
15 Iteration 7 – Location Based Services

Location Based Services gewinnen zunehmend an Bedeutung. Immer mehr Mobiltelefone sind mit einem GPS-Empfänger ausgestattet, und es ist reizvoll, über Anwendungen nachzudenken, die standortspezifische Informationen verwenden.

Wir wollen uns in diesem Kapitel auf zwei Kernbereiche beschränken und diese in unseren Staumelder integrieren. Zum einem ist dies die Ermittlung von Positionsdaten und zum anderen die Anzeige des aktuellen Standorts in einer Straßenkarte (*Google Maps*). Die verwendeten Beispiele lassen sich leicht auf eigene Anwendungen übertragen und entsprechend anpassen.

15.1 Iterationsziel

Für unseren Staumelder heißt das, dass wir bei einer Staumeldung automatisch die aktuelle Position ermitteln und mit der Staumeldung an den Server senden. Wir brauchen also eine Komponente, die uns die Positionsdaten ermittelt. Außerdem wollen wir unsere aktuelle Position in einer Landkarte anzeigen. Das könnte uns helfen, eine Alternativroute zu finden, falls wir kein Navigationsgerät dabei haben oder gar keins besitzen.

Bevor wir jedoch mit der Programmierung starten, sind einige Vorbereitungen nötig, die wir unserer Struktur entsprechend in den theoretischen Teil gepackt haben. Wir wollen die Möglichkeiten des Emulators ausschöpfen, um effizient entwickeln zu können. Außerdem sollten wir ein wenig über GPS und Google Maps wissen.

Unsere Ziele für dieses Kapitel lauten wie folgt:

- verstehen, wie man mit Positionsdaten umgeht
- die Möglichkeiten des Emulators ausschöpfen können
- lernen, wie man die aktuelle Position ermittelt
- lernen, wie man Google Maps verwendet

15.2 Theoretische Grundlagen

15.2.1 GPS, KML und GPX

Längen- und
Breitengrade

Da Android *GPS* (Global Positioning System) unterstützt, werden viele Hersteller ein GPS-Modul in ihre mobilen Computer einbauen. Ist dies der Fall, haben wir einen Lieferanten für Positionsdaten. Die wichtigsten Parameter sind dabei der Längen- und der Breitengrad. Längengrade werden mit »*longitude*« bezeichnet, Breitengrade mit »*latitude*« und können in östlichen Längengraden bzw. in nördlichen Breitengraden (jeweils mit Bogenminuten und -sekunden) oder im Dezimalsystem angegeben werden. Beispielsweise stellt [7°06'54.09" Nord, 50°42'23.57" Ost] denselben Geopunkt dar wie [7.1152637, 50.7066272] im Dezimalsystem.

GPS-Datenformate

Viele GPS-Geräte bieten zudem die Möglichkeit, die Positionsdaten über einen Zeitraum aufzuzeichnen, zu »tracken«. Meist werden diese dann als *GPX*-Datei (GPS Exchange Format) von *Google Earth* exportiert. Eine solche Datei kann viele Wegpunkte einer zurückgelegten Strecke enthalten.

Mit *Google Earth*
KML-Dateien erzeugen

Ein weiteres Format ist *KML* (Keyhole Markup Language) von *Google Earth*. Das Programm *Google Earth* bietet die Möglichkeit, *KML*-Dateien zu erzeugen. Man fügt in dem Programm einfach über »Ortsmarke hinzufügen« eine neue Ortsmarke hinzu. Diese erscheint im linken Panel unter dem Reiter »Orte«. Klickt man auf diesen Ortspunkt mit der rechten Maustaste und wählt »Speichern unter«, kann man als Zieldatei den Typ »*KML-Datei*« auswählen.

Höheninformation

Der *Location Manager* von Android, den wir gleich vorstellen, verwendet nur die Dezimalschreibweise. Neben den Parametern »*longitude*« und »*latitude*« ist »*altitude*« noch ein wichtiger Geopunkt-Parameter. Er gibt die Höhe in Metern über Normal-Null an.

