**Конкурентное исполнение – общее понятие** для всех терминов типа «многопоточность», «параллелизм», «асинхронность», который обозначает, что система может поддерживать более одного действия, которые находятся в процессе исполнения.

**Параллелизм** – свойство системы одновременно поддерживать несколько задач в состоянии исполнения (подразумевает наличие нескольких вычислительных устройств для каждой задачи).

**Асинхронность –** подразумевает выполнение потока в неблокирующем режиме, то есть ожидание ответа от сервиса или другого потока не приводит к простаиванию текущего.

**Многопоточность *–*** реализация конкурентного исполнения путем выделения абстракций – потоков. Подразумевает как параллельное исполнение (вычисления одновременно), так и «конкурентное» - псевдопараллельное (time-slicing – распределение процессорного времени ядра между несколькими потоками и их постепенное последовательное исполнение).

***Процесс и поток***:

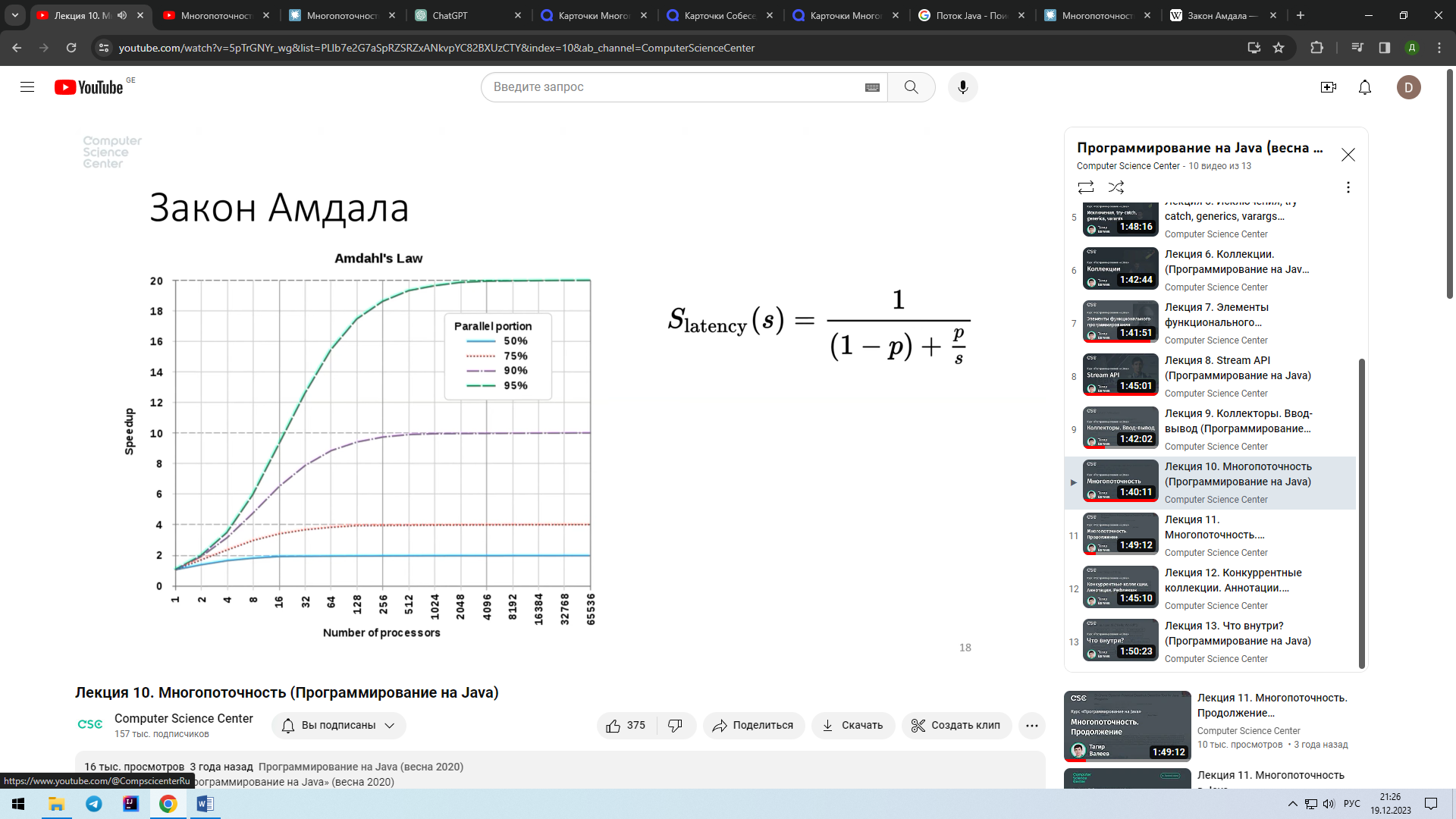
*Процесс* – экземпляр программы, которому выделены собственные неразделяемые ресурсы (изолирован от других процессов).

