10-multi-threading-model

Contents

- Threading
  - Thread
  - Synchronization
  - Communication
  - Control
  - Model
  - Main
  - Creating
  - Join
  - Priority
  - State
  - Volatile
  - Builtin
  - Future

Threading

Basically threading is split into two major categories, with which most people are familiar

- process multi-threading - this is virtually supported by any modern OS out there, this is the ability of the operating system to run multiple processes concurrently, each of which has it's own address and data space, unrelated to the other process, each process shares a specific amount of CPU time and all that is governed by the operating system.
- thread based - this one is related to having multiple threads execute in the process or the program

itself, this allows the program to perform multiple tasks concurrently, usually this is controlled by the run-time the program is running on, for our case this is the JVM.

Thread

A thread in java can exist in multiple states - running, stopped, paused / blocked. In java each thread is assigned a priority, this is some integral value which is used to determine if the thread should be given more time to execute, this is very similar to how operating systems threat multi tasking in process based threading, where there can be certain processes which are given more cpu time (e.g. When a process has been idle for a while, it might be given more cpu time next time it is picked up to effectively "catch up" with execution) thread priority and process priority can be dynamic, meaning that it can change during the execution or lifetime of the thread. When a switching from one thread to another, the run-time does a context switch, this is the same term used in operating systems when one process is paused before control is given to another. There are two ways a thread context switch can occur

- A thread can voluntarily relinquish control - this is usually done explicitly by the thread, yielding, sleeping or blocking on pending I/O. Then a higher priority thread is picked from the pool and it's execution is re-started
- A thread can be preempted - this is done when a higher priority thread can cause a lower priority thread. As soon as a higher priority thread desires to be run, it "overrides" any lower priority thread that might be running at the current time

Synchronization

This is a very common problem where multiple threads might want to operate on the same data/object in this case some sort of resource locking has to be performed, otherwise there is no guarantee what will happen with the state of that object or data if multiple threads start modifying it without any order or structure. The Java run time implements a way to guard against this with something called Synchronization, this is the process of which a given object or some of it's method can be 'locked' for access by any other thread, until the current one relinquishes control over that synchronized block/method or data. This way when a given thread enteres a synchronized block it locks the resource, and any other thread which calls synchronized block on the same data/resource will block/wait until the lock is released.

This model is the so called monitor model, and each Java object has an internal `monitor` associated with it.

```
Any time that you have a method, or group of methods, that manipulates the internal state
 of an object in a multithreaded situation, you should use the synchronized keyword to
guard the state from race conditions. Remember, once a thread enters ANY SYNCHRONIZED
METHOD ON AN INSTANCE, NO OTHER THREAD CAN ENTER ANY OTHER SYNCHRONIZED METHOD on the same
 instance.
```

To synchronize a class or object we have two options really, which provide different capabilities based on the option we go with, in either case we have to know that when an object is locked on, either in synchronize block or a synchronized method is called, the entire object is locked by the lock monitor

- Provide synchronize keyword in front of the relevant class members - variables and/or methods
- Wrap an instance of the object in a synchronized block - a way to synchronize a non-synchronized class

```
class Synced {

    private int k = 0;

    public synchronized void() {
```

```
        // the entire method is locked, actually the entire instance of
           Synced is locked by it's own monitor
        k++;
    }
}
class Synced {

    private int k = 0;

    public void() {
        synchronize(this) {
            // same idea, we are locking the object, however the lock has
               more fine grained control
        }
        // non synchronized code can be added here, unlike the example above
           where the entire method locks
    }
}
```

Communication

An extension of the synchronize model above, java provides a way to have multiple threads communicate with each other when a certain situation has arisen, as discussed the Object class has 3 special final methods called `wait` `notify` and `notifyAll`, these are used to have a way to bounce control between threads when specific conditions are met or not met, in effect instead of having a thread poll for a state, the thread is paused,on certain condition, when that condition has changed or has been fulfilled, the thread can be woken up, and continue its execution the next time the scheduler picks it up, starting exactly from the position `wait` was called

- `wait()` tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify() or notifyAll().
- `notify()` wakes up a thread that called wait() on the same object.
- `notifyAll()` wakes up all the threads that called wait() on the same object. One of the threads will be granted access.

