About This Article

This article contains information on how to target the latest platform version when developing an Android application, while remaining compatible with older platform versions.

To understand the contents of this article one should know the basics of Android application development.

1. Problem overview

Due to device divergence and the rapid evolution of the Android platform there are substantial pitfalls for developers to consider. New sets of APIs occur with each major Android update and often different sets of APIs can become deprecated in the process.
The responsibility to keep the application fully functional on older Android versions while using the features available in latest devices lies with the developer. Thankfully, there are ways that can help reduce the amount of effort required to remain backward compatible.

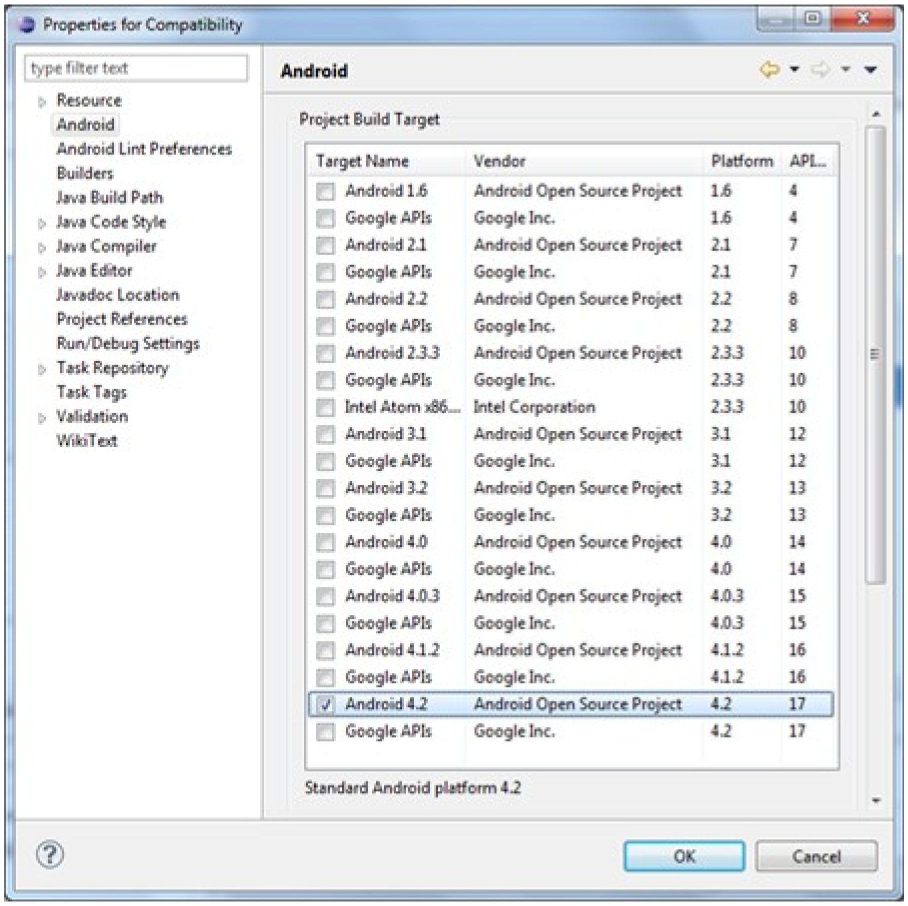
1.1. Declaring minimum and target API versions

In order to allow the application to run on older devices you have to remember to set the minSdkVersion to the minimal Android platform version you want to support. The attribute targetSdkVersion on the other hand hints the platform what kind of behavior you are expecting which can also lead to enabling some compatibility behaviors in some cases.
For example, if we wanted to develop a Live Wallpaper application our users would have to run at least Éclair version of Android (API 7 – 2.1). We also want to use latest APIs available to support the preference headers approach (API 11+) in PreferenceActivity so we set the targetSdkVersion to 17 (the newest version at the time of writing this article).

```
<uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="17" />
```

[Code 1] An example of setting the minimal and target SDK versions (AndroidManifest.xml)

You should also remember to set the Project Build Target in the Project > Properties > Android settings in Eclipse (or by editing the project.properties file manually) that will tell the build process to compile your app against a specified API platform lib.



