# 2 ОПЕРАЦИОННАЯ СИСТЕМА ANDROID. ЗАПУСК ПРИЛОЖЕНИЙ

Мобильное приложение – программа, предназначенная для работы на мобильном устройстве. Она состоит из инструкций, которые должны быть понятны для исполнения процессором. Процессор, в свою очередь, управляет работой операционной системы в соответствии с полученными последовательностями к выполнению. Это учит нас тому, что разработчик должен знать или понимать основы управления той операционной системой, для которой он будет писать программу.

Разные операционные системы диктуют свои правила написания программ. Таким образом, программа, написанная для Android, не может быть исполнена в iOS или Windows.

В некоторых случаях, необходимо не только понимать особенности взаимодействия программы с операционной системы, но также и особенности работы с разными типами процессоров. Хотя существуют стандартизированные виды архитектур, некоторые производители могут вносить свои изменения. В рамках данного курса не будут затрагиваться особенности разработки под конкретные процессоры, однако, разработчиком мобильных приложений приходится учитывать и данный момент.

Подавляющее число студентов имеют в обращении ноутбуки и смартфоны разных фирм, поэтому, в данном курсе упор будет произведен в сторону создания приложений для ОС Android, поскольку разработка под данную ОС не требует наличия специального оборудования и программного обеспечения. В материале ниже рассмотрим теоретические вопросы касательно архитектуры данной операционной системы.

## 2.1 Архитектура операционной системы

Операционная система Android – свободно поставляемое программное обеспечение на базе ядра Linux, изобретенное для широкого перечня устройств разного назначения (бытовая техника, смартфоны, планшеты и т.д.). Она состоит из нескольких слоев (рисунок 4). Каждый уровень предоставляет определенные службы для выполнения конкретных задач и использует возможности нижнего слоя, если это необходимо для выполнения функции.

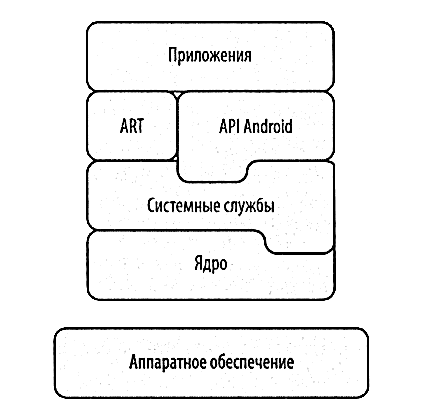


Рисунок 4 – Слои операционной системы Android

Стэк технологий (набор инструментов, применяющийся при работе) операционной системы включает:

* Ядро (Linux Kernel)
* Системные службы (Hardware Abstraction Layer)
* Нативные библиотеки (Native Libraries)
* Виртуальная машина (Android Runtime)
* Интерфейсы Java
* Системные приложения

На рисунке 5 приведена более подробная архитектура операционной системы.

##### Linux Kernel

Ядро – базовый уровень информационных систем, предоставляющий основные службы, драйверы для взаимодействия с оборудованием, файловые системы, службы для управления потоками и процессами, доступами к сети, интерфейсы для аппаратных устройств и т.д.

##### Системные службы

Обеспечивают связь между драйверами и библиотеками. Если рассматривать данный слой детально, то он включает в себя широкий спектр утилит, которые работают и как часть ядра (драйверы и модули ядра), так и приложения с длительным временем запуска для управления служебными задачами или библиотеки для выполнения стандартных функций.

##### Android Runtime (Art)

Перед архитекторами системы стояла задача создать такую платформу, которая поставлялась бы на разные устройства, для которых исполняемая программа могла быть написана в единственном экземпляре (а не столько одинаковых программ, сколько устройств и процессоров). Такое кроссплатформенное поведение уже было реализовано в Java - для запуска приложений на данном языке необходима виртуальная машина – Java Virtual Machine (JIT). Поэтому для Android первым и основным языком написания программ был именно этот язык. А для оптимизации запуска приложений на операционной системе использовалась виртуальная машина Android Runtime (ART).

В основном, выделяют следующие варианты оптимизации, используемые в ART:

* just-in-time (JIT) – файл приложения компилируется (преобразуются) в машинно-читаемый бинарный код (байт-код) во время запуска приложения
* ahead-of-time (AOT) – компиляция байт-кода заранее. Т.е. система сама решает, когда и какой код компилировать.

