

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Компьютерного проектирования
Кафедра Инженерной психологии и эргономики
Дисциплина Технологии программирования приложений

Лабораторная работа №3
«Списки. Создание собственного адаптера. Механизмы обратного вызова»

Студент группы 310901

(подпись)

Усов А.М.

Руководитель

(подпись)

Василькова А.Н.

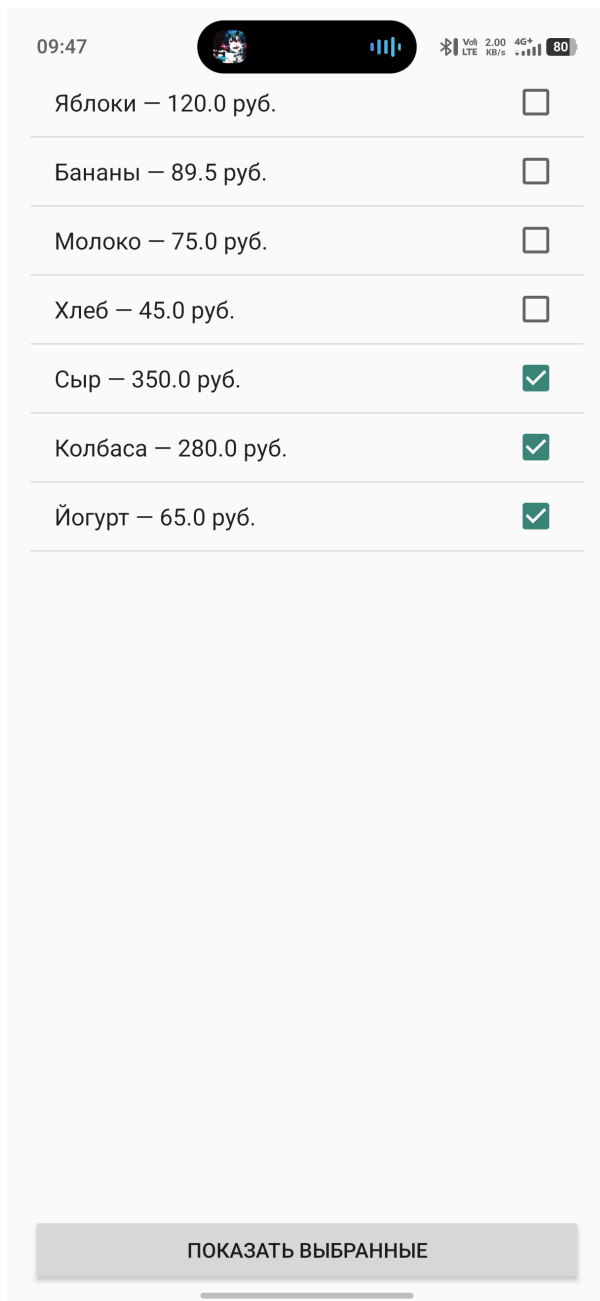
Минск 2025

В рамках данной лабораторной работы разрабатывается приложение MiniShop для платформы Android. Приложение представляет собой магазин товаров с возможностью выбора товаров из списка и просмотра выбранных товаров.

