Разработка под Android в NetBeans IDE без плагинов. Часть 1 tutorial

Разработка под Android*

Обычно у разработчика есть свой любимый инструмент, которым ему пользоваться удобнее, чем другими. Однако бывает так, что платформа заставляет разработчиков брать в руки инструмент, который не так удобен, как ему хотелось бы, или просто чем-то не устраивает. Так получилось, что традиционно приложения под Android пишут при помощи Eclipse, поскольку Google приняли решение о том, что будут разрабатывать официальный плагин, ADT, именно для этого редактора. В результате тем разработчикам, которые им не пользовались, волей-неволей пришлось его освоить.

К счастью, Google также предоставляют систему сборки, которая работает независимо от имеющейся в наличии IDE. А это означает, что можно настроить любой редактор для работы с приложениями Android. Лично я предпочитаю писать код на Java в NetBeans IDE и хочу поведать о том, как его можно настроить для этого. Есть такие плагины, как nbandroid, но разрабатывается он нерегулярно, энтузиастами, так что есть смысл воспользоваться гибкостью NetBeans и задействовать официальную систему сборки напрямую из редактора.

Создание нового проекта

При создании нового проекта, к сожалению, придётся сделать больше действий, чем можно было бы сделать в Eclipse, но это нужно сделать всего лишь один раз. Создание проекта делается в три шага:

1. Создание файлов для сборки из командной строки;
2. Создание проекта в IDE;
3. Добавление дополнительных команд (по вкусу).

Создание файлов для сборки

Прежде всего, необходимо создать новый проект через командную строку. Я буду исходить из предположения, что в `PATH` уже затесалась папка `<Android-SDK>/tools`. Проект создаётся следующей командой:

```
android create project -n <имя проекта> -t android-<уровень API> -p <путь к проекту> -k <пакет программы> -a <название основной активности>
```

На всякий случай поясню, что уровень API — это тот самый, с помощью которого мы будем компилировать проект. То есть если проект в целом рассчитан на уровень API 10, но есть некоторые возможности, которые используются только на аппаратах с уровнем 15 и выше, то нужно именно 15 и выставить. В `AndroidManifest.xml`, кстати, эта 15 не засветится, там будет только 10 как минимально необходимый уровень API.

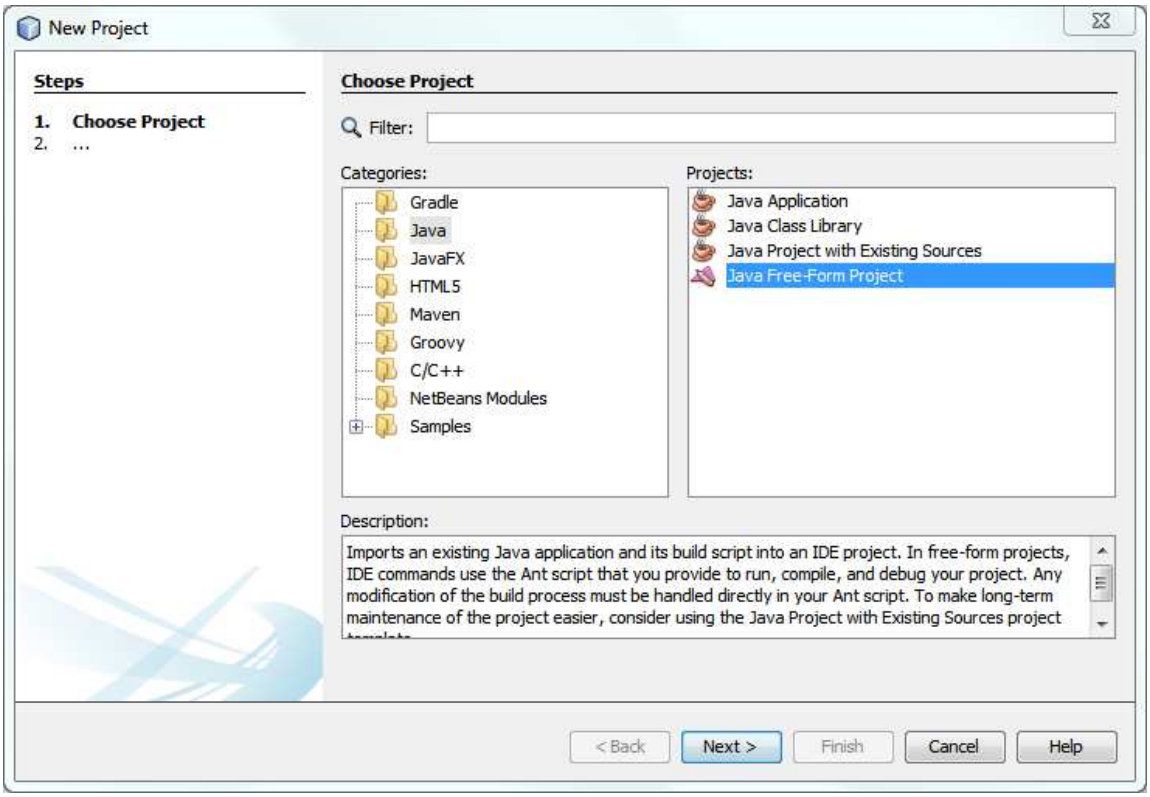
Предположим, что наш проект создаётся для Android 4.0.3 (это уровень 15) и называется KillerApp. Тогда нужно будет ввести следующее:

```
android create project -n KillerApp -t android-15 -p KillerApp -k com.damageinc.killerapp -a MainActivity
```

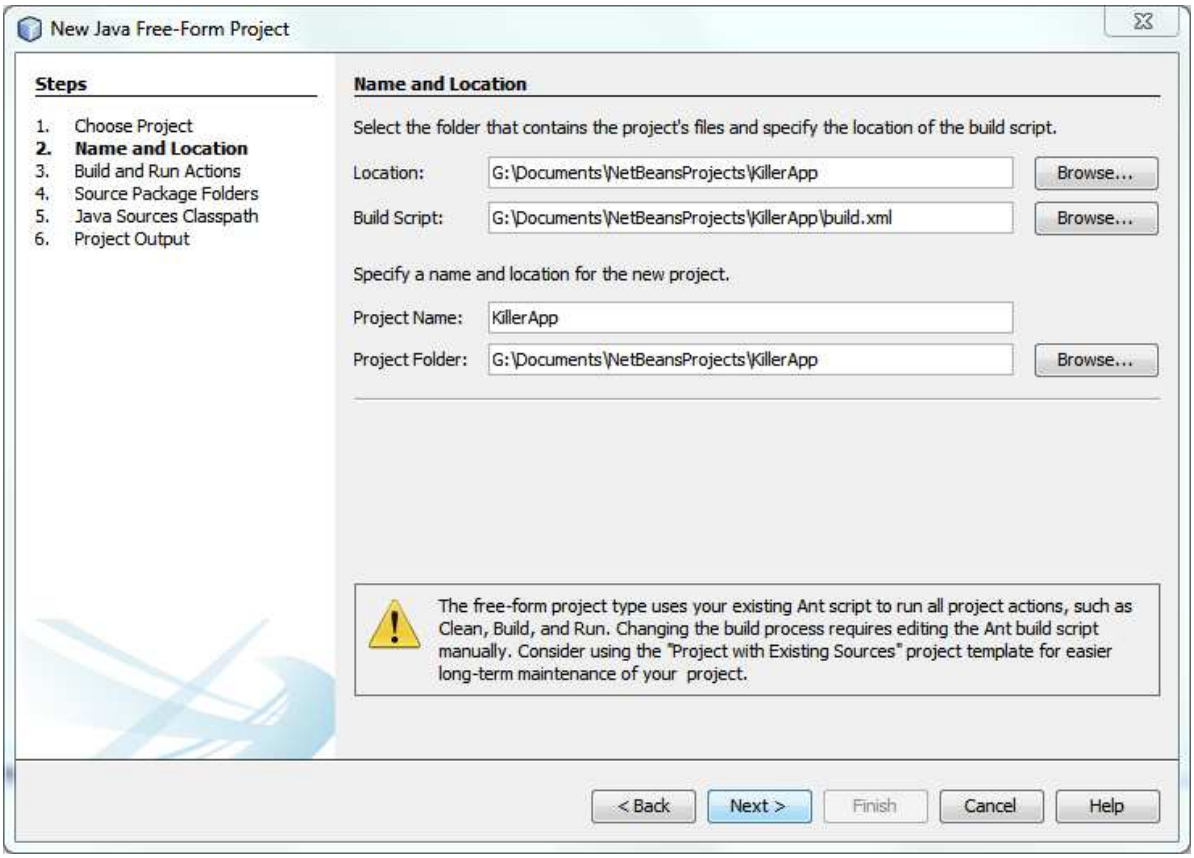
После этой команды в папке проекта завелись все нужные нам файлы: конфигурационные файлы и, самое главное, файл сборки. На этом работа с командной строкой закончена, и мы больше её не увидим. Теперь осталось поколдовать в IDE.