##### Native C/C++ Libraries

Как описано в названии, данный слой предоставляет средства для использования библиотек, написанных на языках C и C++, например, из представленных на рисунке:

* WebKit – представляет из себя движок веб-браузера и другие связанные с ним функции.
* Media Framework – предоставляет медиа кодеки, позволяющие записывать и воспроизводить различные медиа-форматы.
* OpenGL – используется для отображения 2D и 3D графики.
* и т.д.

##### Application Framework

Application Framework (каркас приложений) – набор Java классов, предоставляющий приложениям интерфейс к функциям операционной системы и библиотекам. API (Application Programming Interface) – программный интерфейс приложения в виде исполняемых функций для взаимодействия.

##### Системные приложения

На вершине стека находятся приложения, которые могут вызывать другие приложения, например: камера, контакты, вызов номера.

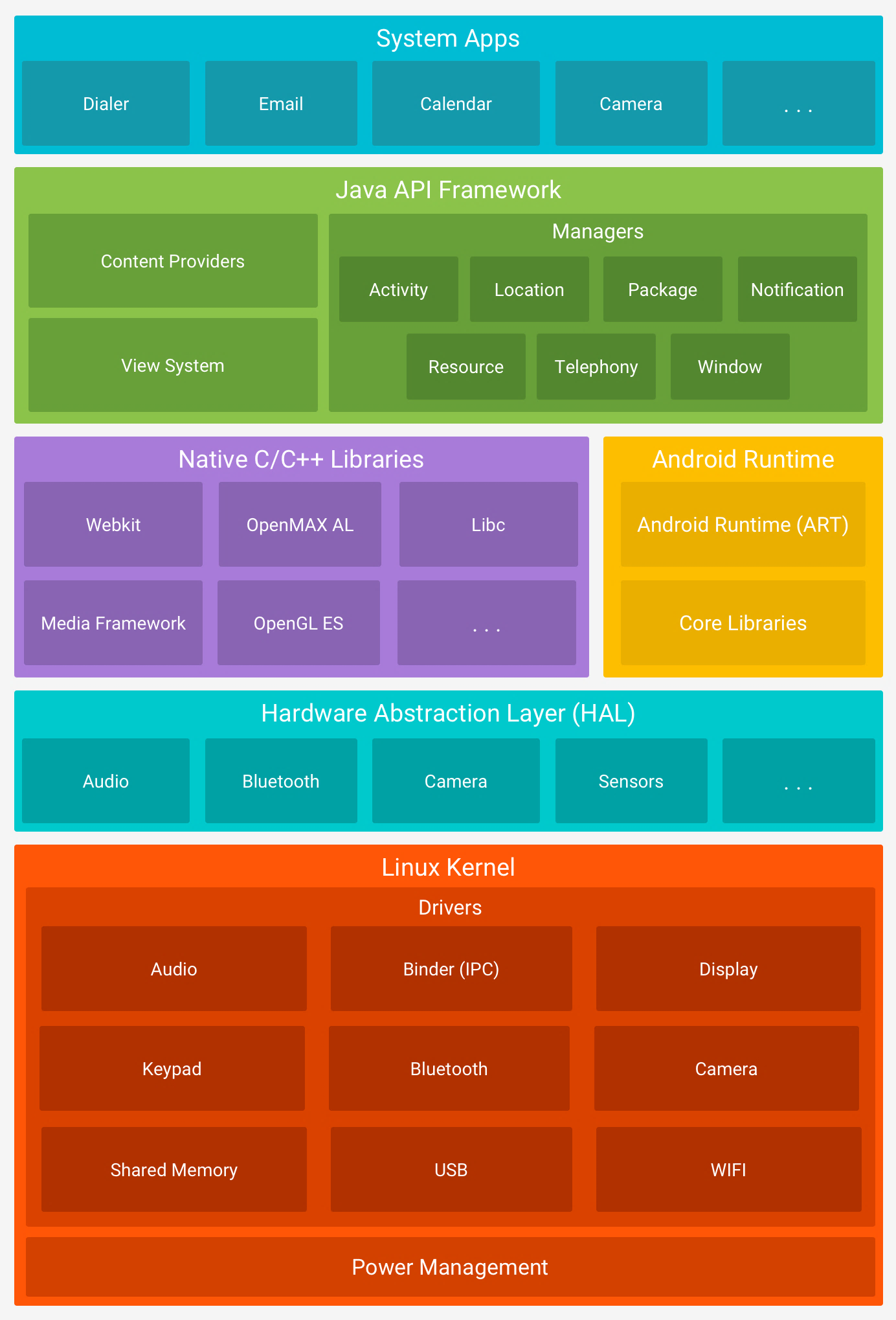


Рисунок 5 – Архитектура операционной системы Android

## 2.2 Компиляция Java приложений

Поскольку основа разработки под платформу Android была заложена с использованием Java, рассмотрим основные моменты, связанные с компилятором и поставляемым инструментальным обеспечением для разработки и запуска.

