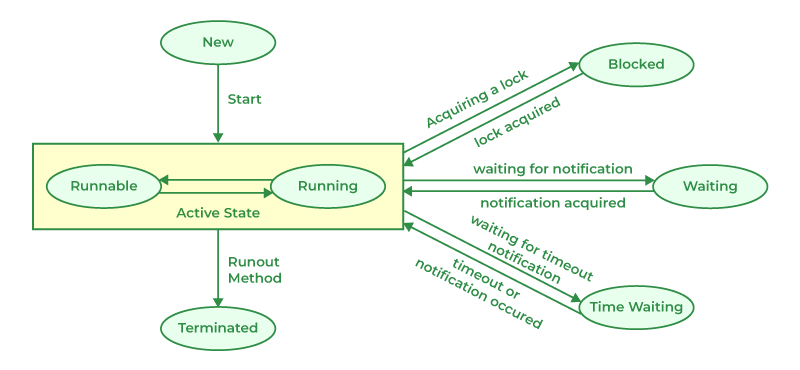
# **Concurrency is everywhere!**

A multithreaded program is always much more complex than a single-threaded program.

What makes multithreading difficult:

* Not all operations are atomic (inconsistent data, race conditions)
* Not all data is shared (Outdated Data)

# **Thread states**



# **Ways to create threads.**

**1. Extending the Thread Class**

The Thread class in Java represents a single thread of execution. You can create a new thread by extending the Thread class and overriding its run() method.

A screen shot of a computer code

Description automatically generated

**2. Implementing the Runnable Interface**

If a class cannot extend Thread (e.g., it’s already extending another class), you can implement the Runnable interface. This approach separates the task from the thread.

A computer screen shot of text

Description automatically generated

**3. Using Lambda Expressions**

If the task to be executed by a thread is small, you can use a lambda expression for concise code. This approach works because Runnable is a functional interface.

A computer screen shot of a computer code

Description automatically generated

# **Wait, Notify, NotifyAll**

Java has a built-in wait mechanism that enable threads to become inactive while waiting for signals from other threads. Class java.lang.Object defines three methods, wait(), notify(), and notifyAll(), to facilitate this.

A screenshot of a computer program

Description automatically generated

A diagram of a signal object

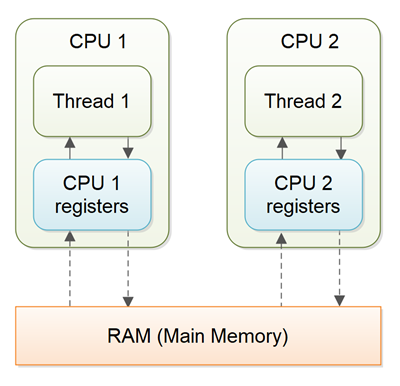
Description automatically generated

Multiple threads can call wait() on the same monitor object - and thus become blocked waiting for a notify() or notifyAll() call. Calling notify() will only awaken a single waiting thread. Calling notifyAll() will awaken all waiting threads.

A thread cannot call wait(), notify() or notifyAll() without holding the synchronization lock on the object the method is called on. If it does, an IllegalMonitorStateException is thrown.

# **Volatile**

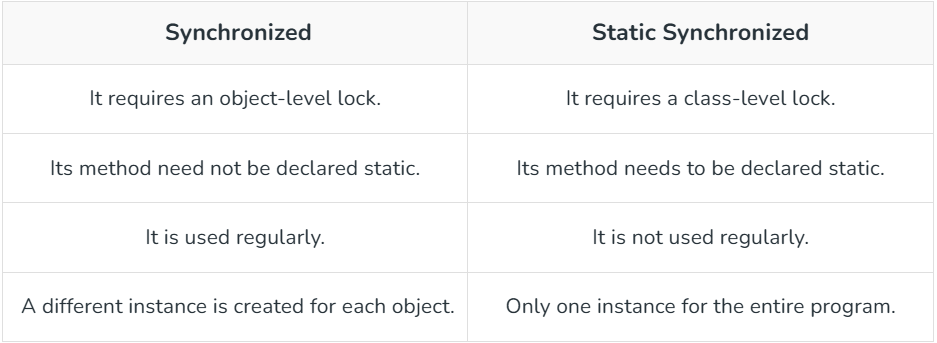
The Java **volatile** keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU registers, and that every write to a volatile variable will be written to main memory, and not just to the CPU registers.



With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU registers, or writes data from CPU registers to main memory.