Создание проекта в IDE

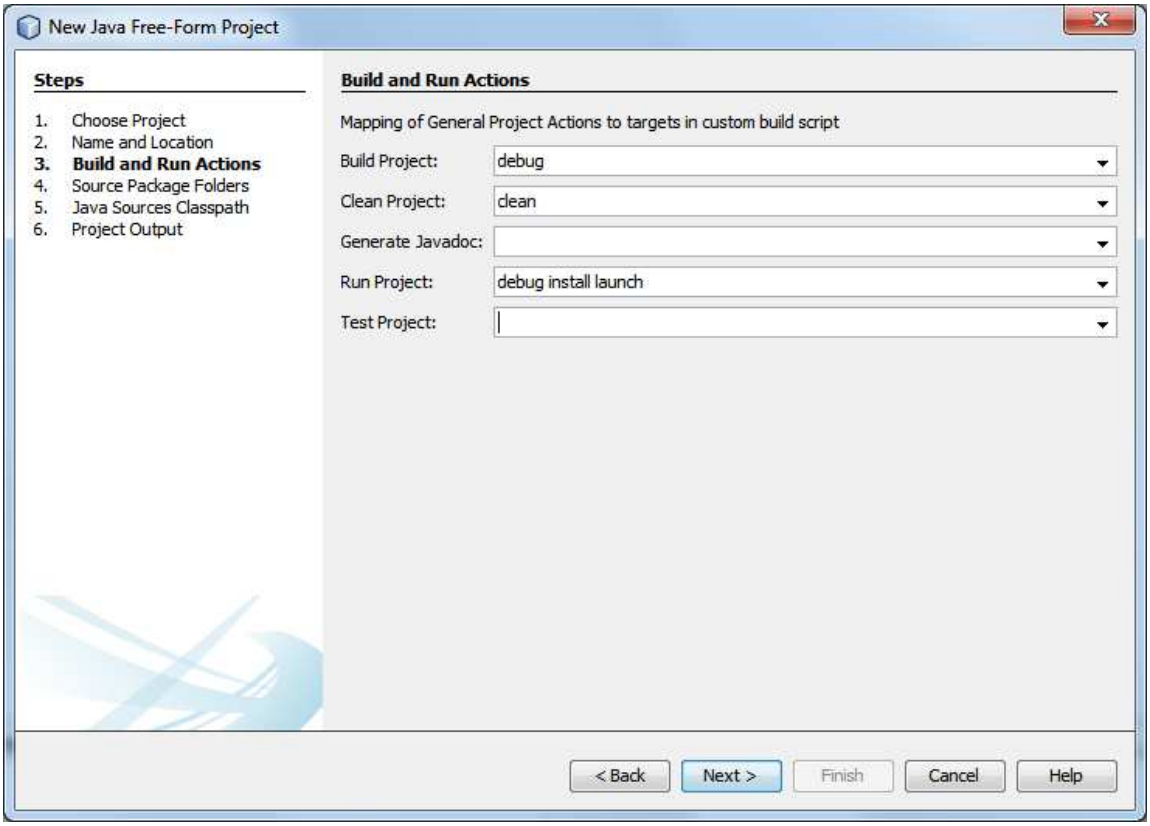
1. На экране создания нового проекта в NetBeans понадобится пункт *Java Free-Form Project*, с помощью которого мы растолкуем IDE, где брать файл сборки.



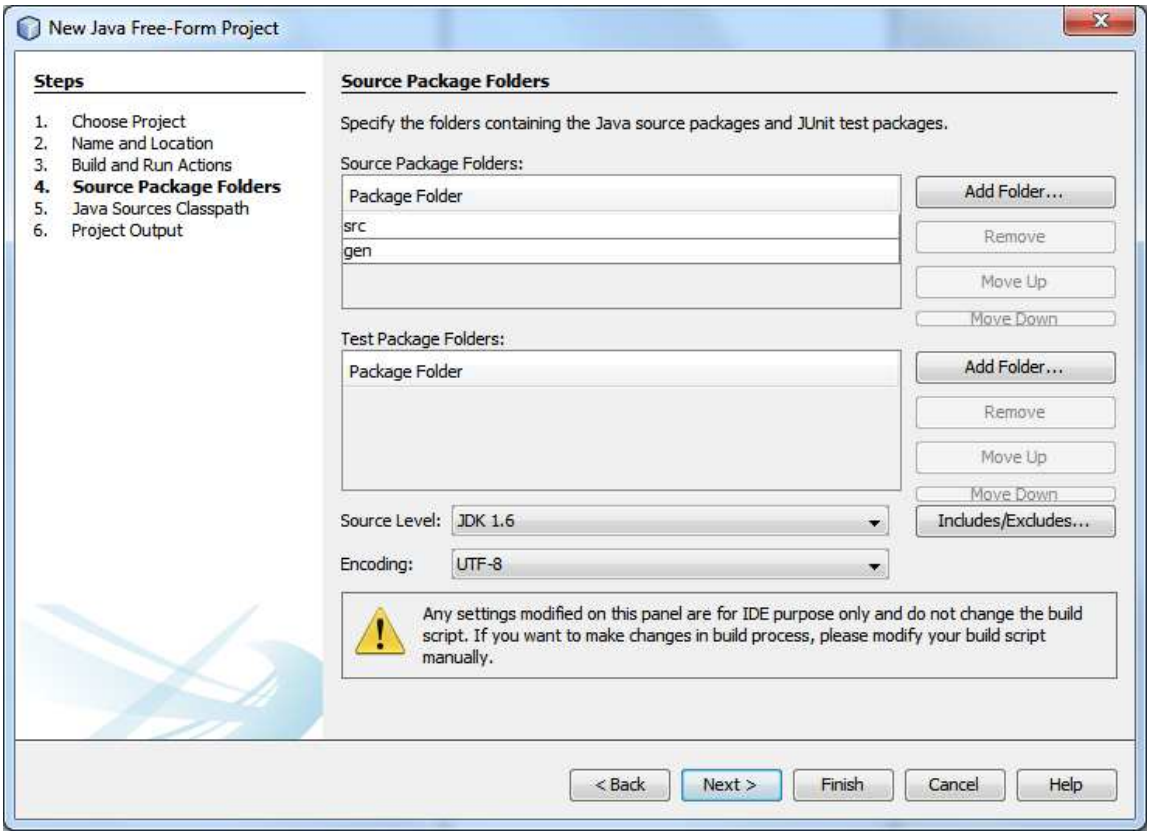
2. Дальше нужно выбрать папку проекта. NetBeans сам найдет файл сборки и сообразит, как называется проект, так что после выбора папки можно спокойно идти на следующий экран.



