Модуль 2. Объектно-ориентированное программирование

Тема 2.6. Наследование. Намерения

4 часа

Оглавление

Тема 2.6. Наследование. Намерения

2.6.1. Понятие наследования

Одно из основных понятий ООП – это наследование. Наследование – это естественная абстракция для большинства людей, потому что с ним (наследованием) мы сталкиваемся в жизни, и возможность перенести принципы наследования в разработку программного обеспечения позволяет значительно упростить декомпозицию сложных систем и создавать такой уровень абстракции, который будет доступен человеку.

Человеческий мозг имеет границы своих возможностей, и крайне сложно удерживать в голове человека видение сложной системы целиком. Психологи отмечают, что мозг человека может держать в кратковременной памяти 7±2 элементов. Отсюда следует, что при разработке программы программист может держать в голове примерно 7 элементов, с которыми он сможет одновременно оперировать.

Отсюда появляется необходимость в синтезировании из нескольких мелких объектов одного более общего и крупного, или анализ массы объектов с целью выделить некоторые общие признаки.

Именно анализ группы объектов и выделение их общих свойств позволяет разработать некоторый обобщающий суперкласс (надкласс, базовый класс), частные реализации которого уже дадут необходимую функциональность.

Рассмотрим пример из жизни. Существуют различные предметы быта, на которых сидят люди: стул, кресло, табуретка, диван и так далее. Объединим их по функции, для которой они предназначены (предметы быта для сидения), и выделим их в единый суперкласс “Sittings”.

Sittings – это класс, который имеет общие свойства и методы для всех сидячих мест: число сидячих мест в одном объекте класса, его габариты, вес и так далее. Таким образом мы в Sittings включили только те свойства, которые относятся ко всем без исключения предметам быта для сидения. И мы не стали включать в него какие-то особенности, которые относятся к конкретным видам (например: количество ножек у табуретки).

Теперь перечислим те классы, которые могут наследоваться от класса Sittings: ArmChair (кресло), Chair (стул), Stool (табуретка), Sofa (диван). Функционально эти три объекта предназначены для сидения, следовательно они являются наследниками класса Sittings. В связи с тем, что они наследуются от Sittings в них помимо их собственных свойств включаются и такие свойства как число сидячих мест, габариты, вес и так далее.

Но помимо общих свойств, которые “пришли” из суперкласса, в наследниках есть так же и их индивидуальные свойства. У табуретки, например, есть свойство – число ножек (3, 4 или даже 5).

Таким образом мы рассмотрели понятия, которые можно описать в следующем определении:

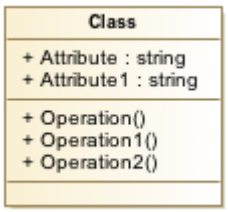
Наследование – принцип объектно-ориентированного программирования, в рамках которого возможно описание новых классов на основе уже существующих классов таким образом, что

свойства (в Java - поля) и методы родителя будут присутствовать и в наследниках. Родителей в данном случае принято называть суперклассом.

2.6.2. Графическое описание структуры классов в UML

Текстовое воспринятие описания классов крайне тяжело воспринимать и поэтому на практике для описания их структуры (в частности такого отношения как наследование) используется специальный графический язык описания UML (Unified Modelling Language). В рамках данного учебника нет задачи научить полноценно использовать диаграммы классов из стандарта UML, но базовые понятия будут полезны для дальнейшего разбора примеров и постановки задач.

Основным элементом диаграммы классов является блок, представляющий собой класс. Ниже представлен блок класса с двумя свойствами и тремя методами.



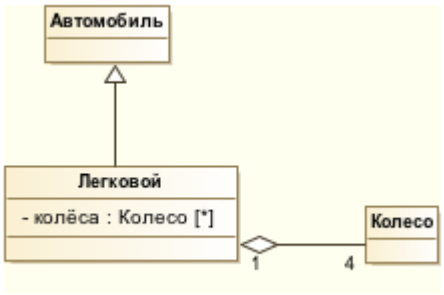
Блок класса представляет из себя прямоугольник разбитый на три части: название (в примере выше – Class), свойства класса (Attribute, Attribute1) и методы (Operation(), Operation1(), Operation2()).

