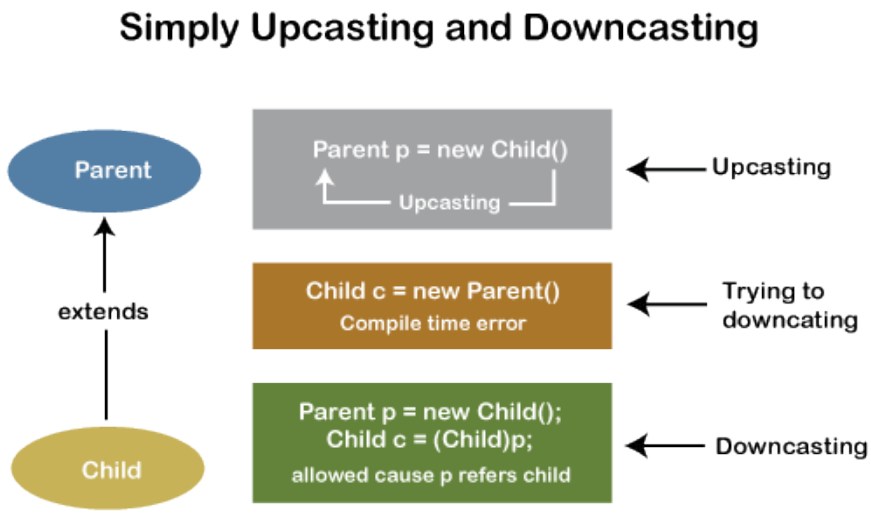
Week 3

Monday, April 3, 2023 5:39 PM

Week 3:

- Multithreading
- Synchronisation, Lock, Re-entrant Lock, Join
- Executor-Framework
- Future & Callable

Typecasting :



Process :

A process is basically a program in execution. It is a sequence of instructions.
A program requires memory and other OS resources to run it. The resources such that registers, program counter, and a stack, and these resources are provided by the OS.

Thread :

Thread is the segment of a process which means a process can have multiple threads and these multiple threads are contained within a process. A thread has three states: Running, Ready, and Blocked. Threads shares the memory with other thread.
The priority of each thread varies. Higher priority threads are executed before lower priority threads.
Thread is critical in the program because it enables multiple operations to take place within a single method. Each thread in the program often has its own program counter, stack, and local variable.

Thread Lifecycle :

The lifecycle of a thread in Java is as follows:
New: When a thread is created, it is in the new state. The thread has not yet started to execute and has not been assigned to a processor.

Runnable: When the thread is ready to run, it moves to the runnable state. In this state, the thread can be scheduled to run on the processor.

Running: Once the thread is scheduled to run by the operating system, it moves to the running state. The thread is now executing its code.

Waiting: When a thread is waiting for another thread or process to complete a task, it enters the waiting state. This state can be caused by calling the wait() method or by waiting for I/O operations to complete.

Timed Waiting: When a thread is waiting for a specified period of time, it enters the timed waiting state. This state can be caused by calling the sleep() or join() methods.

Blocked: When a thread is blocked, it is waiting for a monitor lock to be released by another thread. This state can be caused by calling the synchronized keyword or by waiting for I/O operations to complete.

Terminated: When a thread has completed its task, it enters the terminated state and its resources are released.
It is important to note that the transition from one state to another depends on the underlying operating system and JVM implementation.

Sleep() method :

When sleep() method is invoked on a thread, the thread enters into the TIMED_WAITING state for the specified amount of time. During this time, the thread is not performing any useful work and releases the CPU to other threads that are ready to run.
After the specified time period is over, the thread enters into the RUNNABLE state and will wait for a CPU to become available to execute the thread's tasks. Once the CPU is available, the thread will be picked up by the thread scheduler and run again.
It's important to note that sleep() method does not release the lock on any object that the thread might hold. So, if the thread is holding a lock on an object, other threads will not be able to access that object until the thread holding the lock releases it.