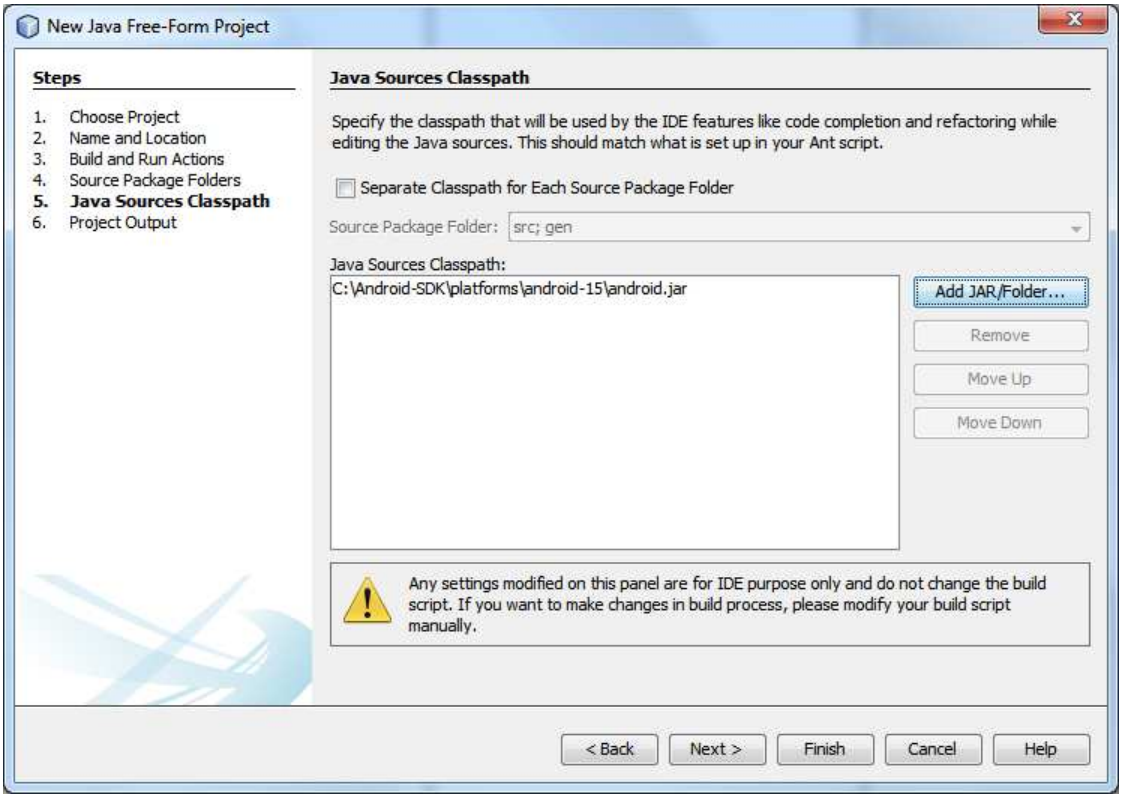
3. А вот теперь надо правильно прописать задания сборки, чтобы IDE знала, что запускать, когда мы заходим собрать проект. Сборка в системе осуществляется заданием *debug*. Это немного странно выглядит, но причина такого названия очень простая: это задание создаёт сборку для отладки, подписанную соответствующим сертификатом. Соответственно, есть задание *release*, до которого мы ещё доберёмся. Запуск проекта в нашем случае означает сборку и установку, что означает выполнение задания *install* после сборки. Там ещё приписано задание `launch`, которого в стандартной системе нет, но мы его сделаем и сами. Очистка — это, вполне ожидаемо, *clean*, а тестировать в Android нужно через отдельный, тестовый проект, поэтому то, что находится в том поле, можно смело стирать.



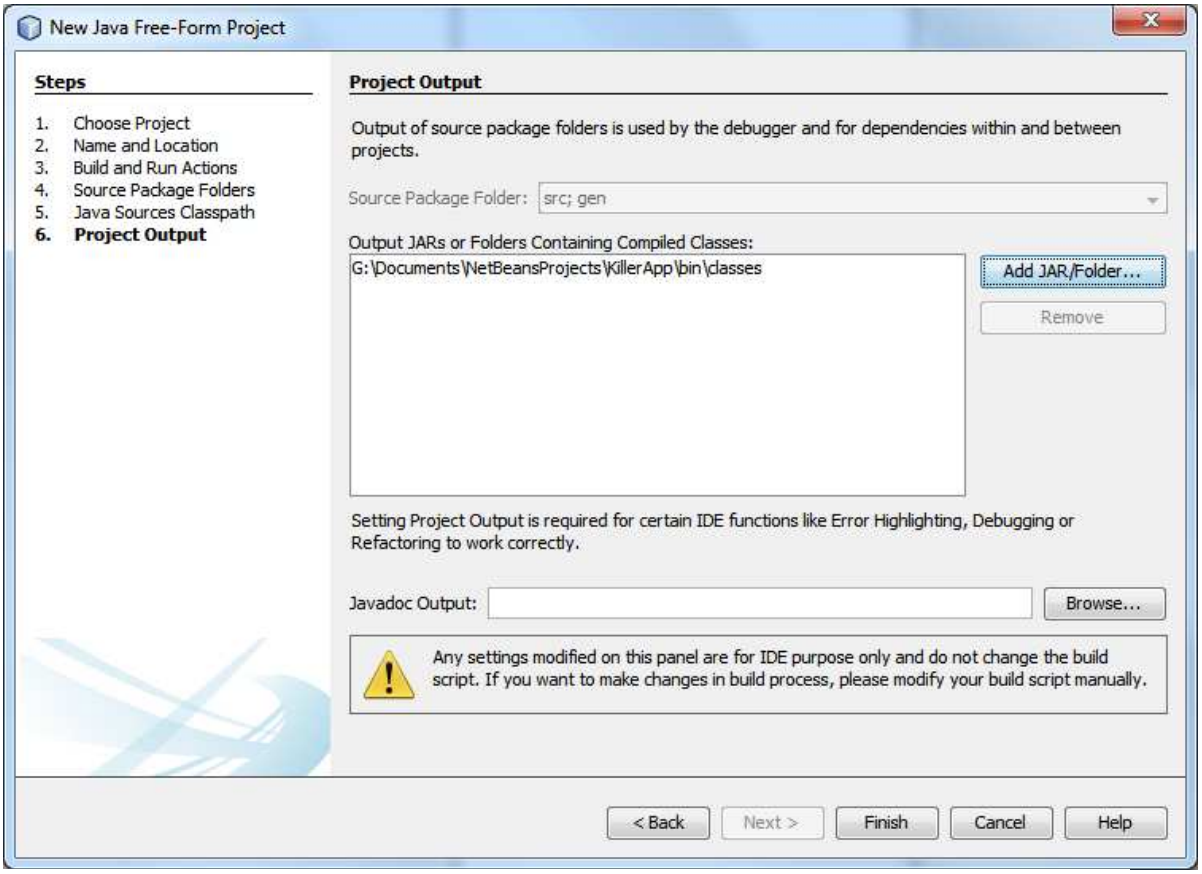
4. На следующем экране нужно добавить папку `gen` к папкам исходников, потому что именно в этой папке будет находится файл `R.java`.