15.2.2 Entwickeln im Emulator

Wenn man Anwendungen erstellt, die Positionsdaten aus dem *Location Manager* verwenden, dann wird man diese meist erst im Emulator testen wollen. Im Emulator müssen Positionsdaten simuliert werden. Den komfortablen Weg beschreiben wir in Abschnitt 16.2. Allerdings funktioniert dieser Weg mit den Releases bis einschließlich *1.1rc1* aufgrund eines Fehlers mit deutschen Spracheinstellungen im Betriebssystem noch nicht. Daher beschreiben wir hier einen alternativen Weg über die *Telnet*-Konsole, da ein Testen der Anwendung sonst nicht möglich ist.

GPS-Modul per *Telnet*
simulieren

Geodaten mittels Telnet an den Emulator senden

Öffnen Sie eine Betriebssystem-Konsole und schauen Sie in der Titelleihe Ihres Emulators, auf welchem Port er läuft (normalerweise 5554). Geben Sie

```
> telnet localhost 5554
```

ein. Dadurch wird die Android-Konsole gestartet. In dieser geben Sie dann beispielsweise

```
> geo fix <longitude> <latitude>\index{geo fix}
```

an. Denken Sie daran, dass Android nur Dezimalzahlen versteht. Eine Eingabe der Form

*Dezimalzahlen als
Eingabe*

```
> geo fix 7.1152637 50.7066272
```

wäre also zulässig.

Wir brauchen dies erst später zum Testen, wenn wir unsere Anwendung erstellt haben. Einfacher geht es natürlich mit einem echten Gerät, aber das Laden der Karten von Google Maps erfolgt über das Internet und kann auf Dauer nicht unbedeutende Kosten verursachen.

15.2.3 Debug Maps API-Key erstellen

Wir werden später Google Maps nutzen und eine Straßenkarte mit unserer aktuellen Position anzeigen. Dazu bedarf es einiger Vorbereitungen. Um von Google Maps Kartenmaterial abrufen zu dürfen, braucht man einen Maps API-Key. Wir zeigen nun, wie man diesen möglichst einfach erstellt. Grundlage für die Erstellung des Maps API-Key ist ein Zertifikat, mit dem man seine fertige Anwendung signiert. Das Signieren von Anwendungen ist Thema des Kapitels 17. Hier müssen wir jedoch schon wissen, dass das Eclipse-Plug-in zum Android-SDK ein Zertifikat für Entwicklungszwecke mitbringt, hier Debug-Zertifikat genannt. Anwendungen, die man mit dem Eclipse-Plug-in erstellt, werden automatisch mit dem Debug-Zertifikat signiert und laufen im Emulator oder in einem per USB-Kabel angeschlossenen Android-Gerät.

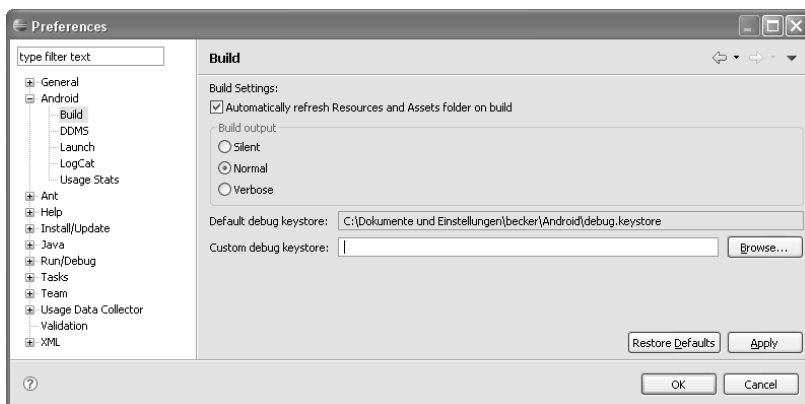
*Mitgeliefertes
Entwicklerzertifikat
verwenden*

Zum Erstellen des Maps API-Keys brauchen wir ein Zertifikat. Da wir zum Entwickeln und Testen ein Debug-Zertifikat zur Verfügung haben, verwenden wir dieses, um den Debug Maps API-Key zu erzeugen.

Debug-Zertifikat finden

Das Debug-Zertifikat befindet sich in einem sogenannten *Keystore*. Wir müssen zunächst den Speicherort für den Keystore ermitteln. Sein Name ist »debug.keystore«. In Eclipse findet man die Angabe des Verzeichnisses unter »*Window->Preferences->Android->Build*« (siehe Abb. 15-1).

