Модуль 4. Алгоритмы и структуры данных

Тема 4.8. Ассоциативные массивы

2 часа

Оглавление

4.8. Ассоциативные массивы

4.8.1. Ассоциативный массив как набор пар «ключ-значение»

Ассоциативный массив – абстрактный тип данных, который предназначен для работы с данными в виде пар: ключ, значение (K,V). Ассоциативные массивы поддерживают операции добавления пары, поиска и удаления пары по ключу. В ассоциативном массиве не могут храниться пары с одинаковыми ключами.

В паре (K,V) значение V (value) называется значением, ассоциированным с ключом K (key). Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться. Некоторые массивы допускают пустые ключи и пустые значения.

Примером ассоциативного массива может служить телефонный справочник. Значением в данном случае является имя абонента, а ключом – номер телефона. Один номер телефона имеет одного владельца, но один человек может иметь несколько номеров. Ассоциативные массивы в профессиональной литературе на английском называют **Map** (как карточка в картотеке).

В библиотеке Java предусмотрено две основные реализации Map: ассоциативный массив на основе хеш-таблиц **HashMap** и ассоциативный массив на основе структуры данных дерево **TreeMap**. Оба класса реализуют интерфейс **Map**.

В нереляционных базах данных в качестве основной структуры данных выступают многомерные ассоциативные массивы с произвольным набором индексов, так же называемые глобалами. Вкратце, глобал — это постоянный, разреженный, динамический, многомерный массив, содержащий текстовые значения.

4.8.1. Интерфейс Map

Интерфейс Map связывает уникальные ключи с значениями. Ключ - это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект Map. После того как это значение сохранено, вы можете получить его по ключу.

```
interface Map<K, V>
```

В параметре K указывается тип ключей, в V - тип хранимых значений.

Методы:

- V **get(Object k)** - возвращает значение, ассоциированное с ключом k. Возвращает значение null, если ключ не найден;
- V **put(K k, V v)** - помещает элемент в ассоциативный массив, переписывая любое предшествующее значение, ассоциированное с таким ключом. Возвращает null, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом;
- void **clear()** - удаляет все пары "ключ-значение" из вызывающего ассоциативного массива;

- boolean **containsKey(Object k)** - возвращает значение true, если вызывающий ассоциативный массив содержит ключ k. В противном случае возвращает false;
- boolean **containsValue(Object v)** - возвращает значение true, если вызывающий ассоциативный массив содержит значение v. В противном случае возвращает false;
- boolean **equals(Object o)** - возвращает значение true, если параметр o - это ассоциативный массив, содержащий одинаковые значения. В противном случае возвращает false;
- int **hashCode()** - возвращает хеш-код вызывающего ассоциативного массив;
- boolean **isEmpty()** - возвращает значение true, если вызывающий массив не содержит пар. В противном случае возвращает false;
- void **putAll(Map<? extends K, ? extends V> m)** - помещает все значения из m в массив;
- V **remove(Object k)** - удаляет элемент, ключ которого равен k;
- int **size()** - возвращает количество пар "ключ-значение" в массиве.

Следующие три метода позволяют получить содержимое ассоциативного массива в различных видах: пар ключ-значение, только ключи или только значения.

- Set<Map.Entry<K, V>> **entrySet()** - возвращает Set, содержащий все пары ключ-значение ассоциативного массива в виде множества объектов интерфейсного типа Map.Entry.

  Каждый элемент ассоциативного массива, описываемого интерфейсом Map, имеет интерфейсный тип Map.Entry, который предоставляет три основных метода:

  - getKey() — возвращает ключ элемента;
  - getValue() — возвращает значение элемента;
  - setValue(Object value) — меняет значение элемента.
- Set<K> **keySet()** - возвращает Set, содержащий все ключи ассоциативного массива;

  Каждый элемент множества ключей, которое возвращает метод keySet(), является объектом класса, реализующего интерфейс Set. Например, можно перебрать все ключи массива:

```
Set keys = map.keySet();
for (String key: keys) {
//действия с ключом
}
```

- Collection<V> **values()** - возвращает коллекцию, содержащую все значения ассоциативного массива.