`Although wait()normally waits until notify()or notifyAll()is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a spurious wakeup. In this case, a waiting thread resumes without notify()or notifyAll()having been called. (In essence, the thread resumes for no apparent reason.)Because of this remote possibility, Oracle recommends that calls to wait()should take place within a loop that checks the condition on which the thread is waiting.`

```
public class Queue {

    private void int value = 0;
    private boolean valueSet = false;

    public synchronized void put(int v) {
        // if a value is set, wait until the value has been `retrieved` first
        while(valueSet) {
            try {
```

```
                wait(); // yield the current thread, until there is no value
                    set
            } catch(InterruptedException e) {
                // do something with this ex
            }
        }

        // at this point, reaching here means the loop has no sufficed,
            therefore a valueSet did become false
        // meaning that if it became false, a value was extracted/retrieved
            see get method below, therefore
        // we are now free to set a new value and reset the valueSet = true
            again
        this.valueSet = true;
        this.value = v;
        this.notifyAll();
    }

    public synchronized void get() {
        // if a value is not set, wait until the value has been `set` first
        while(!valueSet) {
            try {
                wait(); // yield the current thread, until there is value set
            } catch(InterruptedException e) {
                // do something with this ex
            }
        }

        // at this point, reaching here means the loop has not sufficed,
            therefore a valueSet did become true
        // meaning that if it became true, a value was set/updated see put
            method above, therefore
        // we are now free to get the value that was set, and reset the
            valueSet = false
        this.valueSet = false;
        this.notifyAll();
        return this.value;
    }
}
public class Consumer {
    private Queue q;

    Consumer(Queue q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
```

```
}
public class Producer implements Runnable {
    private Queue q;

    public Producer(Queue q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}

// create both consumer and producer, the producer will keep adding items
   that the consumer will take from
// this is very similar to polling, but here the thread's state is actually
   suspended, there is no CPU work
// being done to poll for the state of the producer or consumer doing wasted
   work
Queue q = new Q();
new Producer(q);
new Consumer(q);
```

Control

In the past around JDK 2 and prior the Thread class had a few additional methods, such as suspend, stop and resume, which were used to control the state of the thread, however those were all deprecated since they had design issues and could cause the entire program, even the run-time to terminate execution unexpectedly. The way we do this is to go back to our trusted wait() and state check. From Java 2 onward it is preferred and pretty much mandated that the run() implementation checks on a state or flag based on which it decides to pause the thread with wait(), this way the API is kept simple and we re-use already existing methods to control the state of the thread.

```
class MyProcess implements Runnable {
    Thread t;
    boolean needsSuspend = false;
    boolean workNeedsToBeDone = true;

    public MyProcess() {
        t = new Thread(this, "myprocess");
        t.start();
    }
    public void run() {
        try {
            while(workNeedsToBeDone) {
                // long running work process
                synchronize(this) {
```

```
                    // another thread will either suspend or resume the state
                        of this one
                    while(needsSuspend) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
    public synchronized void suspend() {
        // will be called and set from another thread
        this.needsSuspend = true
    }
    public synchronized void resume() {
        // will be called and set from another thread
        this.needsSuspend = false
        notifyAll();
    }
}
```

With the example above we can see that now once we create our new object of type MyProcess, we can control when the execution of the thread is suspended/paused and when resumed by calling the provided methods, which will either pause it or resume it.

