Многопоточность. Класс Thread, интерфейс Runnable. Модификатор synchronized.

Processes and Threads

In concurrent programming, there are two basic units of execution: processes and threads. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a ProcessBuilder object. Multiprocess applications are beyond the scope of this lesson.

Threads

Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the

application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads

Когда запускается любое приложение, то начинает выполняться поток, называемый *главным потоком* (main). От него порождаются дочерние потоки. Главный поток, как правило, является последним потоком, завершающим выполнение программы.

Несмотря на то, что главный поток создаётся автоматически, им можно управлять через объект класса **Thread**. Для этого нужно вызвать метод **currentThread()**, после чего можно управлять потоком.

Класс **Thread** содержит несколько методов для управления потоками.

- **getName()** - получить имя потока
- **getPriority()** - получить приоритет потока
- **isAlive()** - определить, выполняется ли поток
- **join()** - ожидать завершение потока
- **run()** - запуск потока. В нём пишите свой код
- **sleep()** - приостановить поток на заданное время
- **start()** - запустить поток

When a program calls the start() method, a new thread is created and then the run() method is executed. But if we directly call the run() method then no new thread will be created and run() method will be executed as a normal method call on the current calling thread itself and no multi-threading will take place.

Interrupt
If this thread is blocked in an invocation of the wait(), wait(long), or wait(long, int) methods of the Object class, or of the join(), join(long), join(long, int), sleep(long), or sleep(long, int), methods of this class, then its interrupt status will be cleared and it will receive an InterruptedException.

If this thread is blocked in an I/O operation upon an InterruptibleChannel then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a ClosedByInterruptException.

If this thread is blocked in a Selector then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's wakeup method were invoked.

If none of the previous conditions hold then this thread's interrupt status will be set.

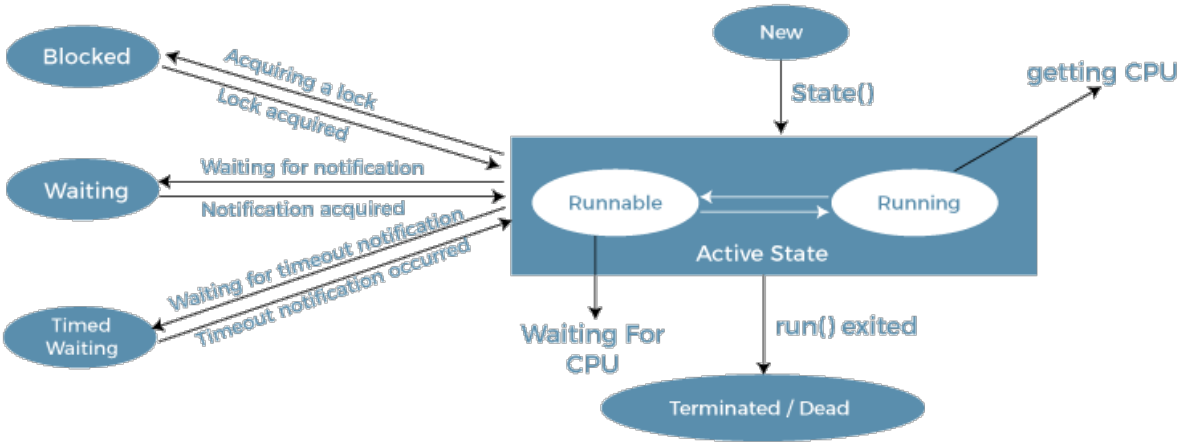
Interrupting a thread that is not alive need not have any effect.

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state. A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.