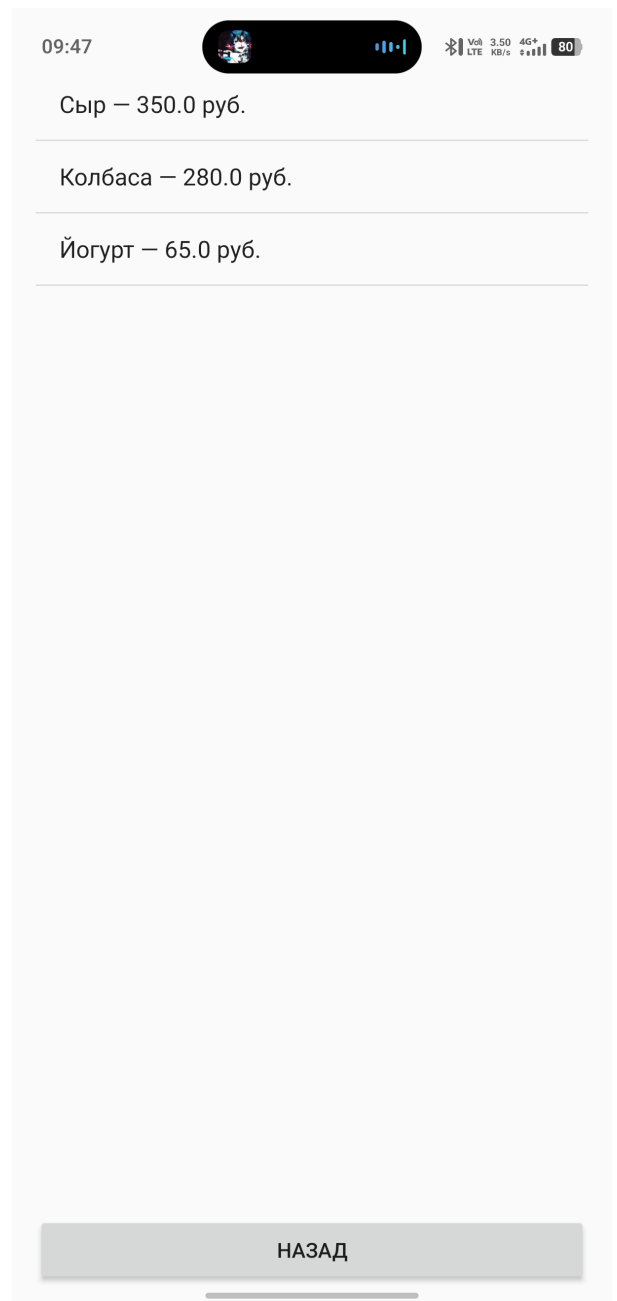
Цель работы — формирование у студентов знаний и навыков создания кастомизированных списков на основе собственного адаптера, реализации механизмов обратного вызова для отслеживания событий в многофункциональных Android-приложениях.

0.1 Скриншоты графических представлений первого и второго Activity в Android Studio

Ниже представлены графические представления первого и второго Activity приложения MiniShop в Android Studio.



(a) Вариант А: Первое Activity (MainActivity)



(b) Вариант Б: Второе Activity (SelectedItemActivity)

Рисунок 1 – Графические представления Activity приложения MiniShop

На рисунке 1 (вариант А) показан интерфейс первого Activity приложения MiniShop. На экране отображается список товаров с возможностью множественного выбора и кнопка "Показать выбранные" для перехода ко второму Activity.

На рисунке 1 (вариант Б) показан интерфейс второго Activity приложения MiniShop. На экране отображается список выбранных товаров и кнопка "Назад" для возврата к первому Activity.

0.2 Код XML-файлов графических представлений

0.2.1 Ниже представлен код XML-файла разметки первого Activity приложения MiniShop.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_marginTop="25sp"
        android:layout_weight="1"
        android:choiceMode="multipleChoice" />

    <Button
        android:id="@+id/showSelectedButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Показать выбранные"
        android:layout_marginTop="16dp" />

</LinearLayout>
```

В данном файле определены следующие элементы:

- `LinearLayout` — контейнер с вертикальной ориентацией для размещения элементов интерфейса;
- `ListView` — список товаров с возможностью множественного выбора (`android:choiceMode="multipleChoice"`);
- `Button` — кнопка для перехода к экрану выбранных товаров.

0.2.2 Ниже представлен код XML-файла разметки второго Activity приложения MiniShop.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <ListView
        android:id="@+id/selectedListView"
        android:layout_width="match_parent"
```

```

        android:layout_height="536dp"
        android:layout_marginTop="25sp"
        android:layout_weight="1" />

<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:text="Назад" />

</LinearLayout>

```

В данном файле определены следующие элементы:

- `LinearLayout` — контейнер с вертикальной ориентацией;
- `ListView` — список выбранных товаров;
- `Button` — кнопка для возврата к первому Activity.

0.2.3 Ниже представлен код XML-файла разметки элемента списка для отображения товаров.

```

<?xml version="1.0" encoding="utf-8"?>
<CheckedTextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:checkMark="?android:attr/listChoiceIndicatorMultiple"
    android:gravity="center_vertical"
    android:paddingStart="16dp"
    android:paddingEnd="16dp"
    android:paddingTop="12dp"
    android:paddingBottom="12dp"
    android:textSize="16sp" />

```

В данном файле определен элемент `CheckedTextView`, который используется для отображения элементов списка с возможностью выбора. Атрибут `android:checkMark` определяет вид индикатора выбора для множественного выбора.

