähler reduzieren, der eigentliche Update aber nach der Reduktion erst erfolgt. Insofern habe ich die Reduktion tatsächlich vor den Update geschrieben im Bewusstsein, dass unser Zähler nun immer einen Wert zu niedrig anzeigen wird. Aus diesem Grunde werden wir beim Reset des Zählers den Wert einfach um 1 erhöhen und die Schleife nicht mit der Bedingung >= 0 sondern > 0 laufen lassen.

```
            iCounter--;
            myTextView.post(new Runnable() {
                public void run() {
                    myTextView.setText(String.valueOf(iCounter));
                }
            });
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        bIamBusy = false;
    }
```

Am Schluss setzen wir unser Busyflag auf „false“, da in diesem Moment unser Thread terminieren wird.

Wenn wir nun unsere Klasse verwenden wollen, dann müssen wir nur noch sicherstellen, dass wir unseren Timer mit einem Zählwert initialisieren und weiterhin unseren TextView übergeben bekommen.

Da unser Timer nur dann sinnvoll funktioniert, packen wir diese beiden Werte gleich in den Konstruktor hinein:

```
public MyTimer(int iValue, TextView myTextViewToSet) {
     iStartValue = iValue;
     myTextView = myTextViewToSet;
}
```

Die beiden Methoden resetMyTimer und isBusy sind nun selbsterklärend. Sehen wir uns nun die MainActivity mit den Änderungen an:

```
public class MainActivity extends Activity {
     private MyTimer myTimer;

     @Override
     protected void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.activity_main);
          myTimer = new MyTimer(10,
            (TextView) findViewById(R.id.text_counter));
     }

     @Override
     public boolean onCreateOptionsMenu(Menu menu) {
          getMenuInflater().inflate(R.menu.activity_main, menu);
          return true;
     }

     public void bttnStartClicked(View myView) {
          if (!myTimer.isBusy()) {
               new Thread(myTimer).start();
          }
     }

     public void bttnResetClicked(View myView) {
          myTimer.resetMyTimer();
     }
}
```

Zuerst sehen wir, dass wir als Klassenvariablen nun unsere Klasse haben – die Zählvariable ist in unsere Klasse gewandert. In der OnCreate Methode erzeugen wir unser Timerobjekt, indem wir den Konstruktor mit unserem Zählwert und unserer TextView aufrufen und das Objekt in die Klassenvariable schreiben:

```
myTimer = new MyTimer(10, (TextView) findViewById(R.id.text_counter));
```

In der bttnStartClicked Methode prüfen wir nun, ob unsere Methode gerade beschäftigt ist. Wenn nicht, können wir einen neuen Thread starten und diesem unseren Timer übergeben – wir erinnern uns, wir haben ja das Interface „Runnable“ implementiert:

```
if (!myTimer.isBusy()) {
     new Thread(myTimer).start();
}
```

In der Methode bttnResetClicked können wir nun bequem auf unser Objekt zugreifen und den Zäher wieder auf den Ursprungszustand setzen. Wenn wir die Applikation nun starten sehen wir, dass alle unsere funktionalen Anforderungen erfüllt sind.

3.3 Fazit zur Thread Lösung

Grundsätzlich ist mit dieser Lösung alles möglich. Wir müssen lediglich unsere Klasse, welche im Thread läuft flexibel genug gestalten. Wichtig zu verstehen ist, dass wir mit der .post Methode den Befehl an unsere View in die Messagequeue schreiben, welche asynchron verarbeitet wird. Weiterhin müssen wir verstehen, dass wir die .post Methode des Views genutzt haben. Die Activity bietet auch eine Möglichkeit, mit ihr zu kommunizieren. Hier eine kurze Aufstellung:

Objekt:	**Methode:**	**Funktion:**
View	.post(Runnable)	Der Befehl im „Runnable“ Objekt wird an die Messagequeue gesendet und im UI thread verarbeitet.
View	.postDelayed(Runnable, long)	Der Befehl im „Runnable“ Objekt wird an die Messagequeue gesendet und nach der angegebenen Zeit (2. Parameter in Millisekunden) im UI thread verarbeitet.
Activity	.runOnUIThread(Runnable)	Der Befehl im „Runnable“ Objekt wird an die Messagequeue gesendet und im UI thread verarbeitet.