# **Synchronized**

A *Java* *synchronized* *block* marks a method or a block of code as *synchronized*. A synchronized block in Java can only be executed a single thread at a time (depending on how you use it). Java synchronized blocks can thus be used to avoid race conditions.



# **DeadLock**

A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding. Deadlock can occur when multiple threads need the same locks, at the same time, but obtain them in different order.

A diagram of a blockchain

Description automatically generated

# **Callable and Future**

**Callable** is a functional interface in Java introduced in **Java 5**. It represents a task that can be executed by a thread and is capable of returning a result or throwing a checked exception.

Callable is similar to Runnable, but it can return a result (V) and throw exceptions.

**Future** is an interface in Java that acts as a placeholder for the result of an asynchronous computation. It is used to represent the pending result of a **Callable** task executed by a thread.

**Future** provides methods to check if the computation is complete, retrieve the result, or cancel the task.

**Relationship Between Callable and Future**

* **Callable** defines the task and its result type.
* **Future** provides a way to manage and retrieve the result of the Callable task once it has been executed.

# **Executor Service**

Executors are a framework in Java for managing and controlling thread execution. They simplify thread creation and management by abstracting the complexity of creating and managing threads manually.

**- newSingleThreadExecutor():** Creates an executor with a single thread.

**- newFixedThreadPool(int n):** Creates a pool with a fixed number of threads.

**- newCachedThreadPool():** Creates a pool with dynamically growing threads.

**- newScheduledThreadPool(int n):** Creates a pool for scheduling tasks at fixed rates or delays.

**ExecutorService** is a subinterface of **Executor**. It provides more advanced methods to manage threads, control task execution, and gracefully shut down the executor.

**- submit(Runnable task):** Submits a task for execution.

**- submit(Callable<T> task):** Submits a task for execution and returns a Future to retrieve the result.

**- invokeAll(Collection<? extends Callable<T>> tasks):** Executes multiple tasks and waits for all to complete.

**- invokeAny(Collection<? extends Callable<T>> tasks):** Executes multiple tasks and returns the result of the first successful one.

**- shutdown():** Gracefully shuts down the executor, completing all submitted tasks.

**- shutdownNow():** Attempts to stop all running tasks immediately.

# **Multithreading, Concurrency, Parallelism and Async**

**Concurrency**

Concurrency means executing multiple tasks at the same time but not necessarily simultaneously. In a concurrent application, two tasks can start, run, and complete in overlapping time periods i.e Task-2 can start even before Task-1 gets completed.

Cartoon mouses on a black background

Description automatically generated

In a single core environment (i.e your processor is having a single core), concurrency is achieved via a process called [context-switching](http://www.linfo.org/context_switch.html). If its a multi-core environment, concurrency can be achieved through parallelism.



**Parallelism**

Parallelism means performing two or more tasks simultaneously. Parallel computing in computer science refers to the process of performing multiple calculations simultaneously.

A red and blue line with black text

Description automatically generated

Concurrency is about managing multiple tasks at once.

Parallelism is about executing multiple tasks simultaneously.

**How is concurrency related to parallelism?**

* Concurrency and Parallelism refer to computer architectures which focus on how our tasks or computations are performed.
* ***In a single core environment, concurrency happens*** with tasks executing over same time period via context switching i.e at a particular time period, only a single task gets executed.
* ***In a multi-core environment, concurrency can be achieved via parallelism***in which multiple tasks are executed simultaneously.

## **Threads**

Threads are a sequence of execution of code which can be executed independently of one another. **It is the smallest unit of tasks that can be executed by an OS. A program can be single threaded or multi-threaded.**

## **Process**

A process is an instance of a running program. **A program can have multiple processes. A process usually starts with a single thread i.e a primary thread but later down the line of execution it can create multiple threads.**

A blue circle with white lines and white text

Description automatically generated

**Synchronous and Asynchronous**

In a synchronous programming model, tasks are executed one after another. Each task waits for any previous task to complete and then gets executed.

In an asynchronous programming model, when one task gets executed, you could switch to a different task without waiting for the previous to get completed.

**Synchronous and Asynchronous in a single and multi-threaded environment**

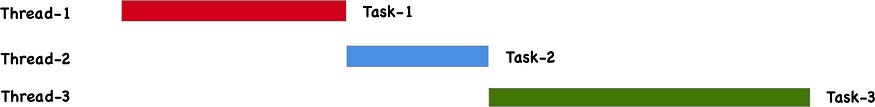
**Synchronous**

***Single Threaded:***



Each task gets executed one after another. Each task waits for its previous task to get executed.