Как видите на картинке, каждое свойство класса имеет имя и тип (указывается после двоеточия). Также каждый метод и свойство имеют знак области видимости:

- + – public (публичный)
- - – private (приватный)
- # – protected (защищенный)

Публичные и приватные методы и свойства классов мы разобрали ранее, а защищенные рассмотрим чуть ниже в этой главе.

Классы не единственное, что можно отобразить на диаграмме классов. Другим важным элементом данной диаграммы является возможность показать на ней отношения между этими классами. Рассмотрим два основных вида отношений между классами: наследование и агрегирование.

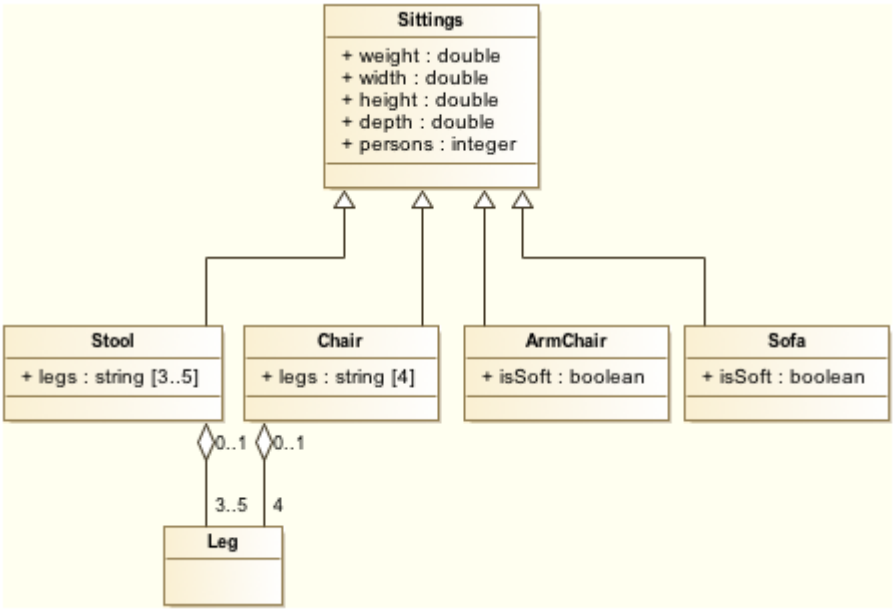


На данном рисунке есть суперкласс Автомобиль и его наследник Легковой. Таким образом, чтобы указать отношение наследования необходимо провести линию от наследника к суперклассу и нарисовать белый треугольник в конце.

Также на диаграмме показано, что у одного легкового автомобиля есть четыре колеса. Поэтому от Колеса к Легковой проводится линия и в конце рисуется белый ромб. Очень часто начинающие программисты путаются в отношениях наследования и агрегирования, но принципиально это два разных отношения:

- **наследование** переносит свойства и методы из суперкласса, то есть по-сути класс-наследник расширяется (дополняется) свойствами и методами суперкласса.
- при **агрегировании** происходит включение класса-агрегата как отдельной части класса-агрегатора. Здесь не случайно используется слово “агрегат”, которое все понимают как “часть”, “деталь” или “кусок”.

Разберём пример со стульями и нарисуем диаграмму классов. Только на этот раз создадим ещё один класс – Leg (ножка стула).



Классы Stool, Chair, ArmChair, Sofa являются наследниками класса Sittings. Это означает, что у них присутствует все свойства, которые были в Sittings хоть они в них явно не указаны. Класс Leg является агрегатом для Stool и Chair, такой прием полезен, потому что и табуретка и стул имеют ножки.

2.6.3. Наследование на языке Java

Для описания наследования на языке Java используется ключевое слово extends, которое на русский переводится как “расширяет”.

Например, наследование выглядит так:

```
public class Chair extends Sittings {
}
```

В данном примере класс Chair наследуется от класса Sittings. Обратите внимание, что фраза “class Chair extends Sittings” имеет вполне понятный смысл: “класс Chair расширяет Sittings”.

Упражнение 2.6.1

Реализуйте набор классов из примера в диаграмме классов, который был рассмотрен в разделе 2.6.2.