Project Build Target

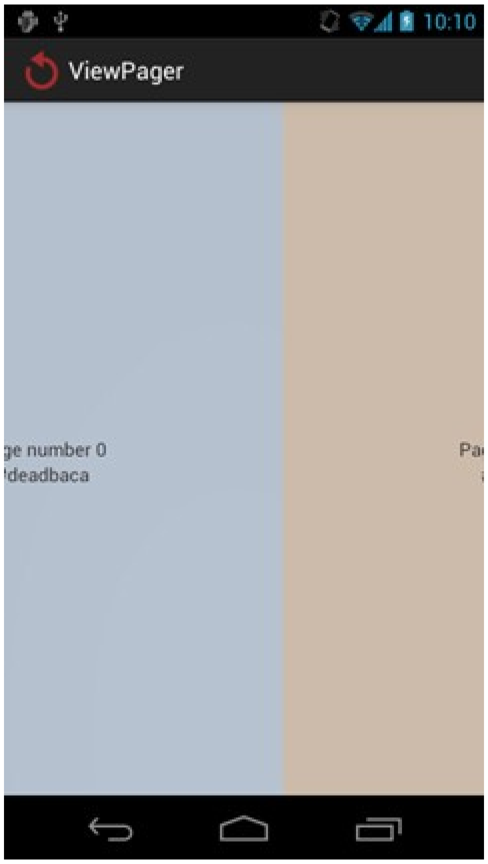
Target Name	Vendor	Platform	API...
Android 1.6	Android Open Source Project	1.6	4
Google APIs	Google Inc.	1.6	4
Android 2.1	Android Open Source Project	2.1	7
Google APIs	Google Inc.	2.1	7
Android 2.2	Android Open Source Project	2.2	8
Google APIs	Google Inc.	2.2	8
Android 2.3.3	Android Open Source Project	2.3.3	10
Google APIs	Google Inc.	2.3.3	10
Intel Atom x86...	Intel Corporation	2.3.3	10
Android 3.1	Android Open Source Project	3.1	12
Google APIs	Google Inc.	3.1	12
Android 3.2	Android Open Source Project	3.2	13
Google APIs	Google Inc.	3.2	13
Android 4.0	Android Open Source Project	4.0	14
Google APIs	Google Inc.	4.0	14
Android 4.0.3	Android Open Source Project	4.0.3	15
Google APIs	Google Inc.	4.0.3	15
Android 4.1.2	Android Open Source Project	4.1.2	16
Google APIs	Google Inc.	4.1.2	16
✓ Android 4.2	Android Open Source Project	4.2	17
Google APIs	Google Inc.	4.2	17

Standard Android platform 4.2

[Image 1] Choosing the Project Build Target version in Project > Properties > Android

2. Support library

Whenever a new set of APIs is introduced in the platform there is always a need to provide the developers with a mechanism of keeping their applications compatible with devices running older platforms while still allowing them to use the latest version’s features.
The Support Library comes in two variants: v4 and v13. The former contains support APIs that will work for any device from API 4 while the latter works for API 13 onwards. Usually, the more appealing solution is the v4 version since it provides a much broader set of support APIs and a much greater amount of devices.
Interestingly, the Support Library also offers a set of APIs unavailable in the regular package set. For example, you will get access to the ViewPager class that allows horizontal scrolling of a set of pages provided by a PagerAdapter.



[Image 2] Using the ViewPager class from the v4 Support Library to offer horizontal page scrolling and a pleasant user experience. Making a fling gesture from right to left causes the ViewPager to scroll to the right page.

Additionally, there also are other classes acting as helpers that are only available in the Support Library. For instance, the NavUtils class helps by implementing recommended Android UI navigation patterns.

2.1. Installation

The Support Library is added to the project by default in the latest version of the ADT plugin for Eclipse so it is very likely you will have access to the APIs available in the support package out of the box. If this is not the case you can add the library in two ways:

- Right-click your project in the Project Explorer and choose Android Tools > Add Support Library…
- Copy the jar file manually from <path_to_android_sdk>/extras/android/support/v4/android-support-v4.jar to your projects's libs directory. If the library doesn't get added to the build path automatically you can go to Project > Properties > Java Build Path > Libraries > Add JARs… and select the .jar file from the list in your project

If you want to use support classes from the v13 jar you don't have to add both libraries explicitly since v13 already includes the v4 variant.

2.2. ViewPager example

In this paragraph you will find basic information about instantiating a ViewPager from the XML layout file and how to provide a proper PagerAdapter instance to it.