***Multi-Threaded:***



Tasks get executed in different threads but wait for any other executing tasks on any other thread.

**Asynchronous**

***Single Threaded:***



Tasks start executing without waiting for a different task to finish. At a given time a single task gets executed.

***Multi-Threaded:***

A purple and black rectangular object with black text

Description automatically generated

Tasks get executed in different threads without waiting for any tasks and independently finish off their executions.

**Concurrency**

Concurrency is the ability of a system to handle multiple tasks at the same time. It doesn't necessarily mean that these tasks are being executed simultaneously, but rather that they are in progress at the same time. Concurrency is about dealing with lots of things at once, and it can be achieved through various means, such as time-slicing, where tasks take turns using the CPU.

**Parallelism**

Parallelism, on the other hand, is a subset of concurrency where tasks are actually executed simultaneously. This requires multiple processors or cores. Parallelism is about doing lots of things at once. For example, if you have a multi-core processor, you can run multiple threads in parallel, each on a different core.

**Single Threading**

Single threading refers to the execution of tasks in a single sequence. In a single-threaded environment, tasks are executed one after another, and only one task is processed at any given time. This is simpler to manage but can be less efficient for tasks that could benefit from being run concurrently or in parallel.

**Multithreading**

Multithreading is the ability of a CPU, or a single core in a multi-core processor, to provide multiple threads of execution concurrently. This is achieved by dividing a program into multiple threads, which can be executed independently. Multithreading can improve the performance of applications by allowing multiple operations to be performed simultaneously, especially on multi-core systems.

**Synchronous**

Synchronous operations are those that are executed one after the other. Each operation must complete before the next one begins. This can lead to inefficiencies, especially if an operation involves waiting (e.g., for I/O operations), as the entire process is blocked until the operation completes.

**Asynchronous**

Asynchronous operations allow tasks to be executed independently of the main program flow. This means that a task can be initiated and then left to complete on its own, allowing the program to continue executing other tasks. Asynchronous programming is particularly useful for I/O-bound operations, where waiting for a response can be done in the background, freeing up the main thread to perform other tasks.

**Summary**

* **Concurrency** is about managing multiple tasks at once, not necessarily simultaneously.
* **Parallelism** is about executing multiple tasks simultaneously, requiring multiple cores.
* **Single Threading** involves executing tasks one at a time in sequence.
* **Multithreading** allows multiple threads to be executed concurrently, improving performance.
* **Synchronous** operations are sequential and blocking.
* **Asynchronous** operations are non-blocking and can run independently of the main program flow.

# **Fork Join Framework**

1. **Purpose**: The Fork/Join Framework is designed to efficiently execute tasks that can be broken down into smaller subtasks, which can be processed in parallel. It's particularly useful for recursive algorithms that can be divided into independent subtasks.
2. **Core Components**:
   * **ForkJoinPool**: This is the executor service that manages the worker threads. It is responsible for executing ForkJoinTask instances.
   * **ForkJoinTask**: This is the base class for tasks that run within a ForkJoinPool. It provides methods like fork() and join().
   * **RecursiveTask**: A subclass of ForkJoinTask used when a task returns a result.
   * **RecursiveAction**: A subclass of ForkJoinTask used when a task does not return a result.

How It Works

1. **Divide and Conquer**: The framework uses a divide-and-conquer approach. A large task is divided into smaller subtasks, which are then processed in parallel. The results of these subtasks are combined to produce the final result.
2. **Work-Stealing Algorithm**: The Fork/Join Framework uses a work-stealing algorithm to balance the load among worker threads. If a thread completes its tasks, it can "steal" tasks from other threads that are still busy, improving efficiency and resource utilization.

# **CompletableFuture**

**Formula to calculate number of threads in thread pool**

A screenshot of a computer code

Description automatically generated

# **Thread Local**

A diagram of a thread

Description automatically generated

The *Java* *ThreadLocal* class enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to the same ThreadLocal variable, the two threads cannot see each other's ThreadLocal variables. Thus, the Java ThreadLocal class provides a simple way to make code [**thread safe**](http://tutorials.jenkov.com/java-concurrency/thread-safety.html) that would not otherwise be so.



The first way to specify an initial value for a Java ThreadLocal variable is to create a subclass of ThreadLocal which overrides its initialValue() method. The easiest way to create a subclass of ThreadLocal is to simply create an anonymous subclass, right where you create the ThreadLocal variable.