Life Cycle of a Thread

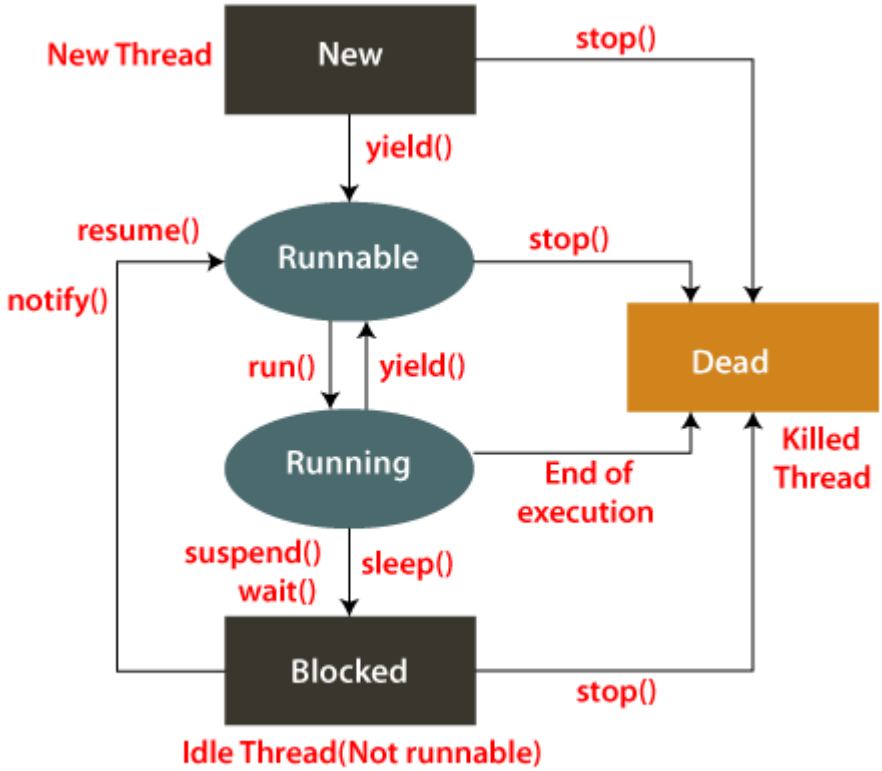


Fig: State Transition Diagram of a Thread

Risks involved in Thread Pools

The following are the risk involved in the thread pools.

Deadlock: It is a known fact that deadlock can come in any program that involves multithreading, and a thread pool introduces another scenario of deadlock.

Consider a scenario where all the threads that are executing are waiting for the results from the threads that are blocked and waiting in the queue because of the non-availability of threads for the execution.

Thread Leakage: Leakage of threads occurs when a thread is being removed from the pool to execute a task but is not returning to it after the completion of the task. For example, when a thread throws the exception and the pool class is not able to catch this exception, then the thread exits and reduces the thread pool size by 1. If the same thing repeats a number of times, then there are fair chances that the pool will become empty, and hence, there are no threads available in the pool for executing other requests.

Resource Thrashing: A lot of time is wasted in context switching among threads when the size of the thread pool is very large. Whenever there are more threads than the optimal number may cause the starvation problem, and it leads to resource thrashing.

Synchronized

A piece of logic marked with synchronized becomes a synchronized block, allowing only one thread to execute at any given time.

We can use the synchronized keyword on different levels:

- Instance methods
- Static methods
- Code blocks

When we use a synchronized block, Java internally uses a monitor, also known as a monitor lock or intrinsic lock, to provide synchronization. These monitors are bound to an object; therefore, all synchronized blocks of the same object can have only one thread executing them at the same time.

Instance methods are synchronized over the instance of the class owning the method, which means only one thread per instance of the class can execute this method.

Static methods are synchronized on the Class object associated with the class. Since only one Class object exists per JVM per class, only one thread can execute inside a static synchronized method per class, irrespective of the number of instances it has.

Reentrancy

The lock behind the synchronized methods and blocks is a reentrant. This means the current thread can acquire the same synchronized lock over and over again while holding it

Volatile

In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared volatile (including long and double variables).

Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelocks

A livelock occurs when two or more threads are actively trying to resolve a conflict or reach a certain state, but their actions prevent any progress from being

made. In a livelock, threads keep responding to each other's actions, resulting in an endless loop of interactions without achieving the desired outcome. This leads to an unproductive execution state, where threads are continuously active but not making any progress.

Lock

Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/notify mechanism, through their associated Condition objects.

The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock. The tryLock method backs out if the lock is not available immediately or before a timeout expires (if specified). The lockInterruptibly method backs out if another thread sends an interrupt before the lock is acquired.