Eine weitere Möglichkeit, die Kommunikation zwischen einem Thread und der Activity zu ermöglichen ist eine Instanz einer Handler Klasse zu erstellen. Vom Konzept her ist es identisch zu den oben gezeigten Methoden – der Vorteil ist, dass wir eine eigene Handlerklasse realisieren können (durch Vererbung aus der Android.os.Handler Klasse). Wenn diese Klasse in der Activity instanziiert wurde, können wir mit handler.post(…) auch Runnables an die Activity schicken. Für’s Erste sollten die oben genannten Möglichkeiten für uns allerdings ausreichend sein.

4 AsyncTask

Die AsyncTask Klasse ermöglicht es uns, die zeitaufwändigen Aktivitäten nicht in einem eigenen Thread, sondern in einem Pool von Hintergrundthreads des main threads laufen zu lassen. Das Handling dieser Klasse ist etwas gewöhnungsbedürftig – aber wir sehen uns das Schritt für Schritt an. Das Grundprinzip ist, dass wir eine Klasse deklarieren, welche unter anderem folgende Methoden aufweist:

onPreExecute()
Diese Methode wird aufgerufen, sobald wir die Methode .execute(params) auf unser Objekt anwenden. Sie muss nicht zwingend von uns überschrieben werden. Üblicherweise wird sie verwendet, um irgendwelche Vorbereitenden Schritte, wie die Anzeige einer Progressbar etc. zu realisieren.

doInBackground(Params... params)
Diese Methode muss umgesetzt sein. Sie erwartet einen Parameter (welcher immer ein Array von Objekten ist), welcher von dem Aufruf der Methode .execute(params) übernommen wird. Der Typ des Parameters wird bei der Klassendeklaration festgelegt (siehe Unten).

onProgressUpdate(Progress... values)
Diese Methode wird nicht vom UI thread aufgerufen, sondern wenn wir „publishProgress“ innerhalb unserer Klasse (im Regelfall ist dies inerhalb der Methode doInBackround) aufrufen. Sie benötigt einen Parameter, welcher ebenfalls ein Array von Objekten ist. Auch dieser Typ wird bei der Klassendeklaration festgelegt (siehe Unten). Die Methode muss nicht überschrieben werden, sollte aber, wenn wir publishProgress nutzen wollen.

onPostExecute(Result result)
Diese Methode wird vom UI thread aufgerufen, sobald doInBackground terminiert. Den Wert für „result“ holt sich die Methode aus dem Rückgabewert der doInBackground Methode. Die Methode muss nicht umgesetzt werden. Wenn sie nicht umgesetzt wird, sollte der Datentyp von result auf Void (Achtung- groß geschrieben) gesetzt werden. Dies geschieht auch in der Klassendeklaration.

Die Klassendeklaration sieht wie folgt aus:

public class MyTimerTask extends AsyncTask<Params, Progress, Result> {

Wie ihr seht,, finden sich die entsprechenden Parameter in dem “Generic” Bereich der Klassendeklaration. Um dem ganzen jetzt das mysteriöse zu nehmen, sehen wir uns eine tatsächliche Implementierung für unsere funktionale Anforderung mal an. Grundsätzlich werden wir viele Parallelen zu der Implementierung aus dem vorigen Kapitel finden. Wir beginnen mit unserer eigenen Klasse:

```
public class MyTimerTask extends AsyncTask<Integer, String, Void> {
    private int iCounter = 0;
    private int iStartValue = 10;
    private boolean bIamBusy = false;
    private TextView myTextView;

    @Override
    protected Void doInBackground(Integer... startValue) {
        bIamBusy = true;
        iStartValue = startValue[0];
        resetMyTimer();

        while (iCounter > 0) {
            iCounter--;
            publishProgress(String.valueOf(iCounter));
            try {
               Thread.sleep(1000);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
        }
        bIamBusy = false;
        return null;
    }

    @Override
    protected void onProgressUpdate(String... progress) {
        myTextView.setText(progress[0]);
    }

    public void setTextView(TextView myTextViewToSet) {
        myTextView = myTextViewToSet;
    }

    public boolean isBusy() {
        return bIamBusy;
    }

    public void resetMyTimer() {
        iCounter = iStartValue + 1;
    }
}
```

Die Klassenvariablen sind gleichbedeutend mit denen aus unserer Klasse MyTimer des vorausgegangenen Kapitels. Auch die dort genutzte Methode run hat hier ein Pendant – die Methode doInBackground. Aber beginnen wir mit der Klassendeklaration:

```
public class MyTimerTask extends AsyncTask<Integer, String, Void> {
```

Wir haben die drei Datentypen für Parameter, Progress und Result definiert. Der Parameter wird der Timer-wert sein – also von welchem Wert werden wir herunterzählen. Der Progresswert wird für das Update des

TextViews verwendet. Da wir keine Aktion bei Ablauf des Timers vorgesehen haben, brauchen wir auch keinen Datentyp für das Result – also hier Void.