This difference is more obvious from the fact that, when a thread calls the **wait() method**, it releases the monitor or lock it was holding on that object, but when a thread calls the **sleep() method**, it never releases the monitor even if it is holding.

Coming back to **yield()**, it's little different than wait() and sleep(), it just releases the CPU hold by Thread to give another thread an opportunity to run though it's not guaranteed who will get the CPU.

It totally depends upon thread scheduler and it's even possible that the thread which calls the yield() method gets the CPU again. Hence, it's not reliable to depend upon the yield() method, it's just on the best effort basis.

The main difference between wait and sleep is that wait() method releases the acquired monitor when the thread is waiting while Thread.sleep() method keeps the lock or monitor even if the thread is waiting. Also, wait for the method in Java should be called from a synchronized method or block while there is no such requirement for sleep() method.

When sleep() method is invoked, the thread goes into the TIMED_WAITING state. This means that the thread is still alive and waiting for a specific period of time before it can continue executing its tasks.

During this time, the thread is temporarily blocked and gives up the CPU to other threads that are ready to run. Once the specified time has elapsed, the thread returns to the RUNNABLE state and continues executing its tasks.

https://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html#axzz7y1jgcGhV

Multi Threading :

Multithreading is a model of program execution that allows for multiple threads to be created within a process, executing independently but concurrently sharing process resources. Depending on the hardware, threads can run fully parallel if they are distributed to their own CPU core.

Critical Section :

A critical section refers to a section of code or a portion of a program where shared resources, such as shared data or shared hardware devices, are accessed or modified by multiple threads or processes concurrently.

Racing Condition :

A racing condition, also known as a race condition, is a type of concurrency-related issue that occurs when multiple threads access shared data concurrently without proper synchronization, leading to unpredictable and undesirable behavior. In other words, racing condition occurs when two or more threads access shared data simultaneously, resulting in unexpected and unintended outcomes due to their interleaved execution.

Example of racing condition :

Increment operation is causing this problem because it is not an atomic operation. It involves reading the value of a variable, incrementing it, and then writing it back to memory. In a multithreaded environment, if two threads read the same value of a variable at the same time, both threads may increment the value and write it back to memory. This can result in lost updates or incorrect results.
For example, if two threads are trying to increment a counter variable, and the counter is initially 0, both threads may read the value of the counter as 0 at the same time. They may both increment the counter and write the value back to memory, resulting in a final value of 1 instead of 2. This is a classic example of a race condition.

Class Level Lock vs Object Level Lock :

Class level lock is achieved by keyword "Static Synchronized", whereas object level is achieved only by synchronized keyword. Object level lock is achieved to restrict same object to operate through different thread, whereas class level lock is achieved to restrict any object to operate.

Object Level Lock :

In Java, every object has a built-in lock that can be used to provide synchronization at the object level. This is called an object-level lock or intrinsic lock.
When a synchronized block or method is executed on an object, the object's lock is acquired by the executing thread. Other threads attempting to execute synchronized blocks or methods on the same object will be blocked until the lock is released by the executing thread.

Daemon Thread :

Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.
In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in life other than to serve user threads.
Example of Daemon Thread in Java: Garbage collection in Java (gc), finalizer, etc.

https://www.geeksforgeeks.org/daemon-thread-java/

Volatile Keyword :

https://www.baeldung.com/java-volatile

Thread Pool :

A thread pool is a collection of pre-initialized threads that are created to perform a number of tasks concurrently. Thread pools are used in programs where a large number of tasks need to be performed, and creating a new thread for each task is inefficient due to the overhead of creating and destroying threads.

When a task is submitted to a thread pool, the pool assigns it to an available thread. Once the task is completed, the thread is returned to the pool and can be used to execute another task. This reduces the overhead of creating and destroying threads for each task and improves performance.

