Практическая работа №10

Работа в многопоточном режиме

1. Что такое многопоточность?

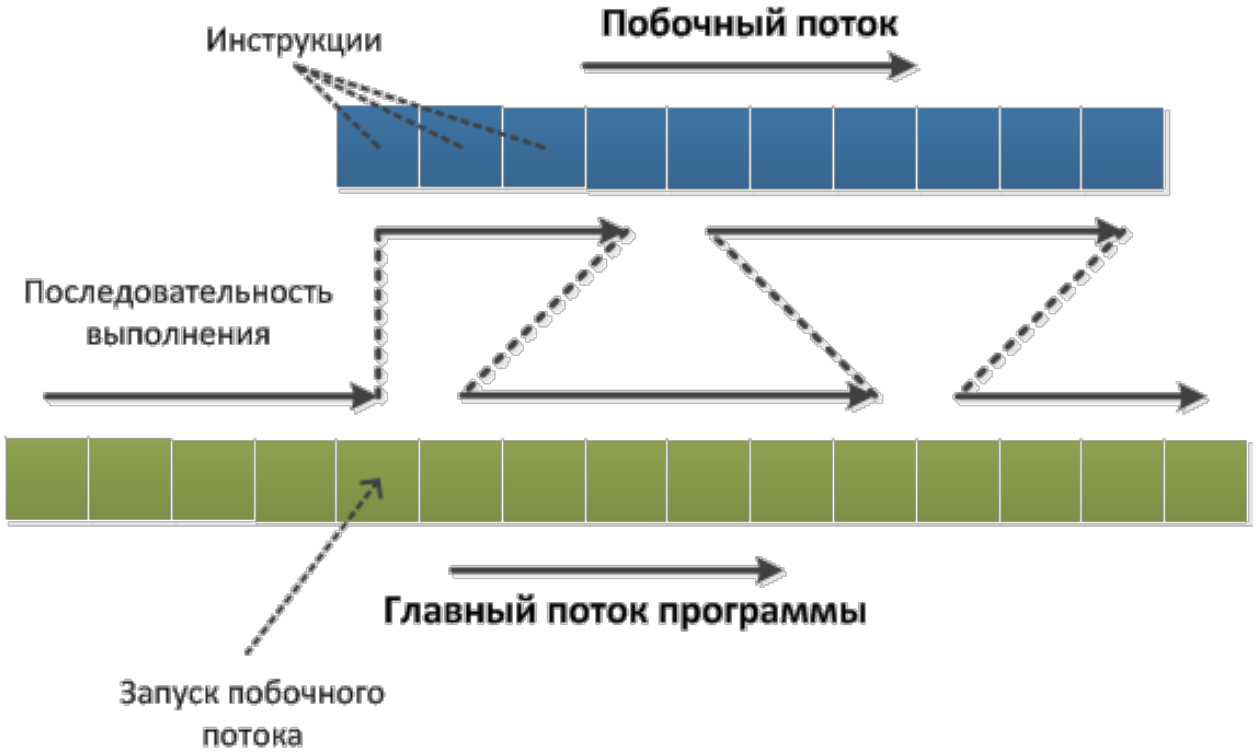
Большинство языков программирования поддерживают такую важную функциональность как многопоточность, и Java в этом плане не исключение. При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому большинство реальных приложений, которые многим из нас приходится использовать, практически не мыслимы без многопоточности.

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

Следует отдельно обговорить фразу «параллельно с другими потоками». Известно, что на одно ядро процессора, в каждый момент времени, приходится одна единица исполнения. То есть одноядерный процессор может обрабатывать команды только последовательно, по одной за раз (в упрощенном случае). Однако запуск нескольких параллельных потоков возможен и в системах с одноядерными процессорами. В этом случае система будет периодически переключаться между потоками, поочередно давая выполняться то одному, то другому потоку. Такая схема называется псевдо-параллелизмом. Система запоминает состояние (контекст) каждого потока, перед тем как переключиться на другой поток, и восстанавливает его по возвращению к выполнению потока. В контекст потока входят такие параметры, как стек, набор значений регистров процессора, адрес исполняемой команды и прочее…

Проще говоря, при псевдопараллельном выполнении потоков процессор мечется между выполнением нескольких потоков, выполняя по очереди часть каждого из них.

Вот как это выглядит:



Цветные квадраты на рисунке – это инструкции процессора (зеленые – инструкции главного потока, синие – побочного). Выполнение идет слева направо. После запуска побочного потока его инструкции начинают выполняться вперемешку с инструкциями главного потока. Кол-во выполняемых инструкций за каждый подход не определено.

То, что инструкции параллельных потоков выполняются вперемешку, в некоторых случаях может привести к конфликтам доступа к данным.