Озаботившись тем, как запускать одну программу на различных устройствах, Sun Microsystem пришла к выводу, что недостаточно создать, к примеру, универсальный компилятор. Необходимо разработать целый комплекс, после установки которого, программа, написанная единожды, сможет быть запущена на любом устройстве, где развернут (установлен) данный комплекс.

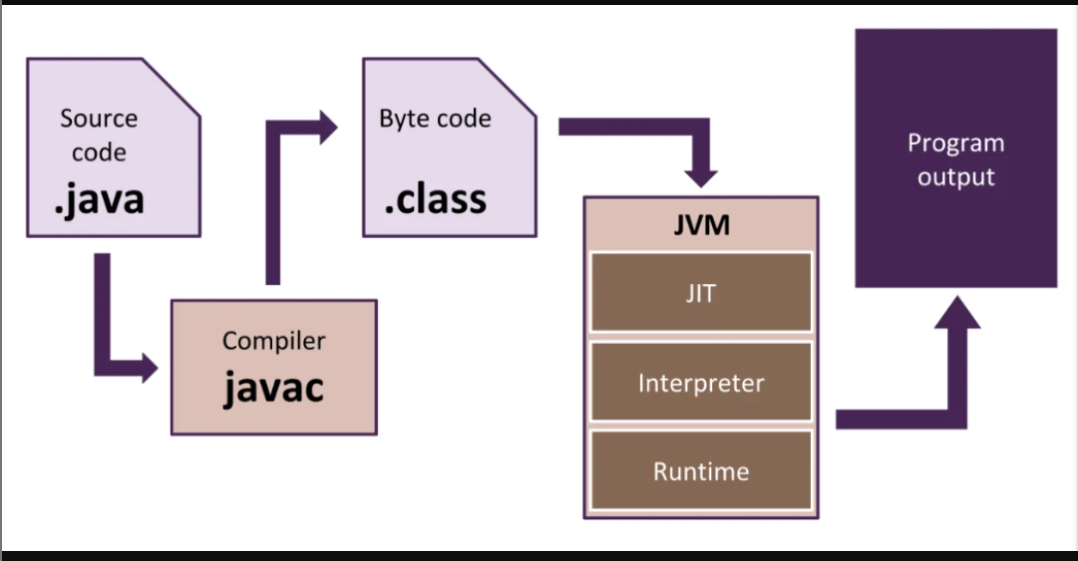


Рисунок 6 – Компиляция приложений в байт-код

То, что программист написал, называется исходным файлом (Java Source File). После этого, компилятор (инструмент javac) переводит данный файл в Java Byte – байт код программы. Однако, данный файл не может быть исполнен на разных устройствах в текущем виде. Как раз для того, чтобы сделать файл переносимым между устройствами, существует JVM.

Ядром инфраструктуры Java является JVM (Java Virtual Machine). Ключевая задача виртуальной машины – обработка и подготовка кода для посредника, который, в свою очередь, будет переводить код для конкретного процессора. Таким образом, байт-код программы java не зависит от операционной системы, а JVM зависит от того, в какой ОС она развернута. То есть, в каждой операционной системе, при условии, что была установлена инфраструктура, уже развернута необходимая виртуальная машина

Подводя итог:

- виртуальная машина делает операцию интерпретации – построчное преобразование исходного кода в машинный с одновременным исполнением