5. Теперь настраиваем подсказки по коду. Прежде всего, важно снять флажок разделения папок с исходниками, иначе IDE будет думать, что файл `R` не должен упоминаться в коде нашей программы, поскольку лежит в отдельной папке. Также нужно добавить правильную платформу Android в список библиотек, в нашем случае это `<Android-SDK>/platforms/android-15/android.jar`.



6. И, наконец, последний шаг, добавляем папку `bin/classes`, чтобы IDE знала, где искать скомпилированный код. В принципе, этот шаг не обязателен, и на него можно смело наплевать. Но для полноты картины я сделаю и его, чтобы NetBeans показывал, какие файлы были недавно изменены и ещё не скомпилированы.



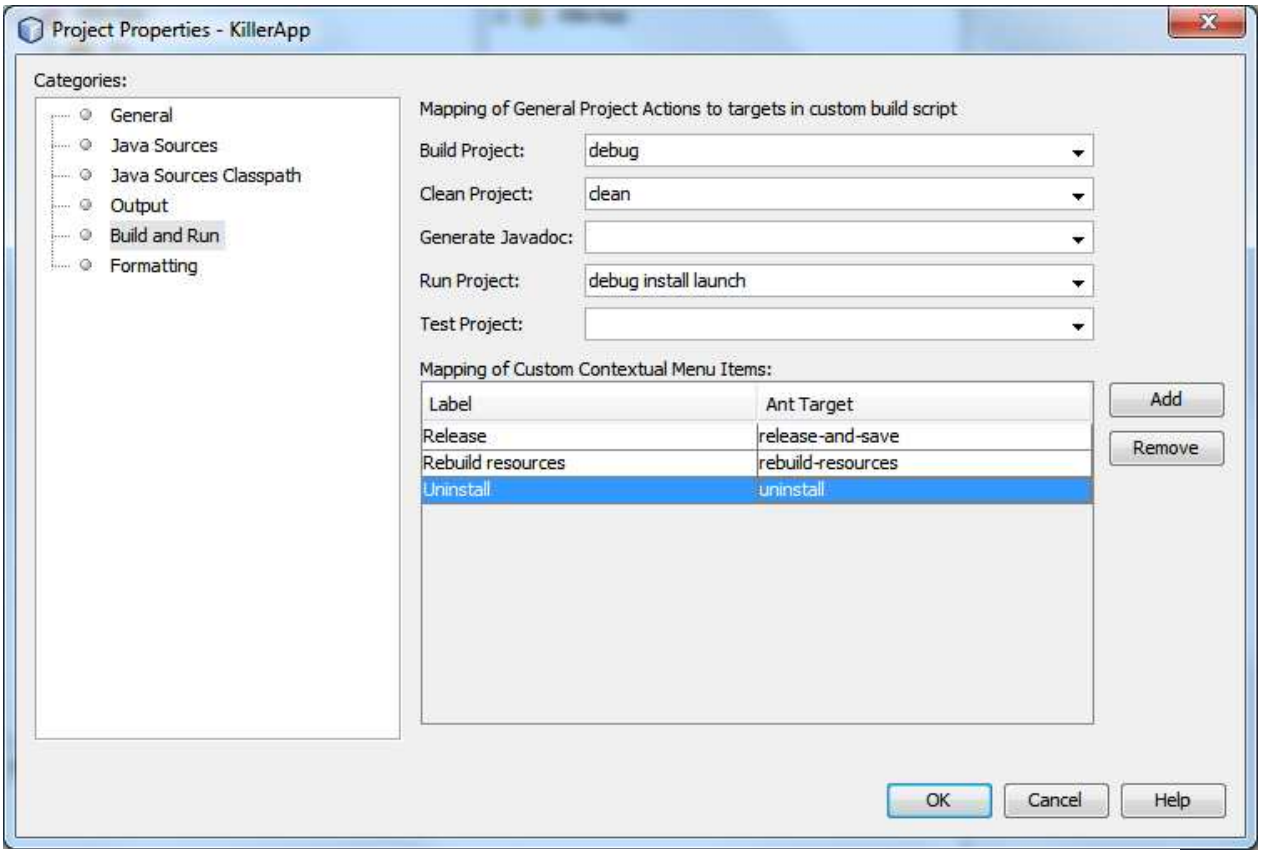
Добавление дополнительных команд

На самом деле, после последнего шага можно больше ничего не делать. Проект создан и уже нормально собирается. Но можно пойти дальше и наделать много удобств. **В папку проекта стоит положить файл** `custom_rules.xml`, а в нём записать необходимые нам задания ant

Эти три задания нам позволяют делать кое-какие весьма полезные вещи. `rebuild-resources` позволяет сгенерировать файл `R` (который в Eclipse, кстати, нередко куда-то исчезает или не обновляется вовремя). `launch`даст нам возможность запускать приложения, а `release-and-save` позаботится о том, чтобы при сборке финальной версии она сохранилась в отдельной папке под соответствующим именем вместе с картой методов ProGuard. Ещё я люблю добавлять такие строчки, чтобы после сборки проигрывалось уведомление:

```
<target name="-post-build">
    <sound>
        <success source="C:\Windows\Media\Windows Notify.wav"/>
    </sound>
</target>
```

Разумеется, этот конкретный вариант звука годится только для Windows, для других ОС стоит выбрать другой файл. Теперь осталось **добавить эти задания в контекстное меню в NetBeans**. В свойствах проекта на вкладке Build and Run мы уже заранее доработали пункт Run Project заданием `launch` в конце. В пользовательские элементы контекстного меню я добавил остальные только что сделанные задания:



Теперь в конекстном меню проекта весь букет необходимых нам команд. Можно запустить эмулятор или подключить смартфон, запустить Run и смотреть, как всё собирается и само запускается. Осталось сделать последние штрихи и**включить обфускацию при сборке финальной версии, раскомментировав строчку в файле**`project.properties`:

```
proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt
```

Также **в** `ant.properties` **стоит прописать строчки для подписи финальных сборок**:

```
key.store = <путь к файлу ключей>
key.alias = <название ключа>
```

Вот теперь у нас проект готов к работе.

Как работает система сборки в Android

На самом деле, системы сборки в Android на данный момент сейчас уже две: одна основана на ant, а другая — на Gradle. В данном конкретном случае мы пользуемся системой ant. Система с Gradle разрабатывается параллельно с новым редактором, идущим на замену Eclipse — Android Studio. Эта система сборки, думаю, пригодится для отдельной статьи.

Возвращаясь к ant, в папке проекта лежат следующие файлы:

- ant.properties
- build.xml
- local.properties
- proguard-project.txt
- project.properties

Самый важный файл здесь — это, разумеется, `build.xml`. На самом деле, это лишь маленький хвостик основной системы, и всё, что он делает — это загрузка свойств из файлов с расширением properties и вызов основной системы, располагающейся в самом SDK.

`local.properties` содержит всего лишь одно свойство: расположение папки с SDK. Этот файл **нужно занести в список исключений системы контроля версий**, потому что он содержит настройки, специфические для отдельной машины. Например, у меня под Windows этот файл содержит строчку

```
sdk.dir=C:\\Android-SDK
```

На самом деле, можно создать переменную окружения `ANDROID_HOME` с содержимым переменной из этого файла и благополучно отправить файл в мусорку. Главное, не забудьте после этого перезапустить NetBeans.