Thread pools are especially useful in server applications where multiple clients can submit tasks concurrently. By limiting the number of threads in the pool, we can avoid resource starvation (resource starvation is a problem encountered in concurrent computing where a process is perpetually denied necessary resources to process its work) and ensure that the system remains responsive. Additionally, thread pools can be used to limit the number of threads that are spawned by an application, which can help reduce resource utilization and improve overall system stability.

The pattern allows us to control the number of threads the application creates and their life cycle. We're also able to schedule tasks' execution and keep incoming tasks in a queue.

New Thread() :
If you create a new Thread object in Java but do not call the start() method on it, then no new operating system thread will be created, and the run() method of the Thread object will not be executed.
In other words, the Thread object will not do anything unless you explicitly start it using the start() method. If you do not call start(), the Thread object will simply be a regular Java object and can be used like any other object.

Thread Pool Types :

There are mainly two types of thread pools in Java:

Fixed Thread Pool:

In this type of thread pool, a fixed number of threads are created in advance and reused for the execution of the submitted tasks. Once a task is completed, the thread returns to the pool and waits for the next task. If all threads are busy executing the tasks, then new tasks are queued until a thread becomes available.
Thread Queue -> Fixed
Task Queue -> Unbounded Queue by default but can be made bounded and blocking

Fixed thread pool uses LinkedBlockingQueue for queueing tasks.
An optionally-bounded blocking queue based on linked nodes.
This queue orders elements FIFO(first-in-first-out).
The head of the queue is that element that has been on the queue the longest time.
The tail of the queue is that element that has been on the queue the shortest time.
New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.
The optional capacity bound constructor argument serves as a way to prevent excessive queue expansion.

The capacity, if unspecified, is equal to Integer.MAX_VALUE (2^31-1)
Linked nodes are dynamically created upon each insertion unless this would bring the queue above capacity.

put(E e) :
Inserts the specified element at the tail of this queue, waiting if necessary for space to become available.
From <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html>
https://medium.com/@amardeepbhowmick92/task-queuing-in-executors-newfixedthreadpool-31bc8c24b4d2

ThreadPoolExecutor.CallerRunsPolicy :
A handler for rejected tasks that runs the rejected task directly in the calling thread of the execute method, unless the executor has been shut down, in which case the task is discarded

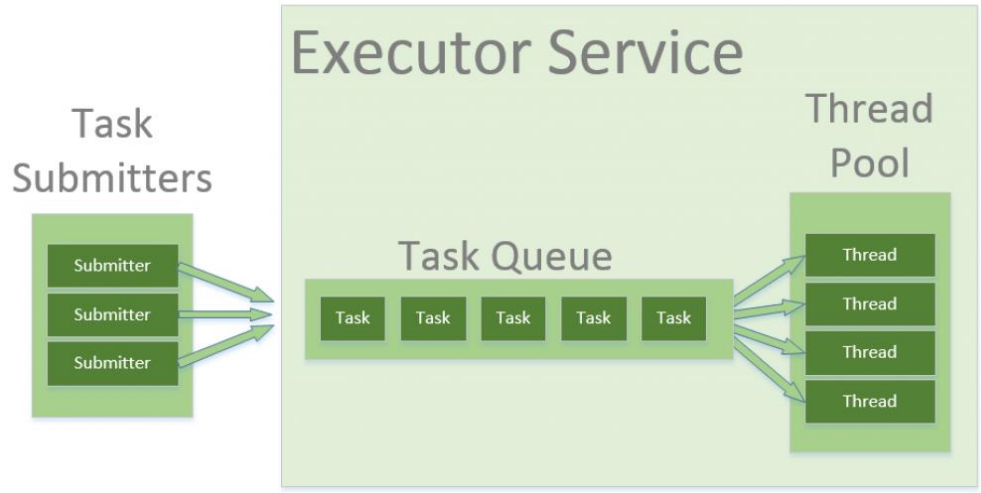
Cached Thread Pool:

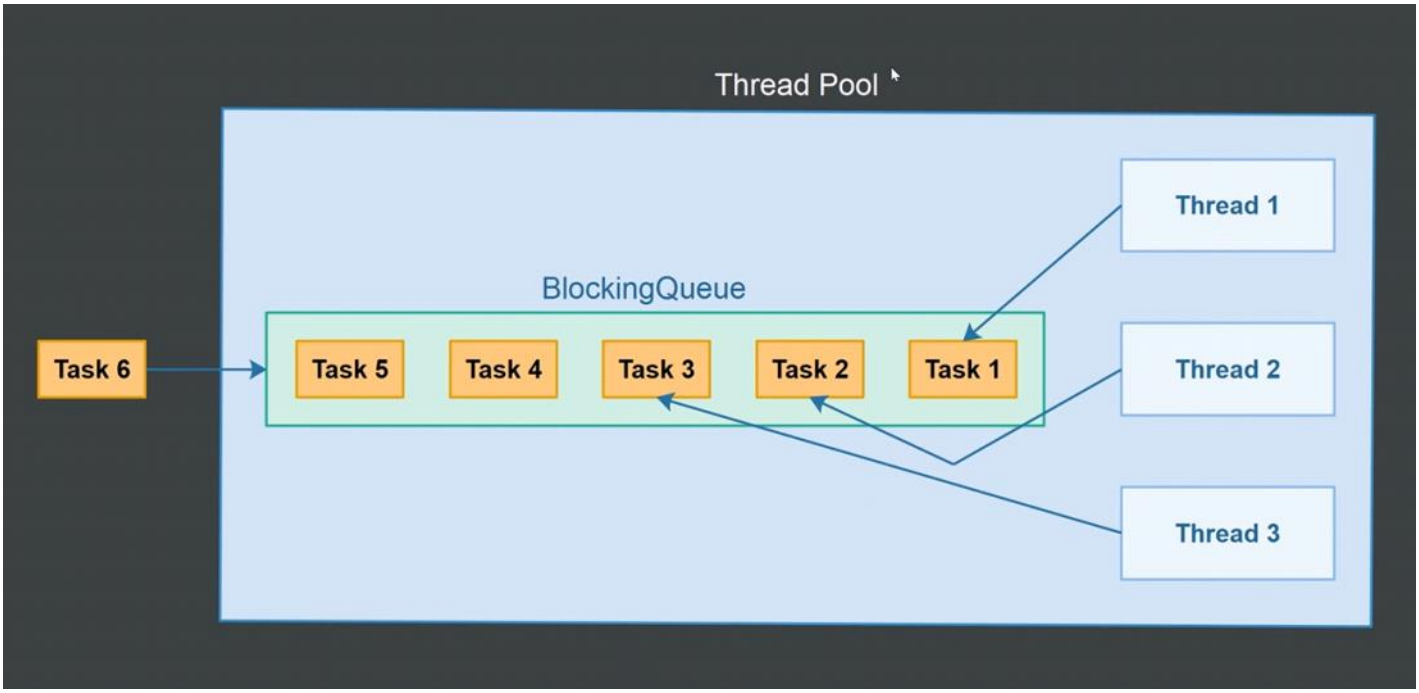
In this type of thread pool, threads are created on-demand as new tasks are submitted for execution. If a thread is idle for a certain amount of time (60 seconds by default), it is terminated and removed from the pool. If new tasks are submitted when all threads are busy, a new thread is created to handle the task. This type of thread pool is useful when there is a large number of short-lived tasks to be executed.

Runnable vs Callable :

Runnable's single method does not throw an exception and does not return a value.
The Callable interface may be more convenient, as it allows us to throw an exception and return a value.

Thread Pools In Java :





Executors, Executor, ExecutorService :

The Executors helper class contains several methods for the creation of preconfigured thread pool instances. Those classes are a good place to start. We can use them if we don't need to apply any custom fine-tuning.

We use the Executor and ExecutorService interfaces to work with different thread pool implementations in Java. Usually, we should keep our code decoupled from the actual implementation of the thread pool and use these interfaces throughout our application.

