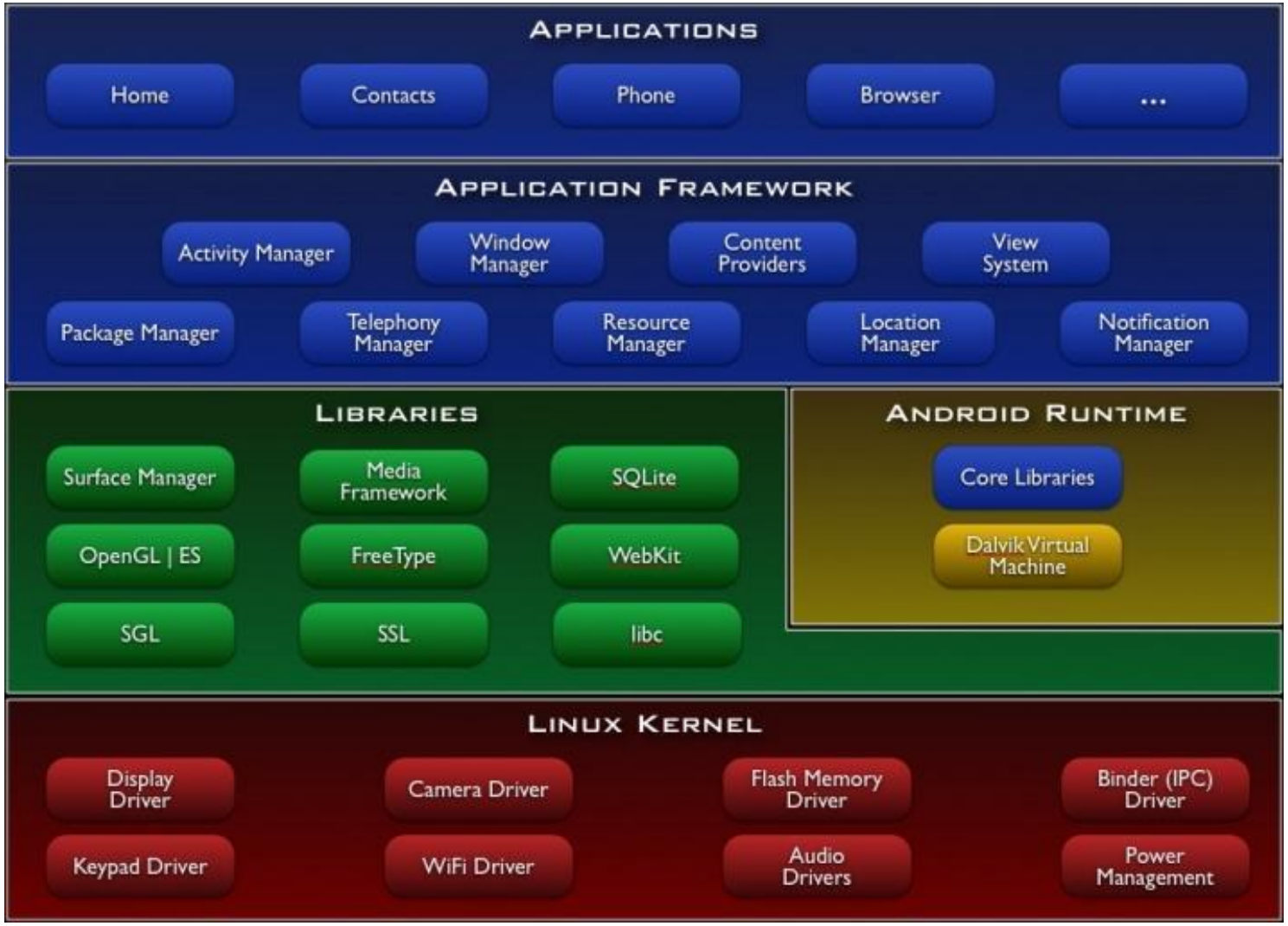
Лекция №4.
по курсу «Жизненный цикл активностей»

Москва 2019

Архитектура Android



Архитектура Android

Уровень абстракции между аппаратным обеспечением и программным стеком:

- В основе лежит ядро ОС Linux (несколько урезанное)
- Обеспечивает функционирование системы;
- Отвечает за безопасность;
- Управляет памятью, энергосистемой и процессами;
- Предоставляет сетевой стек и модель драйверов.

Набор библиотек

Обеспечивает важнейший базовый функционал для приложений:

- Алгоритмы для вышележащих уровней
- Поддержка файловых форматов
- Кодирование и декодирование информации
- Отрисовка графики и т. д.