Die doInBackground Methode ist sozusagen unser Arbeitstier – hier liegt der zeitaufwändige Prozess:

```
@Override
protected Void doInBackground(Integer... startValue) {
     bIamBusy = true;
     iStartValue = startValue[0];
     resetMyTimer();

     while (iCounter > 0) {
          iCounter--;
          publishProgress(String.valueOf(iCounter));
          try {
               Thread.sleep(1000);
          } catch (InterruptedException e) {
               e.printStackTrace();
          }
     }
     bIamBusy = false;
     return null;
}
```

Da die Methode wie die run Methoder unserer Klasse MyTimer funktioniert, werde ich nur auf die Unterschiede eingehen. Zuerst haben wir den Parameter „startValue“. Wir haben in (bei der Klassendefinition) als Integer deklariert und erwarten hier den Timerwert. Da die Parameter immer Arrays sind, nehmen wir hier den Wert aus der Arrayposition 0, um den Startwert zu übernehmen:

```
     iStartValue = startValue[0];
```

Nun kommt der Update unseres TextViews. Hierfür gibt es die Methode publishProgress. Wir haben für unseren Progress einen String Parameter vorgesehen und übergeben dementsprechend den Zählerwert als Stringvariable. Dieser wird weiter Unten in der Methode onProgressUpdate übernommen und weiterverarbeitet.
Zum Schluss wird nun der Rückgabewert festgelegt. Da wir Void vorgesehen haben, weil wir keine Aktion nach dem Beenden des Timers machen möchten, geben wir null zurück.

Die onProgressUpdate Methode sieht recht einfach aus. Hier setzen wir lediglich den übergebenen Progresswert in unser TextView Element rein, um es dem User anzuzeigen:

```
     @Override
     protected void onProgressUpdate(String... progress) {
          myTextView.setText(progress[0]);
     }
```

Nachdem wir keinen Konstruktor vorgesehen haben, brauchen wir noch eine Methode, um den TextView zu übergeben, da wir ihn ja für die Progressanzeige benötigen:

```
     public void setTextView(TextView myTextViewToSet) {
          myTextView = myTextViewToSet;
     }
```

Nun müssen wir das Ganze nur noch in unsere MainActivity einbauen. Der Code hat sich nur marginal im Vergleich zu unserem eigenen Thread Ansatz aus dem vorigen Kapitel geändert:

```
public class MainActivity extends Activity {
    private MyTimerTask myTimerTask;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myTimerTask = new MyTimerTask();
        myTimerTask.setTextView((TextView)
          findViewById(R.id.text_counter));
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }

    public void bttnResetClicked(View myView) {
        myTimerTask.resetMyTimer();
    }

    public void bttnStartClicked(View myView) {
        if (!myTimerTask.isBusy()) {
            myTimerTask.execute(10);
        }
    }
}
```

Die einzigen wirklichen Unterschiede sind beim Instanziieren unserer Klasse:

```
myTimerTask = new MyTimerTask();
myTimerTask.setTextView((TextView)
  findViewById(R.id.text_counter));
```

Wir erzeugen uns ein Objekt und übergeben gleich im Anschluss das TextView Element, damit wir es in unserem Objekt vorhalten können. Der zweite Unterschied ist beim Start unserer Prozedur. In der Methode bttnStartClicked starten wir den Timer – sofern er nicht „busy“ ist:

```
myTimerTask.execute(10);
```

Somit läuft der Timer und zeigt das gleiche Verhalten wie im vorausgegangenen Kapitel.

Wie ihr seht, ist es gar nicht so schlimm die Klasse AsyncTask zu nutzen. Grundsätzlich wird jedoch empfohlen, diese Klasse nur bei eher kurzen Aktionen zu nutzen – also im zweistelligen Sekundenbereich. Bei länger andaurenden Aktionen sollten wir einen eigenen Thread starten, wie im vorausgegangenen Kapitel beschrieben.

5 Lizenz

6 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.