Executor :

The Executor interface has a single execute method to submit Runnable instances for execution.

```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(() -> System.out.println("Hello World"));
```

ExecutorInterface :

CountDownLatch :

CountDownLatch is a synchronization aid provided by Java's concurrency library that allows one or more threads to wait until a set of operations being performed in other threads completes. It is used to synchronize the execution of multiple threads. CountDownLatch is used to make sure that a task waits for other threads before it starts.

A CountDownLatch is initialized with a count, which is the number of times the countDown() method must be invoked before the threads waiting on the latch can proceed. The threads that call the countDown() method do not block, but the threads waiting on the latch will block until the count reaches zero.

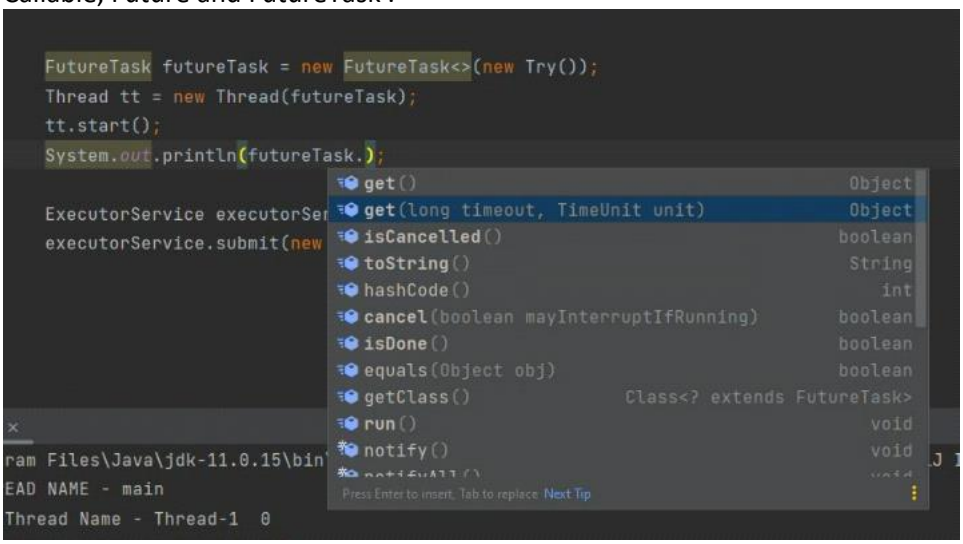
countDown() method decrements the count and await() method blocks until count == 0.

Future :

Future is an interface in Java that represents the result of an asynchronous computation, typically a computation that is executed in a separate thread. It is used for asynchronous programming, where the calling thread can continue its execution without blocking and wait for the result of the computation at a later time.

The Future interface provides methods for checking if the computation is complete, retrieving the result of the computation, and canceling the computation. It is often used in conjunction with ExecutorService to submit tasks for execution in separate threads and obtain Future objects representing the results of those tasks.

Callable, Future and FutureTask :



We use interthread communication in prod cons,

Concurrency and parallelism?

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.

Parallelism is when tasks literally run at the same time, e.g., on a multicore processor.

Quoting Sun's Multithreaded Programming Guide:

> Concurrency: A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.
> Parallelism: A condition that arises when at least two threads are executing simultaneously.

CPU context switch happens in concurrent
Parallelism they execute independently

In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution.

In the same multithreaded process in a shared-memory multiprocessor environment, each thread in the process can run on a separate processor at the same time, resulting in parallel execution.

When the process has fewer or as many threads as there are processors, the threads support system in conjunction with the operating environment ensure that each thread runs on a different processor.

Synchronized vs lock :

Synchronized keyword can be acquired at method level or block level and is released with the same method.

Locks can be acquired in one method and can be released in another method.

For each req, one thread created
Tomcat can handle 200 req i.e. create 200 threads
Sync vs async
Async -> messaging broker
Sync -> rest api call