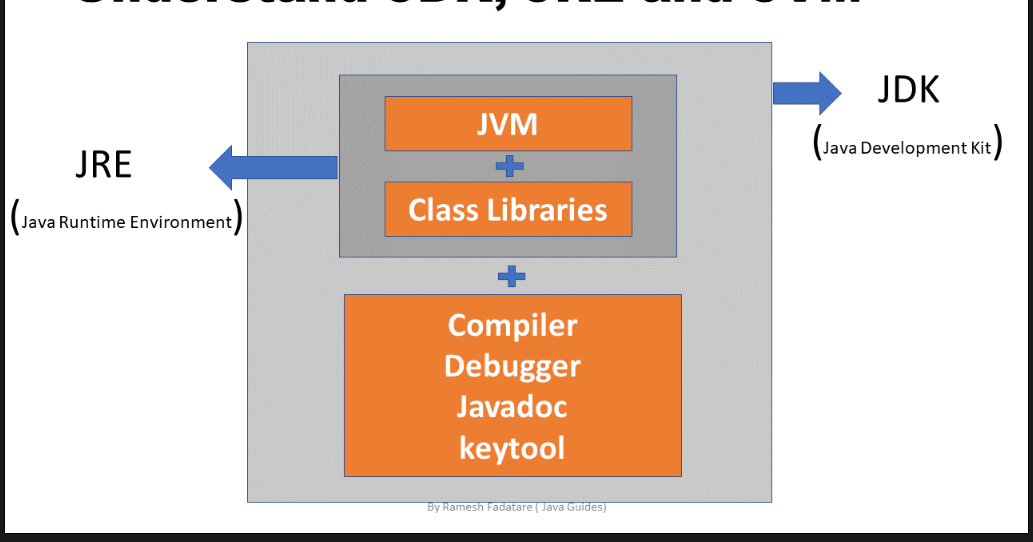
- виртуальная машина является программно-аппаратным эмулятором – средой, в которой запускаются Java прилозжения

- программа Java сама по себе не является исполняемой процессором напрямую

- программа Java может быть выполнена на любом устройстве, где установлена виртуальная машина

***Взаимодействие ОС и JVM***

Операционная система не взаимодействует напрямую с виртуальной машину. Существует определенная инфраструктура для запуска JVM.



Как видно на рисунке, над виртуальной машиной присутствуют JRE (Java Runtime Environment) и JDK (Java Development Kit).

JRE – окружение выполнения программ Java. Это та часть программного пакета инструментов, которая запущена на компьютере постоянно. При запуске программы на java, JRE создает экземпляр JVM и в ней запускает уже выбранную программу. Среда выполнения содержит JVM, корневые библиотеки для запуска и другие компоненты для запуска приложений.

JDK – набор инструментов разработки программ на языке Java: компилятор, дебаггер (программа для отладки, поиска и исправления ошибок в коде), а также библиотеки и инструкции для разработки.

Строго говоря, пользователи, не являющиеся программистами, при установке программ на данном языке (через установщик) ставят изолированно JRE и ее компоненты, без JDK.

***Архитектура JVM***

Приведем еще немного теории для понимания запуска программ Java. Это необходимо для того, чтобы понимать некоторые ошибки при проблемах запуска файла. Например, иногда разработчикам необходимо вручную запускать процесс компиляции файла в байт код для диагностирования, что программа действительно не запускается из-за ошибки внутри кода, а не из-за проблем в работе операционной системы.

Для эффективного запуска программ на разных платформах, виртуальная машина берет на себя управление памятью, сборку мусора и оптимизацию во время выполнения.

В абстрактной архитектуре машины присутствуют: загрузчик классов, область памяти, исполнительный механизм.



Загрузчик классов бывает трех видов:

- Загрузчик классов начальной загрузки – инструмент (BootStrap), загружающий Java-классы из специальной папки rt.jar (файл JRE, содержащий стандартные библиотеки Java)

- Загрузчик классов расширений (Extension) – дочерний класс от предыдущего, загружает расширения основных классов Java из каталога jre/lib/ext или любого другого каталога, на который указывает java.ext.dirs.

- Системный загрузчик классов (System) **–** загружает классы, найденные в переменной среды CLASSPATH, -classpath или параметре командной строки -cp.

Область памяти:

- Область метода предназначена для хранения данных файлов .class: например, метаданные, данные полей и методов, а также код метода. Область метода создаётся автоматически при запуске виртуальной машины, и для каждой ВМ существует только одна область метода.

- Область кучи. В куче хранятся все объекты и соответствующие им переменные экземпляра. Когда мы создаём новый экземпляр класса, он сразу же загружается в область кучи, которая остается единственной во время выполнения задачи.

- Область стека. В неё загружаются все локальные переменные, вызовы методов и частичные результаты.

- Регистры ПК. В регистре ПК хранится адреса виртуальных машин Java, выполняющих операцию в данный момент. В Java каждый поток получает свой собственный регистр ПК.