As with any other View the ViewPager can be initialized either programmatically in the code or in the XML layout file. The latter solution is the preferred way since it lets you avoid the need to recompile your code by externalizing your UI definition. Please note that the ViewPager class has to be specified explicitly with the package name like any other custom View implementation since it does not exist in the platform API directly.

```
res/layout/activity_viewpager.xml:

...
<android.support.v4.view.ViewPager
    android:id="@+id/view_pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
...

src/com/sprc/compatibility/ViewPagerActivity.java:

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

    ViewPager viewPager = ((ViewPager) findViewById(R.id.view_pager));
    viewPager.setAdapter(new PagerAdapter() {

        @Override
        public boolean isViewFromObject(View v, Object o) {
            return v == o;
        }

        @Override
        public int getCount() {
            return PAGE_COUNT;
        }

        @Override
        public Object instantiateItem(ViewGroup container, int position) {
            View v = getLayoutInflater().inflate(R.layout.page_item, container, false);
            container.addView(v);

            TextView tv = ((TextView) v.findViewById(R.id.text));
            tv.setText(String.format(getString(R.string.page_text), position));

            if (position < mColorArray.length())
                v.setBackgroundColor(mColorArray.getColor(position, 0));

            return v;
        }

        @Override
        public void destroyItem(ViewGroup container, int position, Object object) {
            container.removeView((View) object);
        }
    });
}
...
```

[Code 1] A basic example of using the ViewPager class from the v4 support package

2.3. Fragments example

You can find a minimalistic example showing how to use the Fragment mechanism available in the v4 support package in order to remain backward compatible with older platforms (API < 11) below.

```
res/layout/activity_myfragment.xml:

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

    <!--
    Remember that replacing a fragment with a FragmentTransaction is only
    possible for dynamically added Fragments so we rather need a container!

    <fragment
        android:id="@+id/my_fragment"
        android:name="com.sprc.compatibility.MyFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    -->

    <FrameLayout
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

[Code 2] Defining the layout with a placeholder for our fragments

As you can see, the layout of our Activity is rather simple. Since we want to swap fragments dynamically in the code using FragmentTransaction we cannot put the fragment directly in the XML file. Instead, we have to prepare a placeholder (one of ViewGroup’s subclasses) for fragments. In our case we used the simplest container – the FrameLayout.

```
src/com/sprc/compatibility/MyFragmentActivity.java:

public class MyFragmentActivity extends FragmentActivity implements OnSwapFragmentListener {

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

        MyFragment f = MyFragment.newInstance(MyFragment.DEFAULT_NUMBER);
        f.addOnSwapFragmentListener(this);

        getSupportFragmentManager().beginTransaction()
                .replace(R.id.fragment_container, f).commit();
    }

    @Override
    public void onSwapFragment(MyFragment oldFragment) {
        MyFragment f = MyFragment.newInstance(oldFragment.getNumber() + 1);
        f.addOnSwapFragmentListener(this);

        getSupportFragmentManager().beginTransaction()
                .replace(R.id.fragment_container, f).commit();
    }
}
```

[Code 3] MyFragmentActivity class implementation showing the initialization steps and FragmentTransaction usage

In the code snippet above you should note that MyFragmentActivity is not a subclass of the Activity class as it extends the FragmentActivity class coming from the support library. If you were using native APIs you could just use the plain Activity class since it has all the necessary methods to interact with Fragments. As we want our app to be backwards compatible, we stick to the FragmentActivity.

When MyFragmentActivity is created it puts a new instance of the MyFragment class in the container (FrameLayout) by using a FragmentTransaction obtained with the call to the getSupportFragmentManager method from the FragmentManager. We also want to be notified about a particular action performed on the instance of MyFragment by calling the addOnSwapFragmentListener method and providing it with an OnSwapFragmentListener implementation. MyFragment notifies its listeners when a button is clicked and MyFragmentActivity replaces fragments.

```
src/com/sprc/compatibility/MyFragment.java:

public class MyFragment extends Fragment implements OnClickListener {

    ...

    private int mNumber;
    private Set<OnSwapFragmentListener> mListeners = new HashSet<OnSwapFragmentListener(1);

    public MyFragment() {
    }

    public static MyFragment newInstance(int number) {
        MyFragment f = new MyFragment();
        f.mNumber = number;
        return f;
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {

        View v = inflater.inflate(R.layout.frag_my, container, false);
        Button b = ((Button) v.findViewById(R.id.button));
        b.setText(Integer.toString(mNumber));
        b.setOnClickListener(this);

        return v;
    }

    @Override
    public void onClick(View v) {
        for (OnSwapFragmentListener listener : mListeners) {
            listener.onSwapFragment(this);
        }
    }
    public int getNumber() {
        return mNumber;
    }

    public boolean addOnSwapFragmentListener(OnSwapFragmentListener listener) {
        return mListeners.add(listener);
    }

    public boolean removeOnSwapFragmentListener(OnSwapFragmentListener listener){
        return mListeners.remove(listener);
    }