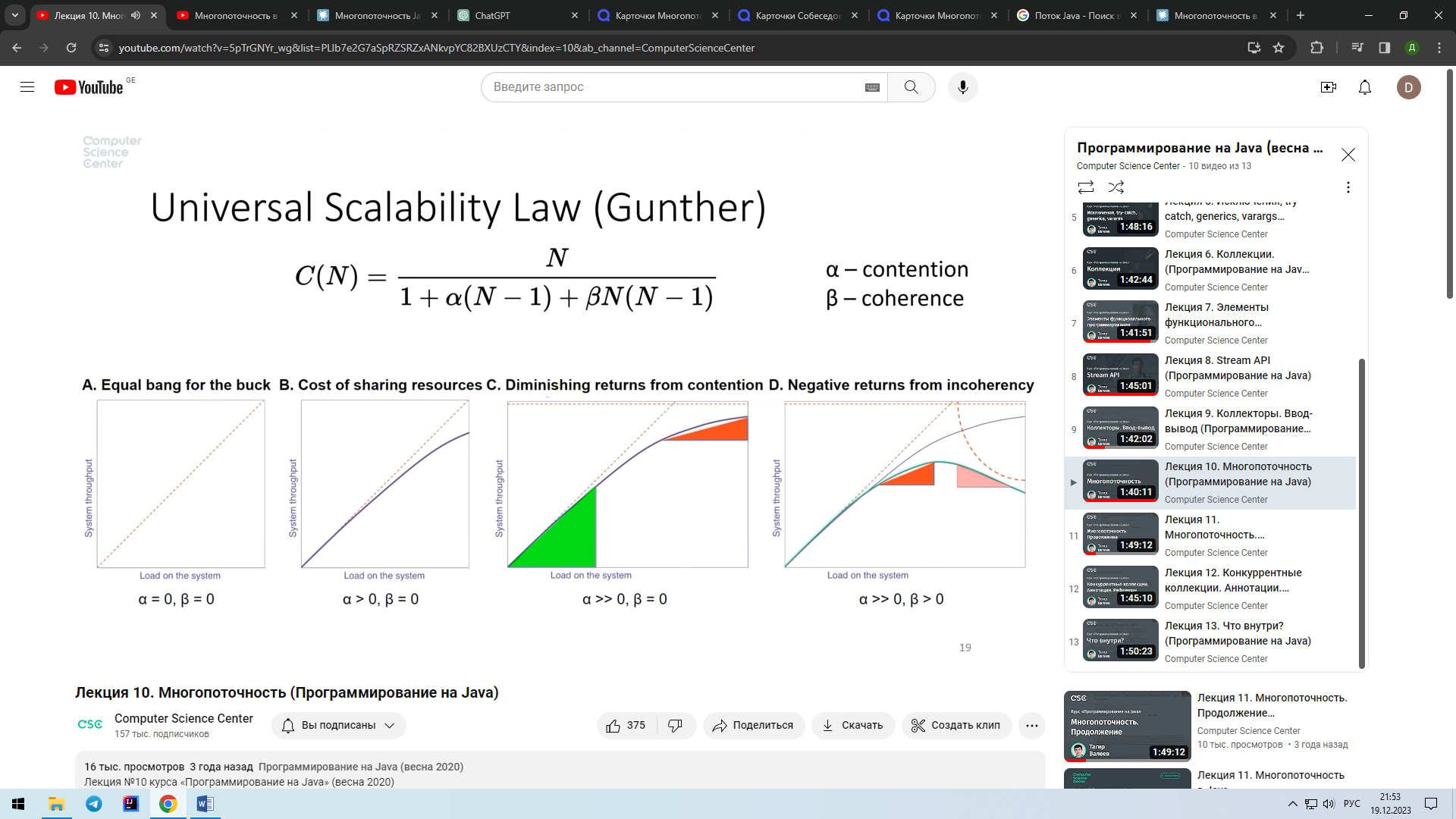
*Поток* – абстракция, инкапсулирующая ресурсы процессора (процесс выделяет ей ресурсы и потоки имеют разделяемую память) и представляющая ход исполнения программы.

