

# Многопоточное (параллельное) программирование. Работа с потоками



Юрий  
Пеньков



**Юрий Пеньков**

Java Software Engineer, InnoSTage



# План занятия

1. [Многопоточное программирование](#)
2. [Работа с потоками](#)
3. [Класс Thread](#)
4. [Пул потоков. Executor. ExecutorService](#)
5. [Пул потоков. ThreadPoolExecutor. ForkJoinPool](#)
6. [Итоги](#)



# Многопоточное программирование



# Многопоточное программирование

Мы уже умеем писать программы на Java, поэтому наша задача научиться писать их эффективными

Вопросы, которые нужно обсудить сегодня:

- Зачем нужно многопоточное программирование?
- Что такое потоки?
- Как работать с потоками на языке Java?
- Подводные камни.

---

# Многопоточное программирование

Допустим, вы управляете рестораном. Однажды утром вам звонят и заказывают банкет, состоящий из **50 блюд**. Вы зовете своего повара и узнаете, что он успеет приготовить **только 25 блюд** за отведенное время.

Что делать?

- Уволить повара за неэффективность
- Отказаться от заказа банкета
- Приготовить 25 блюд, никто не заметит
- Оптимизировать работу повара
- Нанять еще одного повара



---

# Многопоточное программирование

Наняв дополнительного повара, мы поделили задачу на две части и выполняем их одновременно. Это простое решение, но требует дополнительных ресурсов (повара).

x2



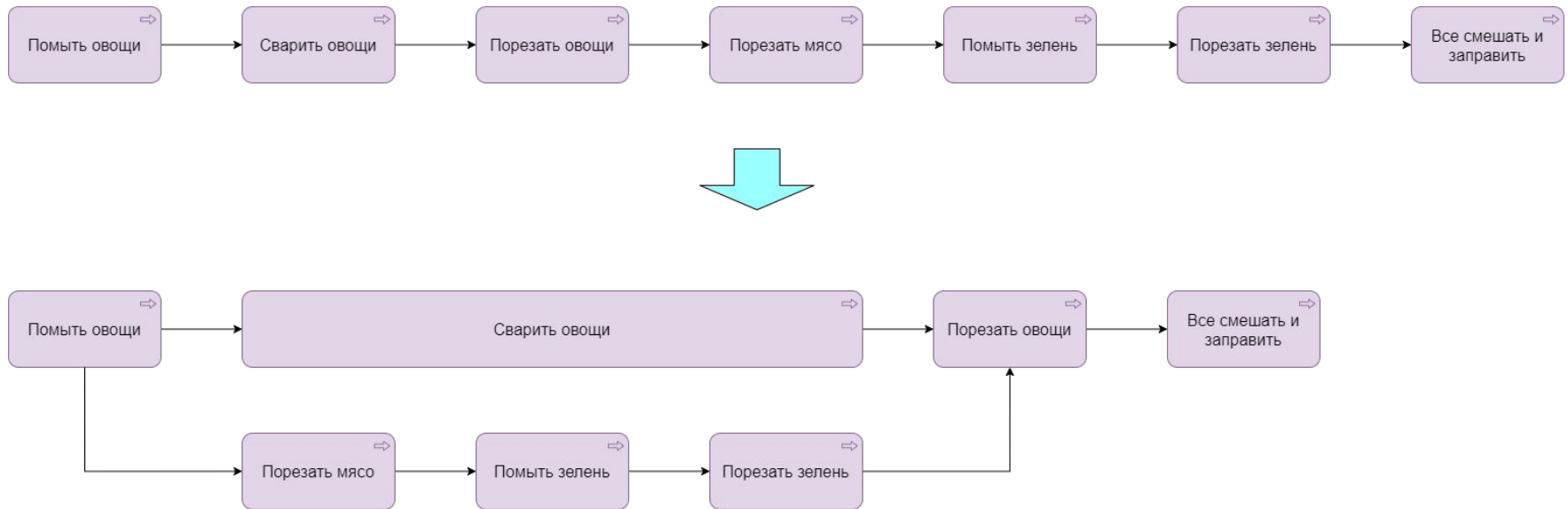
Если у нас нет возможности нанять другого повара, то мы **должны оптимизировать работу первого**, чтоб он стал работать более эффективно.

# Многопоточное программирование

Рассмотрим работу повара во время приготовления салата.

Для начала нужно почистить и отварить овощи, затем все порезать, смешать и заправить.

Что будет делать повар, пока варятся овощи? Он может заняться нарезкой зелени или приготовлением другого блюда. Это еще одно проявление параллелизма в реальном мире: для того, чтобы справиться с общей задачей быстрее, мы уменьшаем время «простоя».





# Многопоточное программирование

А теперь перенесемся обратно в мир программирования и проведем аналогии.

Приготовление блюда - это наша программа. Чтобы приготовить блюдо, нужен повар и рецепт. Аналогично, для выполнения программы нужен процессор и набор инструкций (код).

Приготовление овощей и нарезка зелени – это операции, которые могут выполняться независимо друг от друга и одновременно.

Выполняя операции одновременно, мы существенно сокращаем время выполнения всей программы.

**Для этого нам и нужно уметь писать многопоточные программы.**





# Многопоточное программирование

Как вы думаете,

- все ли задачи можно выполнять одновременно?
- что же такое многопоточность?
- какие виды многопоточности вы знаете?

**Давайте разбираться.**



# Многопоточное программирование

Также как и задачи по нарезке и помывке зелени не могут совершаться одновременно одним поваром, - некоторые задачи компьютер также не может выполнять одновременно.

Некоторые задачи подразумевают работу над одними данными, что усложняет их одновременное выполнение. В следующем уроке мы подробно разберем такие ситуации и научимся с ними работать.