Библиотеки

Реализованы на C/C++ и скомпилированы под конкретное аппаратное обеспечение устройства, вместе с которым они поставляются:

- **Surface Manager**
- **Media Framework**
- **SQLite**
- **3D библиотеки**
- **LibWebCore**
- **SGL (Skia Graphics Engine)**
- **SSL**

Среда выполнения Android Runtime

- Библиотеки ядра, обеспечивающие большую часть низкоуровневой функциональности, доступной библиотекам ядра языка Java
- Виртуальная машина Dalvik, позволяющая запускать приложения

Application Framework – уровень каркаса приложений

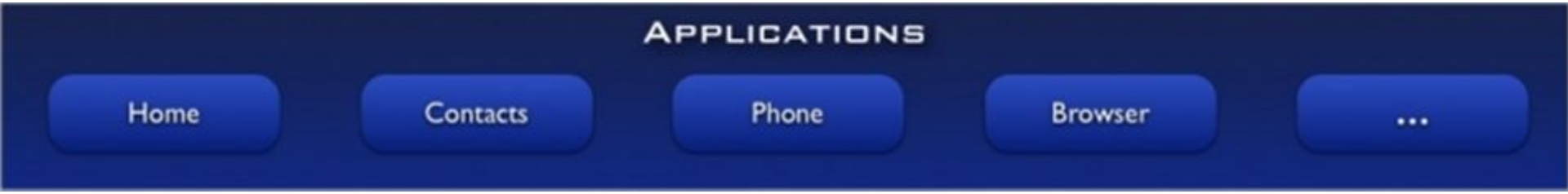
- Обеспечивает разработчикам доступ к API, предоставляемым компонентами системы уровня библиотек
- Любому приложению предоставляются уже реализованные возможности других приложений, к которым разрешено получать доступ

Application Framework

- Богатый и расширяемый набор представлений (**Views**)
- Контент-провайдеры (**Content Providers**)
- Менеджер ресурсов (**Resource Manager**)
- Менеджер оповещений (**Notification Manager**)
- Менеджер действий (**Activity Manager**)
- Менеджер местоположения (**Location Manager**)

Applications – уровень приложений

- Набор базовых приложений, который предустановлен на ОС Android. Например, браузер, почтовый клиент, программа для отправки SMS, карты, календарь, менеджер контактов и др.

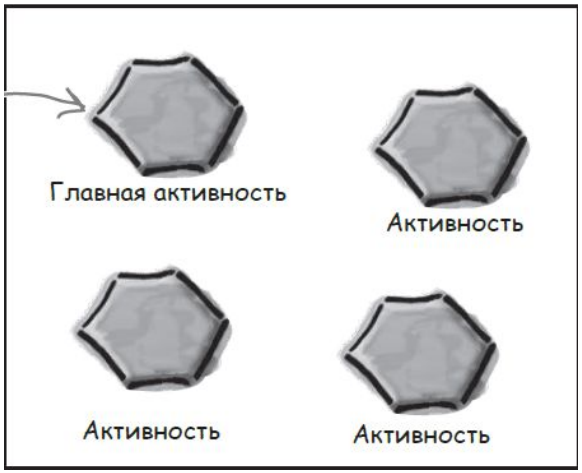


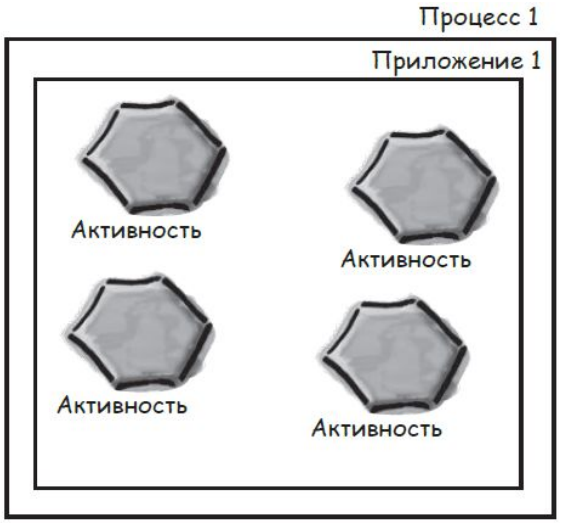
Виды приложений

1. Приложения переднего плана
2. Фоновые приложения
3. Смешанные приложения
4. Виджеты

Приложения под Android разрабатываются на языке программирования Java, компилируется в файл с расширением .apk, после этот файл используется для установки приложения на устройства, работающие под управлением Android.