Abb. 15-1
Speicherort vom
Debug-Keystore
ermitteln



Um den Maps API-Key zu erzeugen, verwenden wir ein Tool, welches bei der Java-Runtime bzw. bei jedem JDK dabei ist und im bin-Verzeichnis von Java liegt. Das Tool heißt »keytool.exe«. Man kann im System die PATH-Variablen erweitern und auf das bin-Verzeichnis zeigen lassen, damit man keytool.exe überall aufrufen kann. Am besten, man kopiert sich die Datei debug.keystore in ein Verzeichnis, welches keine Leerzeichen enthält, da sonst keystore.exe die Parameterliste nicht auflösen kann. Wir haben die Datei nach D:\Projekte\androidbuch kopiert und generieren nun mit keytool.exe einen MD5-Fingerabdruck, den wir für die Erzeugung des Maps API-Key brauchen.

MD5-Fingerabdruck

Ein MD5-Fingerabdruck ist eine Zahl (meist in hexadezimaler Schreibweise angegeben), die eine Checksumme von Daten oder Dateien darstellt. Verändern sich die Daten oder der Inhalt einer Datei, ändert sich auch der Fingerabdruck. Besitzt man den Fingerabdruck einer Datei aus sicherer Quelle und die Datei selbst aus unsicherer Quelle, kann man den Fingerabdruck der Datei erzeugen. Stimmen nun beide Fingerabdrücke überein, besitzt man mit größter Wahrscheinlichkeit die Originaldatei.

```
> keytool -list -alias androiddebugkey -keystore
D:\Projekte\androidbuch\debug.keystore -storepass
android -keypass android
```

Alle anderen Parameter außer dem Pfad zur debug.keystore-Datei bleiben unverändert. Das Ergebnis ist eine Ausgabe in der Konsole der Art:

*Ein
MD5-Fingerabdruck*

```
Zertifikatsfingerabdruck (MD5):
D1:E1:A3:84:B2:70:59:25:66:F0:E5:49:7D:B2:F1:36
```

Um nun an den Maps API-Key zu kommen, geht man auf die Google-Internetseite <http://code.google.com/android/maps-api-signup.html> und gibt den MD5-Fingerabdruck in das dafür vorgesehene Eingabefeld ein. Nach einem Klick auf die Schaltfläche »Generate API Key« erhält man auf der Folgeseite den gewünschten Key. Diesen brauchen wir später, wenn wir ein Layout für eine Activity entwerfen, welche die Straßenkarte anzeigen soll. Daher merken wir uns den Maps API-Key.

15.3 Praxisteil

Im folgenden Praxisteil erweitern wir den Staumelder um einen Service, der uns auf Anfrage die aktuelle Ortsposition zur Verfügung stellt. In einem zweiten Schritt implementieren wir die Funktion hinter dem Hauptmenüpunkt »Straßenkarte anzeigen« des Staumelders.

15.3.1 Vorbereitung

Um GPS in einer Android-Anwendung nutzen zu können, muss man im Android-Manifest ein paar kleine Erweiterungen vornehmen. Zum einen müssen wir folgende Berechtigung setzen:

```
<uses-permission android:name="android.permission.
ACCESS_FINE_LOCATION"/>
```

Dies ermöglicht den Zugriff auf das GPS-Modul des Android-Geräts. Zusätzlich muss auch die Android-Berechtigung android.permission.INTERNET gesetzt sein, wenn wir Google-Maps verwenden, da das Kartenmaterial über das Internet geladen wird. Google-Maps erfordert auch das Einbinden einer zusätzlichen Bibliothek, die Android zwar mitbringt, aber nicht automatisch in eine Anwendung einbindet. Die Bibliothek binden wir innerhalb des <application>-Tags ein:

*Berechtigungen und
Bibliotheken*

```
<application android:icon="@drawable/icon"
            android:label="@string/app_name"
            android:debuggable="true">

    <uses-library
        android:name="com.google.android.maps" />
    ...
```

15.3.2 Der Location Manager

*Provider liefern
Positionsdaten.*

Der Location Manager (`android.location.LocationManager`) ist eine Schnittstelle zu dem GPS-Modul des mobilen Computers. Es kann durchaus mehrere GPS-Module in einem Gerät geben. In den meisten Android-Geräten wird in Zukunft sicherlich ein GPS-Empfänger eingebaut sein. Zusätzlich könnte man über Bluetooth einen zweiten Empfänger anbinden. Darüber hinaus gibt es auch die Möglichkeit, Informationen über die aktuelle Position über das Netzwerk zu beziehen. Der Location Manager verwaltet die Lieferanten von Positionsdaten unter der Bezeichnung »*Provider*«. Ein Provider liefert uns also die Ortsposition.