2. Многопоточность в Java

2.1. Thread и Runnable

В Java функциональность отдельного потока заключается в классе Thread. И чтобы создать новый поток, нам надо создать объект этого класса. Но все потоки не создаются сами по себе. Когда запускается программа, начинает работать главный поток этой программы. От этого главного потока порождаются все остальные дочерние потоки.

Далее мы рассмотрим, как создавать и использовать потоки. Это довольно легко. Однако при создании многопоточного приложения нам следует учитывать ряд обстоятельств, которые негативно могут сказаться на работе приложения.

На некоторых платформах запуск новых потоков может замедлить работу приложения. Что может иметь большое значение, если нам критичная производительность приложения.

Для каждого потока создается свой собственный стек в памяти, куда помещаются все локальные переменные и ряд других данных, связанных с выполнением потока. Соответственно, чем больше потоков создается, тем больше памяти используется. При этом надо помнить, в любой системе размеры используемой памяти ограничены. Кроме того, во многих системах может быть ограничение на количество потоков. Но даже если такого ограничения нет, то в любом случае имеется естественное ограничение в виде максимальной скорости процессора.

Когда мы запускаем приложение на Android, система создает поток, который называется основным потоком приложения или UI-поток. Этот поток обрабатывает все изменения и события пользовательского интерфейса. Однако для вспомогательных операций, таких как отправка или загрузка файла, продолжительные вычисления и т.д., мы можем создавать дополнительные потоки.

Для создания новых потоков нам доступен стандартный функционал класса Thread из базовой библиотеки Java из пакета java.util.concurrent,

которые особой трудности не представляют. Тем не менее трудности могут возникнуть при обновлении визуального интерфейса из потока.

Например, создадим простейшее приложение с использованием потоков. Определим следующую разметку интерфейса в файле activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:textSize="22sp"
        app:layout_constraintBottom_toTopOf="@id/button"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Запустить поток"
        app:layout_constraintTop_toBottomOf="@id/textView"
        app:layout_constraintLeft_toLeftOf="parent"  />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Здесь определена кнопка для запуска фонового потока, а также текстовое поле для отображения некоторых данных, которые будут генерироваться в запущенном потоке.

Далее определим в классе MainActivity следующий код:

```java
package com.example.threadapp;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import java.util.Calendar;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TextView textView = findViewById(R.id.textView);
        Button button = findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Определяем объект Runnable
                Runnable runnable = new Runnable() {
                    @Override
                    public void run() {
                        // получаем текущее время
                        Calendar c = Calendar.getInstance();
                        int hours = c.get(Calendar.HOUR_OF_DAY);
                        int minutes = c.get(Calendar.MINUTE);
                        int seconds = c.get(Calendar.SECOND);
                        String time = hours + ":" + minutes + ":" + seconds;
                        // отображаем в текстовом поле
                        textView.setText(time);
                    }
                };
                // Определяем объект Thread - новый поток
                Thread thread = new Thread(runnable);
                // Запускаем поток
                thread.start();
            }
        });
    }
}
```

Итак, здесь к кнопке прикреплен обработчик нажатия, который запускает новый поток. Создавать и запускать поток в Java можно различными способами. В данном случае сами действия, которые выполняются в потоке, определяются в методе run() объекта Runnable:

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // получаем текущее время
        Calendar c = Calendar.getInstance();
        int hours = c.get(Calendar.HOUR_OF_DAY);
        int minutes = c.get(Calendar.MINUTE);
        int seconds = c.get(Calendar.SECOND);
        String time = hours + ":" + minutes + ":" + seconds;
        // отображаем в текстовом поле
        textView.setText(time);
    }
};
```

Для примера получаем текущее время и пытаемся отобразить его в элементе TextView.

Далее определяем объект потока - объект Thread, который принимает объект Runnable. И с помощью метода start() запускаем поток:

```
// Определяем объект Thread - новый поток
Thread thread = new Thread(runnable);
// Запускаем поток
thread.start();
```

Вроде ничего сложного. Но если мы запустим приложение и нажмем на кнопку, то мы столкнемся с ошибкой:

```
2020-12-24 18:46:49.083 19491-19536/com.example.threadapp E/AndroidRuntime: FATAL EXCEPTION: Thread-2
    Process: com.example.threadapp, PID: 19491
    android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.
        at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:7313)
        at android.view.ViewRootImpl.requestLayout(ViewRootImpl.java:1161)
        at android.view.View.requestLayout(View.java:21995)
        at android.view.View.requestLayout(View.java:21995)
        at android.view.View.requestLayout(View.java:21995)
        at android.view.View.requestLayout(View.java:21995)
        at android.view.View.requestLayout(View.java:21995)
        at android.view.View.requestLayout(View.java:21995)
        at androidx.constraintlayout.widget.ConstraintLayout.requestLayout(ConstraintLayout.java:3239)
        at android.view.View.requestLayout(View.java:21995)
        at android.widget.TextView.checkForRelayout(TextView.java:8531)
        at android.widget.TextView.setText(TextView.java:5394)
        at android.widget.TextView.setText(TextView.java:5250)
        at android.widget.TextView.setText(TextView.java:5207)
        at com.example.threadapp.MainActivity$1$1.run(MainActivity.java:34)
        at java.lang.Thread.run(Thread.java:764)
```

Поскольку изменять состояние визуальных элементов, обращаться к ним мы можем только в основном потоке приложения или UI-потоке.

Для решения этой проблемы - взаимодействия во вторичных потоках с элементами графического интерфейса класс View() определяет метод post(). В качестве параметра он принимает задачу, которую надо выполнить, и возвращает логическое значение - true, если задача Runnable успешно помещена в очередь сообщение, или false, если не удалось разместить в очереди.

Так, изменим код MainActivity следующим образом

```
package com.example.threadapp;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import java.util.Calendar;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TextView textView = findViewById(R.id.textView);
        Button button = findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Определяем объект Runnable
                Runnable runnable = new Runnable() {
                    @Override
                    public void run() {
                        // получаем текущее время
                        Calendar c = Calendar.getInstance();
                        int hours = c.get(Calendar.HOUR_OF_DAY);
                        int minutes = c.get(Calendar.MINUTE);
                        int seconds = c.get(Calendar.SECOND);
                        String time = hours + ":" + minutes + ":" + seconds;
                        // отображаем в текстовом поле
                        textView.post(new Runnable() {
                            public void run() {
                                textView.setText(time);
                            }
                        });
                    }
                };
                // Определяем объект Thread - новый поток
                Thread thread = new Thread(runnable);
                // Запускаем поток
                thread.start();
            }
        });
    }
}
```

Теперь для обновления TextView применяется метод post:

```
textView.post(new Runnable() {
        public void run() {
            textView.setText(time);
    }
});
```

То есть здесь в методе run() передаемого в метод post() объекта Runnable мы можем обращаться к элементам визуального интерфейса и взаимодействовать с ними. Подобным образом можно работать и с другими виджетами, которые наследуются от класса View.

При использовании вторичных потоков следует учитывать следующий момент. Более оптимальным способом является работа потоков с фрагментом, нежели непосредственно с activity. Например, определим в файле activity_main.xml следующий интерфейс:

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/progressBtn"
        android:text="Запуск"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toTopOf="@id/statusView"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>

    <TextView
        android:id="@+id/statusView"
        android:text="Статус"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toTopOf="@id/indicator"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toBottomOf="@id/progressBtn" />
    <ProgressBar
        android:id="@+id/indicator"
        style="@android:style/Widget.ProgressBar.Horizontal"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:max="100"
        android:progress="0"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@id/statusView"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Здесь определена кнопка для запуска вторичной задачи и элементы TextView и ProgressBar, которые отображают индикацию выполнения задачи.

В классе MainActivity определим следующий код:

```java
package com.example.threadapp;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.ProgressBar;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    int currentValue = 0;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ProgressBar indicatorBar = findViewById(R.id.indicator);
        TextView statusView = findViewById(R.id.statusView);
        Button btnFetch = findViewById(R.id.progressBtn);
        btnFetch.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

                Runnable runnable = new Runnable() {
                    @Override
                    public void run() {

                        for(; currentValue <= 100; currentValue++){
                            try {
                                statusView.post(new Runnable() {
                                    public void run() {
                                        indicatorBar.setProgress(currentValue);
                                        statusView.setText("Статус: " + currentValue);
                                    }
                                });

                                Thread.sleep(400);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                    }
                };
                Thread thread = new Thread(runnable);
                thread.start();
            }
        });
    }
}
```

Здесь по нажатию кнопки мы запускаем задачу Runnable, в которой в цикле от 0 до 100 изменяем показатели ProgressBar и TextView, имитируя некоторую долгую работу.

Однако если в процессе работы задачи мы изменим ориентацию мобильного устройства, то произойдет пересоздание activity, и приложение перестанет работать должным образом.

В данном случае проблема упирается в состояние, которым оперирует поток, а именно - переменную currentValue, к значению которой привязаны виджеты в Activity.

Для подобных случаев в качестве решения проблемы предлагается использовать ViewModel. Итак, добавим в ту же папку, где находится файл MainActivity.java, новый класс MyViewModel со следующим кодом:

```
package com.example.threadapp;

import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class MyViewModel extends ViewModel {

    private MutableLiveData<Boolean> isStarted = new MutableLiveData<Boolean>(false);
    private MutableLiveData<Integer> value;
    public LiveData<Integer> getValue() {
        if (value == null) {
            value = new MutableLiveData<Integer>(0);
        }
        return value;
    }
    public void execute(){

        if(!isStarted.getValue()){
            isStarted.postValue(true);
            Runnable runnable = new Runnable() {
                @Override
                public void run() {

                    for(int i = value.getValue();  i <= 100; i++){
                        try {
                            value.postValue(i);
                            Thread.sleep(400);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            };
            Thread thread = new Thread(runnable);
            thread.start();
        }
    }
}
```

Итак, здесь определен класс MyViewModel, который унаследован от класса ViewModel, специально предназначенного для хранения и управления состоянием или моделью.

В качестве состояния здесь определены для объекта. В первую очередь, это числовое значение, к которым будут привязан виджеты MainActivity. И во-

вторых, нам нужен некоторый индикатор того, что поток уже запущен, чтобы по нажатию на кнопку не было запущено лишних потоков.

Для хранения числового значения предназначена переменная value. Для привязки к этому значению оно имеет тип MutableLiveData. А поскольку мы будем хранить в этой переменной числовое значение, то тип переменной типизирован типом Integer.

Для доступа извне класса к этому значению определен метод getValue, который имеет тип LiveData и который при первом обращении к переменной устанавливает 0, либо просто возвращает значение переменной.

Для индикации, запущен ли поток, определена переменная isStarted, которая хранит значение типа Boolean, то есть фактически true или false. По умолчанию она имеет значение false (то есть поток не запущен).

Для изменения числового значения, к которому будут привязаны виджеты, определен метод execute(). Он запускает поток, если поток не запущен. Далее переключает значение переменной isStarted на true, поскольку мы запускаем поток. И в данном случае мы пользуемся преимуществом класса ViewModel, который позволяет автоматически сохранять свое состояние. Причем счетчик цикла в качестве начального значения берет значение из переменной value и увеличается на единицу, пока не дойдет до ста.

В самом цикле изменяется значение переменной value с помощью передачи значения в метод postValue().

В потоке также запускается цикл. Таким образом, в цикле осуществится проход от 0 до 100, и при каждой итерации цикла будет изменяться значение переменной value.

Теперь задействуем наш класс MyViewModel и для этого изменим код класса MainActivity:

```java
package com.example.threadapp;

import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProvider;

import android.os.Bundle;
import android.widget.Button;
import android.widget.ProgressBar;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ProgressBar indicatorBar = findViewById(R.id.indicator);
        TextView statusView = findViewById(R.id.statusView);
        Button btnFetch = findViewById(R.id.progressBtn);
        MyViewModel model = new ViewModelProvider(this).get(MyViewModel.class);

        model.getValue().observe(this, value -> {
            indicatorBar.setProgress(value);
            statusView.setText("Статус: " + value);
        });
        btnFetch.setOnClickListener(v -> model.execute());
    }
}
```

2.2. ExecutorService

ExecutorService — это класс JDK, упрощающий выполнение задач в асинхронном режиме. Вообще говоря, ExecutorService автоматически предоставляет пул потоков и API для назначения ему задач.

Самый простой способ создать *ExecutorService* — использовать один из фабричных методов класса *Executors*.

Например, следующая строка кода создаст пул из 10 потоков:

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

Существует несколько других фабричных методов для создания предопределенной *службы ExecutorService*, соответствующей конкретным случаям использования. Чтобы найти лучший метод для разных нужд, лучше обратиться к официальной документации Oracle.

Поскольку *ExecutorService* — это интерфейс, можно использовать экземпляр любой его реализации. *В пакете java.util.concurrent* есть несколько реализаций на выбор , или вы можете создать свою собственную.

Например, у класса *ThreadPoolExecutor* есть несколько конструкторов, которые мы можем использовать для настройки службы-исполнителя и ее внутреннего пула:

```
ExecutorService executorService =
  new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
  new LinkedBlockingQueue<Runnable>());
```

Вы можете заметить, что приведенный выше код очень похож на исходный код фабричного метода *newSingleThreadExecutor()*. В большинстве случаев детальная ручная настройка не требуется.

ExecutorService может выполнять задачи *Runnable* и *Callable*. Для простоты будут использоваться две примитивные задачи. Обратите внимание, что здесь мы используем лямбда-выражения вместо анонимных внутренних классов:

```
Runnable runnableTask = () -> {
    try {
        TimeUnit.MILLISECONDS.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Callable<String> callableTask = () -> {
    TimeUnit.MILLISECONDS.sleep(300);
    return "Task's execution";
};