    public interface OnSwapFragmentListener {
        void onSwapFragment(MyFragment oldFragment);
    }
}
```

[Code 4] MyFragment implementation snippet that shows the initialization steps and the interface for the Activity to implement

Please note that you should provide your Fragment with an empty public constructor because there are situations when the platform is going to instantiate it on its own.
The most important part that you should implement is the onCreateView method. Its main task is to return a view hierarchy of the Fragment.

You should also remember to make your Fragments activity-independent. This is done mainly by designing an interface which can be implemented by the Activity. In our case the OnSwapFragmentListener performs this function. This approach increases the chances to reuse your code more efficiently.

3. Resource version modifiers

The Android resource system provides a way to choose the appropriate resource files based on the platform version the application is running on. This can be done using the platform version resource modifier (i.e. res/values-v11).

This flexible resource mechanism can be used in many different situations. The list of possibilities includes:

- Selecting application themes/styles based on the platform version. This approach is used when you create a new project in Eclipse. When the application is run on a device with the API level at least at 11, the Theme.Holo theme is used by default (res/values-v11/styles.xml). The fallback version of the resource for older Android platform versions is located under the res/values directory and a simple Theme.Light is used in that case.

```
AndroidManifest.xml:
```

```
<application
    ...
    android:theme="@style/AppTheme" >
    ...

res/values/styles.xml:

<resources>
    ...
    <style name="AppTheme" parent="android:Theme.Light" />
    ...

res/values-v11/styles.xml:

<resources>
    ...
    <style name="AppTheme" parent="android:Theme.Holo.Light"/>
    ...
```

[Code 5] Making Android use a different application theme based on the platform version

•Provide different layouts that reference classes suitable for use on a particular platform version. This can apply to static fragments declared explicitly in the XML layout file. On old platform versions the Fragment's class attribute would point to a fallback Fragment implementation while on the new platform to a class that uses the latest API.

•The configuration files for specialized platform Services like WallpaperService or AccessibilityService. This can prove useful if we want to utilize different Preferences or a different set of attributes based on the platform version.

```
AndroidManifest.xml:


...
<application ...>
    <service ...>
        <meta-data
            android:name="android.service.wallpaper"
            android:resource="@xml/wallpaper_config" />
            ...

res/xml/wallpaper_config.xml:

<wallpaper xmlns:android="http://schemas.android.com/apk/res/android"
    android:description="@string/wallpaper_description"
    android:settingsActivity="com.sprc.wallpaper.OldPreferenceActivity"
    android:thumbnail="@drawable/thumbnail" />

res/xml-v11/wallpaper_config.xml:

<wallpaper xmlns:android="http://schemas.android.com/apk/res/android"
    android:description="@string/wallpaper_description"
    android:settingsActivity="com.sprc.wallpaper.NewPreferenceActivity"
    android:thumbnail="@drawable/thumbnail" />
```

[Code 6] Chosing a different configuration file for the live wallpaper service based on the API version can be utilized by putting customized wallpaper_config.xml files in res/xml and res/xml-v11. This way we can reference different preference Activity classes (OldPreferenceActivity and NewPreferenceActivity respectively).

4. Checking if the hardware supports a particular feature

Android devices differ not only in the case of the platform version. They can provide completely different sets of features, i.e. back and front-facing cameras, the barometer, NFC functionality and many others. In order to correctly handle such differences you should think about your app requirements.

You can just assume that you require such features and will not allow devices without them to run your app. It can significantly decrease the number of potential customers though, so is something that should be avoided. Unfortunately, there are cases where your app must rely on a particular feature to work, for example, a QR code scanner app should require at least one camera. Fortunately, if you decide that a particular feature (like NFC) is optional in your application you can easily check the feature availability during run-time.

```
AndroidManifest.xml:
...
<uses-feature
    android:name="android.hardware.microphone"
    android:required="false" />
...

src/com/sprc/compatibility/FeatureActivity.java:


...
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...

    // Show the record button if this system supports microphone

    if (getPackageManager().hasSystemFeature(PackageManager.FEATURE_MICROPHONE))
        buttonRecord.setVisibility(View.VISIBLE);}
```

```
...
```

[Code 7] An example that shows how to use the PackageManager’s hasSystemFeature method to determine whether an application’s particular functionality should be available at runtime.

In the example above we specify that our application uses the microphone but it is not mandatory for the device to support such a feature since we set the required attribute to false. This makes us responsible for handling the situation when the feature is not available by ourselves. We do this by making a record Button visible when the device supports the microphone.

5. Checking the version code number at runtime

The most obvious solution to check which version the application is currently running on and choose the appropriate approach based on the platform is to compare a constant value available in the API to a specific version code.

```
src/com/sprc/compatibility/FeatureActivity.java:


...
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...