Im Grunde hat der Location Manager drei Funktionen:

- Er liefert die letzte bekannte Ortsposition (Geopunkt).
- Auf Wunsch wirft er einen selbstdefinierten `PendingIntent`, wenn wir den Radius um einen bestimmten Ortspunkt unterschreiten, uns ihm also nähern.
- Man kann bei ihm einen Listener registrieren, der mehr oder weniger periodisch die Ortsposition liefert.

*Listener registriert
Ortsveränderungen.*

Wir schauen uns den letzten Punkt etwas genauer an, da wir diese Funktion für die Anzeige unserer Straßenkarte gut brauchen können. Der Location Manager besitzt unter anderem folgende Methode:

```
public void requestLocationUpdates(String provider,
    long minTime, float minDistance,
    LocationListener listener)
```

Diese Methode veranlasst den Location Manager, in periodischen Abständen die Ortsposition zu ermitteln. Das Ermitteln der Ortsposition verbraucht sehr viel Systemressourcen, sowohl was die Prozessorlast als auch was den Stromverbrauch angeht, denn das GPS-Modul verbraucht sehr viel Energie.

Die Methode hilft, Strom zu sparen, und man sollte die Parameter dem Einsatzzweck entsprechend setzen, wie wir gleich sehen werden.

Parameter	Beschreibung
provider	Name des GPS-Providers
minTime	0: Der Location Manager liefert so oft wie möglich die aktuelle Position. Stromverbrauch ist maximal. >0: Wert in Millisekunden, der festlegt, wie lange der Location Manager mindestens warten soll, bevor er wieder nach der aktuellen Position fragt. Je größer der Wert, desto geringer der Stromverbrauch.
minDistance	0: Der Location Manager liefert so oft wie möglich die aktuelle Position. Stromverbrauch ist maximal. >0: Distanz in Metern, um die sich die Position mindestens verändert haben muss, damit der Location Manager eine neue Position liefert. Je größer der Wert, desto geringer der Stromverbrauch.
listener	Ein selbst programmierter Location Listener überschreibt die Methode <code>onLocationChanged(Location location)</code> . Das Objekt <code>location</code> enthält die aktuelle Position.

Tab. 15-1

Parameter der Methode `requestLocationUpdates`

Die Tabelle 15-1 erklärt die Parameter der Methode. Will man die aktuelle Position so oft wie möglich erhalten, so setzt man die Parameter `minTime` und `minDistance` beide auf Null.

Tipp

Schalten Sie das GPS-Modul des mobilen Computers nur ein, wenn Sie es wirklich brauchen. Der Akku hält dann wesentlich länger! Beenden Sie Ihre Activity mit der Methode `finish` in der `onPause`-Methode. Dadurch greift Ihre Anwendung nicht auf das GPS-Modul zu und Sie sparen Strom.

Wir werden nun den `GpsPositionsServiceLocal` aus Abschnitt 8.3.1 auf Seite 110 erweitern. Ziel ist es, innerhalb einer Activity über einen Callback-Handler (siehe Abschnitt 8.3.2) immer mit aktuellen GPS-Daten aus einem eigenen GPS-Service versorgt zu werden. Wir können dann diesen Service später für andere Anwendungen weiterverwenden und eine Activity schreiben, die den Callback-Handler definiert und die GPS-Daten automatisch vom `GpsPositionsServiceLocal` erhält. Die Activity kann dann die Daten beispielsweise verwenden, um die aktuelle Position in einer Straßenkarte darzustellen.