Executors

The java.util.concurrent package defines three executor interfaces:

- Executor, a simple interface that supports launching new tasks.
- ExecutorService, a subinterface of Executor, which adds features that help manage the life cycle, both of the individual tasks and of the executor itself.
- ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

The Executor Interface

The Executor interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

(new Thread(r)).start();

with

e.execute(r);

However, the definition of execute is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute may do the same thing, but is more likely to use an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available

Thread Pools

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are

submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

Lock

Для управления доступом к общему ресурсу в качестве альтернативы оператору synchronized мы можем использовать блокировки. Функциональность блокировок заключена в пакете java.util.concurrent.locks.

Вначале поток пытается получить доступ к общему ресурсу. Если он свободен, то на него накладывает блокировку. После завершения работы блокировка с общего ресурса снимается. Если же ресурс не свободен и на него уже наложена блокировка, то поток ожидает, пока эта блокировка не будет снята.

Condition

Применение объектов Condition во многом аналогично использованию методов wait/notify/notifyAll класса Object, которые были рассмотрены в одной из прошлых тем. В частности, мы можем использовать следующие методы интерфейса Condition:

- await: поток ожидает, пока не будет выполнено некоторое условие и пока другой поток не вызовет методы signal/signalAll. Во многом аналогичен методу wait класса Object
- signal: сигнализирует, что поток, у которого ранее был вызван метод await(), может продолжить работу. Применение аналогично использованию методу notify класса Object
- signalAll: сигнализирует всем потокам, у которых ранее был вызван метод await(), что они могут продолжить работу. Аналогичен методу notifyAll() класса Object

ForkJoinPool

в основе своей ForkJoinPool – это пул потоков, преимущество которого состоит в том, что он работает на основе принципа `WorkStealing`, что дословно можно перевести как «кража работы». Когда один из потоков ForkJoinPool заканчивает свою работу, он не идёт пить кофе или чилить в ютубчике, он проявляет «сознательность» и берёт из общей очереди работ новую задачу. Это продолжается до тех пор, пока задачи не кончатся.

в него нельзя подать Callable или Runnable задачу. У него есть своя иерархия задач, наследуемая от абстрактного класса ForkJoinTask. Основные

реализации – RecursiveTask и RecursiveAction. У каждого из них есть абстрактный метод compute(), который и надо реализовывать при наследование. RecursiveTask. compute() возвращает некое значение, RecursiveAction. compute() возвращает void.

Concurrent Synchronizers, объекты синхронизации

Синхронизация — это процесс, позволяющий выполнять в программе синхронно параллельные потоки. Несколько потоков могут мешать друг другу при обращении к одним и тем же объектам приложения. Для решения этой проблемы используется мьютекс, он же монитор, имеющий два состояния — объект занят и объект свободен. Монитор (мьютекс) — это высокоуровневый механизм взаимодействия и синхронизации процессов, обеспечивающий доступ к неразделяемым ресурсам. Мьютекс встроен в класс Object и, следовательно, имеется у каждого объекта.

Semaphore

(семафор) — объект синхронизации, ограничивающий одновременный доступ к общему ресурсу нескольким потокам с помощью счетчика. При запросе разрешения и значении счетчика больше нуля доступ предоставляется, а счетчик уменьшается; в противном случае — доступ запрещается. При освобождении ресурса значение счетчика семафора увеличивается. Количество разрешений семафора определяется в конструкторе. Второй конструктор семаформа включает дополнительный параметр «справедливости», определяющий порядок предоставления разрешения ожидающим доступа потокам. Описание с примером представлено здесь.

CountDownLatch

(«защелка с обратным отсчетом») — объект синхронизации потоков, блокирующий один или несколько потоков до тех пор, пока не будут выполнены определенные условия. Количество условий задается счетчиком. При обнулении счетчика, т.е. при выполнении всех условий, блокировки выполняемых потоков будут сняты и они продолжат выполнение кода. Примером CountDownLatch может служить экскурсовод, собирающий группу из заданного количества туристов. Как только группа собрана она отправляется на экскурсию. Необходимо отметить, что счетчик одноразовый и не может быть инициализирован по-новому. Описание с примером представлено здесь

CyclicBarrier

— объект синхронизации типа «Барьер» используется, как правило, в распределённых вычислениях. Барьерная синхронизация останавливает участника (исполняемый поток) в определенном месте в ожидании прихода остальных потоков группы. Как только все потоки достигнут барьера, барьер снимается и выполнение потоков продолжается. Циклический барьер CyclicBarrier, также, как и CountDownLatch, использует счетчик и похож на него. Отличие связано с тем, что «защелку» нельзя использовать повторно после того, как её счётчик обнулится, а барьер можно использовать (в цикле). Описание с примером представлено здесь