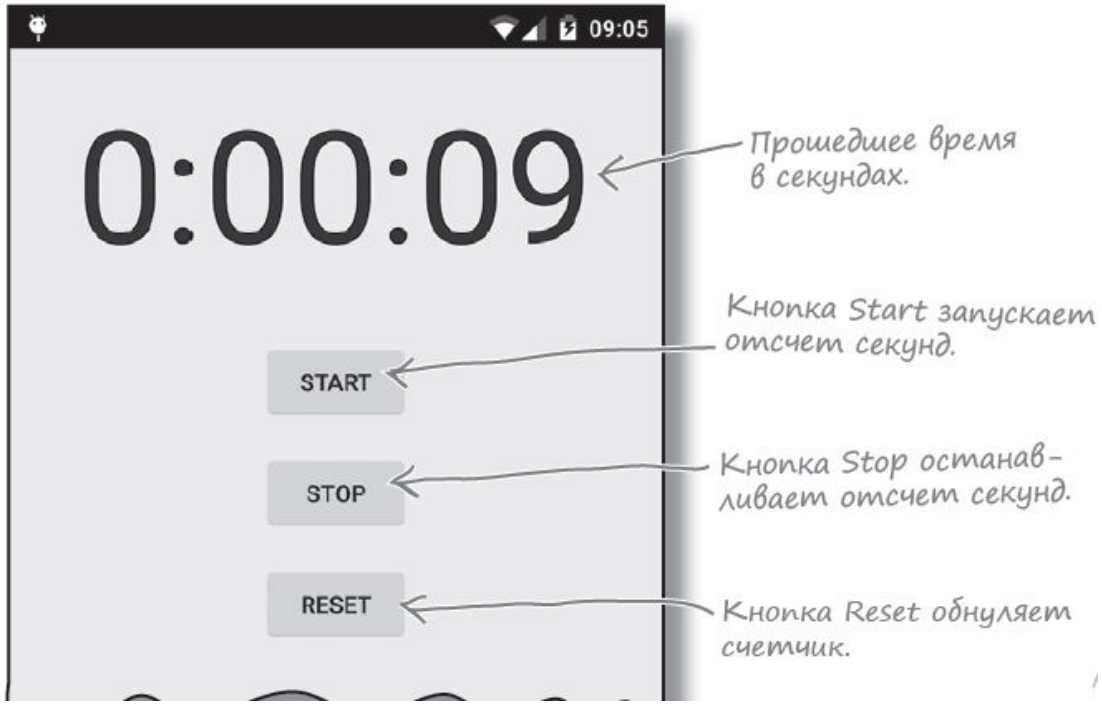
Приложение состоит из активностей





Каждое приложение выполняется в отдельном процессе. Непосредственно перед запуском активности Android проверяет, существует ли процесс для этого приложения. При запуске активности Android вызывает ее метод onCreate().

Секундомер


09:05
0:00:09
START
STOP
RESET
Прошедшее время в секундах.
Кнопка Start запускает отсчет секунд.
Кнопка Stop останавливает отсчет секунд.
Кнопка Reset обнуляет счетчик.

Объекты Handler

В Java-программах, не предназначенных для Android, такие задачи выполняются с использованием фоновых потоков. В мире Android возникает проблема — только главный программный поток Android может обновлять пользовательский интерфейс, а любые попытки такого рода из других потоков порождают исключение `CalledFromWrongThreadException`.

Handler — класс Android, который может использоваться для планирования выполнения кода в некоторый момент в будущем. Также класс может использоваться для передачи кода, который должен выполняться в другом программном потоке. В нашем примере Handler будет использоваться для планирования выполнения кода секундомера каждую секунду.
Чтобы использовать класс Handler, упакуйте код, который нужно запланировать, в объект Runnable, после чего используйте методы post() и postDelayed() класса Handler для определения того, когда должен выполняться код. Давайте поближе познакомимся с этими методами.

Объекты Handler

Метод post()

Метод post() передает код, который должен быть выполнен как можно скорее (обычно почти немедленно). Метод post() вызывается с одним параметром — объектом типа Runnable. Объект Runnable в мире Android, как и в традиционном языке Java, представляет выполняемое задание. Код, который требуется выполнить, помещается в метод run() объекта Runnable, а объект Handler позаботится о том, чтобы код был выполнен как можно быстрее. Вызов метода выглядит так:

```
final Handler handler = new Handler();
handler.post(Runnable);
```

Метод postDelayed()

Метод postDelayed() работает почти так же, как post(), но выполнение кода планируется на некоторый момент в будущем. Метод postDelayed() получает два параметра: Runnable и long. Объект Runnable содержит выполняемый код в методе run(), а long задает задержку выполнения кода в миллисекундах. Код выполняется при первой возможности после истечения задержки. Вызов метода выглядит так:

Объекты Handler

final Handler handler = new Handler();
handler.postDelayed(Runnable, long);
Чтобы обновить секундомер, мы будем многократно планировать выполнение кода с использованием Handler; при этом каждый раз будет назначаться задержка продолжительностью 1000 миллисекунд. Каждое выполнение кода сопровождается увеличением переменной seconds и обновлением надписи.
Полный код метода runTimer():

```
private void runTimer() {
final TextView timeView = (TextView)findViewById(R.id.time_view);
final Handler handler = new Handler();
handler.post(new Runnable() {
```

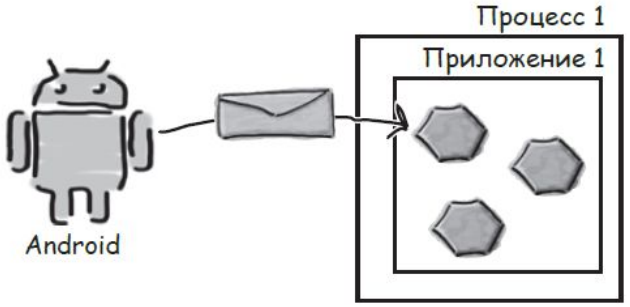
Вызов метода post() с передачей нового объекта Runnable. Метод post() обеспечивает выполнение без задержки, так что код в Runnable будет
выполнен практически немедленно.

```
@Override
public void run() {
```

Запуск приложения

```
int hours = seconds/3600;
int minutes = (seconds%3600)/60;
int secs = seconds%60;
```

1. Пользователь В файле AndroidManifest.xml приложения указано, какая активность должна использоваться как стартовая.
2. Android проверяет, существует ли процесс для этого приложения, и если не существует — создает новый процесс
3. Вызывается метод onCreate() активности
4. При завершении работы onCreate() м
   устройстве

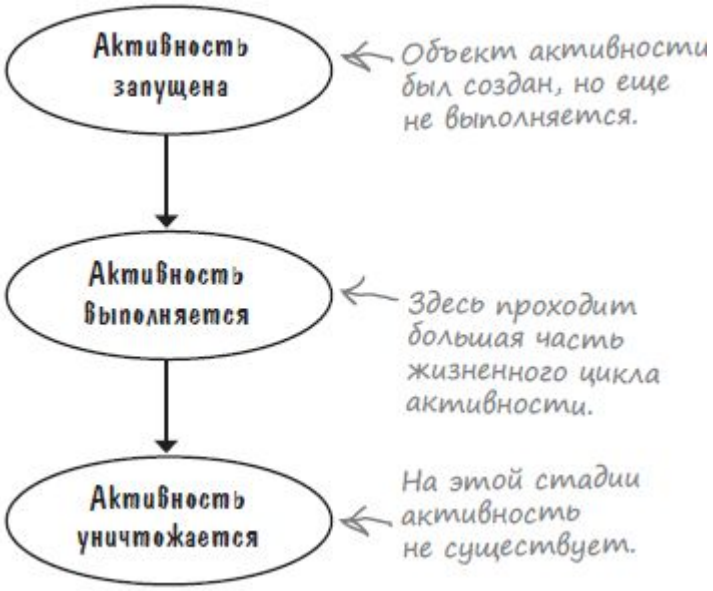


Поворот экрана

1. Пользователь запускает приложение и щелкает на кнопке Start, чтобы секундомер заработал.
2 Пользователь поворачивает устройство
3. Метод onCreate() выполняется заново, и вызывается метод runTimer(). Так как активность была создана заново, переменным seconds и running возвращаются значения по умолчанию (running = false)

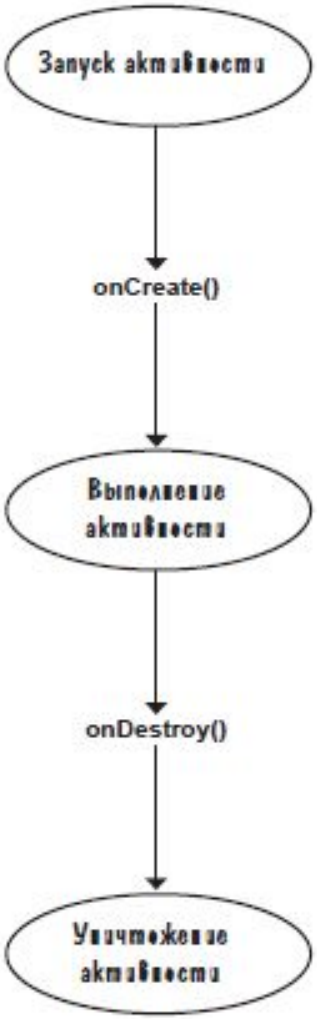
При изменении конфигурации устройства все компоненты приложения, отображающие пользовательский интерфейс, должны быть обновлены в соответствии с новой конфигурацией. Если повернуть устройство, Android замечает, что ориентация и размеры экрана изменились, и интерпретирует этот факт как изменение конфигурации устройства. Текущая активность уничтожается и создается заново, чтобы приложение могло выбрать ресурсы, соответствующие новой конфигурации