Eigener GPS-Service

Wir erweitern zunächst den schon bekannten `GpsPositionsServiceLocal`. Dazu fügen wir folgende Zeile ein:

```
Handler uiServiceCallbackHandler;
```

*Listener registriert
Ortsveränderung.*

Nun fügen wir dem Service eine innere Klasse hinzu. Die Klasse implementiert einen `LocationListener` und muss vier Methoden überschreiben. Davon interessiert uns eigentlich nur eine Methode, die Methode `onLocationChanged(Location location)`. Wie der Name vermuten lässt, wird die Methode aufgerufen, wenn sich die GPS-Position geändert hat. Dem Location Manager übergeben wir gleich den hier implementierten Listener mittels der `requestLocationUpdates`-Methode, die weiter oben beschrieben wurde.

Listing 15.1
*Eigenen Listener für
Ortsveränderungen
implementieren*

```
private class MyLocationListener
    implements LocationListener {

    @Override
    public void onLocationChanged(Location location) {
        if (uiServiceCallbackHandler != null) {
            Message message = new Message();
            message.obj = location;
            Bundle bundle = new Bundle();
            bundle.putParcelable("location", location);
            message.setData(bundle);
            uiServiceCallbackHandler.sendMessage(message);
        }
    }

    @Override
    public void onProviderDisabled(String provider) { }

    @Override
    public void onProviderEnabled(String provider) { }

    @Override
    public void onStatusChanged(String provider,
        int status, Bundle extras) { }
};
```

Nun implementieren wir noch die `onCreate`-Methode des `GpsPositionsServiceLocal` neu:

Listing 15.2
*Den Location Manager
initialisieren*

```
@Override
public void onCreate() {
    locationManager = (LocationManager) getSystemService(
        Context.LOCATION_SERVICE);
```

```

locationListener = new MyLocationListener();
locationManager.requestLocationUpdates (locationManager.
    GPS_PROVIDER, 5000, 10, locationListener);
}

```

Wir holen uns darin vom System einen Verweis auf den Location Manager. Dies erfolgt wie üblich über die Methode `getSystemService(String name)`. Nun erzeugen wir ein Exemplar unseres `MyLocationListeners` und rufen auf dem Location Manager die `requestLocationUpdates`-Methode auf. Mit den verwendeten Parametern wird der `locationListener` minimal alle fünf Sekunden (meist seltener) über die Ortsposition informiert. Er wird ebenfalls aufgerufen, wenn sich die Position um mehr als 10 Meter seit dem letzten Aufruf der Methode `onLocationChanged` verändert hat.

Um nun die GPS-Daten an die Activity zu übertragen, die sich um die Darstellung unserer aktuellen Position kümmert, implementieren wir die `onLocationChanged`-Methode. Wenn unser Callback-Handler nicht null ist, erzeugen wir eine Message und ein Bundle. Wir verzichten hier auf unsere eigene Klasse `GpsData` (siehe Listing 8.8 in Abschnitt 8.3.1), sondern können den Übergabeparameter `location` direkt verwenden, da er das Interface `Parcelable` schon implementiert. Folglich fügen wir dem Bundle die `location` hinzu, vervollständigen die Message und stellen die Message in die Message Queue des aufrufenden Threads, also unserer Activity, die wir gleich implementieren.

Als Nächstes erweitern wir den `IBinder`. Er ist unsere Schnittstelle nach außen und soll auch die Möglichkeit bieten, aktuelle GPS-Daten per Methodenaufruf und nicht über den Callback-Handler zu liefern.

```

public class GpsLocalBinder extends Binder {

    public GpsPositionsServiceLocal getService() {
        return GpsPositionsServiceLocal.this;
    }

    public void setCallbackHandler(Handler
        callbackHandler) {
        uiServiceCallbackHandler = callbackHandler;
    }

    public GpsData getGpsData() {
        if (locationManager != null) {
            Location location = locationManager.
                getLastKnownLocation (LocationManager.
                    GPS_PROVIDER);
            GpsData gpsData = new GpsData(location.getTime(),

```

*Location-Objekt für
Ortspunkte*

Listing 15.3
*Den Location Manager
initialisieren*

```

        (float)location.getLongitude(), (float)location.
        getLatitude(), (float)location.getAltitude());
    return gpsData;
}
return null;
}
}

```

*Letzte bekannte
Position ermitteln*

Zum einen können wir hier unserem Service den Callback-Handler übergeben, den wir oben schon gebraucht haben, zum anderen können wir hier jederzeit aktuelle GPS-Daten abrufen. Die Methode `getLastKnownLocation` liefert uns ebenfalls ein Objekt vom Typ `Location`. Aus Kompatibilitätsgründen und um die wichtigsten Methoden des `Location`-Objekts vorzustellen, erzeugen wir das schon bekannte `GpsData`-Objekt und geben es an den Aufrufer zurück.