- Стеки нативных методов. Нативные методы — это методы, написанные на C или C++. Виртуальная машина JVM хранит стеки, которые поддерживают такие методы, с отдельным стеком собственных методов, выделенным для каждого потока.

Исполнительный механизм используется для тестирования оборудования и программного обеспечения. Состоит из:

- интерпретатора;

- JIT-компилятора;

- сборщика мусора

Перед выполнением программы интерпретатор и компилятор JIT («Just-in-time» — «точно в нужное время») преобразуют байт-код в машинные инструкции. Интерпретатор делает это построчно.

В тот момент, когда в сценарии обнаруживается повторяющийся код, к нему подключается JIT-компилятор для ускорения операции. Затем он компилирует байт-код и заменяет его собственным кодом. Такой процесс повышает производительность всей системы.

За операцию освобождения памяти от объектов без циклических ссылок отвечает сборщик мусора, оптимизируя использование памяти программно.

Важно отметить, что сборка мусора выполняется в JVM автоматически через определённые отрезки времени и не требует отдельного внимания специалистов. Конечно, этот процесс можно попробовать принудительно запустить, вызвав System.gc(), но нет никакой гарантии, что это сработает.

***Процесс разработки и запуска программ***

Подводя итог, чтобы разрабатывать программы на Java, разработчику необходимо поставить JDK, JRE, JVM.

Написание кода

Устройство

JDK

Компиляция

JRE

JVM

Запуск

Compile

Interpret

В процессе написания кода, разработчик использует библиотеки и дебаггер. При запуске (из среды разработки или в JRE при запуске exe-файла программы вручную) происходит компиляция файла исходного кода в байт код и передается в среду исполнения. В среде исполнения запускается виртуальная машина, которая интерпретирует байт-код в файл для запуска программы на данном устройстве

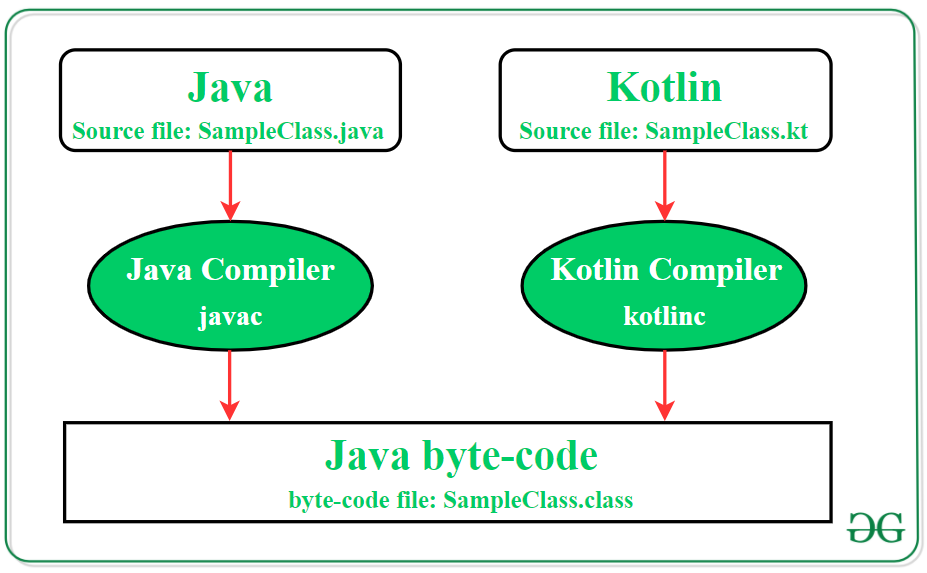
## 2.3 Сборка приложений Android

Разработка приложений для Android включает некоторые процессы, выполняющиеся друг за другом. После написания программы, когда разработчик нажимает кнопку Run в среде разработки, запускаются следующие процессы и операции в «бэкграунде». Дальнейшее описание содержит важнейшие шаги, которые происходят в IDE для запуска разработанного приложения.