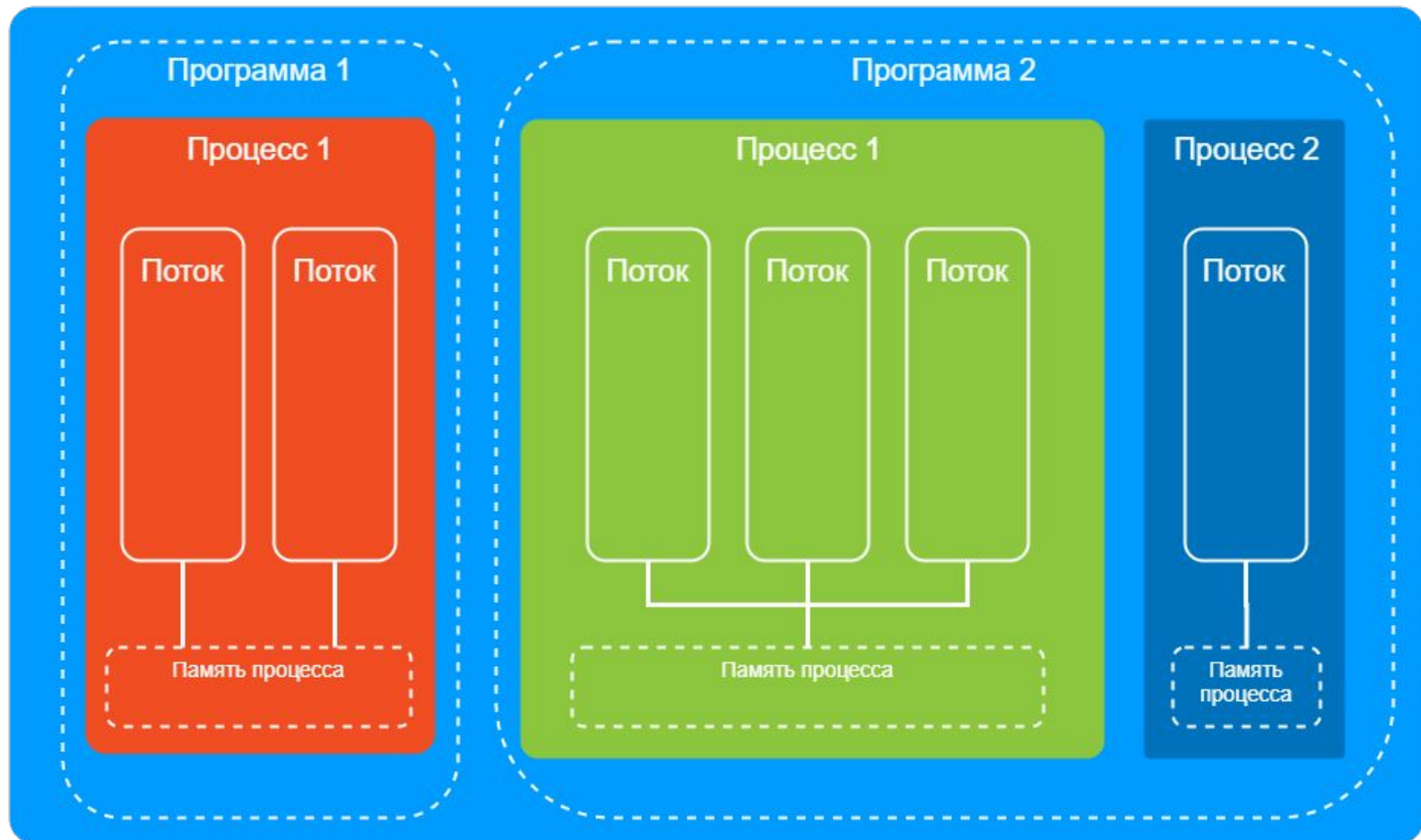


# Многопоточное программирование

Итак, разберемся с основными понятиями:

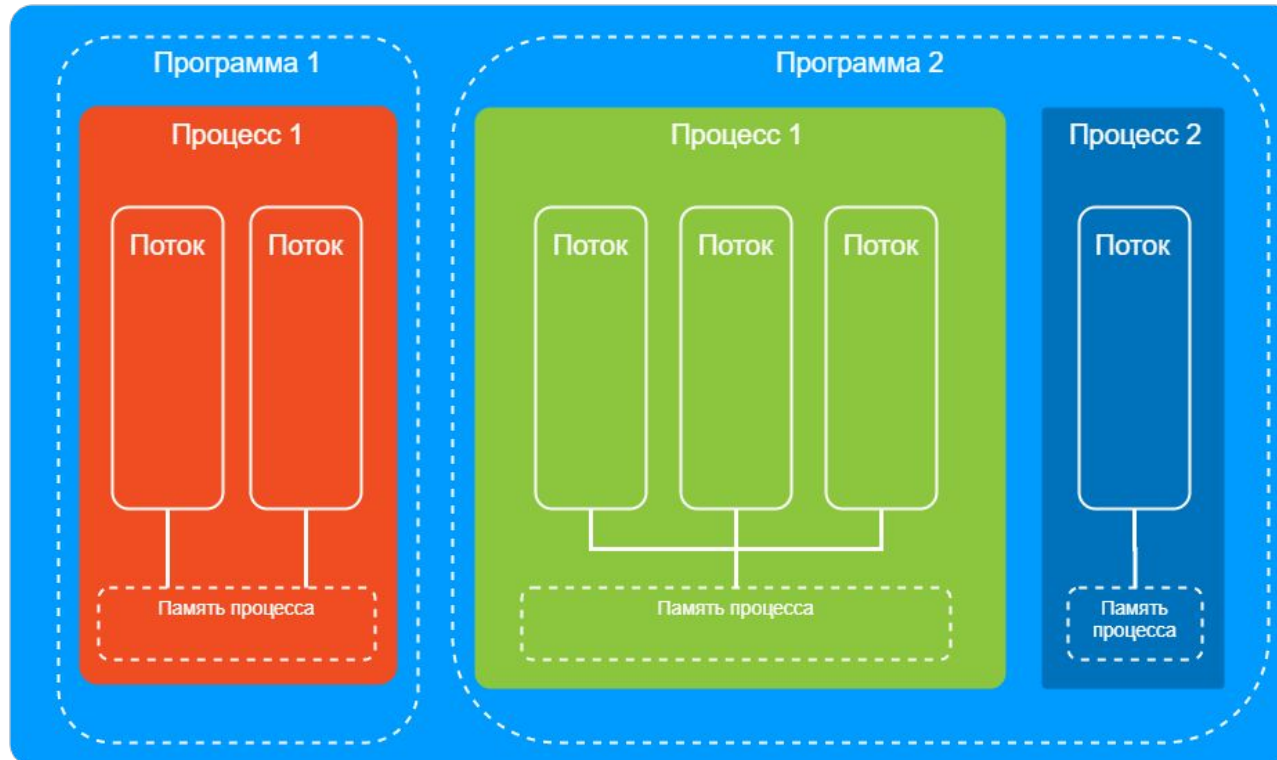
- **Процессорное время** – время, затраченное процессором компьютера на выполнение задачи.
- **Программа во время выполнения** – это один или несколько процессов, которым ОС выделила память для хранения данных и процессорное время
- **Процесс** – это совокупность данных и кода, работающего с этими данными. Процесс создает операционная система (ОС), он имеет свою область памяти, доступную только ему, которую также выделяет ОС. Процесс не следует путать с потоком.
- **Поток** – выполняемая последовательность инструкций, часть процесса. Поток не имеет своей собственной памяти, использует память процесса наравне со всеми потоками этого процесса. Иначе говоря, процесс содержит в себе один или несколько потоков, которые используют одну область памяти.