Поворот экрана



Активность выполняется, когда она находитсяна переднем плане на экране.
Метод onCreate() вызывается при создании активности; именно здесь происходит основная настройка активности.
Метод onDestroy() вызывается непосредственно перед уничтожением активности

Поворот экрана



1. **Запуск активности.**
Android создает объект активности и выполняет его конструктор.

2. **Метод onCreate() выполняется непосредственно после запуска активности.**
В методе `onCreate()` должен размещаться весь код инициализации, так как этот метод всегда вызывается после запуска активности и до начала ее выполнения.

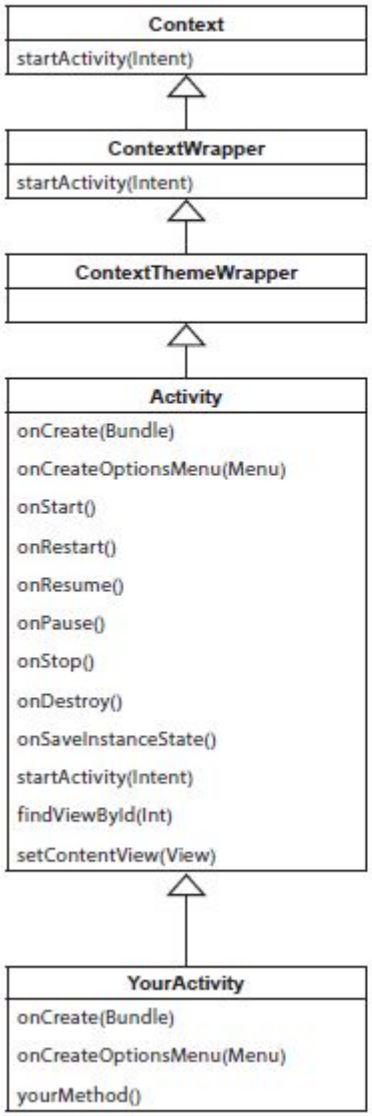
3. **Во время выполнения активность находится на переднем плане, а пользователь может взаимодействовать с ней.**
В этой фазе проходит большая часть жизненного цикла активности.

4. **Метод onDestroy() вызывается непосредственно перед уничтожением активности.**
Метод `onDestroy()` позволяет выполнить любые необходимые завершающие действия — например, освободить ресурсы.

5. **После выполнения метода onDestroy() активность уничтожается.**
Активность перестает существовать.

Наследование активности

Как было показано ранее, ваша активность расширяет класс `android.app.Activity`. Именно этот класс предоставляет активности доступ к методам жизненного цикла Android:



Абстрактный класс Context
(android.content.Context)

Интерфейс к глобальной информации об окружении приложения; открывает доступ к ресурсам приложения классам и операциям уровня приложения.

Класс ContextWrapper
(android.content.ContextWrapper)

Промежуточная реализация для Context.

Класс ContextThemeWrapper
(android.view.ContextThemeWrapper)

ContextThemeWrapper позволяет изменить тему оформления того, что содержится в ContextWrapper.

Класс Activity
(android.app.Activity)

Класс Activity реализует версии по умолчанию для методов жизненного цикла. Он также определяет такие методы, как findViewById(Int) и setContentView(View).

Класс YourActivity
(com.hfad.foo)

Большая часть поведения вашей активности обслуживается методами суперкласса. Остается лишь переопределить те методы, которые нужны вам.