0.3 Код Kotlin-файлов приложения MiniShop

0.3.1 Ниже представлен код класса модели данных `Product`, используемой для представления товара в приложении.

```

package com.example.lt3

data class Product(

```

```

        val name: String,
        val price: Double
    ) {
        override fun toString(): String {
            return "$name — $price руб."
        }
    }
}

```

Класс `Product` представляет собой `data class`, содержащую информацию о товаре: название (`name`) и цену (`price`). Метод `toString()` переопределен для удобного отображения товара в списке.

0.3.2 Ниже представлен код класса `ProductAdapter`, реализующего собственный адаптер для списка товаров на основе `BaseAdapter`.

```

package com.example.lt3

import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.BaseAdapter
import android.widget.CheckedTextView

class ProductAdapter(
    private val products: List<Product>
) : BaseAdapter() {

    override fun getCount(): Int = products.size

    override fun getItem(position: Int): Product = products[position]

    override fun getItemId(position: Int): Long = position.toLong()

    override fun getView(position: Int, convertView: View?, parent:
↳ ViewGroup): View {
        val view: CheckedTextView = if (convertView == null) {
            LayoutInflater.from(parent.context)
                .inflate(R.layout.list_item, parent, false) as CheckedTextView
        } else {
            convertView as CheckedTextView
        }

        val product = getItem(position)
        view.text = product.toString()

        return view
    }
}

```

Класс `ProductAdapter` наследуется от `BaseAdapter` и реализует следующие методы:

- `getCount()` — возвращает количество элементов в списке;
- `getItem(position)` — возвращает объект `Product` по указанной позиции;
- `getItemId(position)` — возвращает уникальный идентификатор элемента;
- `getView(position, convertView, parent)` — создает или переиспользует `View` для отображения элемента списка. Использует механизм переиспользования `View` через параметр `convertView` для оптимизации производительности.

0.3.3 Ниже представлен код класса `MainActivity`, содержащего логику работы первого экрана приложения.

```
package com.example.lt3

import android.content.Intent
import android.os.Bundle
import android.widget.Button
import android.widget.ListView
import android.widget.Toast
import androidx.activity.ComponentActivity

data class Product(
    val name: String,
    val price: Double
) {
    override fun toString(): String {
        return "$name — $price руб."
    }
}

class MainActivity : ComponentActivity() {

    private val products = listOf(
        Product("Яблоки", 120.0),
        Product("Бананы", 89.50),
        Product("Молоко", 75.0),
        Product("Хлеб", 45.0),
        Product("Сыр", 350.0),
        Product("Колбаса", 280.0),
        Product("Йогурт", 65.0)
    )

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main_activity)

        val listView = findViewById<ListView>(R.id.listView)
        val showSelectedButton = findViewById<Button>(R.id.showSelectedButton)
```

```

val adapter = ProductAdapter(products)
listView.adapter = adapter

showSelectedButton.setOnClickListener {
    val selectedProducts = mutableListOf<Product>()
    val checkedPositions = listView.checkedItemPositions

    for (i in 0 until checkedPositions.size()) {
        val position = checkedPositions.keyAt(i)
        if (checkedPositions.valueAt(i)) {
            selectedProducts.add(products[position])
        }
    }

    if (selectedProducts.isEmpty()) {
        Toast.makeText(this, "Ничего не выбрано",
            ↪ Toast.LENGTH_SHORT).show()
    } else {
        val totalPrice = selectedProducts.sumOf { it.price }
        val selectedStrings = ArrayList(selectedProducts.map {
            ↪ it.toString() })

        val intent = Intent(this, SelectedItemsActivity::class.java)
        intent.putStringArrayListExtra("selected_items",
            ↪ selectedStrings)
        intent.putExtra("total_price", totalPrice)
        startActivity(intent)
    }
}
}
}

```

В классе MainActivity реализована следующая функциональность:

- создание списка товаров (products);
- инициализация ListView и установка собственного адаптера ProductAdapter;
- обработка нажатия кнопки "Показать выбранные" через механизм обратного вызова (setOnClickListener);
- получение выбранных элементов списка через checkedItemPositions;
- передача выбранных товаров во второе Activity через Intent с использованием методов putStringArrayListExtra() и putExtra();
- отображение сообщения через Toast, если ничего не выбрано.