# Многопоточное программирование



# Многопоточное программирование

**Многопоточность** - свойство приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно».



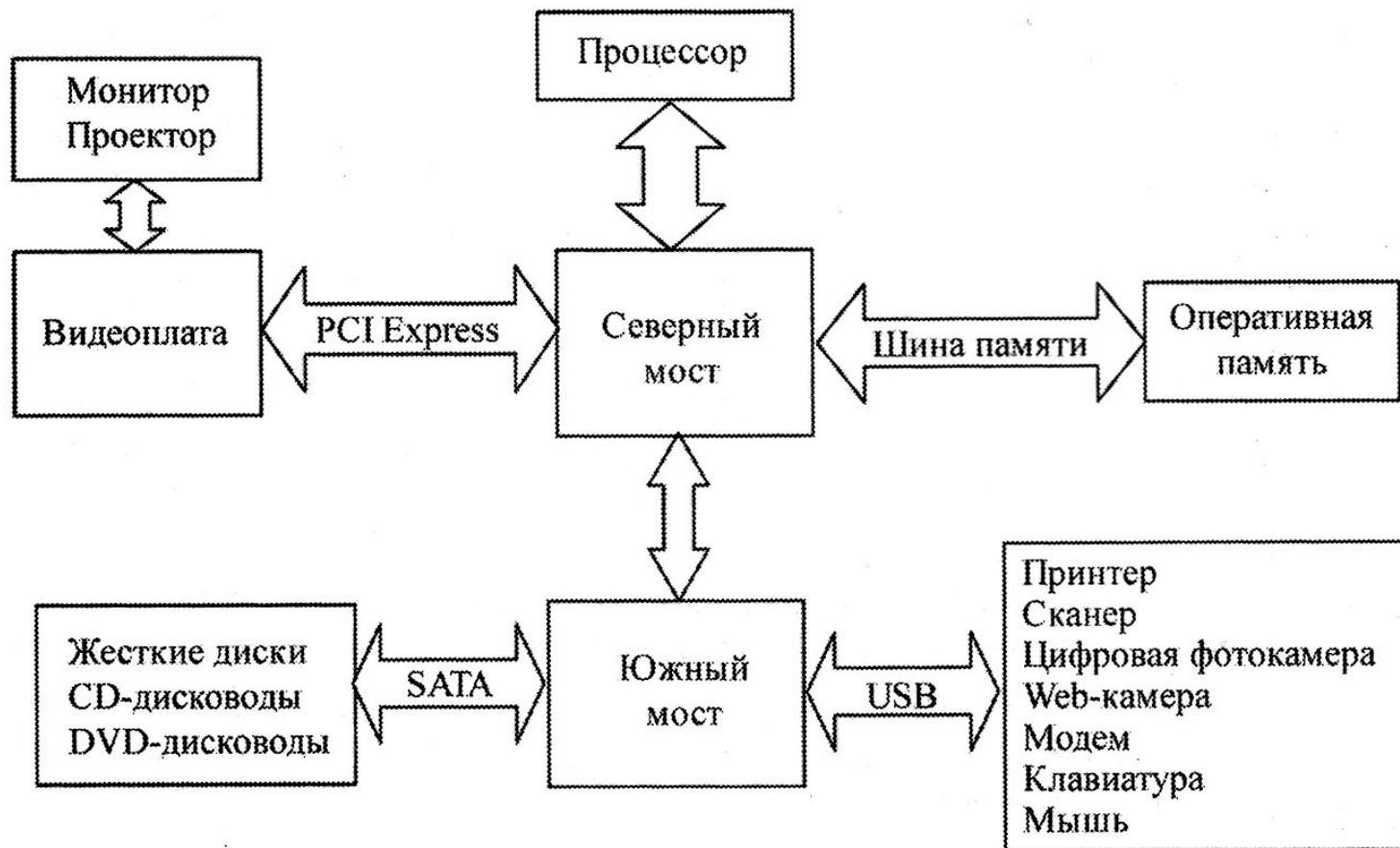


# Многопоточное программирование

Как вы думаете,

- из чего состоит ваш компьютер? Какой компонент компьютера отвечает за выполнение инструкций? А за хранение данных? Кто работает быстрее?
- как еще можно организовать многопоточное решение задач?
- об эволюции компьютеров? О чем говорит закон Мура?
- всегда ли чем больше процессоров - тем быстрее? О чем говорит закон Амдала?