List<Callable<String>> callableTasks = new ArrayList<>();
callableTasks.add(callableTask);
callableTasks.add(callableTask);
callableTasks.add(callableTask);
```

Мы можем назначать задачи ExecutorService, *используя* несколько методов, включая *execute()* , который унаследован от интерфейса *Executor* , а также *submit()* , *invokeAny()* и *invokeAll()*.

Метод ***execute()*** недействителен и не дает никакой возможности получить результат выполнения задачи или проверить статус задачи (выполняется ли она) *:*

```
executorService.execute(runnableTask);
```

submit() отправляет *задачу Callable* или *Runnable* в *ExecutorService* и возвращает результат типа *Future* :

```
Future<String> future =
  executorService.submit(callableTask);
```

invokeAny() присваивает набор задач ExecutorService , вызывая выполнение каждой из них, и возвращает результат успешного выполнения одной задачи (если было успешное выполнение):

```
String result = executorService.invokeAny(callableTasks);
```

invokeAll() *присваивает ExecutorService* набор задач, вызывая выполнение каждой из них, и возвращает результат выполнения всех задач в виде списка объектов типа *Future*:

```
List<Future<String>> futures = executorService.invokeAll(callableTasks);
```

Прежде чем идти дальше, нам нужно обсудить еще два вопроса: закрытие ExecutorService *и* работа с типами возврата *Future*.

Как правило, *ExecutorService* не будет автоматически уничтожен, если нет задачи для обработки. Он останется в живых и будет ждать новой работы.

В некоторых случаях это очень полезно, например, когда приложению необходимо обрабатывать задачи, которые появляются нерегулярно, или количество задач неизвестно во время компиляции.

С другой стороны, приложение может достичь своего завершения, но не быть остановленным, потому что ожидающий *ExecutorService* заставит JVM продолжать работу.

Чтобы правильно закрыть ExecutorService , у нас есть API-интерфейсы *shutdown()* и *shutdownNow()*.

Метод shutdown***()*** не вызывает немедленного уничтожения *ExecutorService*. Это заставит *ExecutorService* прекратить принимать новые задачи и закрыться после того, как все запущенные потоки закончат свою текущую работу:

```
executorService.shutdown();
```

Метод ***shutdownNow()*** пытается немедленно уничтожить ExecutorService , но не гарантирует, что все запущенные потоки будут остановлены одновременно:

```
List<Runnable> notExecutedTasks = executorService.shutDownNow();
```

Этот метод возвращает список задач, ожидающих обработки. Разработчик сам решает, что делать с этими задачами.

Один хороший способ закрыть ExecutorService *(*который также рекомендуется Oracle) — использовать оба этих метода в сочетании с методом ***awaitTermination()***:

```
executorService.shutdown();
try {
    if (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {
        executorService.shutdownNow();
    }
} catch (InterruptedException e) {
    executorService.shutdownNow();
}
```

При таком подходе *ExecutorService* сначала перестанет принимать новые задачи, а затем будет ждать до указанного периода времени, пока все задачи будут выполнены. Если это время истекает, выполнение немедленно останавливается.

3. Многопоточность в Android

3.1. AsyncTask

Класс AsyncTask предлагает простой и удобный механизм для перемещения трудоёмких операций в фоновый поток. Благодаря ему вы получаете удобство синхронизации обработчиков событий с графическим потоком, что позволяет обновлять элементы пользовательского интерфейса для отчета о ходе выполнения задачи или для вывода результатов, когда задача завершена.

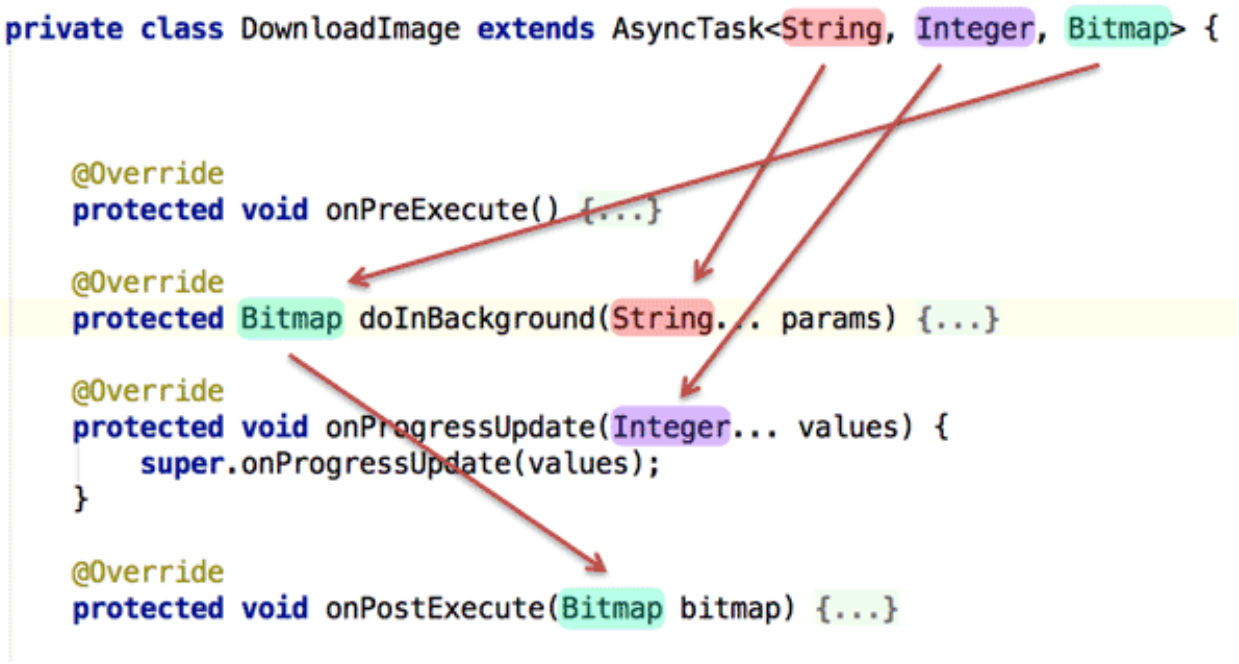
Следует помнить, что AsyncTask не является универсальным решением для всех случаев жизни. Его следует использовать для не слишком продолжительных операций - загрузка небольших изображений, файловые операции, операции с базой данных и т.д.

Напрямую с классом AsyncTask работать нельзя, так как это абстрактный класс, вместо этого нужно наследоваться от него при помощи ключевого слова extends. Реализация должна предусматривать классы для объектов, которые будут переданы в качестве параметров методу execute(), для переменных, что станут использоваться для оповещения о ходе выполнения, а также для переменных, где будет храниться результат. Формат такой записи следующий:

