ЛАБОРАТОРНАЯ РАБОТА №6

Списки. Создание собственного адаптера. Механизмы обратного вызова.

Цель: формирование у студентов знаний и навыков создания кастомизированных списков на основе собственного адаптера, реализации механизмов обратного вызова для отслеживания событий в многофункциональных Android-приложениях.

План занятия

1. Изучить теоретические сведения.
2. Выполнить практическое задание по лабораторной работе.
3. Оформить отчёт и ответить на контрольные вопросы.

Теоретические сведения

Для разработки многофункциональных интерфейсов Android приложений необходимо научиться создавать свои собственные наборы View элементов в отдельном xml-файле с последующим подключением созданной графики в родительскую разметку. Создание View-элемента из содержимого layout-файла. Класс LayoutInflater используется для создания View-элемента из со держимого отдельного layout-файла. Алгоритм действий для достижения вышеуказанной цели следующий:

1. Создаем xml-файл с требуемым View-элементом или набором Viewэлементов (например, my_view.xml).
2. В Activity, к которой требуется добавить содержимое my_view.xml, находим id того layout, в котором и разместиться View-элемент (например, lin_layout).
3. В методе onCreate создаем объект класса LayoutInflater:
   LayoutInflater layoutInflater = LayoutInflater.from(this);
4. Вызываем из-под объекта layoutInflater метод inflater:
   View view = layoutInflater.inflate(R.layout.my_view, lin_layout, true);

Синтаксис метода inflater:

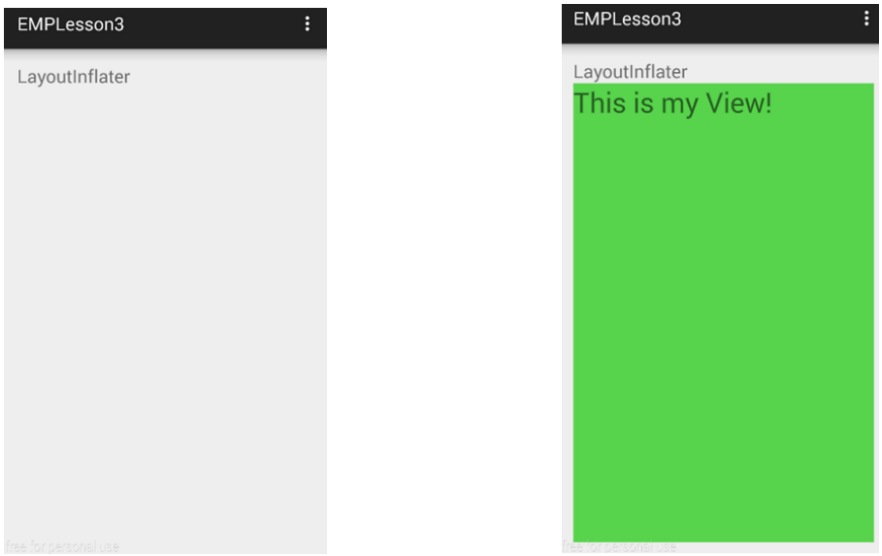
public View inflate (int resource, ViewGroup root, boolean attachToRoot);

resource – ID layout-файла, который будет использован для создания View.

root – родительский ViewGroup-элемент для создаваемого View.

attachToRoot – параметр, определяющий присоединять ли создаваемый View к root.

Чтобы root стал родителем создаваемого View необходимо указать true. Пример полного варианта кода с добавлением View-элемента из содержимого layout-файла как дочернего приведен ниже (рисунок 3.1).



а б

Рисунок 3.1 – Пример первоначального интерфейса (а) и интерфейса с добавлением дочернего View-элемента из содержимого layout-файла (б)

```
public class MainActivity extends ActionBarActivity {
        private LinearLayout lin_layout;
        private LayoutInflater layoutInflater;
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                initView();
  }
  private void initView() {
          lin_layout = (LinearLayout) findViewById(R.id.lin_layout);
          layoutInflater = LayoutInflater.from(this);
```

```
        View view = layoutInflater.inflate(R.layout.my_view, lin_layout, true);
        }
    }
```

Список ListView.

Список ListView содержит однотипные по дизайну пункты и логику обработки взаимодействия с элементами этих пунктов (рисунок 3.2). Каждый пункт списка представляет собой ViewGroup, создаваемый в отдельном layout файле.