# Многопоточное программирование





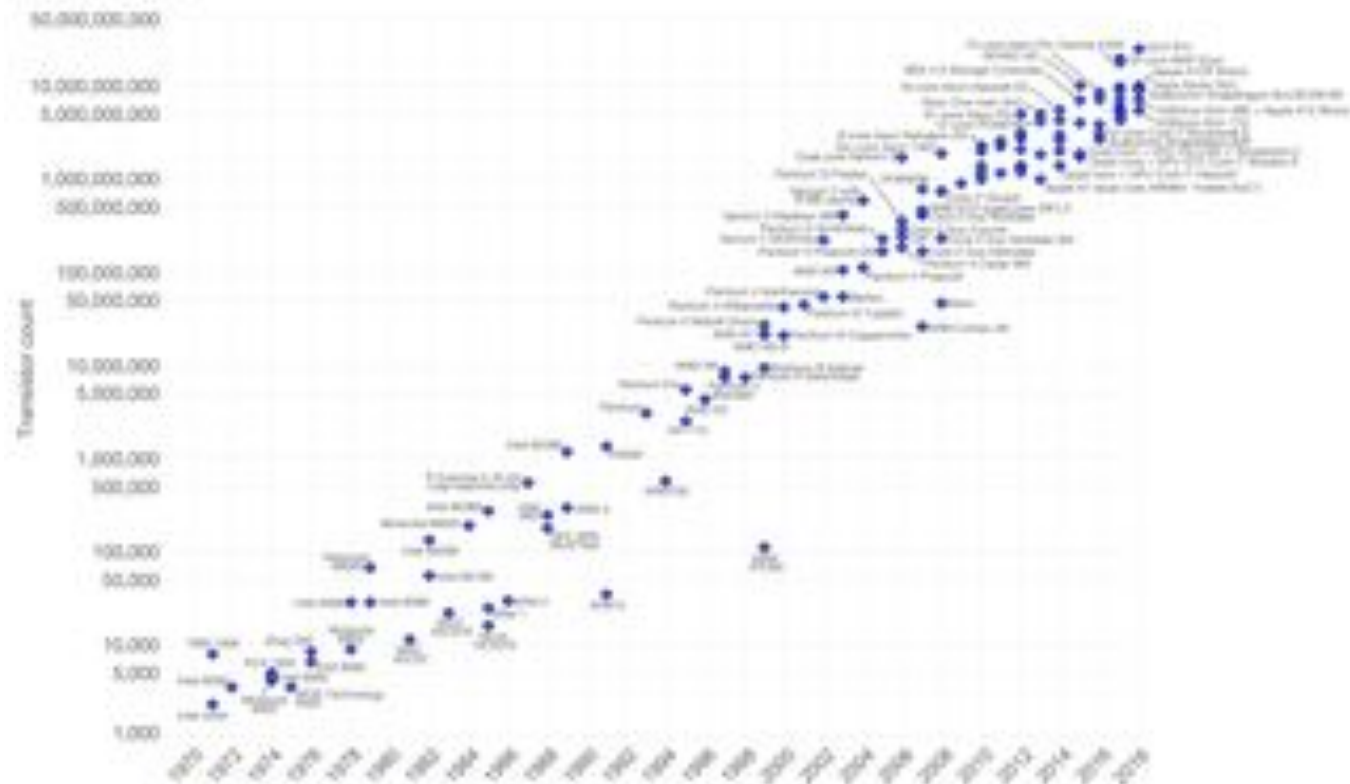
# Многопоточное программирование

- Процессор выполняет инструкции и делает это быстро.
- Память (кэши, оперативная, диск, сеть) записывает, хранит и отдает данные медленно. Кэши быстрее, сеть медленнее. Пока ждем ввод/вывод информации, процессор может заняться другой работой.
- Если компьютер имеет несколько логических или физических процессов - он может выполнять несколько наборов инструкций одновременно.
- В самой доступной формулировке закон Мура говорит о том, что **каждые 18 месяцев мощность процессоров на рынке возрастает вдвое**. В отличие от процессоров, память не увеличивает свою работу так быстро.

# Многоточное программирование

## Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important in other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



Data source: Wikipedia ([https://en.wikipedia.org/wiki/Transistor\\_count](https://en.wikipedia.org/wiki/Transistor_count))

This data visualization is available at [ourworldindata.org](https://ourworldindata.org). Please cite this data visualization and research on this topic.

Licensed under CC-BY-SA by the author Max Roser

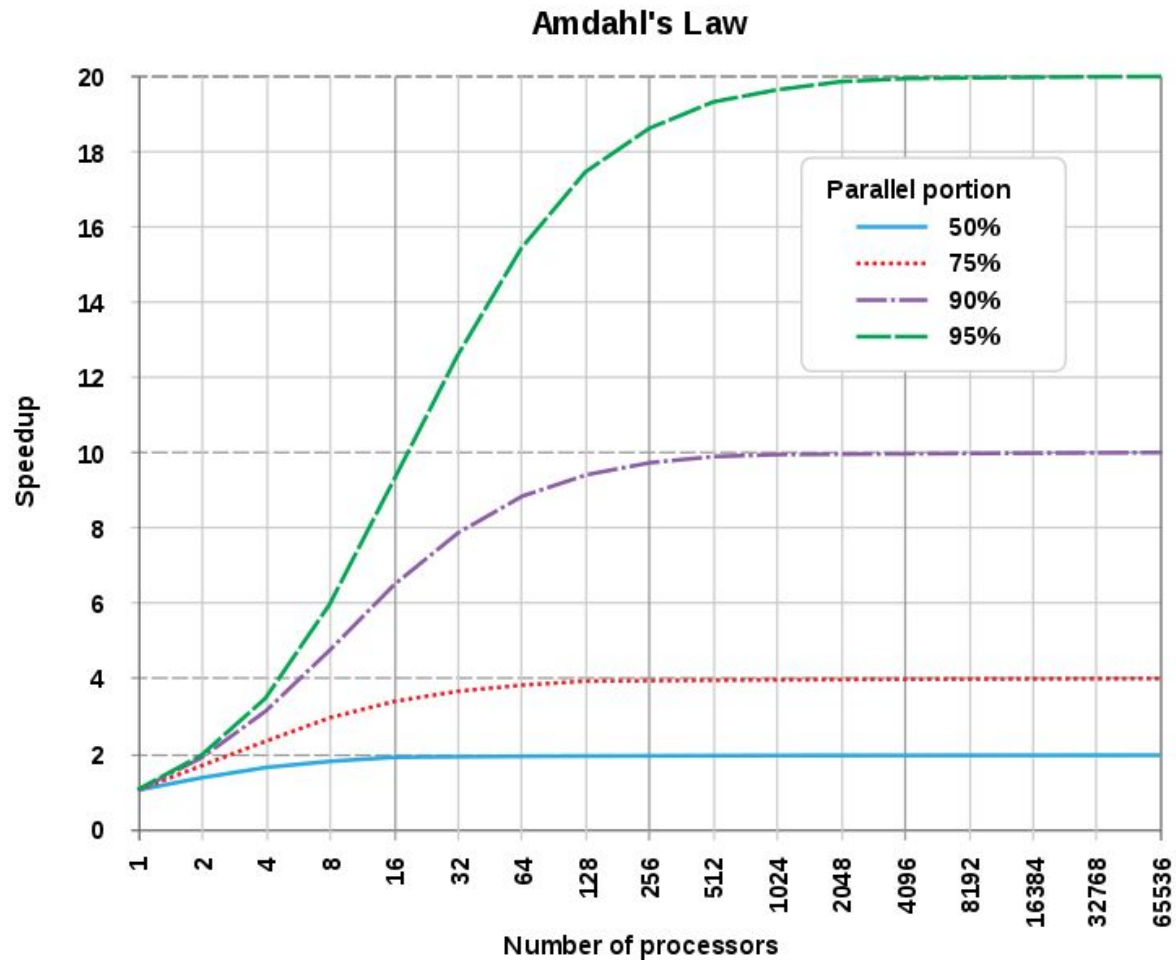


# Многопоточное программирование

Действительно ли чем больше процессоров, тем быстрее выполнится некоторая работа? Не совсем так.

Закон Амдала говорит нам о том, что **В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого медленного фрагмента.**

# Многопоточное программирование





# Многопоточное программирование

Многопоточность позволяет:

- Быстрее решать однотипные задачи
- Эффективнее использовать ресурсы системы

Реализация многопоточности:

- Аппаратная (Intel Hyper-Threading)
- Программная

Программная реализация многопоточности в Java:

- `java.lang` (стандартные средства языка)
- `java.util.concurrent` (пакет работы с потоками и синхронизацией)



# Работа с потоками



# Работа с потоками

Минимальная единица многопоточности – это поток.