У интерфейса **Map** есть расширение - интерфейс **Sortedmap, который** гарантирует, что элементы размещаются в возрастающем порядке значений ключей.

Интерфейс **NavigableMap** в свою очередь расширяет интерфейс **Sortedmap** и позволяет искать и извлекать элементы не только по точному совпадению, но и ближайшие к заданному ключу или ключам.

4.8.2. Классы для Map

На сегодняшний момент в Java ассоциативные массивы реализованы в виде следующих классов:

- **AbstractMap** - абстрактный класс, реализующий большую часть интерфейса Map;
- **EnumMap** - расширяет класс **AbstractMap** для использования с ключами типа enum;
- **HashMap** - для использования хеш-таблицы;
- **TreeMap** - для использования дерева;
- **WeakHashMap** - для использования хеш-таблицы со слабыми ключами;
- **LinkedHashMap** - разрешает перебор в порядке вставки;
- I**dentityHashMap** - использует проверку ссылочной эквивалентности при сравнении документов.

Рассмотрим подробнее два класса **HashMap** и **TreeMap**. Первый обеспечивает максимальную скорость выборки, а порядок хранения его элементов не очевиден. Второй – хранит ключи отсортированными по возрастанию.

4.8.3 Контейнер HashMap

Класс **HashMap** использует хеш-таблицу для хранения ассоциативного массива, обеспечивая быстрое время выполнения запросов **get()** и **put()** при больших наборах. Ключи и значения в данном случае могут быть любых типов, в том числе и **null**. При этом все ключи обязательно должны быть уникальны, а значения могут повторяться. Данная реализация не гарантирует порядка элементов.

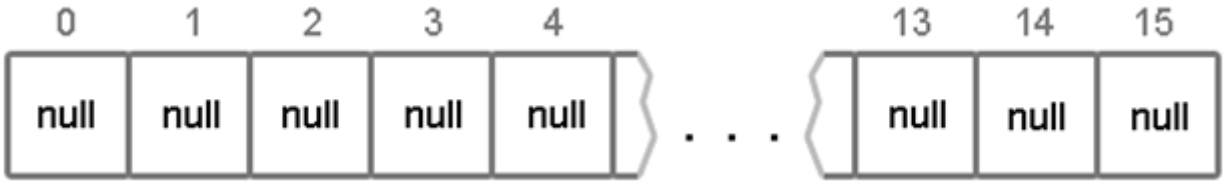
Общий вид **HashMap**:

```
// K - это Key (ключ), V - Value (значение)
class HashMap<K, V>
```

Объект класса можно объявить следующим образом:

```
Map<String, Integer> hm = new HashMap<String, Integer>();
// или так
Map<String, String> hashmap = new HashMap<String, String>();
```

По умолчанию при использовании пустого конструктора создается картотека ёмкостью из 16 ячеек. При необходимости ёмкость увеличивается автоматически.



Можно указать свои ёмкость и коэффициент загрузки, используя конструкторы **HashMap(capacity)** и **HashMap(capacity, loadFactor)**. Максимальная ёмкость, которую вы сможете установить, равна половине максимального значения int (1073741824).

Добавление элементов происходит при помощи метода **put(K key, V value)**. При добавлении нужно указать ключ и его значение.

```
hashmap.put("Moscow", "Russia");
```

Чтобы узнать размер ассоциативного массива, нужно воспользоваться следующим оператором:

```
hashmap.size();
```

Для проверки ключа и значения на наличие необходимо:

```
hashmap.containsKey("Moscow");
hashmap.containsValue("Russia");
```

Фрагмент следующего кода сначала выбирает все ключи, затем все значения, затем все ключи и значения одновременно:

```
for (String key : hashmap.keySet()) {
       System.out.println("Key: " + key);
}
for (int value : hashmap.values()) {
       System.out.println("Value: " + value);
}
for (Map.Entry entry : hashmap.entrySet()) {
       System.out.println("Key: " + entry.getKey() + " Value: "
       + entry.getValue());
}
```

4.8.4 Контейнер TreeMap

Класс **TreeMap** расширяет класс AbstractMap и реализует интерфейс **NavigatebleMap**. Он создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.