Abschließend sei der Vollständigkeit halber noch erwähnt, dass wir die `GpsPositionsServiceLocal`-Klasse noch um folgende Deklaration erweitern müssen:

```

private LocationManager locationManager;
private LocationListener locationListener;

```

15.3.3 Google Maps

In einigen Anwendungen wird man seine aktuelle Position auf einer Landkarte darstellen wollen. Dies können wir mit Hilfe von Google Maps machen. Nachdem wir uns am Anfang des Kapitels den Maps API-Key zum Testen unserer Anwendung besorgt haben, können wir die Anzeige einer Straßenkarte mit unserer aktuellen Position in Angriff nehmen.

*MapView und
MapActivity*

Auf einfache Weise ist dies mit einer `MapView` möglich. Dies ist eine spezielle View, die wir in ein Layout integrieren können. Zur Darstellung eines Layouts mit `MapView` sollte man unbedingt eine `MapActivity` verwenden. Die `MapActivity` hat einen speziellen Lebenszyklus, da im Hintergrund das Kartenmaterial über das Internet geladen und geladenes Kartenmaterial im Dateisystem abgelegt wird. Diese langwierigen Aufgaben erledigen Threads, die im Lebenszyklus der `MapActivity` verwaltet werden.

Schauen wir uns zunächst die Definition eines einfachen Layouts zur Anzeige einer Straßenkarte mit Google Maps an.

Listing 15.4
*Layout zur Anzeige
einer MapView*

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/
    apk/res/android"
    android:orientation="vertical"

```

```

android:layout_width="fill_parent"
android:layout_height="fill_parent">

<com.google.android.maps.MapView
    android:id="@+id/mapview_strassenkarte"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:clickable="true"
    android:apiKey="
        0xYgdiZYM8ZDqh1JWsm7qdgRzBrMEkTJCdqpD6w" />
</LinearLayout>

```

Zwei Attribute sind dabei neu: `android:clickable` und `android:apiKey`. Das erste dient unter anderem dazu, dass wir später die Straßenkarte verschieben und ein Oberflächenelement anzeigen können, welches den Zweck hat, die Karte zu vergrößern oder zu verkleinern. Das zweite Attribut ist der Maps API-Key, den wir am Anfang des Kapitels auf den Google-Maps-Internetseiten haben erzeugen lassen. Fügen Sie hier Ihren eigene Maps API-Key ein, den Sie wie in Abschnitt 15.2.3 beschrieben selbst erzeugt haben.

*Eigenen Maps API-Key
einfügen*

Was wir damit haben, ist ein View-Element, welches wir in unserer Activity ansprechen können.

15.3.4 MapActivity

Wie wir weiter oben gesehen haben, sollten wir für die Darstellung der Straßenkarte eine `MapActivity` einsetzen. Um die `MapActivity` zu implementieren, müssen wir folgende Schritte ausführen:

- Wir müssen die `MapView` in der `MapActivity` anzeigen.
- Wir müssen den Callback-Handler implementieren, so dass er die Straßenkarte mit den aktuellen GPS-Daten versorgt und die Darstellung aktualisiert.
- Wir müssen der Klasse `GpsPositionsServiceLocal` den Callback-Handler übergeben, um in der Activity automatisch aktuelle GPS-Positionsdaten zu erhalten.
- Wir müssen unseren aktuellen Standort in die Karte malen.

Starten wir mit der `MapActivity` und dem Zugriff auf die `MapView`.