2.6.4. Защищенные члены класса

До этого мы изучили, что ключевое слово private означает, что свойства и методы класса будут недоступны вне класса (являются приватными), а public означает, что свойства и методы класса доступны извне (являются публичными). Использование слова private бывает полезно тогда когда разработчик класса хочет скрыть некоторую реализацию класса от его пользователей.

Но пользователями класса могут быть как другие несвязанные напрямую классы, так и классы-наследники. И слово private точно также скрывает методы и свойства от них. Поскольку класс-наследник орудует, можно сказать, во внутренностях суперкласса, то логично было бы дать ему больше прав чем обычным пользователям класса.

Для этих целей служит слово protected, которое говорит, что свойство или метод с этим уровнем доступа будет недоступен извне, но будет доступен наследникам класса.

Рассмотрим следующий пример:

```
public class MyProgram {
        public static class A {
                public int a = 1;
                private int b = 2;
                protected int c = 3;
        }

        public static class B extends A {
                public B() {
                a = 11; // допустимо, так как a - public
                b = 22; // недопустимо, так как b - private
                c = 33; // допустимо, так как c - protected
                }
        }

        public static void main(String args[]) {
                B bObj = new B();
                bObj.a = 111; // допустимо, так как a - public
                bObj.b = 222; // недопустимо, так как b - private
                bObj.c = 333; // недопустимо, так как c - protected
        }
}
```

2.6.5. Понятие интерфейса

В разработке программного обеспечения понятие интерфейса весьма широкое и зачастую имеет достаточно разный смысл в зависимости от контекста. Например, широкоупотребимая фраза “интерфейс программы” подразумевает “графический интерфейс пользователя программы”, а фраза “интерфейс библиотеки” подразумевает “прикладной программный интерфейс библиотеки”. Это абсолютно разные интерфейсы, но сложностей в восприятии обычно не возникает.

В данной главе мы рассмотрим интерфейсы с точки зрения объектно-ориентированного программирования. Начнем с общего понятия интерфейса.

Интерфейс – это совокупность возможностей взаимодействия двух систем.

Таким образом интерфейс является способом коммуникации между двумя субъектами. Интерфейс обычно подразумевает некоторый протокол общения, то есть формальный подход к его описанию. Даже общение через телефон всегда начинается со слова “ало” (или его альтернативы), что предусмотрено протоколом общения по телефону.

Отсюда попытаемся дать определение интерфейса в объектно-ориентированном программировании.

Интерфейс – это формальное описание методов класса без реализации, для обеспечения коммуникации между объектами этого класса и другими классов.

То есть интерфейс – это описание того, какие методы должны присутствовать в классах, которые намереваются реализовывать этот интерфейс. Если класс C реализует интерфейс I, то другие классы смогут пользоваться некоторыми возможностями класса C через интерфейс I. Например, в Java существует интерфейс Comparable, который заставляет все классы, которые реализуют этот интерфейс иметь внутри себя метод compareTo. Благодаря этому обязательству можно смело пользоваться методом compareTo, который сравнивает экземпляры этого класса.

2.6.6. Интерфейсы на языке Java

Упрощенно можно считать, что интерфейсы – это классы, в которых методы не имеют реализации. Отсюда несколько следствий:

- нельзя создать экземпляр интерфейса;
- область видимости полей интерфейса по-умолчанию будет public.

Рассмотрим пример интерфейса, который обязывает все классы, которые его реализуют возвращать и устанавливать некоторое int-овое значение:

```
public interface IntValuable {
        void setValue(int value);

        int getValue();
}
```

Этот интерфейс обязывает классы реализовывать два метода: setValue и getValue. Попробуем написать такой класс:

```
public class MyProgram implements IntValuable {
        private int value = 0;

        public void setValue(int value) {
                this.value = value;
        }

        public void getValue() {
                return value;
        }
}
```

Обратите внимание, что если при наследовании мы использовали слово “extends” (расширяет), то в данном случае мы используем слово “implements” (реализует). То есть фраза “class MyProgram implements IntValuable” означает “класс MyProgram реализует IntValuable”. А раз класс MyProgram реализует интерфейс IntValuable, то в этом классе необходимо написать методы этого интерфейса: setValue и getValue.