Время доступа и извлечения элементов достаточно мало, что делает класс **TreeMap** блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена. Для сортировки используются ключи, а не значения. Общий вид объекта класса схож с объектом класса **HashMap**.

В классе **TreeMap** присутствуют следующие конструкторы: **TreeMap( )**, **TreeMap(Comparator comp)**, **TreeMap(Map m)** и **TreeMap(SortedMap sm)**. Первый конструктор создает коллекцию, в которой все элементы отсортированы в натуральном порядке их ключей. Второй создаст пустую коллекцию, элементы в которой будут отсортированы по закону, который определен в передаваемом компараторе. Третий конструктор создаст **TreeMap** на основе уже имеющегося Map.Четвертый создаст **TreeMap** на основе уже имеющегося **SortedMap**, элементы в которой будут отсортированы по закону передаваемой SortedMap.

Для объектов класа TreeMap существует метод **SubMap(K fromKey, K toKey)**, который возвращает часть дерева, начиная с ключа fromKey включительно до ключа toKey исключительно.

Прототип метода **java.util.TreeMap.subMap()** выглядит следующим образом:

```
public SortedMap<K,V> subMap(K fromKey,K toKey)
```

Упражнение 4.8.1

Рассмотрим пример работы двух контейнеров HashMap и TreeMap.

Для начала создадим класс **TreeAndHashMapExample**. В классе создадим функцию, которая демонстрирует некоторые методы для работы с ассоицативными массивами.

```
public class TreeAndHashMapExample {
```

```
    private static void testMap(Map<String, String> map) {
    }
    public static void main(String[] args) {
    }
}
```

В функции testMap запишем код, который демонстрирует некоторые возможности при работе с ассоциативными массивами:

```
System.out.println("========" + map.getClass() + "==========");
        // Добавление элементов в ассоциативный массив.
        map.put("Russia", "Moscow");
        map.put("USA", "Washington");
        map.put("France", "Paris");
        // Проход по всем элементам массива.
        System.out.println("========");
        for (Entry<String, String> entry : map.entrySet()) {
        System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
        // Быстрый поиск элемента по ключу.
        System.out.println("==== Russia ====");
        System.out.println("Russia -> " + map.get("Russia"));
        // Удаление элемента по ключу.
        map.remove("Russia");
        // Несуществующий элемент обозначается как null.
        System.out.println("Russia -> " + map.get("Russia"));
        System.out.println("========");
        for (Entry<String, String> entry : map.entrySet()) {
        System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
        System.out.println("====================================");
        // Возвращаем Россию обратно.
        map.put("Russia", "Moscow");
```

Далее в функции main вызовем функцию testMap сначала для HashMap, а затем для TreeMap.

```
testMap(new HashMap<String, String>());
        TreeMap<String, String> treeMap = new TreeMap<String, String>();
        testMap(treeMap);
```

Следующий фрагмент кода демонстрирует возможность выбора подмножества из TreeMap.

```
// с TreeMap можно брать подмножество по аналогии с treeSet.
System.out.println("======== SubMap  ==========");
        Map<String, String> submap = treeMap.tailMap("Germany");
        for (Entry<String, String> entry : submap.entrySet()) {
        System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

В итоге, полностью код класса **TreeAndHashMapExample** выглядит так:

```
import java.util.*;
import java.util.Map.Entry;
```

```
public class TreeAndHashMapExample {
   private static void testMap(Map<String, String> map) {
       // Получение название класса.
       System.out.println("========" + map.getClass() + "=========");

       // Добавление элементов в ассоциативный массив.
       map.put("Russia", "Moscow");
       map.put("USA", "Washington");
       map.put("France", "Paris");
       // Проход по всем элементам массива.
       System.out.println("========");
       for (Entry<String, String> entry : map.entrySet()) {
           System.out.println(entry.getKey() + " -> " + entry.getValue());
       }
       // Быстрый поиск элемента по ключу.
       System.out.println("==== Russia ====");
       System.out.println("Russia -> " + map.get("Russia"));
       // Удаление элемента по ключу.
       map.remove("Russia");
       // Несуществующий элемент обозначается как null.
       System.out.println("Russia -> " + map.get("Russia"));
       System.out.println("========");
       for (Entry<String, String> entry : map.entrySet()) {
           System.out.println(entry.getKey() + " -> " + entry.getValue());
       }
       System.out.println("====================================");