```

public class StraßenkarteAnzeigen extends MapActivity {

    private MapView mapView;
    private MapController mapController;
    private MapViewOverlay mapViewOverlay;
    private Paint paint = new Paint();

```

Listing 15.5
*MapActivity zur
Anzeige einer
Straßenkarte*

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.strassenkarte_anzeigen);
    setTitle(R.string.text_strassenkarte_titel);

    Intent intent = new Intent(this,
        GpsPositionsServiceLocal.class);
    bindService(intent, localServiceVerbindung,
        Context.BIND_AUTO_CREATE);

    mapView = (MapView) findViewById(
        R.id.mapview_strassenkarte); // (1)
    mapController = mapView.getController();

    int zoomlevel = mapView.getMaxZoomLevel();
    mapController.setZoom(zoomlevel - 2);

    mapViewOverlay = new MapViewOverlay(); // (2)
    mapView.getOverlays().add(mapViewOverlay);
    mapView.postInvalidate();

    LinearLayout zoomView = (LinearLayout) mapView.
        getZoomControls();
    zoomView.setLayoutParams(new ViewGroup.LayoutParams(
        ViewGroup.LayoutParams.WRAP_CONTENT,
        ViewGroup.LayoutParams.WRAP_CONTENT));
    zoomView.setGravity(Gravity.BOTTOM |
        Gravity.CENTER_HORIZONTAL);
    mapView.addView(zoomView);

    mapView.setStreetView(true);
}
}

```

Wir erweitern nun die Activity um die fehlenden Methoden. Um eine Verbindung zu unserem Service aufzubauen, müssen wir noch das Attribut `localServiceVerbindung` angeben. Sobald die Verbindung zum Service steht, setzen wir unseren Callback-Handler.

Listing 15.6
*Verbindung zum
 lokalen GPS-Service
 aufbauen*

```

private ServiceConnection localServiceVerbindung =
    new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName className,
        IBinder binder) {
        ((GpsPositionsServiceLocal.GpsLocalBinder)binder).
            setCallbackHandler(uiThreadCallbackHandler);
    }
}

```

```

@Override
public void onServiceDisconnected(
    ComponentName className) { }
};

```

Wenn wir zur onCreate-Methode zurückkehren, sehen wir als Nächstes, wie wir uns die MapView, die wir im Layout deklariert haben, holen (1). Eigenschaften der MapView können wir über einen Controller steuern. Nachdem wir die MapView und den MapController haben, können wir die Straßenkarte für die Ansicht präparieren. Wir holen uns mittels der Methode getMaxZoomLevel die höchste Auflösung, die uns für so eine View zur Verfügung steht. Es handelt sich hier um die maximal verfügbare Auflösung in der Mitte der Karte. Da Google Maps nicht für alle Orte auf der Welt die gleiche Auflösung zur Verfügung stellt, bietet es sich an, hier evtl. einen festen Wert zu nehmen. Wenn dieser höher als der höchste verfügbare Zoomlevel liegt, fällt die Anwendung auf diesen Zoomlevel zurück.

Zoomlevel

Als Nächstes instanziiieren wir ein Objekt vom Typ MapViewOverlay (2). Die Deklaration des Objekts fügen wir als innere Klasse unserer Activity hinzu.

```

public class MapViewOverlay extends Overlay {
    @Override
    public void draw(Canvas canvas, MapView mapView,
        boolean shadow) {
        super.draw(canvas, mapView, shadow);

        // Wandel Geopunkt in Pixel um:
        GeoPoint gp = mapView.getMapCenter();
        Point point = new Point();
        // in Point werden die relativen Pixelkoordinaten
        // gesetzt:
        mapView.getProjection().toPixels(gp, point);

        // Zeichenbereich definieren:
        RectF rect = new RectF(point.x - 5, point.y + 5,
            point.x + 5, point.y - 5);

        // roter Punkt fuer eigenen Standort
        paint.setARGB(255, 200, 0, 30);
        paint.setStyle(Style.FILL);
        canvas.drawOval(rect, paint);
    }
}

```

Listing 15.7

Ein Overlay zur Anzeige des eigenen Standorts in der Karte

```

        // schwarzen Kreis um den Punkt:
        paint.setARGB(255,0,0,0);
        paint.setStyle(Style.STROKE);
        canvas.drawCircle(point.x, point.y, 5, paint);
    }
}

```

Was ist ein Overlay?

Ein *Overlay* können wir uns als durchsichtige Folie vorstellen, die wir über unsere *MapView* legen. Mittels der *draw*-Methode können wir wie auf einem Overhead-Projektor zeichnen und malen. Wir können einer *MapView* potenziell beliebig viele *Overlays* hinzufügen und wieder entfernen, ohne dass sich die *MapView* verändert. Daher holen wir uns in der *onCreate*-Methode der *MapActivity* erst alle *Overlays* und fügen dann unseres hinzu.