```
AsyncTask<[Input_Parameter Type], [Progress_Report Type], [Result Type]>
```

Если не нужно принимать параметры, обновлять информацию о ходе выполнения или выводить конечный результат, просто укажите тип Void во всех трёх случаях. В параметрах можно использовать только обобщённые типы, т.е. вместо int необходимо использовать Integer и т.п.

Для лучшего запоминания связи дженерик классов и передаваемых данных можно обратиться к следующей схеме:



Каркас реализации AsyncTask, в котором используются строковой параметр и два целочисленных значения, нужных для оповещения о выполнении работы и о конечном результате:

```
private class MyAsyncTask extends AsyncTask<String, Integer, Integer> {
    @Override
    protected Integer doInBackground(String... parameter) {
        int myProgress = 0;
        // [... Выполните задачу в фоновом режиме, обновите переменную myProgress...]
        publishProgress(myProgress);
        // [... Продолжение выполнения фоновой задачи ...]
        // Верните значение, ранее переданное в метод onPostExecute
        return result;
    }

    @Override
    protected void onProgressUpdate(Integer... progress) {
        // [... Обновите индикатор хода выполнения, уведомления или другой
        // элемент пользовательского интерфейса ...]
    }

    @Override
    protected void onPostExecute(Integer... result) {
        // [... Сообщите о результате через обновление пользовательского
        // интерфейса, диалоговое окно или уведомление ...]
    }
}
```

У AsyncTask есть несколько основных методов, которые нужно освоить в первую очередь. Обязательным является метод doInBackground(), остальные используются исходя из логики вашего приложения.

- doInBackground() – основной метод, который выполняется в новом потоке. Он не имеет доступа к UI. Именно в этом методе должен находиться код для тяжёлых задач. Принимает набор параметров тех типов, которые определены в реализации вашего класса. Этот метод выполняется в фоновом потоке, поэтому в нем не должно быть никакого взаимодействия с элементами пользовательского интерфейса. Здесь размещается трудоёмкий код, а так же при помощи метода publishProgress() позволяется передавать изменения в пользовательский интерфейс посредством передачи данных обработчику onProgressUpdate(). Когда фоновая задача завершена, данный метод возвращает конечный результат для обработчика onPostExecute(), который сообщит о нём в поток пользовательского интерфейса.
- onPreExecute() – выполняется перед doInBackground(). Он имеет доступ к UI.
- onPostExecute() – выполняется после doInBackground(), однако может не вызываться, если AsyncTask был отменен. Имеет доступ к UI. Его используют для обновления пользовательского интерфейса, как только фоновая задача завершена. Данный обработчик при вызове синхронизируется с потоком GUI, поэтому внутри него вы можете безопасно изменять элементы пользовательского интерфейса.
- onProgressUpdate(). Имеет доступ к UI. Переопределение этого обработчика позволяет публиковать промежуточные обновления в пользовательский интерфейс. При вызове он синхронизируется с потоком GUI, поэтому в нём вы можете безопасно изменять элементы пользовательского интерфейса.

- publishProgress() - можно вызвать в doInBackground() для показа промежуточных результатов в onProgressUpdate()
- cancel() - отмена задачи
- onCancelled() - Имеет доступ к UI. Задача была отменена. Имеются две перегруженные версии.

3.2. Worker

WorkManager является частью Android Jetpack и компонентом архитектуры для фоновой работы, требующей сочетания оппортунистического и гарантированного выполнения. Оппортунистическое выполнение означает, что WorkManager выполнит вашу фоновую работу, как только сможет. Гарантированное выполнение означает, что WorkManager позаботится о логике запуска вашей работы в различных ситуациях, даже если вы уйдете из своего приложения.

WorkManager — простая, но невероятно гибкая библиотека, обладающая множеством дополнительных преимуществ. К ним относятся:

- Поддержка как асинхронных разовых, так и периодических задач
- Поддержка ограничений, таких как условия сети, объем памяти и состояние зарядки.
- Цепочка сложных рабочих запросов, включая параллельную работу