С `ant.properties` мы уже познакомились, там хранятся вспомогательные переменные вроде расположения хранилища ключей. Также есть файл `project.properties`. После действий, описанных в создании проекта, там есть только строчки об уровне API Android, для которого собирается проект, и о том, где искать файл конфигурации ProGuard. Когда мы будем добавлять в проект библиотеки, строчки о них окажутся там же.

Наконец, файл `proguard-project.txt`, который, как следует из названия, содержит указания ProGuard. Он изначально пуст, но это вовсе не значит, что ProGuard будет работать вхолостую, поскольку в папке SDK уже есть заранее записанная конфигурация (помните раскомментированную строчку о ProGuard?), а здесь мы можем её конкретизировать. Например, я лично люблю, помимо прочих, добавлять строчки

```
-renamesourcefileattribute MyProject
-keepattributes SourceFile,LineNumberTable
```

Они в дальнейшем помогают собирать отчёты об ошибках, поскольку теперь есть точные данные о том, какая строчка кода в проекте вызывает падение или исключение.

Также `build.xml` загружает файл `custom_rules.xml`, если он есть, в котором мы и добавили все необходимые нам задания. Стоит взглянуть на задания ещё раз.

```
<target name="rebuild-resources" depends="-set-debug-mode, -build-setup, -code-gen" />
```

Эти строчки взяты просто из системы сборки SDK. К сожалению, там они не вынесены в отдельное задание, поэтому пришлось делать это самостоятельно. Интереснее взглянуть на другие два задания:

```
<target name="release-and-save" depends="release">
    <xpath
        input="AndroidManifest.xml"
        expression="/manifest/@android:versionName"
        output="manifest.versionName"
        default="test"/>
    <xpath
        input="AndroidManifest.xml"
        expression="/manifest/@android:versionCode"
        output="manifest.versionCode"
        default="test"/>
    <copy
        file="${out.final.file}"
        tofile="releases/${ant.project.name}-release${manifest.versionCode}-${manifest.versionName}.apk"
        overwrite="true"/>
    <copy
        file="${obfuscate.absolute.dir}/mapping.txt"
        tofile="releases/mapping-release${manifest.versionCode}.txt"
        overwrite="true"/>
</target>
```

С помощью XPath здесь достаются параметры версии и дальше с их помощью генерируются имена файлов. В обычном ant поддержки XPath нет, так откуда же он тут взялся? В Android SDK Google добавили свои собственные инструменты для того, чтобы работать с приложениями было удобнее. Многие из их инструментов сводятся к запуску определённых файлов из SDK, но есть и те, которые облегчают написание файлов сборки, такие как `xpath`. Ещё одним полезным инструментом, например, является `if`, делающий именно то, что делает соответствующая конструкция в языках программирования: выполнение того или иного задания в зависимости от условия.

Второе задание тоже использует `xpath`, на этот раз задача немного сложнее:

```
<target name="-find-main-activity">
    <xpath
        input="AndroidManifest.xml"
        expression="/manifest/@package"
        output="project.app.package"
        default="test"/>
    <xpath
```

```
        input="AndroidManifest.xml"
        expression="/manifest/application/activity[intent-filter/category/@android:name =
'android.intent.category.LAUNCHER'][1]/@android:name"
        output="project.app.mainactivity"
        default="test"/>
    <if>
        <condition>
            <matches pattern="\..+|[^\.].*\..*[^\.]" string="${project.app.mainactivity}"/>
        </condition>
        <then>
            <property name="project.app.mainactivity.qualified" value="${project.app.mainactivity}"/>
        </then>
        <else>
            <property name="project.app.mainactivity.qualified" value=".${project.app.mainactivity}"/>
        </else>
    </if>
    <property name="project.app.launcharg" value="-a android.intent.action.MAIN -n
${project.app.package}/${project.app.mainactivity.qualified}"/>
</target>

<target name="launch" depends="-find-main-activity">
    <exec executable="adb">
        <arg line="shell am start"/>
        <arg line="${project.app.launcharg}"/>
    </exec>
</target>
```

Необходимо найти активность, которая определяет в своем фильтре намерений категорию`android.intent.category.LAUNCHER` — именно так в Android определяются активности, которые должны показываться в меню. Их может быть и несколько (хотя это бывает редко), поэтому задание берёт первую из них.

Есть и ещё одна загвоздка. Активности декларируются в `AndroidManifest.xml` либо записью с полным именем, либо только именем класса с точкой впереди, если активность лежит в главном пакете. По крайней мере, так говорит документация. Только вот проблема в том, что Eclipse и прочий инструментарий позволяет точку опускать и просто писать имя активности, когда она находится в главном пакете. Android-то такое попустительство терпит, а вот команда, запускающая приложения, уже нет. Приходится добавлять точку, когда её не хватает. Вот тут нам и поможет задание `if`, недавно мной упомянутое, и регулярные выражения.

Также было ещё совсем простое задание, которое воспроизводит звук. В нём был использован один из шести хуков, которые дёргаются из файла сборки в SDK:

- -pre-build
- -pre-compile
- -post-compile
- -post-package
- -post-build
- -pre-clean

Хуки отражают этапы, через которые проходит сборка программы: компиляция библиотек, генерирование кода (RenderScript, aidl, R, BuildConfig), компиляция проекта, упаковка APK, подпись и zipalign. Соответственно, `-pre-build` вызывается прежде, чем начнутся все эти действия, `-pre-compile` непосредственно перед компиляцией самого проекта, `-post-compile` — между компиляцией и упаковкой, `-post-build` — после упаковки, но до подписи, и `-post-build` вызывается в самом конце. Ну а когда вызывается `-pre-clean`, думаю, понятно из названия.

Вместо заключения

Сегодня мы посмотрели самое важное: создание проекта. Собственно, уже после этого проект вполне рабочий, можно садиться и строчить код. Но нам нужно будет добавлять в проект библиотеки, отлаживать, а также создавать тесты. Все эти действия тоже отлично получаются в NetBeans. Как это можно сделать, я опишу в следующей части статьи.

- android
- , netbeans
- , ant

Разработка под Android в NetBeans IDE без плагинов. Часть 2 tutorial

Разработка под Android*

Продолжаем начатый эксперимент, посвящённый настройке NetBeans IDE для программирования под Android. В прошлый раз нам удалось создать проект в NetBeans, настроить систему сборки, а также сделать автоматический запуск приложения. Кроме этого мы

немного посмотрели на то, как система сборки построена изнутри. Во второй части статьи мы пойдём дальше и посмотрим, как в NetBeans можно осуществлять отладку, создавать библиотечные проекты, а также добавлять библиотеки к проектам и работать с модульными тестами.