***Компилятор***

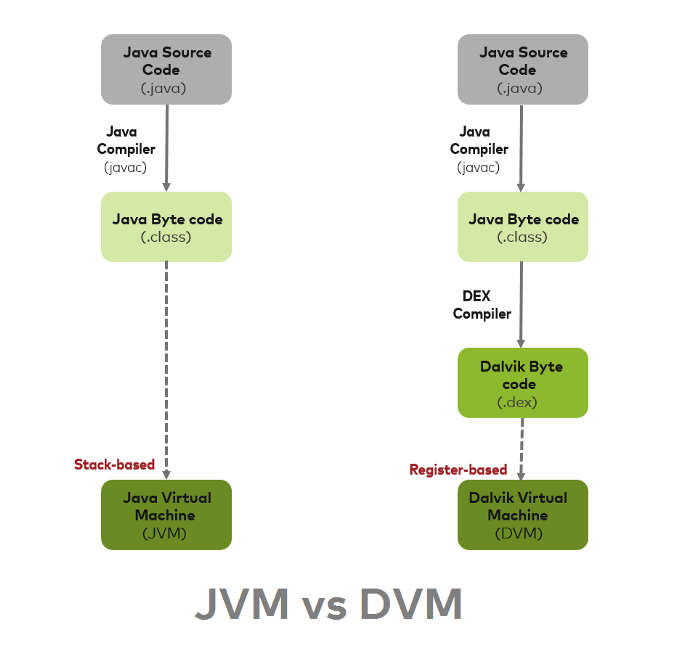
Поскольку программа должна быть понятна компьютеру, ее необходимо преобразовать в двоичный код. Для этого создан компилятор – программа, переводящая исходный код в набор машинных кодов.

Программа на исходном языке (Java или Kotlin) компилируется в файлы типа .java или .kt (соответственно).



***Перевод в Dalvik байткод***

Сгенерированный файл в предыдущем шаге подходит для выполнения JVM, так как содержит стандартные инструкции Oracle JVM Java byte-codes. Этот формат не поддерживается на устройствах Android, поскольку последние работают с файлами уникального формата – Dalvic byte-code. Dex компилятор преобразует Java byte-code в файл формата DEX, который аналогичен файлам .class, но содержит байткод всего приложения (или части приложения в случае Multidex), а не одного класса.



DEX файл – это результат компиляции кода Android-приложения. DEX расшифровывается как Dalvik Executable.

***Среда исполнения***

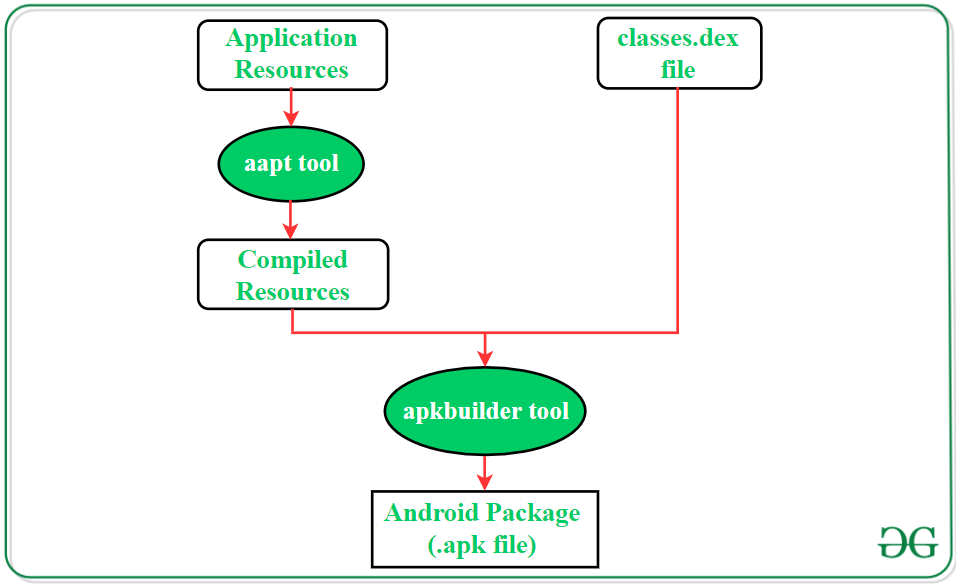
К моменту разработки операционных систем для мобильных устройств, Java зарекомендовала себя как надежное кросплатформенное решение в качестве примера среды выполнения программ. Таким образом, в 2003 году начинается разработка мобильной операционной системы, которая бы могла без привязки к аппаратной части устройства запускать приложения из общего магазина приложений. Вспоминая инфраструктуру Java, можно прийти к выводу, что разработчики Android занялись реализацией похожей виртуальной машины для запуска приложений.

Dalvik – это [виртуальная машина](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0), которая использовалась с первой версии до Android v5.0. Ее стратегия компиляции байткода в машинный код заключалась в использовании JIT-компиляции, при которой байткод компилируется во время исполнения команды.