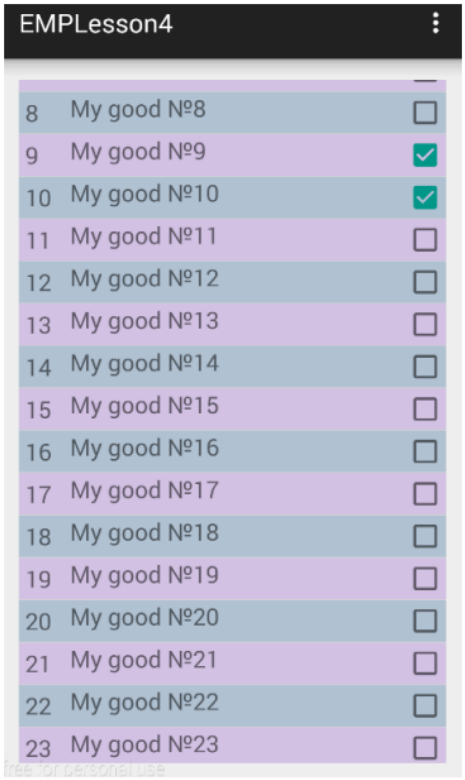


Рисунок 3.2 – Пример реализации списка ListView

Стратегия создания списка следующая:

1. Создаем в папке layout xml-файл с графическим отображением одного пункта списка. Для корректного отображения набора пунктов в результирующем списке в качестве layout_height указываем wrap_content или высоту в dp.

2. Создаем набор данных для последующего наполнения списка.

3. Программируем свой собственный адаптер. Адаптер – структурный шаблон проектирования, предназначенный для создания класса-оболочки с требуемым интерфейсом.

Адаптеру назначаем набор данных для наполнения списка и layout ресурс с визуальным представлением одного пункта списка.

4. Присваиваем адаптер списку ListView. Список при построении запрашивает у адаптера пункты, адаптер их создает (используя данные и layout файл) и возвращает списку. В итоге мы видим готовый список.

Рассмотрим подробно вышеизложенную стратегию. В качестве примера разработаем магазин товаров (см. рисунок 3.2).

Создадим модель данных – класс Good с полями int id (номер товара), String name (наименование товара), boolean check (флаг, отображающий выбор товара пользователем), конструктором класса и методами getId(), getName(), isCheck(), setId(int id), setName(String name), setCheck(boolean check).

Разработаем графическое представление одного пункта списка – файл item_good.xml, содержащий два TextView и один CheckBox для отображения параметров товара.

В activity_main.xml в корневой layout добавляем ListView с размерами wrap content.

Создаем класс GoodsAdapter для реализации кастомизированного адаптера.

В MainActivity.java находим listView по id; создаем динамический массив ArrayList arr_goods и имитируем интернет-магазин путем генерации объектов класса Good. Создаем объект goodsAdapter от класса GoodsAdapter: для этого в конструктор класса передаем параметр context (т.е. this) и коллекцию объектов arr_goods. Далее присваиваем goodsAdapter списку с помощью метода setAdapter.

```
public class MainActivity extends ActionBarActivity {

        private ListView listView;

        private ArrayList arr_goods = new ArrayList();

        private final int SIZE_OF_ARR = 25;

        private GoodsAdapter goodsAdapter;

        @Override

        protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initView(); createMyListView();
    }
    private void initView() {
        listView = (ListView) findViewById(R.id.listView);
    }
    private void createMyListView() {
        fillData();
        goodsAdapter = new GoodsAdapter(this, arr_goods);
        listView.setAdapter(goodsAdapter);
    }
    private void fillData(){
        int i=0;
        while (i<SIZE_OF_ARR) {
        i++;
        arr_goods.add(new Good(i," " + "My good №" + i, false));
        }
    }
}
```