***Предпосылки появления –*** повысить скорость выполнения программы можно 2мя способами:

1) увеличить скорость вычисления за счет увеличения тактовой частоты процессора (скорость вычислений в секунду) и оптимизации работы самого процессора, но мешают физические ограничения – тепловыделение и скорость распространения сигнала (чем выше тактовая частота, тем быстрее должны перемещаться сигналы).

2) распараллеливанием вычислительных задач – вычисления параллельно на разных процессорах. Распараллеливание так же имеет свой предел по закону Амдала (получаемое ускорение от распараллеливания зависит от кол-ва работы, которую можно исполнять параллельно – часть работы, которая выполняется последовательно ограничивает возможное ускорение). Универсальный закон масштабируемости (Гюнтера) (дополняет закон Амдала и определяет, что распараллеливание работы не приносит линейного ускорения из-за наличия конкуренции и необходимости когерентности).





**Проблемы многопоточности:**

Проблемы многопоточности связаны с наличием разделяемых ресурсов (критическая секция).

**Race Condition –** ситуация, когда несколько потоков одновременно пытаются изменить разделяемый ресурс, и результат изменения зависит от порядка выполнения операций каждого потока. Проблема Race condition решается путем упорядочивания этих операций посредством блокировок.

Подходы к управлению доступом:

Пессимистическая блокировка – предполагает, что конфликт при доступе к ресурсу вероятен. Поток, получающий доступ к ресурсу блокирует его на время выполнения операции и получает к нему эксклюзивный доступ (mutual exclusion – взаимное исключение).

Минусы: приводит к остановкам других потоков в случае, если ресурс занят, возможность появления Deadlock и LiveLock.

Реализации:

**Ключевое слово synchronized** – ключевое слово для управления доступом к разделяемым ресурсам, которое выражено механизмом «монитор» - логика, управляющая состоянием мьютекса и обеспечивающая mutual exclusion.

Мьютекс – примитив синхронизации, который имеется в каждом java объекте, является частным случаем семафора, может быть захвачен только одним потоком.

Семафор - примитив синхронизации, выраженный счетчиком количества потоков, которым может быть предоставлен единовременный доступ к критической секции (представлен одноименным классом)

Ключевым словом может быть помечен блок кода или метод. Необходимо указать объект, мьютекс которого будет захвачен. Указание слова в сигнатуре метода – синтаксический сахар с синхронизацией на объекте this или .class.

**Использование классов из пакета concurrent.locks –** предоставляют семантику synchronized, но с расширенным функционалом: возможности более гибкого управления блокировками: таймаут на удержание блокировки, таймаут на ожидание освобождение блокировки, разделение блокировок на чтение и на запись, повторный захват блокировки, наличие условий для выхода из блокировки Condition – класс предоставляющий более гибкие условия для координации потоков, имеет аналоги методов wait, notify, notifyAll в виде await, signal, signalAll, работают по такой же механике.

Реализации: *ReentrantLock, RentrantReadWriteLock.*

Оптимистическая блокировка – предполагает, что конфликты при доступе к ресурсу редки. Работа с ресурсом происходит без блокировки, но при сохранении результата происходит проверка на наличие конфликтов (принцип CAS).

Минусы: Затраты на повторение операций в случае конфликтов (если конкурируют много потоков, то приводит к задержкам выполнения).

Реализации:

CAS (Compare and Swap) – инструкция процессора, одноименная принципу, которая реализует атомарную операцию сравнения и замены значения. Инструкция используется в реализации Atomic классов, которые благодаря этой инструкции обеспечивают атомарные математические операции (без использования классов Atomic математические операции неатомарны, т.к. включают 3 операции: чтение из переменной, изменение ее значения и присваивание.)

Версионное управление – реализация CAS подхода, при котором каждое обновление ресурса связано с изменением его версии, для успешного изменения требуется, чтобы текущая версия ресурса совпала с версией, прочитанной ранее.