ART до Android v7.0 использует [AOT-компиляцию (Ahead-of-Time)](https://ru.wikipedia.org/wiki/AOT-%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F), при которой приложение компилируется в машинный код во время установки. В Android v7.0 в ART был добавлен JIT-компилятор в дополнение к AOT. Смешанный подход помог смягчить недостатки AOT-компиляции.

***Генерация .apk***

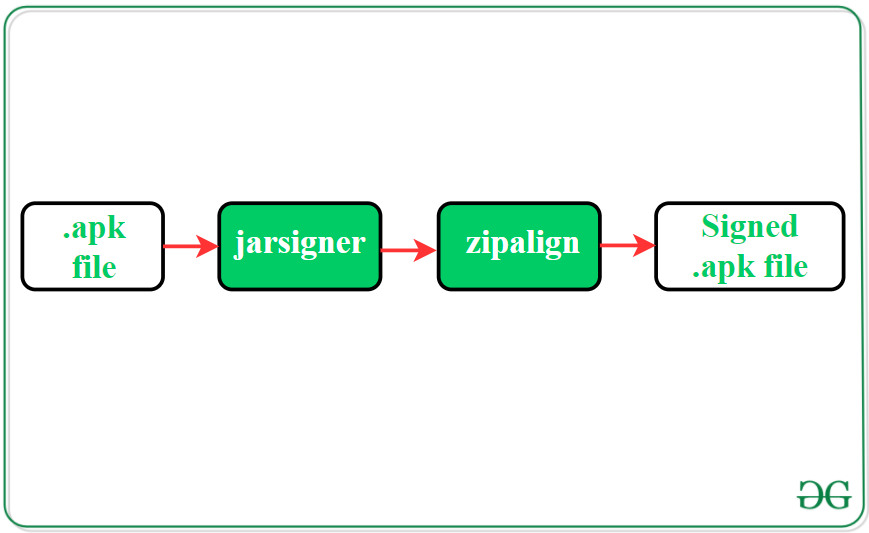
Файлы ресурсов приложения (картинки, шрифты, XML разметки и т.д.) компилируются в один исходник через Android Assret Packaging Tool(aapt). Данный инструмент так же отвечает за создание R.java файла приложения Android. Далее, скомпилированные отдельно ресурсы вместе с classes.dex файлами компануются средствами apkbuilder tool в zip-файл, который называется Android Package (.apk файл). Сгенерированный архив содержит все необходимые данные для запуска Android-приложения.



***Публикация приложения***

Подготовленный на предыдущем шаге .apk файл является готовым к исполнению пакетом и разработчик может использовать его для размещения своего приложения в магазине приложений. Важно помнить, что на каждое размещаемое приложение необходимо создать цифровой сертификат, который устанавливается на каждом устройстве пользователя. Данный сертификат используется Android для определения автора приложения. Разработчик-автор помещает закрытый ключ сертификата в виде дополнительного файла в пакете .apk.

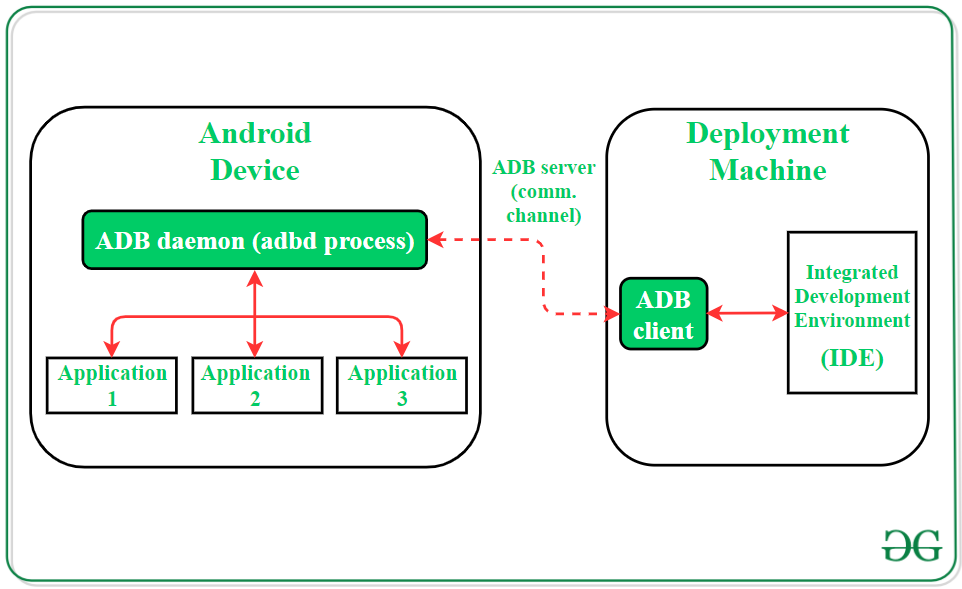
Пакет инструментов разработчика Java (JDK) предоставляют jarsinger tool для «подписания» файлов .jar и .apk. Кроме того, сжатые файлы signed apk.file читаются определенным образом операционной системой. При чем инструмент, благодаря которому байты файла-архива могут быть прочитаны, позволяет запустить именно подписанный файл таким образом, что система читает остальные файлы архива без его распаковки. Инструмент называется zipalign.