Exchanger

— объект синхронизации, используемый для двустороннего обмена данными между двумя потоками. При обмене данными допускается null значения, что позволяет использовать класс для односторонней передачи объекта или же просто, как синхронизатор двух потоков. Обмен данными выполняется вызовом метода *exchange*, сопровождаемый самоблокировкой потока. Как только второй поток вызовет метод *exchange*, то синхронизатор *Exchanger* выполнит обмен данными между потоками. Описание с примером представлено здесь

Phaser

— объект синхронизации типа «Барьер», но, в отличие от *CyclicBarrier*, может иметь несколько барьеров (фаз), и количество участников на каждой фазе может быть разным. Описание с примером представлено здесь

Атомарные классы пакета util.concurrent

Операция называется **атомарной**, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию synchronized.

Блокировка подразумевает **пессимистический** подход, разрешая только одному потоку выполнять определенный код, связанный с изменением значения некоторой «общей» переменной. Таким образом, никакой другой поток не имеет доступа к определенным переменным. Но можно использовать и **оптимистический** подход. В этом случае блокировки не происходит, и если поток обнаруживает, что значение переменной изменилось другим потоком, то он повторяет операцию снова, но уже с новым значением переменной. Так работают атомарные классы.

Каждый атомарный класс включает метод **compareAndSet**, представляющий механизм *оптимистичной блокировки* и позволяющий изменить значение value только в том случае, если оно равно ожидаемому значению (т.е. current). Если значение value было изменено в другом потоке, то оно не будет равно ожидаемому значению. Следовательно, метод compareAndSet вернет значение false.

Коллекции из пакета java.util.concurrent

Один из подходов к улучшению масштабируемости коллекции при сохранении потокобезопасности состоит в том, чтобы обходиться **без общей блокировки всей таблицы**, а использовать блокировки для каждого hash backet (или, в более общем случае, пула блокировок, где каждая блокировка защищает несколько бакетов). Это позволяет нескольким потокам обращаться к различным частям коллекции одновременно, без соперничества за единственную на всю коллекцию блокировку. Данный подход улучшает масштабируемость операций вставки, извлечения и удаления.

Класс ConcurrentHashMap появился в пакете java.util.concurrent в JDK 1.5, является потокобезопасной реализацией Map и предоставляет намного большую степень масштабирования (параллелизма), чем synchronizedMap. Отличие ConcurrentHashMap связано с внутренней структурой хранения пар key-value. **ConcurrentHashMap** использует несколько сегментов, и данный класс нужно рассматривать как группу HashMap'ов. Количество сегментов по умолчанию равно 16. Если пара key-value хранится в 10-ом сегменте, то ConcurrentHashMap заблокирует, при необходимости, только 10-й сегмент, и не будет блокировать остальные 15.

Класс CopyOnWriteArrayList следует использовать вместо ArrayList в потоконагруженных приложениях, где могут иметь место нечастые операции вставки и удаления в одних потоках и одновременный перебор в других. Это типично для случая, когда коллекция ArrayList используется для хранения списка объектов.

При использовании обычной ArrayList в многопоточном приложении необходимо либо блокировать целый список во время перебора, либо клонировать его перед перебором; оба варианта требуют дополнительных ресурсов. CopyOnWriteArrayList вместо этого создаёт новую копию списка при выполнении модифицирующей операции и гарантирует, что её итераторы вернут состояние списка на момент создания итератора и не выкинут ConcurrentModificationException. Это так называемый алгоритм CopyOnWrite. Нет необходимости клонировать список до перебора или блокировать его во время перебора, т.к. используемая итератором копия списка изменяться не будет. Другими словами, CopyOnWriteArrayList содержит изменяемую ссылку на неизменяемый массив, поэтому до тех пор, пока эта ссылка остаётся фиксированной, вы получаете все преимущества потокобезопасности от неизменности без необходимости блокировок.

Неблокирующие очереди

Потокобезопасные и неблокирующие очереди на связанных узлах (linked nodes) реализуют интерфейс Queue и его наследника Deque.