Model

The Java thread model is build around the Thread class and it's companion the Runnable interface. The Thread class provides a way for the user to interact with the state of the Thread, the Thread object itself is spawned by other means which will be examined later.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Main

The very first thread which is automatically created by the run-time is the Main thread, it is the one created first, and very often the one that is terminated last. One can obtain the current thread, from any other thread/context by using the public static method of the Thread class called - `currentThread`, it will always return, well the current thread from where it was invoked.

```
public static void main(String args[]) {
    Thread t = Thread.currentThread();
```

```
    System.out.println("Current thread: " + t);
    System.out.println("After name change: " + t);
    try {
        for(int n = 5; n > 0; n--) {
            System.out.println(n);
            Thread.sleep(1000); // sleep throws checked exception for
               interrupt
        }
    } catch (InterruptedException e) {
        // this might occur if another thread interrupts the current thread,
           in that case it has to be handled somehow
        System.out.println("Main thread interrupted");
    }
}
```

Creating

To create a thread we have two options really, either subclass Thread class, which is really not ideal, since when creating a thread we usually mean to execute some sort of code. The second option which is the prefered one is to actually implement the `runnable` interface, which has only one method called `run` that will reprent the code that needs to be run by the thread. When a new Thread object is created it is not immediately run, instead the caller has to manually call the `start` method on the thread object.

```
class NewThread implements Runnable {
    Thread t;
    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
class ThreadDemo {
    public static void main(String args[ ] ) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
```

```
        } catch (InterruptedException e) {
            // really the interrupted exception here has to be re-thrown or
               the state of the current thread set to
            // interrupted again, if we enter here it means we return the
               control back to this thread which was interrupted
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

The second option as mentioned is to inherit from Thread instead, which implements runnable itself, the Thread class actually is neither abstract nor does it require us to override runnable. What it does is it is a wrapper around a runnable object instance, if it was set during the creation of the thread it is run, below is the default run implementation of Thread. Meaning that we can actually create a thread object that does nothing by passing `nil` as the target runnable.

```
public void run() {
    if (target != null) {
        target.run();
    }
}
```

`The catching of InterruptedException is exceptional case, if a thread is blocking, the run-time will silently pass control to another thread while the current one is sleeping or otherwise occupied, this will NOT cause interrupted exception, the exception is for situations where the thread is manually interrupted or killed by another one, which is a case we have to handle if we expect to terminate the thread prematurely`

Join

Priority

The thread priority specifies how a given thread is threated by the java run-time when it comes time for a thread to be paused or even preempted, in the scenarios above all threads are created with default priority and in general all threads must be picked up by default at least for an equal amount of time, however that is highly dependent on the system on which the program is running, and is not something that can be relied on. Further more if the operating system does not support preempting (overruling the execution of the current thread, for one with higher priority) then specifying the priority of a thread is somewhat meaningless

To set a thread's priority, use the setPriority( ) method, which is a member of Thread. This is its general form: `final void setPriority(int level)` Here, level specifies the new priority setting for the calling thread. The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as static final variables within Thread.

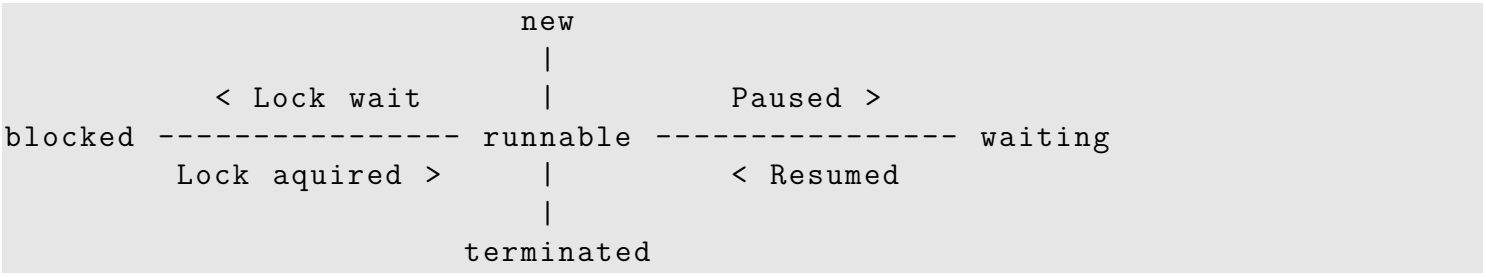
`One should not rely on threads being controlled by preempting, meaning that we should avoid setting priority and let the threads automatically relinquish or gain control instead`

State

The `getState()` method existing on a thread is used to obtain more information about the thread, which provides the exact state of the thread at the current time of invocation. Note that we have other means of

getting some sort of state from a thread such as `isAlive, isDaemon or isInterrupted`

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called sleep( ). This state is also entered when a timeout version of wait( ) or join( ) is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of wait( ) or join( ).



Volatile

A very interesting modifier keyword, that can be placed on member variables, the idea behind it is that in a threaded environment, when an object is shared between multiple threads, each thread might create a thread local copy of each variable of an object which it accesses, this is usually done on a per `cpu-core level`, in the L-level caches, however, this can be error prone. Reason being is that if only one thread modifies a variable and every other thread reads it, when the thread that modifies it, changes the variable, it might not become immediately visible to the `reader` threads, `volatile` ensures that all threads see the same value, in other words the value is not cached, meaning it is likely written out to main memory and read from memory on each access, that can be somewhat slow, so use the volatile keyword with caution.

It is important to note that when a `synchronize` block is exited, the member variables are updated, the cache is flushed, and data written out to main memory, but that can often be way too much for simple reader threads, which do not wish to lock the entire object and synchronize on it simply to guarantee they see the most up-to date value. This is where volatile comes in handy, also it is on `per` member, instead of `per` entire object, it does not force the entire object to be flushed out to memory and out of the cpu cache