Способы создать поток в Java:

- реализация наследника класса Thread (поток)
- реализация интерфейса Runnable
- реализация интерфейса Callable

# Работа с потоками

Способ №1. Наследование от класса Thread:

```
public class MyThread extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("Привет из потока!");  
    }  
}
```

Запуск потока:

```
public class Main {  
  
    public static void main(String[] args) {  
        new MyThread().start();  
    }  
}
```



# Работа с потоками

Обратите внимание, что только метод *start()* вызовет выполнение задачи в отдельном потоке.

Если вы вызовете метод *run()*, то задача выполнится **в текущем** потоке и никакого параллельного исполнения не будет!

```
public class Main {  
    public static void main(String[] args) {  
        //Запуск задачи в отдельном потоке  
        new MyThread().start();  
  
        //Выполнение задачи в этом же потоке  
        new MyThread().run();  
    }  
}
```

# Работа с потоками

Способ №2. Реализация интерфейса Runnable:

```
public class MyRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        // TODO: код, который должен выполнить поток  
    }  
}
```

Запуск потока:

```
public class Main {  
  
    public static void main(String[] args) {  
        MyRunnable myRunnable = new MyRunnable();  
        new Thread(myRunnable).start();  
    }  
}
```

# Работа с потоками

В чем разница в способах? Если по-простому:

**Runnable** – это абстрактная **работа**, которую можно выполнить

**Thread** – это абстрактный **процесс** выполнения какой-то работы

Работа:  
разгрузить вагон



Процесс:  
разгрузка вагона



Обратите внимание, что когда мы создаем свою реализацию Runnable, мы все равно оборачиваем ее в Thread, чтобы можно было ее выполнить. Зачем нужны оба способа мы рассмотрим чуть позже.

# Работа с потоками

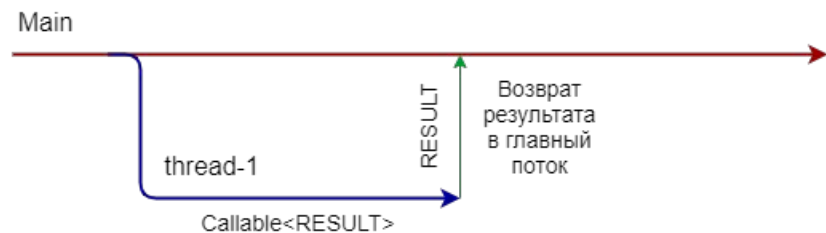
Иногда требуется **не просто выполнить работу** в отдельном потоке, **но и получить результат**.

*Например, разгружая вагон, мы ничего нового не получаем: тот же вагон и тот же груз, но отдельно друг от друга. Но если нужно посчитать доход компании за год – требуется вернуть какой-то результат, кол-во полученных денег.*

Для этого в Java существует похожий на Runnable интерфейс – Callable.

Главное отличие состоит в том, что:

- **Runnable-задача** - не может вернуть результат,
- **Callable-задача** – может вернуть результат.



# Работа с потоками

## Способ №3. Реализация интерфейса Callable

Callable – это задача, результатом работы которой является некоторый объект. Так как тип результата может быть разным, интерфейс Callable является дженериком (generic).

```
public class MyCallable implements Callable<String> {  
  
    @Override  
    public String call() throws Exception {  
        // TODO: код, который должен выполнить поток  
        return "Результат";  
    }  
}
```

В данном случае, задача MyCallable возвращает результат типа *String*.

# Работа с потоками

Но как получить результат? Просто вызвать метод `call()` у задачи мы не можем, так как она будет выполняться не в отдельном потоке, а в текущем.

А как запустить в отдельном потоке и получить оттуда результат, когда задача будет решена? Для этого в Java предусмотрен интерфейс `java.util.concurrent.Future`, который имеет следующие методы:

- `isCancelled()` – вернет `true`, если задача была отменена
- `isDone()` – вернет `true`, если задача была завершена
- `get()` – получить результат после завершения задачи
- `cancel()` – отмена задачи

Отмененную задачу запустить нельзя. Если задача во время отмены уже выполнялась, то она завершится сразу, если был вызван `cancel()` с параметром `mayInterrupt == true`

Важно! Завершенной задача считается не только в случае ее правильного завершения с результатом, но и в случае выброса исключения во время выполнения. Будьте внимательны!

# Работа с потоками

Одной из самых популярных реализаций интерфейса `Future` является класс `FutureTask`. Его преимущество еще и в том, что он также реализует интерфейс `Runnable`. То есть, если мы обернем нашу `Callable`-задачу во `FutureTask`-объект, мы сможем запустить ее в отдельном потоке и в конце получить еще и результат ее выполнения.

```
public class Main {  
    public static void main(String[] args) throws ExecutionException, InterruptedException {  
  
        final MyCallable myCallable = new MyCallable();  
        final FutureTask<String> stringFutureTask = new FutureTask<>(myCallable);  
        new Thread(stringFutureTask).start();  
  
        final String resultOfTask = stringFutureTask.get();  
        System.out.println("Результат выполнения задачи: " + resultOfTask);  
    }  
}
```

Напоминаем, что вызов методов `run()` или `call()` не приводит к выполнению задач в отдельном потоке.

# Класс Thread

Теперь подробнее остановимся на потоках. Можно ли ими как-то управлять? Рассмотрим основные **методы класса Thread**:

- **start()** – запуск потока
- **interrupt()** – остановка потока
- **getId()** – получение идентификатора процесса
- **setName()/getName()** – установка и получение имени потока
- **setPriority()/getPriority()** – установка и получение приоритета
- **getState()** – получение статуса потока
- **isInterrupted()** – вернет true, если выполнение потока было остановлено
- **isAlive()** – вернет true, если поток «жив» (работает)
- **isDaemon()** – вернет true, если поток является потоком-демоном



# Класс Thread

Чтобы остановить поток, существует метод *interrupt()*. Он выставляет флаг потоку, что тот должен завершиться. При этом, само завершение должно быть реализовано самостоятельно.

```
public class Main {  
  
    public static void main(String[] args) throws Exception {  
        MyThread myThread = new MyThread();  
        myThread.start();  
        myThread.interrupt();  
    }  
}
```

```
public class MyThread extends Thread {  
  
    @Override  
    public void run() {  
        while(true) {  
            if(isInterrupted()) return;  
            //TODO: код выполняемого процесса  
        }  
    }  
}
```

## Класс Thread

Почему нельзя завершать поток немедленно, а требуется самостоятельная реализация?

Дело в том, что поток во время работы может открывать сетевые соединения или открывать какие-то файлы. Неожиданное завершение потока может повлечь потерю данных и некорректную дальнейшую работу других потоков.