Рассмотрим реализацию класса GoodsAdapter. Наследуемся от базового адаптера BaseAdapter. Для обработки событий от чек-боксов реализуем интерфейс CompoundButton.OnCheckedChangeListener.

```
public class GoodsAdapter extends BaseAdapter implements CompoundButton.OnCheckedChangeListener{
private Context context;
private ArrayList arr_goods_adapter;
```

```
private LayoutInflater layoutInflater;
public GoodsAdapter(Context context, ArrayList arr_goods_adapter) {
        this.context = context;
        this.arr_goods_adapter = arr_goods_adapter;
        this.layoutInflater = LayoutInflater.from(context);
} // кол-во элементов
@Override
 public int getCount() {
        return arr_goods_adapter.size();
 } // элемент по позиции
 @Override
 public Object getItem(int position) {
        return arr_goods_adapter.get(position);
 } // id по позиции
 @Override
 public long getItemId(int position) {
        return position;
 } // пункт списка
 @Override
 public View getView(int position, View convertView, ViewGroup viewGroup) {
        View view = convertView;
        if (view == null) {
                view = layoutInflater.inflate(R.layout.item_good, null, false);
        }
        Good good_temp = arr_goods_adapter.get(position);
```

```
        TextView tv_goodId = (TextView) view.findViewById(R.id.tv_goodId);
        tv_goodId.setText(Integer.toString(good_temp.getId()));
        TextView tv_goodName = (TextView) view.findViewById(R.id.tv_goodName);
        tv_goodName.setText(good_temp.getName());
        CheckBox cb_good = (CheckBox) view.findViewById(R.id.cb_good);
        cb_good.setChecked(good_temp.isCheck());
        cb_good.setTag(position);
        cb_good.setOnCheckedChangeListener(this); return view;
    } @Override
    public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked) {
        if (compoundButton.isShown()) {
            int i = (int) compoundButton.getTag();
            arr_goods_adapter.get(i).setCheck(isChecked);
            notifyDataSetChanged();
        }
    }
}
```

В конструкторе заполняем внутренние переменные и получаем LayoutInflater для работы с layout-ресурсами. В arr_goods_adapter хранится список товаров.

Методы, отмеченные аннотацией @Override, необходимо реализовать при наследовании BaseAdapter. Эти методы используются списком и должны работать корректно.

Метод getCount должен возвращать количество элементов списка (в текущем примере это количество товаров).

Метод getItem должен возвращать элемент по указанной позиции. Используя позицию, получаем конкретный элемент из arr_goods_adapter.

Метод getItemId должен возвращать id элемента (возвращаем текущую позицию).

Метод getView должен возвращать View пункта списка. Для этого ранее создавался layout-ресурс R.layout.item_good. В этом методе необходимо из R.layout.item_good создать View, заполнить его данными и отдать списку. Но перед тем как создавать, пробуем использовать convertView, который идет на вход метода. Это уже созданное ранее View, но неиспользуемое в данный момент. Например, при прокрутке списка, часть пунктов уходит за экран и их уже не надо прорисовывать. View из этих «невидимых» пунктов используются для воссоздания графики и обновления данных. Это значительно ускоряет работу приложения, т.к. не надо прорисовывать inflate лишний раз.

Если же convertView не поступил (null), то создаем сами view.

Далее заполняем параметры товара, чекбоксу присваиваем обработчик, сохраняем в Tag позицию элемента. Tag – это подобие Object-хранилища у каждого View, куда можно поместить требуемые данные. В нашем случае для каждого чекбокса помещаем в его Tag номер позиции пункта списка. Далее в обработчике чекбокса будет возможность этот номер позиции извлечь и определить, в каком пункте списка был нажат чекбокс.

В итоге, метод getView возвращает списку полностью заполненное view, и список его отобразит как очередной пункт.

onCheckedChanged – обработчик для чекбоксов. При нажатии на чекбокс в списке он срабатывает, читает из Tag позицию пункта списка и помечает соответствующий товар как положенный в корзину. Без этого обработчика не работало бы помещение товаров в корзину, а на экране значения чекбоксов в списке терялись бы при прокрутке. Потому что пункты списка пересоздаются, если они уходят «за экран» и снова появляются. Это пересоздание обеспечивает метод getView, а он для заполнения View берет данные из товаров. Значит при нажатии на чекбокс, обязательно надо сохранить в данных о товаре то, что он теперь в корзине. Метод notifyDataSetChanged() уведомляет список об изменении данных для обновления списка на экране.

Header и Footer в списках.

Header и Footer – это View-элементы, которые могут быть добавлены к списку сверху и снизу. Для этого необходимо создать графические

представления header_mygoods.xml и footer_mygoods.xml, преобразовать их в Viewэлементы и предоставить списку с помощью методов addHeader или addFooter. Обязательным условием отображения Header и Footer является их добавление к списку до того, как присваивается адаптер.