2.6.7. Ключевое слово final

Ключевое слово final имеет различные интерпретации в зависимости от того в каком месте программы оно используется. Но суть этого слова одна – запрет на изменение.

Рассмотрим следующий пример:

```
public class MyProgram {
        public final int MAGIC_CONST = 100;
}
```

Здесь значение MAGIC_CONST нельзя изменить. Таким образом, если написать слово final возле примитивной переменной, то она является константой.

А что будет, если использовать ссылочный тип?

```
import java.util.ArrayList;

public class MyProgram {
        public final ArrayList<Integer> list;

        public MyProgram() {
                list = new ArrayList<Integer>();
        }
}
```

Данный код не будет вызывать ошибки, но только до тех пор пока не присвоить новое значение переменной list. То есть слово final для ссылочных типов запрещает изменение указателя, а не значения по указателю. Также стоит отметить, что присваиваение допустимо в любом месте программы, но только один раз.

Другие способы использования ключевого слова final касаются наследования. И первый из них – это слово final возле метода.

```
public class MyProgram {
        public final int helloWorld() {
        System.out.println(“Hello world!!!”);
        }
}
```

Слово final возле метода запрещает переопределение этого метода в классах-наследниках. То есть если создать класс-наследник от MyProgram и написать в нём следующий код, то произойдет ошибка:

```
public class ExtendedMyProgram extends MyProgram {
        public int helloWorld() {
                System.out.println("Extended hello world!!!");
        }
}
```

Следующим примером когда можно написать слово final – возле класса.

```
public final class MyProgram {
}
```

Финальный класс – это класс, который не может быть суперклассом, то есть запрещается писать для него наследников.

2.6.8. Контекст в Android

С данным понятием мы с вами уже сталкивались. Например, когда хотели вывести всплывающее сообщение:

```
Toast.makeText(this,”qwert”,Toast.LENGTH_SHORT).show();
```

Класс android.context.Context представляет из себя интерфейс для доступа к глобальной информации об окружении приложения.

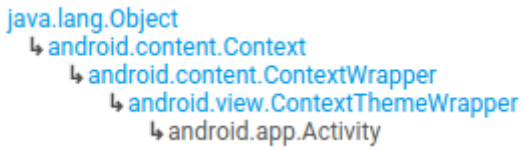
Это абстрактный класс, реализация которого обеспечивается системой Android.

Context позволяет получить доступ к специфичным для данного приложения ресурсам и классам, а также для вызова операций на уровне приложения, таких, как запуск Activity, отправка широковещательных сообщений, получение намерений (Intent) и прочее.

Данный класс также является базовым для классов Activity, Application и Service.

Получить доступ контексту можно с помощью методов getApplicationContext, getContext, getBaseContext, а также просто с помощью свойства this (изнутри Активности или Сервиса).

Именно Context является родительским классом для Activity, Service, что является отличным примером применения наследования в Android-разработке.



Объект Context часто применяется в программировании под Android.

1. Динамическая развертка пользовательского интерфейса, создание новых объектов, адаптеров и т.д.

```
TextView myTextView = new TextView(this);
ListAdapter adapter = new SimpleCursorAdapter(getApplicationContext(),...);
```

2. Доступ к стандартным глобальным ресурсам.

```
//Доступ из класса Activity -- наследника Context
getSystemService(LAYOUT_INFLATER_SERVICE);
//Доступ с использованием Контекста Приложения
SharedPreferences prefs = getApplicationContext().
getSharedPreferences("PREFS", MODE_PRIVATE);
```

В Android существует несколько видов Контекста, отличающихся длительностью жизненного цикла, а также возможностями областью применимости.

Неоправданное использование Контекста с более длительным жизненным циклом может привести к утечке ресурсов в приложении.

2.6.9. Намерения (Intents) Android

Намерения (Intent) в Android используются в качестве механизма передачи сообщений, который может работать как внутри одного приложения, так и между приложениями. Намерения могут применяться для:

- объявления о желании (необходимости) вашего приложения запуска какой-то Активности или Сервиса для выполнения определенных действий;
- извещения о том, что произошло какое-то событие;
- явного запуска указанного Сервиса или Активности.