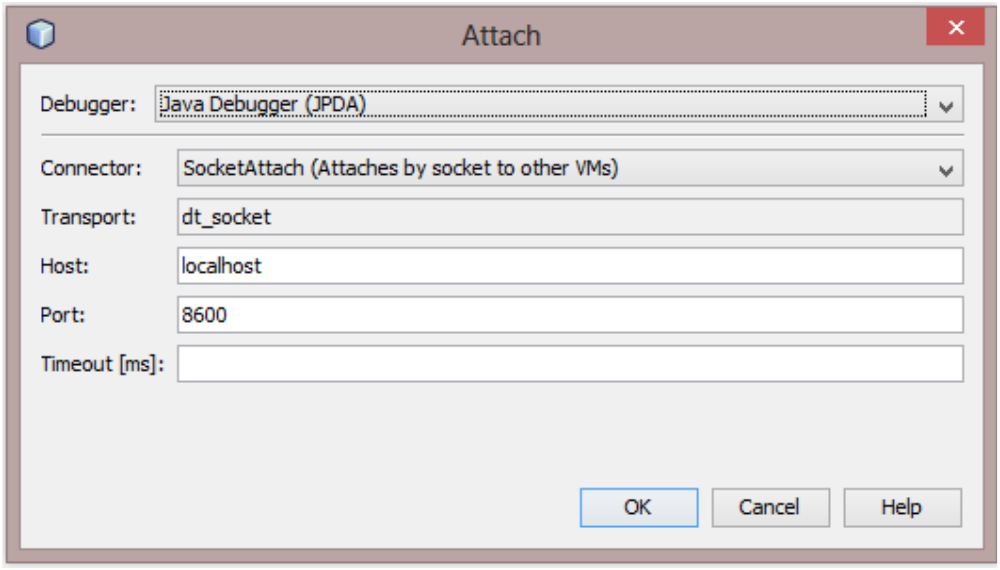
Отладка

Интересно, что отладку в NetBeans можно делать двумя способами. Первый способ даже не потребует никакого колдовства в ant. Второй будет немного посложнее настроить, но в некоторых случаях он пригождается, да и больше похож на то, что есть в Eclipse.

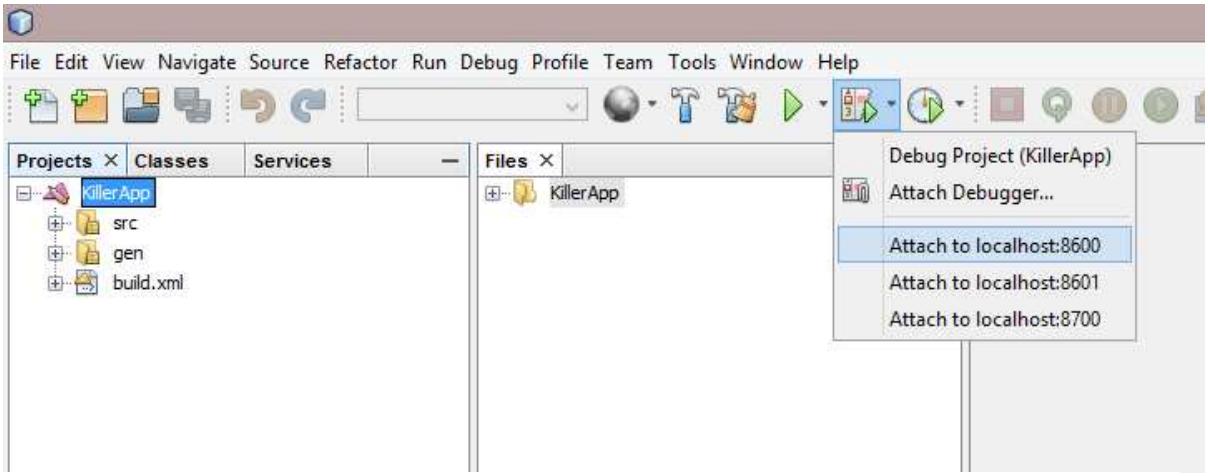
Способ №1

Перед тем, как начинать что-то делать, нужно вспомнить об инструменте под названием monitor. Находится он в папке `<Android-SDK>/tools`. Как я писал в прошлой части статьи, она уже должна быть в PATH, так можно запускать прямо из командной строки или строки поиска в Windows, но никто, конечно, не мешает и создать ярлык. Те, кто работал в Eclipse, сразу узнают все панельки в этом инструменте. Самые важные — это logcat и Devices.

После того, как monitor открыт, нужно запустить приложение, если оно ещё не запущено, и посмотреть, через какой порт нужно подключаться к отладчику. Порт пишется напротив приложения. По умолчанию они имеют схему 860x. Можно также щёлкнуть одно из приложений, чтобы назначить ему порт 8700. После этого в NetBeans нужно подключиться к этому порту через команду Attach Debugger. Выбираем параметры Socket Attach, localhost, необходимый порт… И всё, дальше можно спокойно заниматься отладкой.



На кнопке в панели запоминаются недавно введённые конфигурации, так что в следующий раз даже не придётся ничего вводить.



Отдельно обращаю внимание, что monitor вызывается не только для того, чтобы посмотреть на порты. Он также выступает посредником в соединении с виртуальной машиной, так что, если он не запущен, подключаться наугад к порту 8600, увы, не выйдет.

У этого способа отладки есть то преимущество, что можно подключаться и отключаться в любой момент. Это важно, потому что с подлкючённым отладчиком Dalvik VM начинает заметно тормозить. Иногда это бывает не критично, но не всегда, поэтому возможность дойти до определённой точки выполнения в программе без отладчика, бывает, вовсе и не лишнее.

Есть и ещё один инструмент, который помогает подключить отладчик именно в определённой точке. Можно, конечно, создать точку остановки по условию, но, как я уже говорил, с подключённым отладчиком всё работает очень не шустро. Поэтому можно в коде вставить вызов `Debug.waitForDebugger()`. Как только программа дойдёт до этого метода, она застопорится, и исполнение продолжится дальше только после подключения отладчика.

Но иногда нужно начать отладку в момент запуска программы. Можно воспользоваться этим же вызовом вышеупомянутого метода, а можно и настроить в NetBeans второй способ запуска отладки.

Способ №2

Второй способ будет действовать так же, как и в Eclipse: запускаем отладку, и запускается само приложение. После этого оно ждёт подключения отладчика, и только потом продолжает запуск. NetBeans нам поможет и здесь. Если попробовать выполнить отладку (`CTRL+F5`), NetBeans предложит сгенерировать ant-файл, который подскажет ему, как её нужно проводить в нашем проекте. Именно это нам и нужно. После этого в подпапке проекта `nbproject`заведётся файл `ide-file-targets.xml`, содержимое которого нужно заменить на следующее:

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir=".." name="KillerApp-IDE">
    <import file="../build.xml"/>

    <target name="-load-props">
        <property file="nbproject/debug.properties"/>
    </target>

    <target name="-check-props">
        <fail unless="jpda.host"/>
        <fail unless="jpda.address"/>
        <fail unless="jpda.transport"/>
    </target>

    <target name="-init" depends="-load-props, -check-props"/>

    <target name="-launch-monitor">
        <if>
            <condition>
                <not>
                    <socket server="localhost" port="8700"/>
                </not>
            </condition>
            <then>
                <exec executable="${android.tools.dir}/monitor${bat}"/>
                <waitfor maxwait="20" maxwaitunit="second">
                    <socket server="localhost" port="8700"/>
                </waitfor>
                <sleep seconds="2"/>
            </then>
        </if>
    </target>

    <target name="-launch-debug" depends="-find-main-activity">
        <exec executable="adb">
            <arg line="shell am start -D"/>
            <arg line="${project.app.launcharg}"/>
        </exec>
    </target>

    <target name="debug-nb" depends="-init, -launch-monitor, -launch-debug">
        <nbjpdaconnect
            address="${jpda.address}"
            host="${jpda.host}"
            name="${ant.project.name}"
            transport="${jpda.transport}"
        />
```

```
    </target>
</project>
```

Файл начинается с загрузки свойств из файла `debug.properties`, который нужно кинуть в ту же папку со следующим содержимым:

```
jpda.host=localhost
jpda.address=8700
jpda.transport=dt_socket
```

Теперь можно поразбираться, что этот файл делает. Основное задание здесь — это `debug-nb`, которое NetBeans запускает, когда начинается отладка, и зависит оно от заданий `-init`, `-launch-monitor` и `-launch-debug`. В `-init`ничего особенно интересного нет, задание просто загружает и проверяет переменные из файла `debug.properties`. А вот `-launch-monitor` уже позанятнее: нам ведь необходимо запустить monitor, если он ещё не запущен, и это задание как раз и берёт задачу на себя. В ant есть хорошее задание, которое позволяет посмотреть, слушает ли программа на определённом порте или нет — `socket`. По этому признаку как раз можно определить, работает ли monitor или нет. Если нет, то нужно его запустить и подождать (задание `wait-for`). После запуска ещё стоит подождать секунды две для того, чтобы monitor начал принимать соединения (значение, возможно, придётся немного скорректировать в зависимости от конкретной конфигурации оборудования).

После этого можно запускать само приложение. В прошлой статье мы уже это проделывали из ant с помощью командной строки. Для этого используется команда `adb shell am start -a android.intent.action.MAIN -n <пакет приложения>/<активность>`. В этот раз разберём команду немного подробнее. `adb shell` — это команда, позволяющая работать напрямую с командной строкой внутри Android. `am` — это менеджер активностей, у которого есть довольно впечатляющий набор возможностей; о них можно почитать в официальной документации. Нам же нужна лишь команда `start` для запуска нужной активности, которую мы указываем после ключа `-n`, а ключ `-a`задаёт, как уже, наверное, стало понятно, намерение.

В файле `custom_rules.xml` уже есть задание, которое выдаёт нужные для запуска параметры: `-find-main-activity`. В этот раз нам нужно запустить приложение точно так же, как и в прошлый раз, но с ключом `-D`, чтобы после запуска приложение не сразу продолжало работу, а сначала подождало отладчик.

Таким образом, после выполнения всех этих махинаций к запуску `debug-nb` уже всё готово: работает monitor, приложение запущено и ждёт отладчик. Осталось только его подключить с помощью задания `nbjpdaconnect`. Как понятно из названия, это задание сугубо специфическое для NetBeans.

Я сам пользуюсь вторым способом намного реже, чем первым, за счёт того, что, как я уже сказал, при подключении отладчика Dalvik VM начинает изображать тугодума, поэтому добраться до отлаживаемого участка в приложении становится дольше. Но, если проблема происходит при запуске приложения, этот способ — как раз то, что нужно.

Добавление библиотек и создание библиотечных проектов

Библиотека может быть прекомпилированным jar-файлом, а может быть и отдельным проектом Android, который нужно компилировать перед включением в проект. Подключаются они, соответственно, разными способами.

В случае **прекомпилированного файла** шагов очень мало:

1. Нужно бросить файл в папку `libs` основной папки проекта.
2. В свойствах проекта на вкладке Java Sources Classpath необходимо добавить путь к файлу. На самом деле, можно этого и не делать, но тогда IDE нам не будет подсказывать по коду из этой библиотеки, что сводит на нет преимущества использования IDE.

В случае **библиотечного проекта** всё немного похитрее. Можно его добавлять командой (и это официальный способ), а можно добавлять строчкой в конфигурационном файле. Для тех, кто любит **официальный способ**, нужна следующая команда:

```
android update project -p <путь к проекту> -l <путь к библиотеке относительно проекта>
```

Попробуем добавить ради примера библиотеку поддержки v7 appcompat, которую Google сделали для тех, кто хочет видеть панель действий на версиях Android до 3.0. Она как раз распространяется не как прекомпилированный jar-файл, а как библиотечный проект, поскольку там есть дополнительные ресурсы. Допустим, что он лежит в той же папке, что и наш основной проект.

```
android update project -p KillerApp -l ../AndroidCompatibilityPackage-v7-appcompat
```

Всё! Можно уже компилировать проект. Если мы заглянем в файл `project.properties`, то обнаружим **в этом конфигурационном файле строчку**

```
android.library.reference.1=../AndroidCompatibilityPackage-v7-appcompat
```

Собственно, это всё, что та команда и сделала. Точно таким же способом можно добавлять новые библиотеки безо всякой команды, главное только не забывать увеличивать номер библиотеки на единичку:`android.library.reference.2`, `android.library.reference.3` и так далее.

Разумеется, как и с прекомпилированным файлом, нужно не забыть добавить на владке Java Sources Classpath упоминаемые в проекте папки исходников библиотечного проекта, а также библиотеки, которые использует он (если мы их также используем). То есть стоит добавить папки `src`, `gen` и jar-файлы в папке `libs` библиотечного проекта.

Что если мы хотим создать свой такой же проект? **Создание библиотечного проекта** происходит в точности так же, как и создание обычного проекта с тем исключением, что нужна немного другая команда:

```
android create lib-project -n <имя проекта> -t android-<уровень API> -p <путь к проекту> -k <пакет программы>
```

Основное отличие в том, что вводится `lib-project` вместо `project`. Кроме того, не нужно указывать название главной активности, поскольку библиотеку не придётся запускать напрямую. Дальше создание проекта продолжается так же, как и для обычного проекта.

Создание проектов для тестов

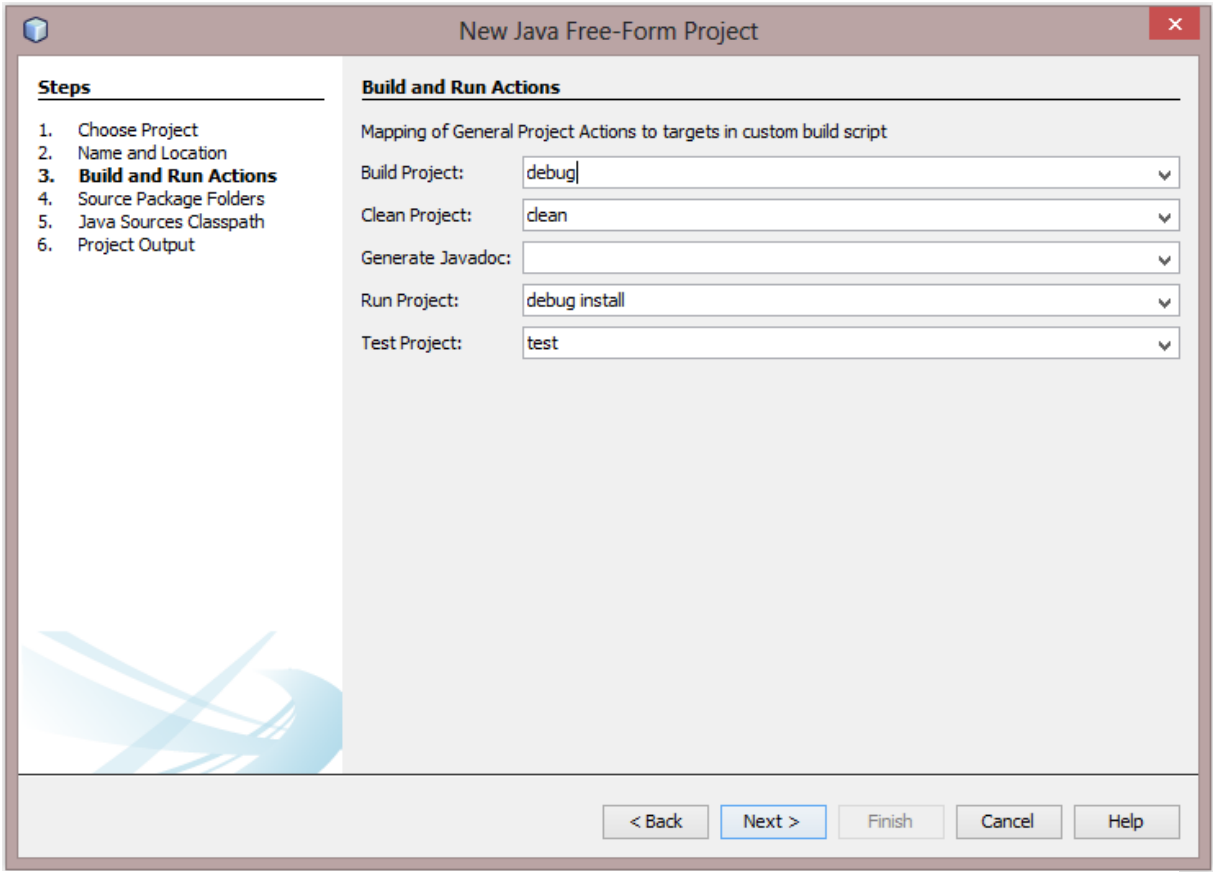
Как известно, в Android, к сожалению, нельзя встраивать модульные тесты непосредственно в проект, и нужно создавать отдельный проект для этого действа. Как и создание библиотечного проекта, все шаги очень похожи на создание обычного проекта, но немного больше нюансов. Потребуется следующая команда:

```
android create test-project -p <путь к проекту> -n <название проекта> -m <путь к основному проекту относительно проекта для тестов>
```

Проекты для тестов обычно создаются в подпапке основного проекта, поэтому создадим такой проект из папки основного проекта:

```
android create test-project -p tests -n KillerAppTest -m ..
```

Дальше можно продолжить создание нового проекта в NetBeans точно так же, как и в случае обычного проекта. Но на этот раз мы сможем оставить на третьем шаге пункт `test`, когда мы назначаем задания ant разным пунктам меню. А вот из `Run Project` теперь стоит убрать `launch` и оставить только `debug install`, поскольку запускать нам тут всё равно нечего.



Для обычного проекта после этого мы ещё добавляли файлы, связанные с запуском приложения, но в этот раз это нам ни к чему. А вот что можно сделать, так это добавить файлы, которые нам помогут отлаживать тесты и запускать их выборочно.

Для начала нужно сгенерировать файл для дополнительных заданий в NetBeans. Нам интересен запуск отдельных файлов, отладка и отладка отдельных файлов. Все эти действия можно сгенерировать нажатиями на `CTRL+F6`,`CTRL+F5` и `CTRL+SHIFT+F5`. После этого в папку `nbproject` нужно опять закинуть файлы, как и при добавлении отладки в обычный проект по второму способу, только файл `ide-file-targets.xml` будет немного другой. Начало файла такое же, как и в случае отладки обычного проекта, поэтому весь файл целиком я не копирую. Желающие могут посмотреть его на BitBucket. А вот дальше у нас другие задания:

```
<target depends="-setup" name="run-selected-file-in-src">
    <fail unless="run.class">Must set property 'run.class'</fail>
    <echo level="info">Running tests in ${run.class}...</echo>
    <run-tests-helper>
        <extra-instrument-args>
            <arg value="-e"/>
            <arg value="class"/>
            <arg value="${run.class}"/>
        </extra-instrument-args>
    </run-tests-helper>
</target>

<macrodef name="launch-debug-and-connect">
    <element name="debugged-class" optional="yes"/>
    <sequential>
        <parallel>
            <run-tests-helper>
                <extra-instrument-args>
                    <debugged-class/>
                    <arg value="-e"/>
                    <arg value="debug"/>
                    <arg value="true"/>
                </extra-instrument-args>
            </run-tests-helper>
            <sequential>
                <sleep seconds="5"/>
                <nbjpdaconnect
                    address="${jpda.address}"
                    host="${jpda.host}"
                    name="${ant.project.name}"
                    transport="${jpda.transport}"
                />
            </sequential>
        </parallel>
    </sequential>
</macrodef>

<target depends="-setup, -init, -launch-monitor" name="debug-selected-file-in-src">
    <fail unless="debug.class">Must set property 'debug.class'</fail>
    <echo level="info">Debugging tests in ${debug.class}...</echo>
    <launch-debug-and-connect>
        <debugged-class>
            <arg value="-e"/>
            <arg value="class"/>
            <arg value="${debug.class}"/>
        </debugged-class>
    </launch-debug-and-connect>
</target>

<target depends="-setup, -init, -launch-monitor" name="debug-nb">
    <launch-debug-and-connect/>
</target>
```

Задание `run-selected-file-in-src` нужно для запуска отдельных тестов. Оно использует макрос `run-tests-helper`, который определён в системе сборки Android с дополнительными параметрами. На самом деле, всё, что делает этот макрос — это запускает команду `adb shell am instrument` с параметрами для тестирования программы (да, это опять менеджер активностей). Мы добавляем к запуску команды аргументы `-e class <тестируемый класс>`, так что аппарат не будет гонять все тесты без разбора, а сосредоточится на конкретном файле.

Дальше остались задания, в которых нам нужно выполнить отладку. Для того, чтобы её сделать, нужно запустить сначала тестирование с указанием подождать отладчик, а потом подключиться. Но тут есть маленькая загвоздка: запуск тестирования происходит с блокировкой, а нам нужно запускать другое задание. Нас спасёт задание`parallel`, которое запускает разные задания вместе. Результат оформлен как макро, чтобы можно было регулировать, с какими параметрами вызывается тестирование.

Соответственно, наши задания для отладки просто его вызывают, с дополнительными параметрами, если нужно.

Итог

Теперь можно подвести итоги тому, что мы натворили. Суммарно получился очень неплохой объём возможностей:

- Полноценная сборка и запуск проектов;
- Генерация `R.java` вручную;
- Запуск отладки проектов;
- Добавление библиотек и библиотечных проектов;
- Создание проектов для тестирования;
- Запуск отдельных тестов с возможностью отладки;
- Подсказки и автозаполнение.

Проекты создаются одной командой из командной строки и парой дополнительных файлов, которые универсальны для каждого проекта, так что всё довольно просто по трудозатратам. Что у нас отсутствует по сравнению с тем, что уже есть в Eclipse или Android Studio:

- Редактирование интерфейса;
- Редактирование XML-файлов с подсказками;
- Переход к объявлению ресурса.

Редактирование XML-файлов — это не так критично, но, разумеется, без WYSIWYG-редактора довольно грустно редактировать интерфейс. Поэтому я лично импортирую проект в Eclipse и редактирую интерфейс там, когда это требуется.

Ещё хочется сказать пару слов насчёт применимости подобных инструментов. В комменатриях к предыдущей статье подобные вопросы возникали, поэтому ещё раз напомню: это эксперимент. Если существует официальная система сборки через ant, независимая от IDE, то для меня было сложно удержаться и не попробовать с её помощью настроить инструмент, который изначально вовсе не был предназначен для работы с Android.

Кроме того, на самом деле, ведь необязательно пользоваться NetBeans, чтобы задействовать эту систему. После настройки проекта можно просто набирать в командной строке, например, `ant debug install launch`, чтобы собрать и запустить проект. И, в отличие от сборки самодельным скриптом, это будет полноценная сборка — в точности такая же, которую делает и Eclipse с ADT: с генерацией интерфейсов из AIDL, BuildConfig, RenderScript, zipalign, ProGuard и всем прочим. Что касается использования её для того, чтобы программировать в NetBeans, это уже, конечно, сильно на любителя. Но, в любом случае, мне лично было очень интересно провести этот эксперимент и надеюсь, что другим было интересно о нём прочитать.