       // Возвращаем Россию обратно.
       map.put("Russia", "Moscow");
   }

   public static void main(String[] args) {
       testMap(new HashMap<String, String>());
       TreeMap<String, String> treeMap = new TreeMap<String, String>();
       testMap(treeMap);
       // с TreeMap можно брать подмножество по аналогии с treeSet.
       System.out.println("======== SubMap  =========");
       Map<String, String> submap = treeMap.tailMap("Germany");
       for (Entry<String, String> entry : submap.entrySet()) {
           System.out.println(entry.getKey() + " -> " + entry.getValue());
       }
   }
}
```

Результат работы программы:

```
========class java.util.HashMap=========
========
France -> Paris
USA -> Washington
Russia -> Moscow
==== Russia ====
Russia -> Moscow
Russia -> null
========
France -> Paris
USA -> Washington
====================================
```

```
========class java.util.TreeMap=========
========
France -> Paris
Russia -> Moscow
USA -> Washington
==== Russia ====
Russia -> Moscow
Russia -> null
========
France -> Paris
USA -> Washington
====================================
======== SubMap  =========
Russia -> Moscow
USA -> Washington
```

4.8.5. Синхронизация ассоциативных массивов

Если вы обращаетесь к ассоциативному массиву из нескольких потоков, необходимо принять меры, чтобы не повредить информацию в массиве. Это неминуемо произойдет, если, например, один поток будет пытаться включить элемент в хэш-таблицу, а другой – перегенерировать ее.

Вместо того чтобы реализовать классы наборов данных, обеспечивающих безопасную работу с потоками, разработчики библиотеки предпочли использовать для этого механизм представлений. Например, статический метод **synchronizedMap()** класса **java.util.Collections** может преобразовать любой ассоциативный массив в Map с синхронизированными методами доступа.

```
Map synchMap = Collections.synchronizedMap(new HashMap());
// или
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

Всего вспомогательный класс java.util.Collections обеспечивает следующие синхронизирующие методы

```
public static Collection synchronizedCollection(Collection c);
public static List synchronizedList(List list);
public static Map synchronizedMap(Map m);
public static Set synchronizedSet(Set s);
public static SortedMap synchronizedSortedMap(SortedMap m);
public static SortedSet synchronizedSortedSet(SortedSet s);
```

После синхронизации можно обращаться к объекту Map из различных потоков. Каждый метод должен полностью закончить свою работу перед тем, как другой поток сможет вызвать подобный метод. При разработке необходимо следить, чтобы ни один поток не обращался к структуре данных посредством обычных не синхронизированных методов.

Задание 4.8.1

Реализовать многопоточную программу, которая добавляет в synchronizedMap элементы с ключами от 100 * THREAD_NUM до 100 * (THREAD_NUM + 1), где THREAD_NUM -- номер потока. Изучить, что происходит, если использовать обычный TreeMap.

4.8.6. Хранение данных в Android Preferences

Android предоставляет несколько вариантов для вас, чтобы сохранить постоянные данные приложений:

Shared Preferences – сохранение пар ключ-значение для примитивных типов данных;

Внутренняя память – сохранение данных во внутренней памяти устройства;

Внешняя память – сохранение данных на внешней памяти устройства;

База данных SQLite – сохранение структурированных данных в БД.

Решение по выбору одного из вариантов хранения зависит от конкретных потребностей, таких, как должны ли данные быть доступными только для вашего приложения или нет, сколько места требуется для ваших данных и др.

В данном параграфе рассмотрим способ хранения данных, основанных на хранении ассоциативных массивов – **Shared Preferences**. Класс **SharedPreference**s позволяет создавать в приложении именованные ассоциативные массивы типа «ключ — значение», которые могут быть использованы различными компонентами приложения.