Намерения бывают двух видов:

- неявные;
- явные.

Явные намерения

При использовании явного Intent указывается:

1) Контекст, предоставляющий нужную информацию об окружающей среде приложения.
2) Класс целевого компонента (будь то сервис, активность), которую нужно будет запустить.

Рассмотрим три варианта работы с явными намерениями:

1. переход из одного Activity в другой Activity

2. открытие и передача каких-либо данных в открываемую Activity
3. открытие и возврат какого-то результата при закрытии открываемого Activity

Рассмотрим эти варианты на примере приложения. Создадим приложение, разметку приводим к такому виду:


```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:paddingBottom="@dimen/activity_vertical_margin"
android:paddingLeft="@dimen/activity_horizontal_margin"
android:paddingRight="@dimen/activity_horizontal_margin"
android:paddingTop="@dimen/activity_vertical_margin"
tools:context=".MainActivity">

<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:text="Начать" />

<Button
    android:id="@+id/button2"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentEnd="true"
    android:layout_alignParentRight="true"
    android:layout_below="@+id/button"
    android:text="Об авторе" />

<Button
    android:id="@+id/button3"
```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentEnd="true"
    android:layout_alignParentRight="true"
    android:layout_below="@+id/button2"
    android:text="Держи" />

<Button
    android:id="@+id/button4"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentEnd="true"
    android:layout_alignParentRight="true"
    android:layout_below="@+id/button3"
    android:text="Отдай обратно" />

<EditText
    android:id="@+id/et"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@+id/button4" />
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/et"
    android:id="@+id/tv"/>

</RelativeLayout>
```

Первые две кнопки рассчитаны просто на открытие Activity. Поле для ввода нужна для ввода строки, передаваемой в другую Activity. TextView - для отображения полученного из дочерней Activity текста.

Добавляем 4 дочерних активности - MainActivity2, MainActivity3, ToInfActivity, ComeBackActivity. Для каждой добавленной Activity в AndroidManifest.xml прописываем:

```
<activity
       android:name=".НазваниеАктивности"
       android:label="Текст, который будет отображаться в заголовке Активности" >
</activity>
```

Реализуем первую кнопку. Вторая - на самостоятельную работу. В java-коде прописываем связывание с компонентами из разметки. Также реализуем метод обработки нажатия кнопки.

```
public void onClick(View v) {
    Intent i;
    switch (v.getId()) {
        case R.id.button:
            i = new Intent(this, MainActivity2.class);
            startActivity(i);
            break;
...
```

i - переменная Intent - представляет собой намерение. Конструктор Intent() принимает два аргумента - объект типа Context и class-объект дочернего компонента.

Стоит обратить внимание на то, как система понимает, что за компонент ему передан в качестве цели. В данном случае путеводителем для системы является файл AndroidManifest.

Например, при создании записи Activity в манифест-файле указывается имя класса. Если указать тот же класс в Intent – то система, просмотрев манифест-файл обнаружит соответствие и покажет соответствующий Activity. Но если такого компонента не окажется в манифест-файле, или тип компонента не будет соответствовать совершаемому действию, система не сможет проделать нужные действия.

Теперь рассмотрим следующий случай - передача параметра в дочернюю Активность.

Как было упомянуто ранее, Intent является основным способом передачи данных из одного компонента (активность, сервис) в другой.

В качестве дочерней возьмем следующую нами добавленную активность ToInfAct.

В качестве передатчика параметров используется намерение, у которого есть метод putExtra(); этот метод получает в качестве параметров ключ, по которому значение можно извлечь и само значение. Ключ имеет строковый тип, а значение может иметь либо примитивный тип, либо строковый тип, либо являться массивом, либо иметь объектный тип, реализующий интерфейс Parcelable или Serializable.