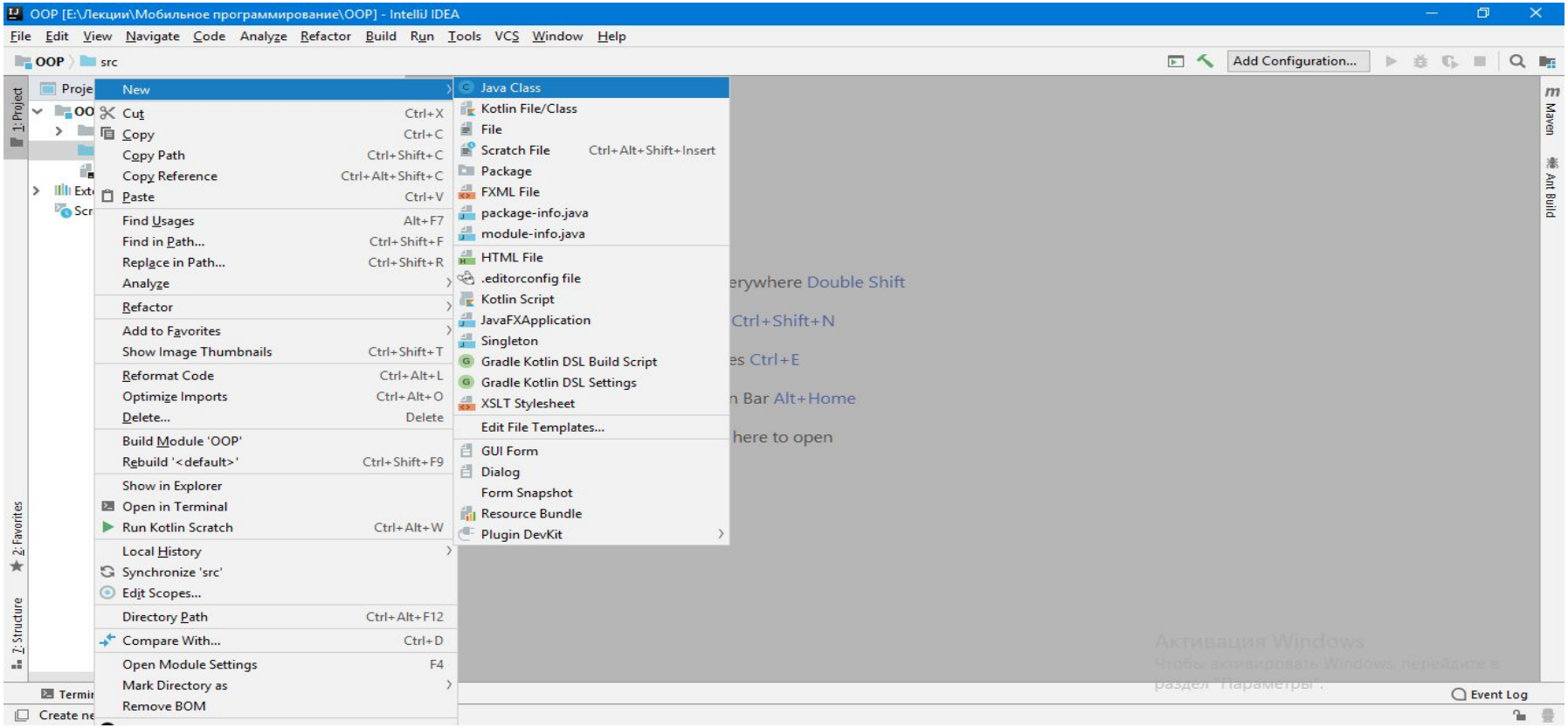
Классы

- *Классы* позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что позволяет абстрагироваться от деталей реализации.
- Данные и операции над ними образуют определенную сущность, и они не разносятся по всей программе, как нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- *Инкапсуляция* позволяет привнести свойство *модульности*, что облегчает распараллеливание выполнения задачи между несколькими исполнителями и обновление версий отдельных компонентов.

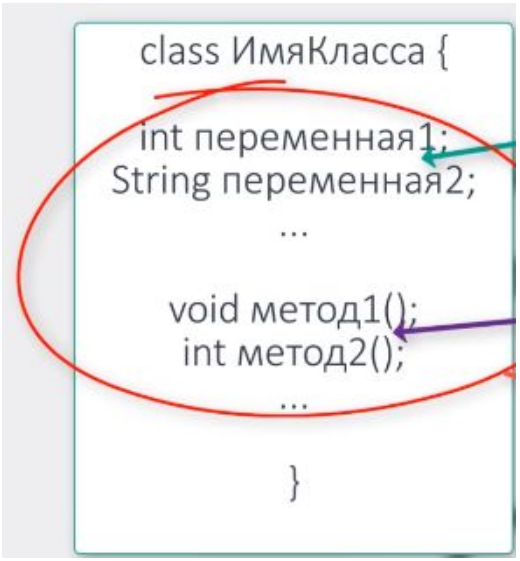
Полиморфизм оказывается полезным преимущественно в следующих ситуациях.