Общие настройки поддерживают базовые типы boolean, string, float, long и integer, что делает их идеальным средством для быстрого сохранения значений по умолчанию, переменных экземпляра класса, текущего состояния UI или пользовательских настроек.

Чтобы получить экземпляр класса SharedPreferences для получения доступа к настройкам в коде приложения используются три метода:

- **getPreferences()** – внутри активности, чтобы обратиться к определенному для активности предпочтению;
- **getSharedPreferences()** – внутри активности, чтобы обратиться к предпочтению на уровне приложения;

getDefaultSharedPreferences() – из объекта PreferencesManager, чтобы получить общедоступную настройку, предоставляемую Android.

По отношению к введенным данным можно выбрать 3 уровня доступности:

- **MODE_PRIVATE** – только это приложение может читать настройки с xml файла;
- **MODE_WORLD_READABLE** – все приложения могут читать с xml файла;
- **MODE_WORLD_WRITEABLE** – все приложения могут выполнять запись в xml файл.

Все эти методы возвращают экземпляр класса SharedPreferences, из которого можно получить соответствующую настройку с помощью ряда методов:

getBoolean(String key, boolean defValue);

getFloat(String key, float defValue);

getInt(String key, int defValue);

getLong(String key, long defValue);

getString(String key, String defValue)

Чтобы создать или изменить Общие настройки, нужно вызвать метод getSharedPreferences в контексте приложения, передав имя общих настроек (имя файла):

```
SharedPreferences sharedPreferences = getSharedPreferences(APP_PREFERENCES,
Context.MODE_PRIVATE);
```

Упражнение 4.8.2

Разработаем приложение, в котором покажем пример сохранения строки. В приложении будет поле для ввода текста и две кнопки – Save и Load. По нажатию на Save мы будем сохранять значение из поля, по нажатию на Load – загружать.

Откроем **main.xml** и создадим поле ввода и две кнопки:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical">
   <EditText
       android:id="@+id/etText"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">
   </EditText>
   <Button
       android:id="@+id/save"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="Save">
   </Button>
   <Button
       android:id="@+id/load"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="Load">
   </Button>
</LinearLayout>
```

В **MainActivity.java** пишем следующий код:

```
package com.example.myapplication;

import android.content.SharedPreferences;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends ActionBarActivity implements
View.OnClickListener {
   EditText etText;
   Button btnSave, btnLoad;
   SharedPreferences sharedPreferences;
   final String SAVED_TEXT = "TEXT";
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       etText = (EditText) findViewById(R.id.etText);

       btnSave = (Button) findViewById(R.id.save);
       btnSave.setOnClickListener((View.OnClickListener) this);
```

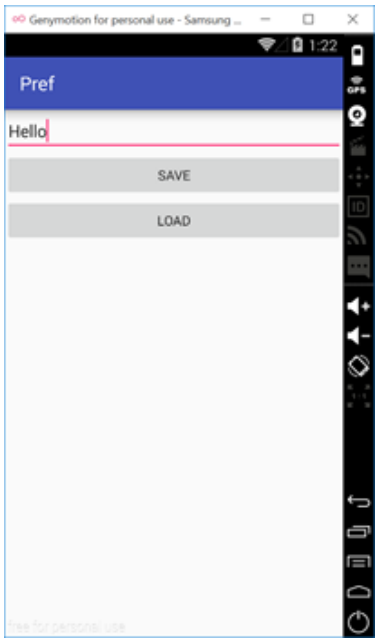
```
        btnLoad = (Button) findViewById(R.id.load);
        btnLoad.setOnClickListener((View.OnClickListener) this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.save:
                saveData();
                break;
            case R.id.load:
                loadData();
                break;
            default:
                break;
        }
    }

    void saveData() {
        sharedPreferences = getPreferences(MODE_PRIVATE);
        SharedPreferences.Editor editor = sharedPreferences.edit();
        editor.putString(SAVED_TEXT, etText.getText().toString());
        editor.commit();
        Toast.makeText(this, "Text saved", Toast.LENGTH_SHORT).show();
    }