Если ваш поток использует какие-то ресурсы, вы должны освободить их прежде, чем поток прекратит выполнение – закрыть сетевые соединения, файлы и т.п.



# Класс Thread

Для идентификации потока можно использовать метод `getId()`. Метод вернет идентификатор типа `long`

Иногда неудобно работать с идентификаторами. Например, при логировании неудобно читать идентификатор потока, а вот осмысленное имя – удобно.

Для назначения и последующего получения имени потока используются методы `getName()` и `setName()`

```
public class Main {  
    private static final Logger LOG = LoggerFactory.getLogger(Main.class);  
  
    public static void main(String[] args) throws Exception {  
        MyThread myThread = new MyThread();  
        myThread.setName("Поток-загрузчик 1");  
        LOG.debug("Запускаю поток с именем {} и идентификатором {}", myThread.getName(), myThread.getId());  
        myThread.start();  
    }  
}
```

# Класс Thread

Мы уже кратко ознакомились с понятием процессорного времени.

Одноядерный процессор не может одновременно выполнять несколько потоков. Чтобы все потоки работали, процессор постоянно переключается между ними.

По умолчанию, все потоки получают одинаковое процессорное время, но это можно изменить, задав потокам разный приоритет. Для этого существуют методы *getPriority()*, *setPriority()*

```
public class Main {  
  
    public static void main(String[] args) throws Exception {  
  
        MyThread myThread = new MyThread();  
  
        //Приоритет потока от 1 до 10  
        myThread.setPriority(Thread.MAX_PRIORITY);  
    }  
}
```

По умолчанию все потоки имеют одинаковый приоритет  
`Thread.NORM_PRIORITY == 5`




# Класс Thread

Для отладки программы иногда требуется знать текущее состояние потока. Для этого существует метод `getState()`, который вернет одно из следующих значений

- NEW – поток создан, но не запущен
- RUNNABLE – поток запущен
- BLOCKED – поток заблокирован и ждет освобождения ресурсов
- WAITING – поток в режиме ожидания
- TIMED\_WAITING – поток в режиме ожидания фиксированного времени
- TERMINATED – процесс завершил выполнение

Более подробно статусы потока мы разберем в следующем уроке.



Важно! Не рекомендуется использовать метод для построения логики программы. Только для цели отладки, так как статус потока меняется очень быстро.

# Класс Thread

Как остановить поток мы уже рассмотрели и использовали для этого метод `isInterrupt()`. Есть похожий метод `interrupted()`, который также возвращает `true`, если поток помечен на завершение.

Отличие метода состоит в том, что он принадлежит классу `Thread` (является статическим). Его можно использовать не имея ссылки на поток, метка завершения вернется **для текущего потока**.

```
public class Main {  
  
    public static void main(String[] args) throws Exception {  
  
        MyThread myThread = new MyThread();  
  
        //Приоритет потока от 1 до 10  
        myThread.setPriority(Thread.MAX_PRIORITY);  
    }  
}
```

Важно! Вызов метода `interrupted()` сбрасывает метку завершения у потока. Если метод выполнен раньше завершения потока, то поток не узнает, что ему нужно прекратить выполнение задачи.

# Класс Thread

Иногда бывает ситуация, когда нам нужно выполнять некоторые вычисления в отдельном потоке, но только тогда, когда главный поток работает.

Например, пока работает UI-поток (не закрыто окно в Windows), требуется выполнять некоторые вычисления в отдельном потоке. Как только окно закрыто пользователем (завершился UI-поток), данные из другого потока нам больше не нужны. Чтобы не завершать поток вручную, можно определить поток-демон.

Поток-демон – это поток, который JVM сама убьет при завершении породившего его потока.

Чтобы поток стал потоком-демоном, надо вызвать метод `setDaemon()`

```
public class Main {  
  
    public static void main(String[] args) throws Exception {  
  
        MyThread myThread = new MyThread();  
  
        //Помечаем поток как поток-демон  
        myThread.setDaemon(true);  
    }  
}
```

# Класс Thread

Узнать, является ли поток демоном, можно вызвав метод *isDaemon()*. Если это так, метод вернет *true*.

**isDaemon()**

Важно! Поток-демон не завершается безопасно. Не гарантируется даже выполнение блока *finally{}*. Не создавайте поток-демон, если он работает с файлами или другими ресурсами, требующими освобождения.



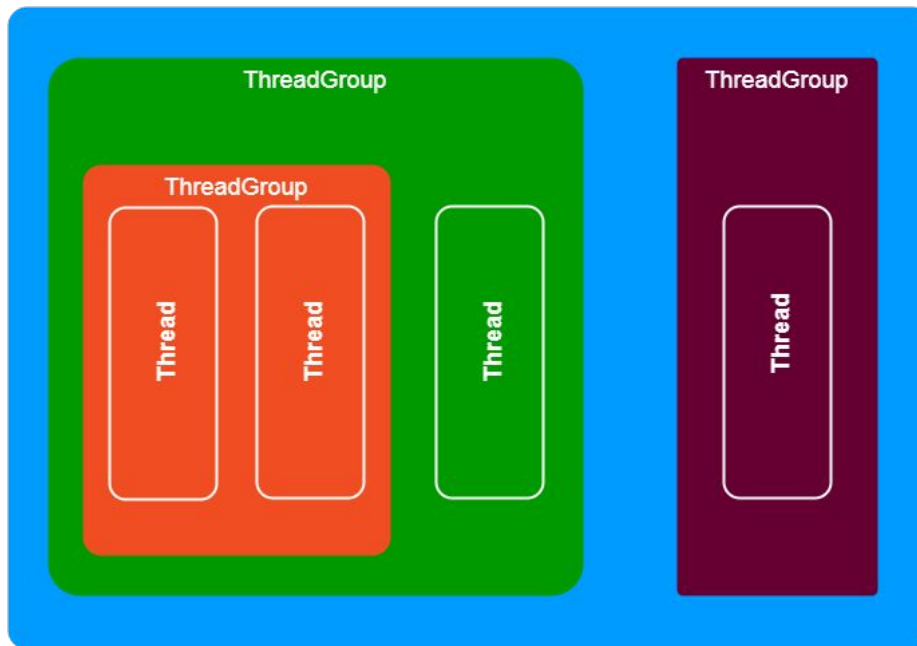
# Класс Thread

Еще один полезный метод, относящийся к классу Thread (статический), - это метод `sleep()`. Во время изучения принципов работы потока он бывает крайне полезен. Метод приостанавливает выполнение потока на заданное время (в миллисекундах)

Важно! Вызов метода `sleep()` может породить выброс исключения, если текущий поток получил метку на завершение во время паузы. Чтобы обработать это исключение, используйте блок `try-catch`

# Группировка потоков - ThreadGroup

Когда потоков много, а еще и когда они занимаются разными задачами, удобнее их **логически группировать**. Для этого существует класс ThreadGroup, объекты которого могут содержать потоки и группы потоков, помогая строить иерархию потоков.