- Обработка разнородных структур данных. Программы могут работать, не различая вида *объектов*, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.
- Изменение *поведения* во время исполнения. На этапе исполнения один *объект* может быть заменен другим, что позволяет легко, без изменения кода, адаптировать алгоритм в зависимости от того, какой используется *объект*.
- Реализация работы с наследниками. Алгоритмы можно обобщить настолько, что они уже смогут работать более чем с одним видом *объектов*.
- Создание "каркаса" (framework). Независимые от приложения части предметной области могут быть реализованы в виде набора универсальных *классов*, или каркаса (framework), и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Классы


OOP [E:\Лекции\Мобильное программирование\OOP] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
OOP src
Add Configuration...
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Shift+C
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Find Usages Alt+F7
Find in Path... Ctrl+Shift+F
Replace in Path... Ctrl+Shift+R
Analyze
Refactor
Add to Favorites
Show Image Thumbnails Ctrl+Shift+T
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Build Module 'OOP'
Rebuild '<default>' Ctrl+Shift+F9
Show in Explorer
Open in Terminal
Run Kotlin Scratch Ctrl+Alt+W
Local History
Synchronize 'src'
Edit Scopes...
Directory Path Ctrl+Alt+F12
Compare With... Ctrl+D
Open Module Settings F4
Mark Directory as
Remove BOM
Java Class
Kotlin File/Class
File
Scratch File Ctrl+Alt+Shift+Insert
Package
FXML File
package-info.java
module-info.java
HTML File
.editorconfig file
Kotlin Script
JavaFXApplication
Singleton
Gradle Kotlin DSL Build Script
Gradle Kotlin DSL Settings
XSLT Stylesheet
Edit File Templates...
GUI Form
Dialog
Form Snapshot
Resource Bundle
Plugin DevKit
erywhere Double Shift
Ctrl+Shift+N
es Ctrl+E
n Bar Alt+Home
here to open
Maven
Ant Build
1: Project
2: Favorites
7: Structure
Event Log

Классы



Main.java | Box.java

```
1   public class Box {
2       double width;
3       double height;
4       double length;
5   }
```

```
public class Main {
    public static void main(String[] args) {
        Box myBox = new Box();
        myBox.height = 55;
        Box box4 = myBox;

    }
}
```

void - не возвращает никаких данных

тип имяМетода(*список_параметров*) {
(int, boolean, Box...)
тело_метода
...
}

Конструктор

```
public class Box {
    double width;
    double height;
    double length;

    Box() {
        width = 10;
        height = 10;
        length = 10;
    }
```

Модификаторы доступа

По умолчанию все модификаторы доступа public.
'Generate...' and then 'Getter and Setter'.
Default – это доступ по умолчанию , доступность внутри пакета

```
public class Box {
    private double width;
    double height;
    double length;

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }
}
```

Пакеты

Элементами *пакета* являются содержащиеся в нем классы и интерфейсы, а также вложенные *пакеты*. Чтобы получить *составное имя пакета*, необходимо к полному имени *пакета*, в котором он располагается, добавить точку, а затем его собственное *простое имя*. Например, *составное имя* основного *пакета* языка Java – `java.lang` (то есть *простое имя* этого пакета `lang`, и он находится в объемлющем пакете `java`). Внутри него есть вложенный пакет, предназначенный для *типов* технологии reflection, которая упоминалась в предыдущих главах. Простое название пакета `reflect`, а значит, составное – `java.lang.reflect`.

Простое имя классов и интерфейсов дается при объявлении, например, `Object`, `String`, `Point`. Чтобы получить *составное имя* таких *типов*, надо к *составному имени пакета*, в котором находится *тип*, через точку добавить *простое имя типа*. Например, `java.lang.Object`, `java.lang.reflect.Method` или `com.myfirm.MainClass`. Смысл последнего выражения таков: сначала идет обращение к *пакету* com, затем к его *элементу* – вложенному *пакету* `myfirm`, а затем к *элементу пакета* `myfirm` – классу `MainClass`. Здесь `com.myfirm` – *составное имя пакета*, где лежит класс `MainClass`, а `MainClass` — *простое имя*. Составляем их и разделяем точкой – получается полное имя класса `com.myfirm.MainClass`.

Пакеты

Программа на *Java* представляет собой набор *пакетов* (packages). Каждый *пакет* может включать вложенные *пакеты*, то есть они образуют иерархическую систему.