ConcurrentLinkedQueue реализуют интерфейс *Queue* и формирует неблокирующую и ориентированную на многопоточное исполнение очередь. Размер очереди *ConcurrentLinkedQueue* не имеет ограничений. Имплементация очереди использует wait-free алгоритм от Michael & Scott, адаптированный для работы с garbage collector'ом. Данный алгоритм довольно эффективен и очень быстр, т.к. построен на CAS (Compare-And-Swap). Описание с примером представлено здесь.

ConcurrentLinkedDeque реализует интерфейс Deque (Double ended queue), читается как «Deck». Данная реализация позволяет добавлять и получать элемента с обеих сторон очереди. Соответственно, класс поддерживает оба режима работы : FIFO (First In First Out) и LIFO (Last In First Out). *ConcurrentLinkedDeque* следует использовать в том случае, если необходимо реализовывать LIFO, поскольку за счет двунаправленности данный класс проигрывает по производительности очереди *ConcurrentLinkedQueue*. Описание с примером представлено здесь.

Блокирующие очереди

При обработке большого количества потоков данных использование неблокирующих очередей иногда может оказаться явно недостаточным : разгребающие очереди потоки перестанут справляться с наплывом данных, что может привести к «out of memory» или перегрузить IO/Net настолько, что производительность упадет в разы, пока не наступит отказ системы по таймаутам или из-за отсутствия свободных дескрипторов в системе. Самое неприятное в данном случае то, что возникающая ситуация является нестабильной, сложной для отладки. Для таких случаев нужна блокирующая очередь с возможностью задать её размер и/или условия блокировки.

Блокирующие очереди реализуют интерфейсы BlockingQueue, BlockingDeque, TransferQueue. Интерфейс *BlockingQueue* хранит элементы в порядке «первый пришел, первый вышел». Добавленные в очередь элементы в определенном порядке, будут извлечены из неё в том же самом порядке. Реализация *BlockingQueue* гарантирует, что любая попытка извлечь элемент из пустой очереди заблокирует вызывающий поток до тех пор, пока не появится доступный элемент. Аналогично , любая попытка вставить элемент в заполненную очередь заблокирует вызывающий поток до тех пор, пока не освободится место для нового элемента. Интерфейс *BlockingDeque* включает дополнительные методы для двунаправленной блокирующей очереди, у которой данные можно добавлять и извлекать с обоих сторон очереди.

Интерфейсы блокирующих очередей наряду с возможностью определения размера очереди включают методы, по-разному реагирующие на незаполнение или переполнение queue. Так, например, при добавлении элемента в переполненную очередь, один из методов вызовет *IllegalStateException*, другой вернет *false*, третий заблокирует поток, пока не появится место, четвертый же заблокирует поток на определенное время (таймаут) и вернет *false*, если место так и не появится.

ArrayBlockingQueue — блокирующая очередь, реализующая классический кольцевой буфер. Параметр *fair* в конструкторе позволяет управлять справедливостью очереди для

упорядочивания работы ожидающих потоков производителей (вставляющих элементы) и потребителей (извлекающих элементы). Описание с примером представлено здесь.

LinkedBlockingQueue — блокирующая очередь на связанных узлах, реализующая «two lock queue» алгоритм : один lock добавляет элемент, второй извлекает. За счет двух lock'ов данная очередь показывает более высокую производительность по сравнению с *ArrayBlockingQueue*, но и расход памяти повышается. Размер очереди задается через конструктор и по умолчанию равен *Integer.MAX_VALUE*. Описание с примером представлено здесь.

LinkedBlockingDeque — двунаправленная блокирующая очередь на связанных узлах, реализованная как простой двунаправленный список с одним локом. Размер очереди задается через конструктор и по умолчанию равен *Integer.MAX_VALUE*. Описание с примером представлено здесь.

SynchronousQueue — блокирующая очередь, в которой каждая операция добавления должна ждать соответствующей операции удаления в другом потоке и наоборот. Т.е. очередь реализует принцип «один вошел, один вышел». *SynchronousQueue* не имеет никакой внутренней емкости, даже емкости в один элемент. Описание с примером представлено здесь.