    void loadData() {
        sharedPreferences = getPreferences(MODE_PRIVATE);
        String savedText = sharedPreferences.getString(SAVED_TEXT, "");
        etText.setText(savedText);
        Toast.makeText(this, "Text loaded", Toast.LENGTH_SHORT).show();
    }
}
```

После объявления метода работы с Shared Preferences нужно создать объект Editor, который нужен для создания пар имя-значение, которые будут записаны в xml файл для сохранения с помощью метода put(). Для успешного внесения данных в файл сохранения в конце нужно выполнить команду commit().



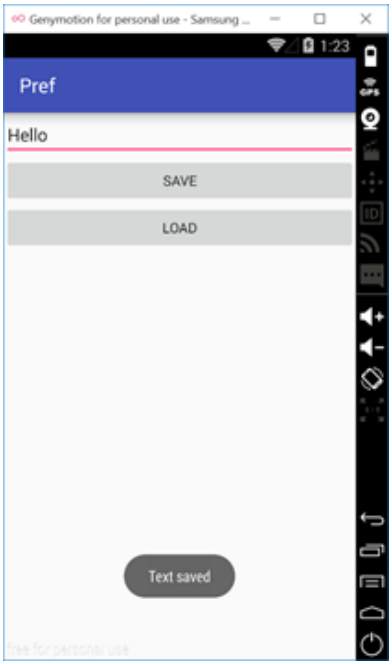
Для того, чтобы извлечь сохраненные данные, нужно обратиться к ним с помощью команды get (), ссылаясь на необходимые пары-значения.

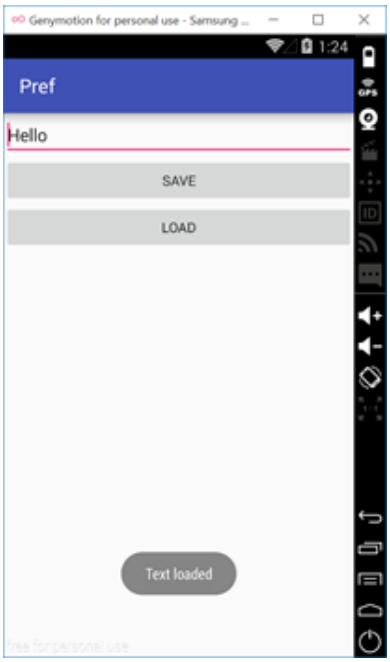
При сохранении и загрузке сохраненных данных будет высвечиваться Toast сообщение с информацией о выполненной операции.

Для начала убедимся, что будет если данные не сохранять. Введите текст «Hello» в поле ввода и, не нажимая кнопку **Save**, закройте приложение.

Теперь найдите приложение в общем списке приложений и запустите снова. Поле ввода пустое. То, что мы вводили – пропало при закрытии программы.

Давайте попробуем сохранять. Снова введите значение и нажмите Save. Значение сохранилось в системе. Теперь закроем приложение (Назад), снова откроем и нажмем Load. Значение считалось и отобразилось.





Чтобы сохранение и загрузка происходили автоматически при закрытии и открытии приложения метод loadData будем вызывать в onCreate, а метод saveData - в onDestroy.

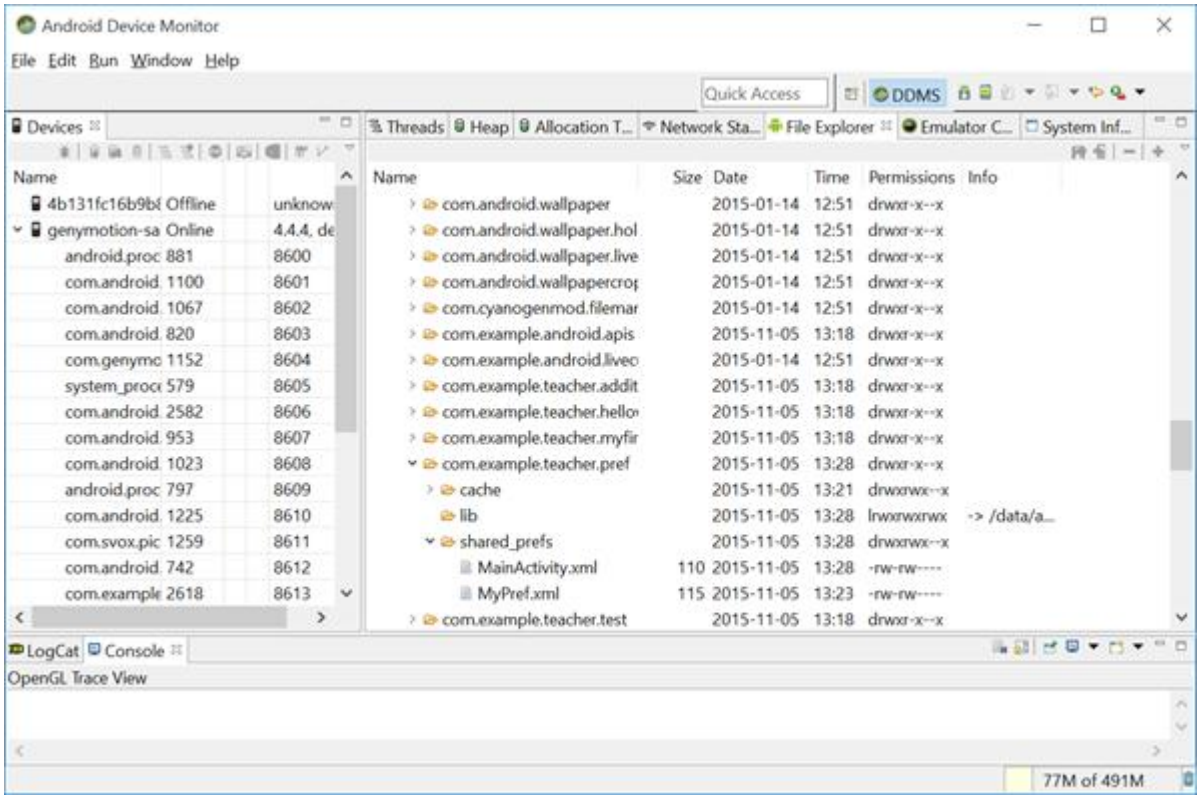
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    etText = (EditText) findViewById(R.id.etText);

    btnSave = (Button) findViewById(R.id.save);
    btnSave.setOnClickListener((View.OnClickListener) this);

    btnLoad = (Button) findViewById(R.id.load);
    btnLoad.setOnClickListener((View.OnClickListener) this);
    loadData();

}
@Override
protected void onDestroy() {
    saveData();
    super.onDestroy();
}
```