```
case R.id.button3:
   i = new Intent(this, ToInfoAct.class);
   String eText = et.getText().toString();
   i.putExtra("et",eText);
   startActivity(i);
   break;
```

В дочерней Activity данные извлекаются:

```
String str = getIntent().getStringExtra("et");
```

Извлечение строковых данных происходит методом `getStringExtra` по ключу.

Для остальных типов по аналогии: `getТипExtra()`.

Также через намерения можно передавать и полноценные объекты собственных классов. Однако для такой возможности класс этого объекта должен расширять интерфейс `Serializable` или `Parcelable`.

Рассмотрим теперь возврат результатов из дочерней Activity. В качестве дочерней будем использовать `ComeBackAct`.

Запуск дочерней Activity из родительской имеет вид:

```
case R.id.button4:
    i = new Intent(this, ComeBackAct.class);
    startActivityForResult(i, REQ_C);
```

```
    break;
```

Обратите внимание: для запуска дочерней Activity в метод startActivityForResult() передаются два параметра: Намерение и код запроса. Данный код запроса нужен при обработке возвращенных Activity результатов для подходящей обработки.

После выполнения нужных действий и задач дочерняя Activity завершает свою работы следующим образом:

```
Intent i = new Intent();
i.putExtra(MainActivity.NAME,et.getText().toString());
setResult(RESULT_OK, i);
finish();
```

В данном случае Намерение создается для передачи результата обратно в родительскую Activity. В качестве результата передается код результата и Намерение. Код результата нужен для обработки возвращаемых данных.

Для обработки резултьтата переопределяется метод onActivityResult().

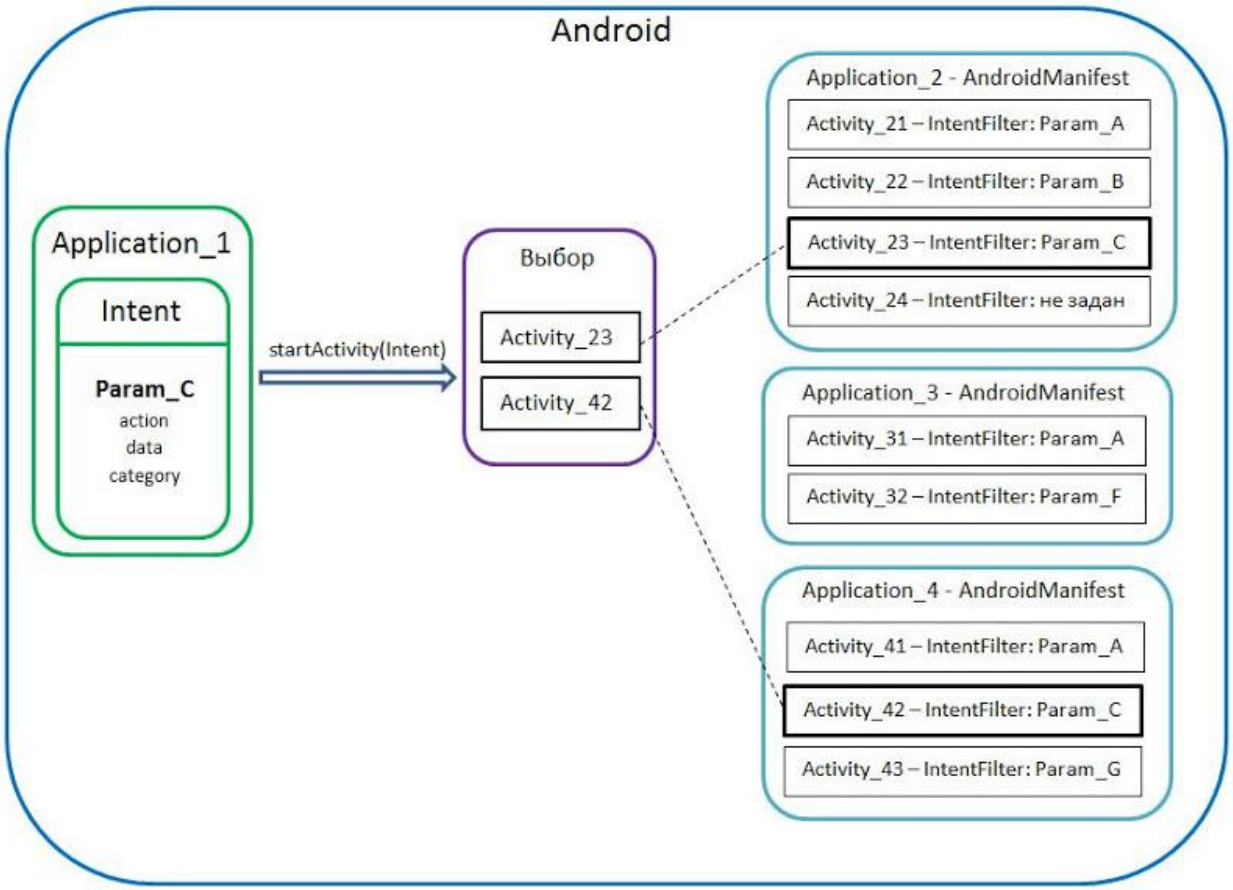
```
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        switch (resC) {
        case RESULT_OK:
            tv1.setText(d.getStringExtra(NAME));
            break;
        }
        super.onActivityResult(reqC, resC, d);
    }
```

Данный метод в качестве аргументов получает код запроса, код результата и Намерение из дочерней Activity.

Неявные намерения

Неявный Intent отличается тем, что при создании Intent не используется класс, а заполняются параметры action, data, category определенными значениями. Комбинация этих значений определяют цель, которую нужно достичь.

Например: отправка письма, открытие гиперссылки, редактирование текста, просмотр картинки, звонок по определенному номеру и т.д. В свою очередь для компонента (например, Activity) прописывается Intent Filter - это набор тех же параметров: action, data, category (но значения уже свои - зависят от того, что умеет делать Activity). И если параметры нашего Intent совпадают с условиями этого фильтра, то Activity вызывается. Но при этом поиск уже идет по всем Activity всех приложений в системе. Если находится несколько, то система предоставляет вам выбор, какой именно программой вы хотите воспользоваться. Схематично это можно изобразить так:



В Application_1 создается Intent, заполняются параметры action, data, category. Для удобства, получившийся набор параметров назовем Param_C. С помощью `startActivity` этот Intent отправляется на поиски подходящей Activity, которая сможет выполнить то, что нам нужно (т.е. то, что определено с помощью Param_C). В системе есть разные приложения, и в каждом из них несколько Activity. Для некоторых Activity определен Intent Filter (наборы Param_A, Param_B и т.д.), для некоторых нет. Метод `startActivity` сверяет набор параметров Intent и наборы параметров Intent Filter для каждой Activity. Если наборы совпадают (Param_C для обоих), то Activity считается подходящей.

Если в итоге нашлась только одна Activity – она и отображается. Если же нашлось несколько подходящих Activity, то пользователю выводится список, где он может сам выбрать какое приложение ему использовать.

Например, если в системе установлено несколько музыкальных плееров, и вы запускаете mp3, то система выведет вам список Activity, которые умеют играть музыку и попросит выбрать, какое из них использовать. А те Activity, которые умеют редактить текст, показывать картинки, звонить и т.п. будут проигнорированы.

Если для Activity не задан Intent Filter (Activity_24 на схеме), то Intent с параметрами ему никак не подойдет, и оно тоже будет проигнорировано.

Если проводить аналогии - можно сравнить Intent с ключом, а Intent Filter с замкОм, за которым сидит прекрасное Activity.

В качестве примера рассмотрим использование Google Now (распознавание речи) в простеньком приложении. Приложение будет представлять из себя одну активность с одной кнопкой и текстовым полем, куда будет выводиться распознанная речь.

По нажатию на кнопку создается неявный Intent, который и запустит активность распознавания. Приведем код layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:gravity="center"
android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingBottom="5dp"
            android:text="Задать вопрос"
            android:fontFamily="calibri"
            android:textColor="@color/white"
            android:textSize="18sp"
            android:textStyle="bold" />

        <ImageButton
            android:id="@+id/question"
            android:layout_width="55dp"
            android:layout_height="55dp"
            android:background="@drawable/btn_selected"
            android:src="@android:drawable/ic_btn_speak_now"
            android:tint="@color/white" />


        <ScrollView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content">

            <TextView
                android:id="@+id/text"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_margin="10dp"
                android:text="(после нажатия произнесите вопрос)"
                android:textAlignment="center"
                android:textColor="@color/grey"
                android:textSize="12sp" />
        </ScrollView>
</LinearLayout>
```