# Группировка потоков - ThreadGroup

Группы потоков позволяет не только составлять иерархию процессов, но и управлять множеством процессов как одним. Например, можно остановить все потоки, входящие в одну группу, переназначить приоритет одновременно всем или определить все потоки в группе как потоки-демоны.

```
// Главная группа потоков
ThreadGroup mainGroup = new ThreadGroup("main group");

// Вторая группа потоков, входящая в главную
ThreadGroup group1 = new ThreadGroup(mainGroup, "group-1");

// Добавление потоков в группы
final Thread thread1 = new Thread(group1, myRunnable);
final Thread thread2 = new Thread(group1, myRunnable);
final Thread thread3 = new Thread(mainGroup, myRunnable);

// Запуск потоков
thread1.start();
thread2.start();
thread3.start();


// Понижаем приоритет у всех потоков группы group-1
group1.setMaxPriority(4);

// Завершаем все потоки одним вызовом
mainGroup.interrupt();
```



## Группировка потоков - ThreadGroup

Группу также можно уничтожить, вызвав метод `destroy()`. Обратите внимание, что если в группе содержатся потоки или группы с потоками, то вызов метода породит ошибку. Поток сам выйдет из группы, когда завершит свое выполнение. Таким образом, прежде чем вызвать `destroy()`, следует убедиться в отсутствии потоков или вызвать `interrupt()` для завершения всех потоков.



Существует много споров, относительно полезности группировки потоков. Цель данных слайдов – показать возможности работы с потоками. Далее мы рассмотрим примеры как вообще не создавать потоки вручную, а значит и не группировать их.

# Пул потоков

Создание новых потоков – дорогая операция с точки зрения времени.

Создавать под каждую задачу свой поток слишком накладно, особенно, если задач таких у нас много. Что же делать? Java предлагает переиспользовать потоки, подобно тому, как переиспользуются контейнеры для личных вещей в аэропорту при досмотре.

В аэропорту контейнеры не создаются под каждого пассажира, а используются свободные на данный момент. Когда пассажир видит свободный контейнер – он складывает в него свои вещи, проходит досмотр и освобождает контейнер для следующего пассажира. Точно также освободившиеся потоки берут нерешенные задачи на выполнение.



# Executor

Прежде чем узнать больше про пул потоков и работу с ним, следует ознакомиться с интерфейсами Executor и ExecutorService.

В Java определен стандартный интерфейс, описывающий метод запуска Runnable-объекта без создания специального потока - *java.util.concurrent.Executor*

```
public interface Executor {  
    void execute(Runnable command);  
}
```

# ExecutorService

Также, у `Executor` имеется важный наследник – `ExecutorService`. Он предлагает больше методов для запуска и управления выполнением задач. Он позволяет запускать `Runnable` и `Callable` объекты без лишних оборачиваний в `Thread` и `FutureTask`. Достаточно использовать метод *submit()*

```
// запуск Callable-объекта
// сразу получаем объект Future для управления задачей и получения результата
<T> Future<T> submit(Callable<T> task);

// запуск Runnable-объекта
// сразу получаем объект Future для управления задачей
Future<?> submit(Runnable task);

// запуск Runnable-объекта (например FutureTask<T>)
// сразу получаем объект Future для управления задачей и получения результата
<T> Future<T> submit(Runnable task, T result);
```

# ExecutorService

ExecutorService позволяет определять методы для запуска нескольких задач сразу или запуска всех задач ради решения хотя бы одной.

```
// запуск нескольких Callable-объектов
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
    throws InterruptedException;

// запуск нескольких Callable-объектов
// задачи, решение которых заняло больше времени - отменяются автоматически
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
    long timeout, TimeUnit unit)
    throws InterruptedException;

// запуск нескольких Callable-объектов
// возврат результата первой успешно выполненной задачи. Остальные отменяются
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
    throws InterruptedException, ExecutionException;

// запуск нескольких Callable-объектов
// задачи, решение которых заняло больше времени - отменяются автоматически
// возврат результата первой успешно выполненной задачи. Остальные отменяются
<T> T invokeAny(Collection<? extends Callable<T>> tasks,
    long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException;
```





# ExecutorService

Если нам больше не нужен ExecutorService, мы должны правильно его «выключить». Для этого определено несколько методов:

- **shutdown()** – перестает запускать задачи, отменяет все запущенные, соблюдая порядок
- **shutdownNow()** – отменяет все запущенные задачи, возвращает их список
- **awaitTermination()** – ожидает завершение всех задач или таймаута и выключает сервис

Существуют и методы для определения состояния сервиса:

- **isShutdown()** – вернет true, если сервис «выключен» одним из описанных выше методов.
- **isTerminated()** – вернет true, если сервис закончил выполнение всех отправленных задач

# Пул потоков

Итак, мы теперь знаем возможности `ExecutorService` и как с ним работать, но что это нам дает?

Дело в том, что любой пул потоков так или иначе реализует интерфейс `ExecutorService`, а значит все методы, описанные в нем, можно использовать у любого создаваемого пула. Как же создать пул потоков? В этом нам поможет класс `Executors` и его статический метод `newFixedThreadPool`

```
public class Main {  
    public static void main(String[] args) {  
  
        // Создаем пул потоков фиксированного размера  
        final ExecutorService threadPool = Executors.newFixedThreadPool(4);  
    }  
}
```

Здесь мы создали пул потоков фиксированного размера (4). Можно было указать и другое кол-во потоков.

# Пул потоков

Теперь попробуем запустить нашу задачу с помощью пула:

```
public class Main {  
    public static void main(String[] args) throws ExecutionException, InterruptedException {  
  
        // Создаем задачу с результатом типа String  
        Callable<String> myCallable = new MyCallable<>();  
  
        // Создаем пул потоков фиксированного размера  
        // В данном случае у нас будет 4 потока  
        final ExecutorService threadPool = Executors.newFixedThreadPool(4);  
  
        // Отправляем задачу на выполнение в пул потоков  
        final Future<String> task = threadPool.submit(myCallable);  
  
        // Получаем результат  
        final String resultOfTask = task.get();  
  
        // Завершаем работу пула потоков  
        threadPool.shutdown();  
    }  
}
```

# ThreadPoolExecutor

На самом деле, для запуска одной задачи не стоит создавать целый пул потоков. Но когда задач много, тогда использование пула потоков оправданно.