0.3.4 Ниже представлен код класса SelectedItemsActivity, содержащего логику работы второго экрана приложения.

```
package com.example.lt3
```



```

import android.os.Bundle
import android.widget.ArrayAdapter
import android.widget.Button
import android.widget.ListView
import androidx.activity.ComponentActivity

class SelectedItemsActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.selected_items)

        val selectedListView = findViewById<ListView>(R.id.selectedListView)
        val button = findViewById<Button>(R.id.button)

        val selectedItems = intent.getStringArrayListExtra("selected_items")
            ↪ ?: arrayListOf()

        val adapter = ArrayAdapter(
            this,
            android.R.layout.simple_list_item_1,
            selectedItems
        )

        selectedListView.adapter = adapter

        button.setOnClickListener {
            finish()
        }
    }
}

```

В классе `SelectedItemsActivity` реализована следующая функциональность:

- получение данных из `Intent` через метод `getStringArrayListExtra()`;
- создание и установка `ArrayAdapter` для отображения выбранных товаров;
- обработка нажатия кнопки "Назад" через механизм обратного вызова (`setOnClickListener`), который вызывает метод `finish()` для закрытия `Activity`.

0.4 Ответы на контрольные вопросы

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

Для создания View-элемента из содержимого layout-файла

используется класс `LayoutInflater` и его метод `inflate()`. Пример использования:

```
val view = LayoutInflater.from(context)
    .inflate(R.layout.list_item, parent, false)
```

Метод `inflate()` принимает следующие параметры:

- ID ресурса layout-файла (`R.layout.list_item`);
- родительский контейнер (`parent`);
- флаг, указывающий, нужно ли добавлять созданный View в родительский контейнер (`false` – не добавлять сразу).

Это необходимо в следующих случаях:

- при создании собственных адаптеров для списков (в методе `getView()`);
- при динамическом создании View-элементов в коде;
- при создании кастомных диалогов и всплывающих окон;
- при работе с Fragment и программным созданием их интерфейсов.

2. Как создать и обеспечить работу списка `ListView`?

Для создания и обеспечения работы списка `ListView` необходимо выполнить следующие шаги:

1. Добавить `ListView` в XML-разметку Activity:

```
<ListView
    android:id="@+id/listView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:choiceMode="multipleChoice" />
```

2. Создать или использовать готовый адаптер (например, `ArrayAdapter` или собственный адаптер, наследующийся от `BaseAdapter`):

```
val adapter = ProductAdapter(products)
```

3. Установить адаптер для `ListView`:

```
listView.adapter = adapter
```

4. (Опционально) Установить обработчик событий для элементов списка:

```
listView.setOnItemClickListener { parent, view, position, id ->
    // Обработка нажатия на элемент
}
```

3. Как реализовать собственный кастомизированный список?

Для реализации собственного кастомизированного списка необходимо:

1. Создать класс адаптера, наследующийся от `BaseAdapter` или `ArrayAdapter`:

```
class ProductAdapter(private val products: List<Product>) :  
    ↳ BaseAdapter() {  
        // Реализация методов  
    }
```

2. Переопределить необходимые методы адаптера:
 - `getCount()` — возвращает количество элементов;
 - `getItem(position)` — возвращает элемент по позиции;
 - `getItemId(position)` — возвращает ID элемента;
 - `getView(position, convertView, parent)` — создает `View` для элемента.
3. Создать XML-разметку для элемента списка (например, `list_item.xml`).
4. В методе `getView()` использовать `LayoutInflater` для создания `View` из layout-файла и заполнения его данными.
5. Использовать механизм переиспользования `View` через параметр `convertView` для оптимизации производительности.

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

Адаптер в Android-приложениях — это класс, который связывает данные с `View`-элементами списка (`ListView`, `RecyclerView`, `Spinner` и др.). Адаптер преобразует данные из источника (массив, список, база данных) в `View`-элементы, которые отображаются на экране.

Основные функции адаптера:

- предоставление данных для отображения в списке;
- создание и настройка `View`-элементов для каждого элемента списка;
- управление переиспользованием `View`-элементов для оптимизации памяти и производительности;
- обеспечение связи между данными и их визуальным представлением.