## 2.4 Развертывание приложений при отладке

***ADB Server***

Andrid Debug Brige (ADB) разворачивает приложение на устройстве. Это инструмент командной строки, который предоставляет удобный инструмент разработчику для взаимодействия с устройством. Для того, чтобы начать разворачивать приложение, клиент сервера проводит первую проверку того, запущены ли процессы ADB server на устройстве и запустит их, если они не были запущены ранее. Сервер стартует на локальном порту 5037. Все взаимодействия и команды переводятся от сервера к клиенту через использование данного порта. После того, как соединение установлено со всеми запущенными устройствами, начинается сканирование всех портов для того, чтобы определить ADB daemon (adbd: фоновый процесс на эмуляторе или экземпляре устройства). Связанный adbd process с устройством android предоставляет возможность коммуникации с приложениями, их отладку и сбор их выходных логов (log output).



***Перенос сборки проекта на устройство***

ADB команда переносит .apk в локальную файловую систему на контрольное устройство Android. Расположение приложения на устройстве определяется именем его пакета, т.е., для приложения с именем com.example.sampleapp система расположит его apk-file   
по пути /data/app/com. example.sampleapp.

## 2.5 Запуск приложения

***App launch request***

Zygote process – родительский процесс для всех приложений Android, которое запускает приложение, которое запросил пользователь. Данный процесс – специальный тип процесса данной операционной системы, который позволяет делиться кодом между разными экземплярами приложений, которые запущены в Android Virtual Devices (Dalvik/Andtoid Runtime). Те ресурсы, классы и библиотеки, которые потребуются для приложений предварительно загружаются в область памяти процесса zygote. Когда бы ни произошел запрос на запуск нового приложения, процесс делает собственный «форк» (разветвляется, делает копию) с использованием системного вызова fork() (создание процессов в системе Linux) и запускает новое приложение. Так как необходимые ресурсы уже загружены в память, запуск происходит быстро.

***Переход кода .dex в OAT формат***

Когда новое приложение установлено, Android оптимизирует его данные и генерирует соответствующий файл OAT. Этот файл, сгенерированный операционной системой, ускоряет время загрузки приложения. Процесс генерации OAT файла запускается вместе с процессом извлечения classes.dex, предоставляемого из файла сборки. Classes.dex файл размещается в отдельной директории и операционная система компилирует Dalvik байт-код ahead-of-time компиляцией (AOT так же имеет аббревиатуру OAT) в нативный машинный код. Android система использует такой подход для того, чтобы усилить восприятие пользователя о том, что приложения запускаются быстро и плавно.

До появления AOT, использовался инструмент dexopt для преобразования .dex в odex (оптимизированный DEX файл). Как становится из описания, последний формат говорит о том, что исходник – это оптимизированный байт-код. С появлением AOT, оптимизацию и преобразование делает инструмент dex2oat. Теперь .dex становится AOT файлом, который написан в формате ELF (Executable and Linkable format – исполняемый и связываемый формат). Нативная библиотека «мапится»[[1]](#footnote-1) в памяти процесса приложения. Данный файлы сохраняется в директории: /data/dalvik-cache/.

Только после всех перечисленных шагов и процедур, приложение запускается и initial activity приложения и оно отображается на экране устройства.

1. «Мапинг» в данном случае необходимо воспринимать как разбиение информации в контексте А и Б и дальнейшее сопоставление данных частей в виде мапы вида: часть контекста А -> часть контекста Б.

   Для понимания контекста, можно воспользоваться информацией по структуре Map в языках программирования. [↑](#footnote-ref-1)