- Выходные данные одного рабочего запроса используются в качестве входных данных для следующего
- Обеспечивает совместимость уровня API с уровнем API 14 (см. примечание).
- Работает с сервисами Google Play или без них
- Соответствует рекомендациям по обеспечению работоспособности системы
- Поддержка LiveData для удобного отображения состояния рабочего запроса в пользовательском интерфейсе.

WorkManager работает поверх нескольких API, таких как JobScheduleri AlarmManager. WorkManager выбирает правильные API для использования, основываясь на таких условиях, как уровень API пользовательского устройства.

Библиотека WorkManager — хороший выбор для задач, которые полезно выполнять, даже если пользователь уходит с определенного экрана или вашего приложения.

Некоторые примеры задач, которые хорошо использовать WorkManager:

- Загрузка журналов
- Применение фильтров к изображениям и сохранение изображения
- Периодическая синхронизация локальных данных с сетью

WorkManager предлагает гарантированное выполнение, и не все задачи требуют этого. Таким образом, это не универсальное решение для запуска каждой задачи вне основного потока. Дополнительные сведения о том, когда использовать WorkManager.

Для корректной работы WorkManager требуется подключение зависимости в gradle.

```
dependencies {
    // WorkManager dependency
    implementation "androidx.work:work-runtime:$versions.work"
}
```

Есть несколько классов WorkManager, о которых вам нужно знать:

- Worker – здесь вы размещаете код для фактической работы, которую хотите выполнять в фоновом режиме. Вы расширите этот класс и переопределите doWork() метод.
- WorkRequest – представляет запрос на выполнение некоторой работы. Вы передадите свой Worker как часть создания вашего WorkRequest. При создании WorkRequest также можно указать такие вещи, как время запуска Worker.
- WorkManager – Этот класс на самом деле планирует ваш WorkRequest и запускает его. Он планирует WorkRequest сеансы таким образом, чтобы распределить нагрузку на системные ресурсы, соблюдая при этом заданные вами ограничения.

В данном случае определяется новый BlurWorker, который будет содержать код для размытия изображения. При нажатии кнопку WorkRequest создает файл, а затем ставится в очередь с помощью WorkManager.

В пакете workers необходимо создать новый класс с именем BlurWorker. Он должен наследоваться от Worker. Также необходимо реализовать конструктор. Для определения многопоточной задачи необходимо переназначить метод doWork(). Итоговый код программы можно увидеть ниже.

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.net.Uri;
import android.util.Log;

import com.example.background.R;

import androidx.annotation.NonNull;
import androidx.work.Worker;
import androidx.work.WorkerParameters;

public class BlurWorker extends Worker {
    public BlurWorker(
            @NonNull Context appContext,
            @NonNull WorkerParameters workerParams) {
        super(appContext, workerParams);
    }

    private static final String TAG = BlurWorker.class.getSimpleName();

    @NonNull
    @Override
    public Result doWork() {

        Context applicationContext = getApplicationContext();

        try {

            Bitmap picture = BitmapFactory.decodeResource(
                    applicationContext.getResources(),
                    R.drawable.android_cupcake);

            // Blur the bitmap
            Bitmap output = WorkerUtils.blurBitmap(picture, applicationContext);

            // Write bitmap to a temp file
            Uri outputUri = WorkerUtils.writeBitmapToFile(applicationContext, output);

            WorkerUtils.makeStatusNotification("Output is "
                    + outputUri.toString(), applicationContext);

            // If there were no errors, return SUCCESS
            return Result.success();
        } catch (Throwable throwable) {

            // Technically WorkManager will return Result.failure()
            // but it's best to be explicit about it.
            // Thus if there were errors, we're return FAILURE
            Log.e(TAG, "Error applying blur", throwable);
            return Result.failure();
        }
    }
}
```