Модифицируем код MainActivity.java:

public class MainActivity extends ActionBarActivity implements View.OnClickListener { // добавляем новые переменные класса

private LayoutInflater layoutInflater;

private View view_header, view_footer;

private Button btnShow; private TextView tv_count; // добавляем Header и Footer

private void createMyListView() {

fillData(); goodsAdapter = new GoodsAdapter(this, arr_goods, this);

layoutInflater = LayoutInflater.from(this); view_header = layoutInflater.inflate(R.layout.header_mygoods, null);

view_footer = layoutInflater.inflate(R.layout.footer_mygoods, null);

btnShow = (Button) view_footer.findViewById(R.id.btnShow);

btnShow.setOnClickListener(this); tv_count = (TextView) view_footer.findViewById(R.id.tv_count);

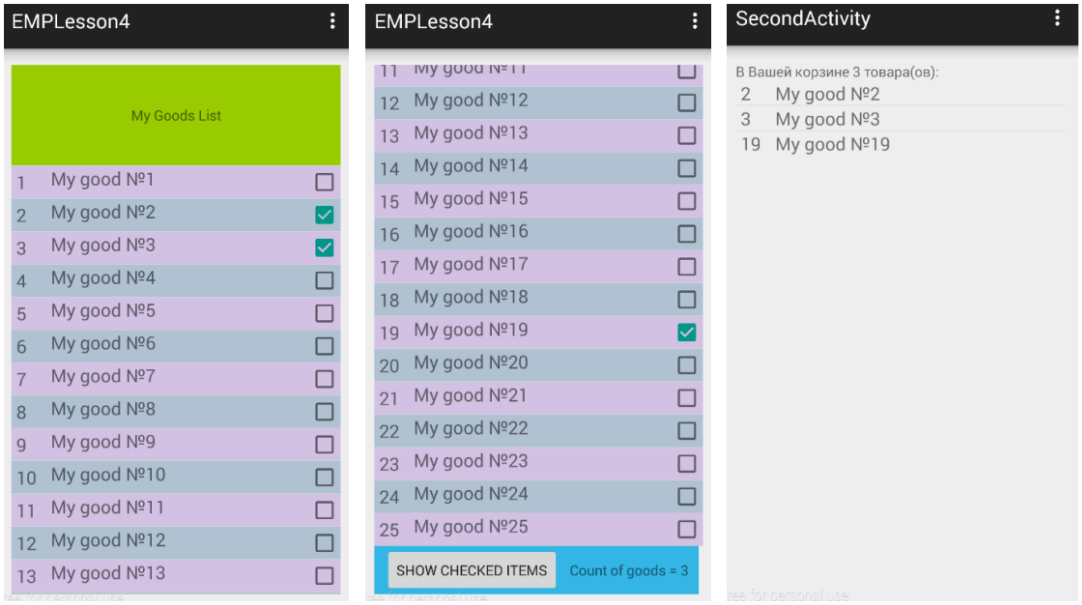
listView.addHeaderView(view_header);

listView.addFooterView(view_footer);

listView.setAdapter(goodsAdapter);

}

Пример реализации списка с Header приведен на рисунке 3.3, а, с Footer – на рисунке 3.3, б



а б в

Рисунок 3.3 – Пример реализации списка ListView с Header (а), Footer (б) и переходом в корзину товаров (в)

Механизмы обратного вызова для обработки событий в Android приложениях.

В продолжение примера со списком товаров магазина поставим следующую задачу: отмеченные товары должны сохраняться в отдельном динамическом массиве для последующего отображения корзины товаров пользователя в новом Activity (см. рисунок 3.3, в).

Особенностью реализации данной задачи является необходимость обработки события, наступление которого отслеживает класс GoodsAdapter, в классе MainActivity.

Простейший механизм callback (обратный вызов функции) позволяет вызвать из одного класса метод другого класса. Для этого достаточно иметь объект от класса, что дает полный доступ ко всем его методам.

В частности, в GoodsAdapter добавляем логику формирования динамического массива выбранных товаров arr_checked_goods_adapter и создаем метод getCheckedGoods(), возвращающий данный массив.