**Оптимизации на уровне компилятора или железа:**

**Видимость изменений –** изменения, совершенные потоком, могут не сразу отразится в общей памяти (другой поток может не увидеть этих изменений), что связано с кэшированием этих изменений на уровне процессора или взятие значения из стека потока (а не из общей памяти) для повышения производительности (все кэши рано или поздно синхронизируются по MESI протоколу (эта операция достаточно дорогая, поэтому не происходит постоянно) – обеспечивается свойство когерентности). При этом доставка изменений в основную память не гарантируется.

**Нарушение порядка операций и параллелизация –** компилятор или процессор могут изменять порядок выполнения инструкций для увеличения производительности, такие оптимизации не приведут к видимым последствиям в однопоточной среде, но могут отразится на многопоточной.

Для абстрагирования от особенностей реализаций аппаратных и программных оптимизаций существует Java Memory Model, которая стандартизирует поведение java-приложения на различных устройствах и системах.

**Java memory model** – описывает взаимодействие потоков с общей памятью.

Вводит понятие атомарных операций:

*Атомарность* – атомарной операция считается, если невозможно наблюдать частичный результат ее выполнения. Гарантирует атомарность на чтение и запись всех типов данных (включая Object), кроме long и double, объявление поля как volatile гарантирует атомарность для всех типов. Для всех остальных операций атомарность не гарантирована (арифметические операции).

*Видимость* – результат операции записи, выполненной в потоке А, виден при чтении в потоке Б, видимость определена только для конкретных потоков А и Б, не существует «глобальной видимости». Достигается с помощью ключевого слова volatile.

**Volatile** – запрещает оптимизации такие как кэширование значения и переупорядочивание инструкций, тем самым гарантируя актуальность значения переменной для всех потоков-наблюдателей. Также гарантирует атомарность операций чтения и записи переменных типа long и double в 32 битных системах.

*Порядок* (ordering) – определение последовательности операций в многопоточной программе.

Правила формализованы при помощи happens-before (hb) - все операции записи, произошедшие до точки видны в любой операции чтения после точки B.

Правила happens-before:

1) A hb B, если A раньше B в тексте программы (program order) – порядок исполнения программы.

2) Thread.start() happens-before Thread.run()

3) Последнее действие в потоке hb Thread.join()

4) Инициализация объекта по умолчанию hb любое другое действие

5) Между синхронизациями по одному монитору установлен полный порядок (total order), то есть между двумя критическими секциями можно точно сказать, какая выполнилась раньше, какая позже. Завершение синхронизации hb начало последующей синхронизации по тому же объекту

6) Запись volatile поля happens-before чтение этого поля.

7) Запись в final-переменную происходит до публикации ссылки на этот объект.

8) Инициализация объекта по умолчанию hb любое другое действие.

**Возможность возникновения DeadLock и LiveLock –** проблемы синхронизации, приводящее к ситуации взаимного ожидания освобождения ресурсов. Такие виды проблем возникают только в случае пессимистических блокировок.

**DeadLock -**ситуация, когда оба потока находятся в состоянии вечного ожидания ресурсов другого потока. То есть подразумевается наличие 2х потоков, каждый удерживает по 1 ресурсу и при этом пытается захватить ресурс другого потока.

**Livelock** – ситуация, когда потоки в ожидании освобождения ресурсов выполняют какие-то действия с точки зрения приложения (то есть не просто ожидают), но при этом полезной работы не происходит.

Результатом обоих процессов является Starvation – ситуация, когда потоку не удается получить доступ к ресурсам в результате его блокировки, и он не способен продолжать выполнение.

**Как избежать –** использовать таймауты на время захвата ресурсов, соблюдать последовательность захвата и освобождения ресурсов.

**Создание и работа с потоками:**

Два основных способа создания потока – это наследование от класса Thread и реализация его метода run или имплементация интерфейса Runnable и реализация его метода run.

Thread - это абстракция над физическим потоком, то есть напрямую наследоваться от потока следует только в том случае, если необходимо внести изменения в механизм выполнения потока.

Runnable - это абстракция над выполняемой задачей, отделяет механику исполнения потока от логики исполняемой задачи.