Исходный код java:

```
public class QuestionActivity extends BaseActivity {
```

```
	protected static final int RESULT_SPEECH = 1;
	private ImageButton btnSpeak;
	private TextView textView;

	@Override
	   public void onCreate(Bundle savedInstanceState) {
	       super.onCreate(savedInstanceState);
	       setContentView(R.layout.activity_question);
	       btnSpeak = (ImageButton) findViewById(R.id.question);
	       textView = (TextView) findViewById(R.id.text);
	       btnSpeak.setOnClickListener(v -> {
	              Intent intent = new Intent(
	                     RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
	              intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,      "en-
US");
	              try {
	               startActivityForResult(intent, RESULT_SPEECH);
	              } catch (ActivityNotFoundException a) {
	              Toast.makeText(getApplicationContext(),
	                     R.string.error_voice_recognize_not_support,
	                     Toast.LENGTH_SHORT).show();
	              }
	       });
	   }

	@Override
	protected  void  onActivityResult(int  requestCode,  int  resultCode,  Intent
data) {
	       super.onActivityResult(requestCode, resultCode, data);
	       switch (requestCode) {
	       case RESULT_SPEECH: {
	              if (resultCode == RESULT_OK && null != data) {
	                     ArrayList<String> text = data

	.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
	                     textView.setText(text.get(0));
	              }
	              break;
	       }

	       }
	}
}
```

Не забудьте добавить строчку в манифесте:

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

Задание 2.6.1

В соответствии с заданным вариантом задания разработать собственный класс, аналог которого общеупотребителен в окружающем мире или в математике и имеет хорошо знакомое поведение. Для школьника задание должно быть сформулированно учителем.

Возможные варианты заданий:

1. Реализовать класс “Билет на общественный транспорт”.
   Реализовать классы-наследники: автобусный билет, билет на метро, единый билет (на несколько видов транспорта одновременно). Должны быть реализованы методы для прохода в в автобус и метро.
2. Реализовать класс “Переводчик” для перевода из десятичной системы счисления в другую. Его наследники: в двоичную, в римскую.
3. Реализовать класс "Нота".
   Поля: перечислимый тип для ноты (до,ре, ми, фа, соль, ля, си); перечислимый тип для тональности (контроктава, большая, малая, первая, вторая третья, четвертая); перечислимый тип для знака альтерации (отсутствует, диез, бемоль). Конструктор проверяет корректность задания ноты - не может быть до-бемоль, фа-меболь, ми-диез, си-диез. Класс Нота обязательно сделать реализующем интерфейс Comparable и написать метод compareTo для сравнения нот: нота с меньшей высотой предшествует (меньше) ноты с большей высотой. Реализовать класс "Музыкальный интервал" как контейнер, содержащий две ноты. Конструктор должен проверять, что первая нота ниже (меньше) второй ноты. Метод: вычисление длины интервала в тонах. Реализовать и как метод класса, и как метод экземпляра.
4. Реализовать иерархию классов: "Шахматная фигура", "Пешка", "Слон", "Конь", "Ладья", "Король", "Ферзь".
   Для каждого производного класса должна быть реализована своя функция "Сделать ход". Члены класса: поле, на котором стоит Фигура (реализовать приватный класс "Поле"); цвет фигуры. Если ход возможен, поле фигуры меняется. Если невозможен - не меняется. Поле характеризуется буквой (a-h и цифрой 1-8). В конструкторах обеспечить контроль ввода (чтобы не поставить, например, белую пешку на первую горизонталь)
5. Реализовать иерархию классов: "Шашка", "Простая шашка" (производный класс), "Дамка" (другой производный класс).
   Для каждого производного класса должна быть реализована своя функция "Сделать ход". Члены класса: поле, на котором стоит Шашка (реализовать приватный класс "Поле"); цвет шашки. Если ход возможен, поле шашки меняется. Если невозможен - не меняется. Поле характеризуется буквой (a-h и цифрой 1-8). В конструкторах обеспечить контроль ввода (чтобы не поставить шашку на белое поле).

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Реймерову Сергею Юрьевичу, Маннапову Илназу Магсумовичу.