    // Show the NFC button if this system is at least 4.1 (JellyBean)

    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.JELLY_BEAN) {
        buttonNfc.setVisibility(View.VISIBLE);
    }
}
...
```

[Code 8] An example that shows how the version code can be used to determine whether an application’s particular functionality should be available at runtime.

The minimalistic example above performs a check similar to the previous one. The NFC settings Activity is guaranteed to be available on devices running Android 4.1 (API level 16) and above. Thus, we check the version code to determine if the Button that runs this Activity should be visible or not.

However, using the two previous approaches extensively with many customizations based on the platform version can lead to a lot of if-else clauses around the code. The following approach allows avoiding such problems.

6. Designing an abstraction layer for a feature

If your application uses a lot of optional features and/or supports older platforms by performing some platform version-dependent logic you can suffer from hard to read and error-prone code. One possible solution you should consider is to design an abstraction layer for your application’s behavior. The example provided below uses different SharedPreferences.Editor’s methods to save the state of a shared prereference based on the underlying API level. The VersionedEditor class is the base class for the version-dependent code.

```
src/com/sprc/compatibility/content/VersionedEditor.java:

public abstract class VersionedEditor {
    public abstract void save(SharedPreferences.Editor editor);
    ...
}
```

[Code 9] The abstract VersionedEditor class that should be subclassed by specific API-dependent versions

Below is the logic that determines which VersionedEditor implementation should be instantiated during runtime. The instance is accessible through the getInstance method.

```
src/com/sprc/compatibility/content/VersionedEditor.java:

public abstract class VersionedEditor {
    ...
    private static class SingletonHolder {
        static {
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD)
                INSTANCE = new GingerbreadEditor();
            else
                INSTANCE = new FallbackEditor();
        }

        private static final VersionedEditor INSTANCE;
    }

    public static VersionedEditor getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

[Code 10] Using a Singleton Design Pattern to instantiate a specific VersionedEditor implementation at runtime

The code provided below contains the actual version-dependent implementation. For the API 9 (Gingerbread and newer) the apply method is used and to support older platforms we fall back to using the commit method instead.

```
src/com/sprc/compatibility/content/FallbackEditor.java:

public class FallbackEditor extends VersionedEditor {
    @Override
    public void save(Editor editor) {
        editor.commit();
    }
}

src/com/sprc/compatibility/content/GingerbreadEditor.java:

public class GingerbreadEditor extends VersionedEditor {
    @Override
    public void save(Editor editor) {
        editor.apply();
    }
}
```

[Code 11] Two version-dependent implementations utilizing the best approach for the available API

The actual usage of this mechanism is shown in the code snippet below.

```
src/com/sprc/compatibility/PrefsActivity.java:

...
SharedPreferences prefs = getPreferences(MODE_PRIVATE);

SharedPreferences.Editor editor = prefs.edit();
editor.putBoolean(SP_KEY_SOME_VALUE, true);

/*
 * Instead of calling editor.apply() or editor.commit() explicitly we
 * use our VersionedEditor class to choose the approach that fits the
 * current platform.
 */
VersionedEditor.getInstance().save(editor);
...
```

[Code 12] Using the VersionedEditor's getInstance method to retrieve

The main benefit of such an approach becomes clearer when you have to provide different customizations for multiple API versions.
You can find it useful to use this mechanism for new UI designs for Android apps. For instance, you could provide a dual-pane layout for a tablet device and a single-pane one for a handset. The logic that would support this approach could be hidden behind an abstraction layer similar to the one provided in the example.

7. External libraries

The Android platform is backed by a great community which provides developers with matured and well-tested libraries decreasing the need to reinvent the wheel in many cases. As you may suspect, others have already stumbled upon backward-compatibility issues and there are couple of projects that can aid you in maximizing the number of your app customers.

- ActionBarSherlock – allows the use of the ActionBar and new Holo styles available in the 11 version on older Androids
- NineOldAndroids – ports animation APIs available in the 11 version to older platforms

Please remember to read the license of these projects before using them. Be a good community player and don’t forget to give credit to the author somewhere in your application!

8. Summary and tips

As mentioned in the first chapter, Android is a constantly evolving platform. Knowing the good principles of keeping your apps backward-compatible and still using the latest features can keep your users base as wide as possible.
When you publish your app to Samsung Apps it will have to pass a certification process. This includes professional testing of your application performed by our employees. Nonetheless, you should remember to test first by yourself to improve the user experience. Fewer errors mean more satisfied customers!

Ref:http://developer.samsung.com/android/technical-docs/Providing-compatibility-with-older-Android-versions