В Android используются следующие типы адаптеров:

- `ArrayAdapter` — простой адаптер для работы с массивами и списками строк;
- `BaseAdapter` — базовый класс для создания собственных адаптеров;
- `CursorAdapter` — адаптер для работы с данными из базы данных через `Cursor`;

- `RecyclerView.Adapter` — адаптер для работы с `RecyclerView` (современный подход).

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

При создании кастомизированных списков на основе `BaseAdapter` необходимо переопределить следующие методы:

- `getCount() : Int` — возвращает количество элементов в списке. Этот метод определяет, сколько элементов будет отображено.
- `getItem(position: Int) : Any` — возвращает объект данных по указанной позиции. Используется для получения данных элемента списка.
- `getItemId(position: Int) : Long` — возвращает уникальный идентификатор элемента по позиции. Обычно возвращает позицию как ID.
- `getView(position: Int, convertView: View?, parent: ViewGroup) : View` — самый важный метод, который создает или переиспользует `View` для отображения элемента списка. В этом методе происходит:
 - создание `View` из layout-файла (если `convertView == null`);
 - переиспользование существующего `View` (если `convertView != null`);
 - заполнение `View` данными из источника;
 - возврат настроенного `View`.

6. Для чего нужен метод `getView` в адаптере?

Метод `getView()` в адаптере выполняет следующие функции:

- **Создание View-элементов** — создает `View` для каждого элемента списка на основе layout-файла с помощью `LayoutInflater`.
- **Переиспользование View** — оптимизирует использование памяти путем переиспользования уже созданных `View`-элементов через параметр `convertView`. Это критически важно для производительности при работе с большими списками.
- **Заполнение данными** — связывает данные из источника (массив, список) с `View`-элементами, устанавливая тексты, изображения и другие свойства.
- **Настройка внешнего вида** — позволяет кастомизировать внешний вид каждого элемента списка в зависимости от данных или позиции.

Пример реализации:

```
override fun getView(position: Int, convertView: View?, parent: ViewGroup):  
    View {  
        val view: CheckedTextView = if (convertView == null) {  
            LayoutInflater.from(parent.context)  
                .inflate(R.layout.list_item, parent, false) as CheckedTextView  
        } else {  
            convertView as CheckedTextView  
        }  
  
        val product = getItem(position)  
        view.text = product.toString()  
  
        return view  
    }
```

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

Header (заголовок) в списках — это специальный элемент, который отображается в начале списка и обычно содержит статическую информацию или элементы управления.

Для реализации Header в `ListView` можно использовать следующие подходы:

1. Использование метода `addHeaderView()`:

```
val headerView = LayoutInflater.from(this)  
    .inflate(R.layout.header_layout, listView, false)  
listView.addHeaderView(headerView)
```

2. Создание кастомного адаптера с поддержкой Header:

- добавить специальный тип элемента в адаптер для Header;
- в методе `getViewTypeCount()` вернуть количество типов (обычно 2: Header и обычный элемент);
- в методе `getItemViewType(position)` определить тип элемента;
- в методе `getView()` создавать разные View в зависимости от типа.

3. Использование `RecyclerView` с `ConcatAdapter` или кастомным адаптером, который объединяет несколько адаптеров.

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

Footer (подвал) в списках — это специальный элемент, который отображается в конце списка и обычно содержит статическую информацию, элементы управления или индикатор загрузки.

Для реализации Footer в `ListView` можно использовать следующие подходы:

1. Использование метода `addFooterView()`:

```
val footerView = LayoutInflater.from(this)
    .inflate(R.layout.footer_layout, listView, false)
listView.addFooterView(footerView)
```

2. Создание кастомного адаптера с поддержкой Footer:

- добавить специальный тип элемента в адаптер для Footer;
- в методе `getViewTypeCount()` вернуть количество типов;
- в методе `getItemViewType(position)` определить тип элемента (обычный элемент или Footer);
- в методе `getView()` создавать разные View в зависимости от типа.

3. Использование `RecyclerView` с поддержкой Footer через кастомный адаптер или библиотеки.

Важно: Методы `addHeaderView()` и `addFooterView()` должны вызываться до установки адаптера для `ListView`.