Для дальнейшей удобной работы необходимо создать переменную для экземпляра класса WorkManager во ViewModel и присвоить ей значение в ViewModel конструкторе:

```
private WorkManager mWorkManager;

// BlurViewModel constructor
public BlurViewModel(@NonNull Application application) {
  super(application);
  mWorkManager = WorkManager.getInstance(application);

  //...rest of the constructor
}
```

Имея экземпляр WorkManager можно приступить к созданию WorkRequest и указанию WorkManager как его запустить. Есть два типа WorkRequests:

- OneTimeWorkRequest – подразумевает разовое выполнение WorkRequest.
- PeriodicWorkRequest – WorkRequest, который будет повторяться в цикле.

Мы хотим, чтобы изображение размывалось только один раз при нажатии кнопку. В связи с этим внутри метода applyBlur, который вызывается при нажатии на кнопку, создается OneTimeWorkRequest, внутрь которого кладется BlurWorker. Затем, используя экземпляр WorkManager, для установки в очередь созданного OneTimeWorkRequest.

```
void applyBlur(int blurLevel) {
   mWorkManager.enqueue(OneTimeWorkRequest.from(BlurWorker.class));
}
```

В создаваемый WorkRequest можно передать данные на вход и получить данные по окончанию его работы. Ввод и вывод данных происходит через Data объекты. Data объекты — это легкие контейнеры для пар ключ/значение.

В данном примере произведем передаче URI изображения через входные данные. Необходимый URI будет хранится в переменной с именем mImageUri.

Создадим приватный метод с именем createInputDataForUri. Этот метод должен:

- Создать Data.Builder объект.
- Если mImageUri не null, то при помощи putString() метода добавить URI картинки в Data объект. Этот метод принимает ключ и значение. Для указания ключа будет использоваться значение, хранящиеся в константе String KEY_IMAGE_URI.
- Вызовите метод build() у объекта Data.Builder, чтобы собрать Data объект.

Ниже приведен готовый createInputDataForUri метод:

```
/**
 * Creates the input data bundle which includes the Uri to operate on
 * @return Data which contains the Image Uri as a String
 */
private Data createInputDataForUri() {
    Data.Builder builder = new Data.Builder();
    if (mImageUri != null) {
        builder.putString(KEY_IMAGE_URI, mImageUri.toString());
    }
    return builder.build();
}
```

Далее необходимо изменить applyBlur метод так, чтобы он:

- Создавал новый OneTimeWorkRequest.Builder.
- Вызовал setInputData, передавая результат из createInputDataForUri.
- Строил OneTimeWorkRequest.
- Ставил этот запрос в очередь, используя WorkManager.

Ниже приведен готовый applyBlur метод:

```
void applyBlur(int blurLevel) {
   OneTimeWorkRequest blurRequest =
               new OneTimeWorkRequest.Builder(BlurWorker.class)
                       .setInputData(createInputDataForUri())
                       .build();

   mWorkManager.enqueue(blurRequest);
}
```

Чтобы получить данные в BlurWorker и использовать их в методе doWork(), необходимо обновить метод, чтобы он получал вложенный Data объект и выгружал из него URI, который мы передали в нем.

Для этого обновим наш класс BlurWorker:

```
public Result doWork() {

       Context applicationContext = getApplicationContext();

        // ADD THIS LINE
       String resourceUri = getInputData().getString(Constants.KEY_IMAGE_URI);

        //... rest of doWork()
}
```

По окончанию работы Worker есть возможность передать результат работы при помощи метода Result.success(). В связи с тем, что наш BlurWorker производил работу с картинкой, в качестве результата его работы можно предоставить OutputURI, чтобы сделать выходное изображение легко доступным для других рабочих процессов и для дальнейших операций. Обновим наш код следующим образом:

- Создадим новый Data, как делали с входными данными, и сохраним его outputUri как String. Используйте тот же ключ, KEY_IMAGE_URI.
- Передайте это Worker методу Result.success().

```
Data outputData = new Data.Builder()
    .putString(KEY_IMAGE_URI, outputUri.toString())
    .build();
return Result.success(outputData);
```

Задание на практическую работу

1) В разрабатываемом приложении необходимо реализовать многопоточную задачу, обоснованную предметной областью, основанную на классе ExecutorService.
2) В разрабатываемом приложении необходимо реализовать многопоточную задачу, обоснованную предметной областью, основанную на классе Worker.