Кроме пула фиксированного размера есть также:

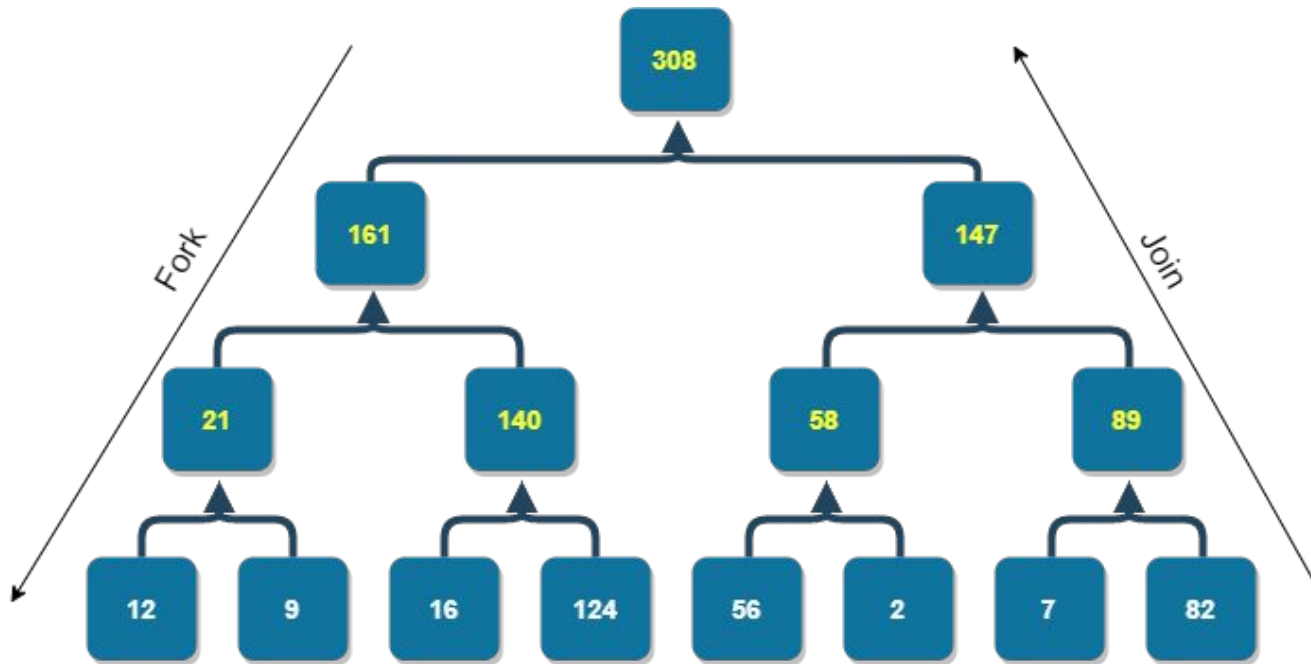
- **`Executors.newSingleThreadExecutor()`** - пул, состоящий из одного потока
- **`Executors.newScheduledThreadPool()`** - пул для задач по расписанию
- **`Executors.newCachedThreadPool()`** - пул потоков переменного количества

Эти варианты пулов на самом деле объекты одного класса – `ThreadPoolExecutor`, имеющие разный набор свойств.

Пул фиксированного размера, несмотря на свое название, не создает сразу нужное количество потоков. Благодаря ленивой инициализации, потоки будут созданы только тогда, когда нужны

# ForkJoinPool

Кроме `ThreadPoolExecutor`'а есть еще один важный класс – `ForkJoinPool`. Этот пул потоков был написан для эффективного решения тех задач, которые могут рекурсивно разбиваться на части до некоторых пор, а результат потом собирался из результатов вычисления всех частей. Например, задача на нахождение суммы элементов массива.



# ForkJoinPool

ForkJoinPool иногда называют целым фреймворком, потому что сама идея пула сильно отличается от других реализаций и включает в себя целую экосистему классов. Чтобы понять, как с ней работать, ознакомимся с интерфейсами RecursiveAction и RecursiveTask.

**RecursiveAction** – это некоторое действие. Действие не возвращает никакого результата. Главный метод, который описывает инструкции всего действия – compute

```
protected abstract void compute();
```

**RecursiveTask** – это некоторая задача, предполагающая результат.

```
V result;  
protected abstract V compute();
```

# ForkJoinPool

Алгоритм задач для ForkJoinPool строится по одной схеме. Если задача достаточно небольшая (например, посчитать сумму двух элементов массива), то она решается. Если нет – делится на части. Вот пример задачи, которая считает сумму элементов массива.

```
@Override
protected Integer compute() {
    final int diff = end - start;
    switch (diff) {
        case 0: return 0;
        case 1: return array[start];
        case 2: return array[start] + array[start+1];
        default: return forkTasksAndGetResult();
    }
}
```

Если задача поделилась до тех пор, что осталось сложить 2 элемента массива или даже меньше – то можно быстро вычислить и вернуть результат. В противном случае, запускаем метод деления задачи

# ForkJoinPool

Если задача достаточно большая, ее стоит разделить, например, на две части. В данном случае, делим диапазон суммирования напополам. Создаем задачу под каждую часть диапазона и методом `invokeAll()` запускаем обе. Как только результаты обеих задач будут вычислены, можно будет вернуть их сумму

```
private int forkTasksAndGetResult() {  
    final int middle = (end - start) / 2 + start;  
  
    // Создаем задачу для левой части диапазона  
    ArraySumTask task1 = new ArraySumTask(start, middle, array);  
  
    // Создаем задачу для правой части диапазона  
    ArraySumTask task2 = new ArraySumTask(middle, end, array);  
  
    // Запускаем обе задачи в пуле  
    invokeAll(task1, task2);  
  
    // Суммируем результаты выполнения обеих задач  
    return task1.join() + task2.join();  
}
```





# ForkJoinPool

Более подробное изучение ForkJoinPool и как именно он работает, мы оставим на следующий урок, когда поговорим о синхронизации потоков.

Стоит отметить только, что ForkJoinPool используется не только для решения рекурсивных задач, а получить его экземпляр можно вызвав `Executor.newWorkStealingPool()`



# Итоги



# Итоги

- Сегодня мы с вами погрузились в сложный, но интересный мир многопоточной разработки
- Это важная тема, поскольку знания, полученные сегодня, помогут нам писать свои программы более эффективно.
- Вместе с этим, следует понимать, что в многопоточной разработке есть много подводных камней, о которых стоит помнить. Но о большей части из них мы поговорим на следующей лекции.



# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в **чате**!
- Задачи можно сдавать **по частям**.
- Зачет по домашней работе проставляется после того, как приняты **все задачи**.

**Задавайте вопросы и  
пишите отзыв о лекции!**

**Юрий Пеньков**