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

В Android-приложениях используются следующие механизмы обратного вызова для обработки событий:

1. Анонимные классы (Anonymous Classes):

```
button.setOnClickListener(object : View.OnClickListener {
    override fun onClick(v: View?) {
        // Обработка события
    }
})
```

2. Lambda-выражения (современный подход в Kotlin):

```
button.setOnClickListener {
    // Обработка события
}
```

3. Реализация интерфейса в Activity/Fragment:

```
class MainActivity : ComponentActivity(), View.OnClickListener {
    override fun onClick(v: View?) {
        when (v?.id) {
            R.id.button -> { /* обработка */ }
        }
    }
}
```

4. Ссылки на методы (Method References):

```
button.setOnClickListener(this::onButtonClick)
```

```
fun onClick(view: View) {
    // Обработка события
}
```

5. Именованные классы-слушатели:

```
class MyClickListener : View.OnClickListener {
    override fun onClick(v: View?) {
        // Обработка события
    }
}
button.setOnClickListener(MyClickListener())
```

6. Обработчики событий для списков:

```
listView.setOnItemClickListener { parent, view, position, id ->
    // Обработка нажатия на элемент списка
}
```

7. Lifecycle Callbacks для Activity и Fragment:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Инициализация
}
```

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

Для передачи динамического массива объектов из одного Activity в другое с помощью Intent можно использовать следующие подходы:

1. Передача ArrayList строк (если объекты можно сериализовать в строки):

```
// В первом Activity
val selectedStrings = ArrayList(selectedProducts.map { it.toString()
    → })
val intent = Intent(this, SecondActivity::class.java)
intent.putStringArrayListExtra("selected_items", selectedStrings)
startActivity(intent)

// Во втором Activity
val selectedItems = intent.getStringArrayListExtra("selected_items")
    → ?: arrayListOf()
```

2. Передача массива примитивных типов:

```
// Передача массива целых чисел
intent.putExtra("numbers", intArrayOf(1, 2, 3))

// Получение
val numbers = intent.getIntArrayExtra("numbers")
```

3. Использование Parcelable (для сложных объектов):

```
// Класс должен реализовывать интерфейс Parcelable
data class Product(
```

```

        val name: String,
        val price: Double
    ) : Parcelable {
        // Реализация Parcelable
    }

    // Передача ArrayList объектов
    val intent = Intent(this, SecondActivity::class.java)
    intent.putParcelableArrayListExtra("products", ArrayList(products))
    startActivity(intent)

    // Получение
    val products =
        ↪ intent.getParcelableArrayListExtra<Product>("products")

```

4. Использование **Serializable** (альтернатива **Parcelable**, но медленнее):

```

// Класс должен реализовывать интерфейс Serializable
data class Product(
    val name: String,
    val price: Double
) : Serializable

// Передача
intent.putExtra("products", ArrayList(products) as Serializable)

// Получение
val products = intent.getSerializableExtra("products") as?
    ↪ ArrayList<Product>

```

5. Передача через **Bundle**:

```

val bundle = Bundle()
bundle.putStringArrayList("items", selectedStrings)
intent.putExtras(bundle)

```

Примечание: В примере приложения MiniShop используется первый подход — передача `ArrayList<String>` через `putStringArrayListExtra()`, так как объекты `Product` преобразуются в строки через метод `toString()`.

0.5 Заключение

В ходе выполнения лабораторной работы было разработано Android-приложение MiniShop, реализующее функциональность магазина товаров с возможностью выбора товаров из списка и просмотра выбранных товаров.

Были изучены и применены следующие технологии и подходы:

- создание собственного адаптера на основе `BaseAdapter` для кастомизированного отображения списка товаров;

- работа с `ListView` и механизмами множественного выбора элементов;
- реализация механизмов обратного вызова через `setOnClickListener` для обработки событий нажатия кнопок;
- передача данных между `Activity` через `Intent` с использованием методов `putStringArrayListExtra()` и `putExtra()`;
- использование `LayoutInflater` для создания `View`-элементов из `layout`-файлов;
- оптимизация производительности списков через механизм переиспользования `View` в методе `getView()`.

Приложение успешно демонстрирует работу со списками, использование собственного адаптера и механизмов обратного вызова для обработки событий в Android-приложениях.