Die eigene Position anzeigen

Das *Overlay* selbst gehört zur *MapView* und bekommt diese als Parameter in der *draw*-Methode übergeben. Wir holen uns mittels *mapView.getMapCenter* das Zentrum als Geo-Punkt (Objekt *GeoPoint*) und nutzen die *MapView* für die Umrechnung in Pixel (Methode *getProjection(GeoPoint gp, Point point)*). Der Rest der Methode dient dem Zeichnen eines roten Punkts mit schwarzer Umrandung in der Mitte der Karte, also dort, wo sich laut GPS-Modul unser aktueller Standort befindet.

Was uns nun noch fehlt, ist der *Callback-Handler*. Er liefert uns in seiner *handleMessage*-Methode den Ortspunkt. Da diese Methode automatisch durch den Service aufgerufen wird, entweder wenn wir uns bewegen oder wenn eine bestimmte Zeitspanne vergangen ist, brauchen wir nur dafür zu sorgen, dass die Straßenkarte in der *MapView* auf unsere aktuellen GPS-Koordinaten zentriert wird. Den Rest übernimmt dann das *Overlay*, indem es unseren Standort als roten Punkt einzeichnet.

Listing 15.8

Der Callback-Handler liefert uns die GPS-Daten aus unserem GPS-Service.

```

private final Handler uiThreadCallbackHandler =
    new Handler() {
        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            Bundle bundle = msg.getData();
            if (bundle != null) {
                Location location =
                    (Location)bundle.get("location");
                GeoPoint geoPoint = new GeoPoint(
                    (int) (location.getLatitude() * 1E6),
                    (int) (location.getLongitude() * 1E6));
            }
        }
    };

```

```
mapController.animateTo(geoPoint);
mapView.invalidate();
}
}
};
```

Um die MapView zu zentrieren, holen wir uns das Location-Objekt aus dem Bundle, welches uns der Service liefert. Das Zentrieren erledigt der Controller der MapView, und dieser erwartet ein GeoPoint-Objekt. Die Umrechnung ist einfach. Das location-Objekt liefert die Werte longitude und latitude in Grad. GeoPoint erwartet aber Mikrograd. Durch die Multiplikation mit dem Wert 1E6 (1 Million) erhalten wir aus Grad Mikrograd. Abschließend rufen wir noch auf der MapView invalidate auf und erzwingen dadurch ein Neuzeichnen der Straßenkarte. Die Overlays werden nach wie vor angezeigt, so dass uns auch der Standortpunkt nicht verloren geht.



Abb. 15-2
Anzeige der aktuellen
Position mit Google
Maps im Emulator

Abbildung 15-2 zeigt das fertige Ergebnis unserer Bemühungen im Emulator, nachdem wir mit telnet eine Ortskoordinate eingegeben haben.

15.4 Fazit

Wir haben das Kapitel mit ein wenig Theorie über GPS begonnen, damit wir später beim Entwickeln und Testen mit Ortskoordinaten umgehen können. Denn wir können auch im Emulator Location Based Services testen, da wir den Location Manager über die Android-Konsole mit Geodaten füttern können.

Wir haben dann einige Vorbereitungen getroffen, damit wir später Google Maps in unserer Anwendung zur grafischen Anzeige von Karten nutzen können. Wir haben uns einen Maps API-Key aus unserem Debug-Zertifikat des Android-SDK erstellt.

Nach diesen Vorbereitungen haben wir uns dem Location Manager gewidmet. Er liefert uns die Ortskoordinaten unseres aktuellen Standorts. Normalerweise greift er dazu auf das GPS-Modul des Android-Geräts zurück, allerdings sind auch andere Verfahren der Positionsermittlung möglich, z.B. die Berechnung der Position über die Funkmasten, in die das Android-Gerät gerade eingebucht ist.

Wir haben dann mit der `MapActivity` experimentiert. Durch sie war es uns möglich, die von Google Maps geladene Karte darzustellen. Dank Overlays konnten wir dann der Karte noch Informationen in grafischer Form hinzufügen, was wir beispielhaft durch einen roten Punkt in der Kartenmitte, der unsere aktuelle Position anzeigt, getan haben.