GoodsAdapter.java:

```
private ArrayList arr_checked_goods_adapter = new ArrayList();

@Override
```

```
public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked) {
    if (compoundButton.isShown()) {
        int i = (int) compoundButton.getTag();
        arr_goods_adapter.get(i).setCheck(isChecked);
        notifyDataSetChanged();
        if(isChecked){
            arr_checked_goods_adapter.add(arr_goods_adapter.get(i));
        }else {
            arr_checked_goods_adapter.remove(arr_goods_adapter.get(i));
            }
        }
    }
    public ArrayList getCheckedGoods() { return arr_checked_goods_adapter; }
```

Тогда в MainActivity реализуем логику обработки нажатия на кнопку Show checked items.

MainActivity.java:

```
private ArrayList arr_checked_goods = new ArrayList();
@Override
public void onClick(View view) {
    arr_checked_goods = goodsAdapter.getCheckedGoods();
    Intent intent = new Intent(this, SecondActivity.class);
    intent.putParcelableArrayListExtra("MyList", arr_checked_goods);
    startActivity(intent);
}
```

Усложним задачу: количество выбранных товаров должно динамически изменяться в TextView, размещенным в Footer списка (см. рисунок 3.3, б). В

этом случае потребуется использовать интерфейс в качестве прослойки для отслеживания событий. Такой подход является распространенным и подробно описан в шаблоне проектирования Наблюдатель (Observer). Для этого класс реализует специально созданный интерфейс; тем самым у класса появляется механизм, который позволяет получать экземпляру объекта этого класса оповещения от объектов других классов об изменении их состояния, тем самым наблюдая за ними.

Создаем свой собственный интерфейс:

```
public interface OnChangeListener {
    public void onDataChanged();
}
```

В MainActivity.java имплементируем данный интерфейс, а в методе onDataChanged() запрашиваем размер массива с выбранными пользователем товарами и выводим это число в соответствующий TextView.

```
public class MainActivity extends ActionBarActivity implements OnChangeListener, View.OnClickListener {
@Override
public void onDataChanged() {
    int size = goodsAdapter.getCheckedGoods().size();
    tv_count.setText("Count of goods = " + size + "");
}
```

В конструктор GoodsAdapter.java необходимо передать объект от интерфейса OnChangeListener. Так как MainActivity.java имплементирует указанный интерфейс, то в качестве объекта от интерфейса OnChangeListener выступает собственно MainActivity.java , т.е. передаем this:

```
goodsAdapter = new GoodsAdapter(this, arr_goods, this);
```

Изменения в GoodsAdapter.java следующие: добавляем в поля класса и, соответственно, конструктор, объект от интерфейса OnChangeListener: onChangeListener. В методе обработки нажатий на чекбоксы вызываем из-под объекта onChangeListener метод onDataChanged().

GoodsAdapter.java:

```
private OnChangeListener onChangeListener;

public GoodsAdapter(Context context, ArrayList arr_goods_adapter, OnChangeListener onChangeListener) {

    this.context = context;

    this.arr_goods_adapter = arr_goods_adapter;

    this.layoutInflater = LayoutInflater.from(context);

    this.onChangeListener = onChangeListener;

}

@Override

public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked) {

    if (compoundButton.isShown()) {

    int i = (int) compoundButton.getTag();

    arr_goods_adapter.get(i).setCheck(isChecked);

    notifyDataSetChanged(); if(isChecked){

        arr_checked_goods_adapter.add(arr_goods_adapter.get(i));

        }else {

            arr_checked_goods_adapter.remove(arr_goods_adapter.g
        et(i));

        } onChangeListener.onDataChanged();

    }

}
```

Таким образом, при каждом нажатии на чек-бокс любого товара автоматически сработает реализация метода onDataChanged() в MainActivity и произойдет перерасчет количества выбранных пользователем товаров.

Передаем Parcelable объекты с помощью Intent.

Для передачи динамического массива объектов между Activity используются следующие методы:

// для отправки данных

intent.putParcelableArrayListExtra("MyList", arr_checked_goods); // для извлечения данных

arr_checked_goods = getIntent().getParcelableArrayListExtra("MyList");

При этом в классе Good необходимо реализовать интерфейс Parcelable:

public class Good implements Parcelable{ private int id; private String name;

private boolean check; // обычный конструктор

public Good(int id, String name, boolean check){

this.id = id;

this.name = name; this.check = check;

}

@Override

public int describeContents() { return 0; } // упаковываем объект в Parcel

@Override

public void writeToParcel(Parcel parcel, int i) {

parcel.writeInt(id);

parcel.writeString(name);

}

public static final Parcelable.Creator CREATOR = new Parcelable.Creator() { // распаковываем объект из Parcel

public Good createFromParcel(Parcel in) { return new Good(in); }

public Good[] newArray(int size) { return new Good[size]; }

}; // конструктор, считывающий данные из Parcel

private Good(Parcel parcel) {

id = parcel.readInt(); name = parcel.readString(); check = false;

}

}

В методе writeToParcel в объект Parcel упаковывают поля объекта Good.