Кроме того, *пакеты* могут содержать классы и интерфейсы и таким образом группируют *типы*. Это необходимо сразу для нескольких целей. Во-первых, чисто физически невозможно работать с большим количеством классов, если они "свалены в кучу". Во-вторых, модульная *декомпозиция* облегчает проектирование системы. К тому же, как будет показано ниже, существует специальный уровень доступа, позволяющий *типам* из одного *пакета* более тесно взаимодействовать друг с другом, чем с классами из других *пакетов*. Таким образом, с помощью *пакетов* производится логическая группировка *типов*. Из *ООП* известно, что большая *связность* системы, то есть среднее количество классов, с которыми взаимодействует каждый *класс*, заметно усложняет развитие и поддержку такой системы. Используя *пакеты*, гораздо проще организовать эффективное взаимодействие подсистем друг с другом.

Наконец, каждый *пакет* имеет свое *пространство* имен, что позволяет создавать одноименные классы в различных *пакетах*. Таким образом, разработчикам не приходится тратить время на разрешение конфликта имен.

Пакеты

Исходный код располагается в файлах с расширением .java, а бинарный – с расширением .class

исходный код классов

```
space.sunsystem.Moon
space.sunsystem.Sun
space.sunsystem.Test
```

хранится в файлах

```
space\sunsystem\Moon.java
space\sunsystem\Sun.java
space\sunsystem\Test.java
```

Используйте == для сравнения два примитива, или посмотреть,
если два ссылки относятся к тот же объект.
Используйте equals чтобы увидеть если два объекта равны.

```
Integer.parseInt("3")

for (int cell : locationCells)
```

Не принято, чтобы классы находились не внутри пакетов

Пакеты

Package – указывает в каком пакете находится данный класс

```
package Main;


import box.Box;

public class Main {
    public static void main(String[] args) {
        Box myBox = new Box();


        Box box4 = myBox;

    }
}
```

Отношения между классами

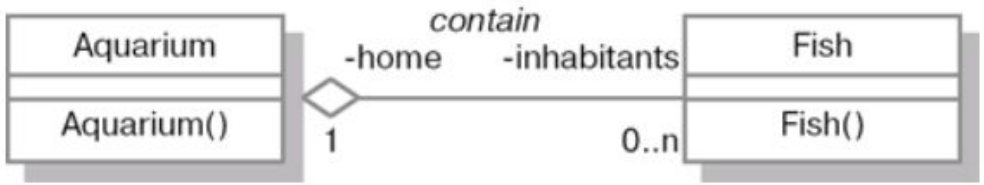
агрегация (Aggregation);
ассоциация (Association);
наследование (Inheritance);
метаклассы (Metaclass).

Приняты также обозначения:

- " `1..n` " - от единицы до бесконечности;
- " `0` " - ноль;
- " `1` " - один;
- " `n` " - фиксированное количество;
- " `0..1` " - ноль или один.

Агрегация

Отношение между *классами* типа "содержит" (contain) или "состоит из" называется агрегацией, или включением. Например, если аквариум наполнен водой и в нем плавают рыбки, то можно сказать, что аквариум агрегирует в себе воду и рыбок.




```
// определение класса Fish
public class Fish {
   // определения поля home
   // (ссылка на объект Aquarium)
   private Aquarium home;

   public Fish() {
   }
}
```

```
// определение класса Aquarium
public class Aquarium {
   // определения поля inhabitants
   // (массив ссылок на объекты Fish)
   private Fish inhabitants[];
   public Aquarium() {
   }
}
```

Отношения между классами

Ассоциация

Если *объекты* одного *класса* ссылаются на один или более *объектов* другого *класса*, но ни в ту, ни в другую сторону отношение между *объектами* не носит характера "владения", или контейнеризации, такое отношение называют ***ассоциацией*** (association). Отношение *ассоциации* изображается так же, как и отношение агрегации, но линия, связывающая *классы*,- простая, без ромбика.

В качестве примера можно рассмотреть программиста и его компьютер. Между этими двумя *объектами* нет агрегации, но существует четкая взаимосвязь. Так, всегда можно установить, за какими компьютерами работает какой-либо программист, а также какие люди пользуются отдельно взятым компьютером. В рассмотренном примере имеет место *ассоциация* "многие-ко-многим".

В данном случае между экземплярами *классов* `Programmer` и `Computer` в обе стороны используется отношение " `0..n` ", т.к. программист, в принципе, может не работать с компьютером (если он теоретик или на пенсии). В свою очередь, компьютер может никем не использоваться (если он новый и еще не установлен).

```
public class Programmer {
   private Computer computers[];
   public Programmer() {
   }
}
public class Computer {
   private Programmer programmers[];
   public Computer() {
   }
}
```