LinkedTransferQueue — блокирующая очередь с реализацией интерфейса TransferQueue, который позволяет при добавлении элемента в очередь заблокировать вставляющий поток до тех пор, пока другой поток не заберет элемент из очереди. Блокировка может быть как с таймаутом, так и с проверкой ожидающего потока. Таким образом может быть реализован механизм передачи сообщений с поддержкой как синхронных, так и асинхронных сообщений. Подробное описание представлено здесь.

DelayQueue — специфичный вид очереди, позволяющий извлекать элементы только после некоторой задержки, определенной в каждом элементе через метод getDelay интерфейса Delayed. Подробное описание представлено здесь.

PriorityBlockingQueue — является многопоточной оберткой интерфейса PriorityQueue. При размещении элемента в очереди, его порядок определяется в соответствии с «натуральным» упорядочиванием, либо логикой *Comparator'a* или имплементации *Comparable* интерфейса. Подробное описание представлено здесь.

Callable, Future

JDBC. Порядок взаимодействия с базой данных. Класс DriverManager. Интерфейс Connection

Интерфейсы Statement, PreparedStatement, ResultSet, RowSet

Корутины

Channel
A Channel is conceptually very similar to `BlockingQueue`. One key difference is that instead of a blocking `put` operation it has a suspending send, and instead of a blocking `take` operation it has a suspending receive.

Unlike a queue, a channel can be closed to indicate that no more elements are coming.

The channels shown so far had no buffer. Unbuffered channels transfer elements when sender and receiver meet each other (aka rendezvous). If send is invoked first, then it is suspended until receive is invoked, if receive is invoked first, it is suspended until send is invoked.

Send and receive operations to channels are *fair* with respect to the order of their invocation from multiple coroutines. They are served in first-in first-out order, e.g. the first coroutine to invoke `receive` gets the element.

Job - Launch
So job is sort of an object that represents a coroutine's execution

Deffered- Async
Deferred is some kind of analog of Future in Java: in encapsulates an operation that will be finished at some point in future after it's initialization.

Deferred value is a non-blocking cancellable future — it is a Job that has a result.

Корутины

Прежде всего для определения и выполнения **корутины** нам надо определить для нее контекст, так как корутина может вызываться только в контексте корутины (coroutine scope). Для этого применяется функция coroutineScope() - создает контекст корутины. Кроме того, эта функция ожидает выполнения всех определенных внутри нее корутин. Стоит отметить, что coroutineScope() может применяться только в функции с модификатором suspend

Корутина не привязана к конкретному потоку. Она может быть приостановить выполнение в одном потоке, а возобновить выполнение в другом.

RunBlocking
Кроме функции coroutineScope для создания контекста корутины может применяться функция runBlocking

Функция runBlocking блокирует вызывающий поток, пока все корутины внутри вызова runBlocking { ... } не завершат свое выполнение. В этом собственно основное отличие runBlocking от coroutineScope: **coroutineScope не блокирует вызывающий поток, а просто приостанавливает выполнение, освобождания поток для использования другими ресурсами**.

Flow
Using the `List<Int>` result type, means we can only return all the values at once. To represent the stream of values that are being computed asynchronously, we can use a `Flow<Int>` type just like we would use a `Sequence<Int>` type for synchronously computed value

CoroutineDispatcher
Base class to be extended by all coroutine dispatcher implementations.

The following standard implementations are provided by kotlinx.coroutines as properties on the Dispatchers object:

- Dispatchers.Default — is used by all standard builders if no dispatcher or any other ContinuationInterceptor is specified in their context. It uses a common pool of shared background threads. This is an appropriate choice for compute-intensive coroutines that consume CPU resources.
- Dispatchers.IO — uses a shared pool of on-demand created threads and is designed for offloading of IO-intensive blocking operations (like file I/O and blocking socket I/O).
- Dispatchers.Unconfined — starts coroutine execution in the current call-frame until the first suspension, whereupon the coroutine builder function returns. The coroutine will later resume in whatever thread used by the corresponding suspending function, without confining it to any specific thread or pool. The Unconfined dispatcher should not normally be used in code.
- Private thread pools can be created with newSingleThreadContext and newFixedThreadPoolContext.
- An arbitrary java.util.concurrent.Executor can be converted to a dispatcher with the asCoroutineDispatcher extension function.