CREATOR типа Parcelable.Creator используется для создания экземпляра Good и заполнения его данными из Parcel. Для этого используется его метод createFromParcel, который необходимо реализовать. На вход поступает Parcel, а вернуть нужно готовый Good. В нашем примере используется конструктор Good(Parcel parcel), который реализован отдельно.

Конструктор Good(Parcel parcel) принимает на вход Parcel и заполняет объект Good данными из Parcel.

Итоговая структура проекта приведена на рисунке 3.4.

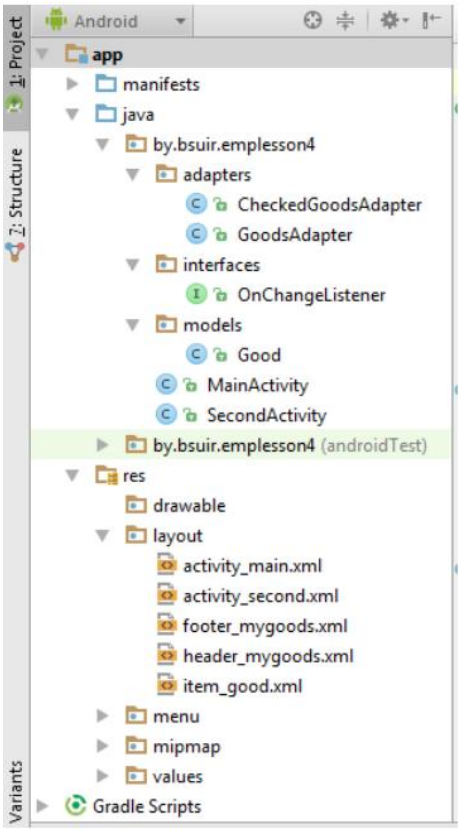


Рисунок 3.4 – Итоговая структура проекта MiniShop

Практическое задание

1. Разработать приложение MiniShop, состоящее из двух Activity (рисунок 3.3, 3.4).

2. В первом Activity создать список ListView с Header и Footer.

3. В Footer разместить текстовое поле (TextView) для ввода количества активированных пользователем товаров, кнопку Show Checked Items для перехода в корзину товаров.

4. Реализовать кастомизированный список ListView с помощью собственного адаптера, наследующего класс BaseAdapter.

5. В каждом пункте списка отобразить следующую информацию о товаре: идентификационный номер, название, стоимость, чек-бокс для возможности выбора товара пользователем.

6. В текстовом поле (TextView) Footer списка динамически отображать общее текущее количество активированных товаров.

7. При нажатии на кнопку Show Checked Items реализовать переход во второе Activity с корзиной товаров

8. Корзину товаров реализовать в виде нового кастомизированного списка с выбранными товарами.

9. Продемонстрировать работу приложения MiniShop на эмуляторе или реальном устройстве.

Содержание отчета

1. Скриншоты графических представлений первого и второго Activity в Android Studio, демонстрирующие логику работы приложения MiniShop.

2. Код xml-файлов графических представлений, используемых в приложении MiniShop.

3. Код java-файлов приложения MiniShop, включая классы Activity, собственные интерфейсы, адаптеры, модели данных.

Контрольные вопросы

1. Как создать View-элемент из содержимого layout-файла? В каких случаях это необходимо?

2. Как создать и обеспечить работу списка ListView?

3. Как реализовать собственный кастомизированный список?

4. Что собой представляет и для чего нужен адаптер в Android приложениях?

5. Какие методы класса BaseAdapter необходимо переопределить при создании кастомизированных списков?

6. Для чего нужен метод getView в адаптере?

7. Что собой представляет и как реализовать Header в списках?

8. Что собой представляет и как реализовать Footer в списках?

9. Какие Вы знаете механизмы обратного вызова для обработки событий в Android-приложениях?

10. Как передать динамический массив объектов из одного Activity в другое с помощью Intent?