**Методы Thread.**

|  |  |
| --- | --- |
| start() | Запускает поток |
| run() | Вызывается после старта потока (должен быть переопределен), если вызвать напрямую, то это не приведет к многопоточному исполнению, метод вызовется в текущем потоке. |
| Interrupt() | Если поток в состоянии WAIT, WAIT\_TIMED или SLEEPING – то вылетит InterruptedException  В остальных состояниях на поток устанавливается флаг interrupted, что сигнализирует планировщику потоков о необходимости прервать поток.  \*Не завершает поток\* |
| Sleep() | Переводит текущий поток в состояние SLEEPING, при этом поток не отпускает синхронизацию. |
| Join() | Вызвавший поток ожидает окончания потока, на котором вызвал метод.  Механика: вызывавший поток блокируется на мониторе потока на котором вызвали join (каждый поток захватывает свой монитор), и ожидает вызова notify, который произойдет, когда поток завершится и освободит свой монитор. |
| Yield() | Говорит планировщику потоков, что он может забрать у него процессорное время |
| stop, suspend, resume, destroy | Устаревшие |
| Wait()  Notify()/NotifyAll() | переводит состояние потока в WAIT или TIMED WAITING и отпускает монитор.  Пробуждает один/все потоки, заблокированные на текущем мониторе.  wait, notify, notifyAll – работают внутри synchronized блока, если вызвать их не получив монитор, то выбросит IllegalMonitorStateException (вызываются на объекте, по которому синхронизируются)  из-за spurious exit (ложное пробуждение) – wait нужно вызывать в цикле. |

**Жизненный цикл потока** – представлен состояниями потока

New – поток создан, но еще не запущен  
Runnable - у потока был вызван метод start и он ожидает выделения процессорного времени от планировщика потоков или исполняется  
Blocked – поток ожидает освобождения ресурса.  
Waiting – состояние потока после вызова методов wait и join без указания таймера, приводят к ожиданию вызова метода notify или notifyAll на ожидаемом ресурсе.  
TimedWaiting – состояние потока после вызова методов wait, join с последующим notify или метод sleep.  
Terminated – исполнение метода run завершилось и поток считается завершенным.

**Приоритет потока** – определяет порядок выполнения потоков, когда несколько из них готовы к исполнению, то есть планировщик потоков выделяет процессорное время для потока на основании его приоритета (окончательное решение в распределении процессорного времени принимает ОС, планировщик дает рекомендации). Устанавливается с помощью setPriority, по умолчанию 5.

**Daemon-поток** – специальный вид потока, который не препятствует завершению работы программы (завершением программы считается завершение всех исполняемых потоков. JVM не дожидается их завершения, их исполнение просто прерывается). Предназначены для исполнения фоновых процессов (задачи, которые необходимы пока работает программа) – потоки для мониторинга, GC.

**Фреймворк Thread executors:**

Предоставляет механизм распределения задач между потоками и вводит понятие пул потоков – некая область хранения для ранее созданных потоков, что дает возможность их переиспользования, что в свою очередь дает уменьшение затрат по памяти. Также предоставляет возможность асинхронного исполнения с помощью Future.

ExecutorService представляет из себя абстракцию для управления исполнением задач, используя пул потоков. Выбор реализации осуществляется с помощью статической фабрики Executors.

Реализации ExecutorService:

*SingleThreadPool* – имеет 1 поток в пуле  
*FixedThreadPool* – фиксированное кол-во потоков, накапливает задачи в LinkedBlockingQueue.  
*ScheduledThreadPool* – пул, позволяющий планировать исполнение задач (через фиксированные задержки или периодически).  
*CachedThreadPool* – динамически увеличивает/уменьшает кол-во потоков в зависимости от кол-ва задач.

Задачи, которые могут быть определены для этих потоков выражены через Runnable – не возвращает результат, Callable – возвращает результат своей работы в виде исключения или возвращаемого значения. Для возврата значения Callable используется абстракция **Future** – абстракция, которая предоставляет текущий результат выполнения асинхронной операции (работает как Promise в JS – отложенный результат). Попытки получения, отсутствующего (еще не исполненного) результата с помощью метода get приведут к блокировке потока до того момента, пока задача не будет исполнена.

**ForkJoinPool** – реализация ExecutorService для параллельной обработки, т.е. требует задач, исполнение которых является независимым, использует принцип «разделяй и властвуй» - рекурсивно разделяет задачи на подзадачи, также используется концепция work stealing – исполнители «воруют» работу друг у друга, не дожидаясь прямого делегирования работы (исполнители заняты и не простаивают). В качестве возвращаемого результата используется RecursiveAction (не возвращает результат) и RecursiveTask (возвращает результат).

Справочный материал: [статья](https://habr.com/ru/companies/otus/articles/353414/), [видео](https://www.youtube.com/watch?v=m-7EljqdxpA).