Preferences-данные сохраняются в файлы и вы можете посмотреть их. Для этого в Eclipse откройте меню **Window** > **Show View** > **Other** и выберите **Android** > **File Explorer**. В Android Studio **Tools > Android > Android Device Monitor** и выберете вкладку **File Explorer.** Отобразилась файловая система устройства (или эмулятора).



Если открыть папку data/data/package_name/shared_prefs, то мы увидим там файл MainActivity.xml. Следует обратить внимание на то, что в пути к файлу используется наш package. Если его открыть - увидим следующее:

```
<?xml version="1.0" encoding="utf-8"?>
  <map>
  <string name="TEXT">Hello</string>
  </map>
```

Разберемся, откуда взялось наименование файла MainActivity.xml. Дело в том, что кроме метода getPreferences, который мы использовали, есть метод getSharedPreferences. Он выполняет те же функции, но позволяет указать имя файла для хранения данных. Если бы мы в saveData использовали для получение SharedPreferences такой код:

```
  sharedPreferences = getSharedPreferences("MyPref", MODE_PRIVATE);
```

то данные сохранились бы в файле MyPref.xml. Таким образом, мы научились как в Android выполнять сохранение данных приложения с помощью стандартного интерфейса под названием Shared Preferences.

Задание 4.8.2

Создать приложение для регистрации пользователей с указанием имени и пароля. В приложении предусмотреть возможность авторизации. При успешной (не успешной) авторизации выдать соответствующее сообщение. Данные зарегистрированных пользователей хранить с использованием Shared Preferences.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Митякову Евгению Сергеевичу.