```
class MyClass {
    int b; // this is not marked as volatile, meaning that writes to this
       value, might not be immedately visible to other reading threads
    volatile byte k; // ensure that k is not cached in L-level cache in the
       cpu, and threads see the most recent value
}
```

`Writes to a volatile variable happen-before subsequent reads of that volatile variable by any thread.`

Builtin

The standard concurrency library provides means of wrapping common threading patterns around a well defined interfaces and classes. One such is the `Executor` and `ExecutorService`. These interfaces provides

means of executing tasks in a concurrent manner without having to worry about the underlying implementation. However there are some key implementations of this interface which find common use in the wild. The `Executor` interface is the top level interface which provides a singular method, to execute a Runnable task, the `ExecutorService` provides more sophisticated means of:

1. `Task Submission`: You can submit `Runnable` and Callable tasks for execution.
2. `Life cycle Management`: It provides methods to control the `lifecycle` of the executor, such as starting, stopping, and checking if it is terminated.
3. `Future Management`: The `ExecutorService` returns Future objects that represent the result of the asynchronous computation, allowing you to retrieve the result or handle exceptions.
4. `Thread Pool Management`: It can manage a pool of threads, allowing for efficient reuse of threads and reducing the overhead associated with thread creation.

ThreadPerTaskExecutor The most basic executor, it implements the `Executor` interface, not the `ExecutorService` which means that this class exposes only one single method `execute` and each is run on a separate thread. Should not really be used in production environment

ThreadPoolExecutor The pool executor is a more sophisticated verision of the `ThreadPerTaskExecutor` which uses pooling of threads to execute tasks. Used for one short, short lived, non repeatable tasks, the pool executor can be configured, meaning it can be created with a specific fixed number of threads in the pool, and various other parameters can be changed.

```
ExecutorService executor = new ThreadPoolExecutor(
    5, // core pool size
    10, // max pool size
    60, // keep-alive time for extra threads
    TimeUnit.SECONDS,
    new LinkedBlockingQueue<Runnable>() // task queue
);
executor.submit(() -> {
    // Task implementation
});

executor.shutdown();
```

ScheduledThreadPoolExecutor The scheduled executor is a sub-class of the `ThreadPoolExecutor` and is meant to be used for execution of tasks after a given delay, or to execute tasks periodically after a given interval, one can use the `ScheduledThreadPoolExecutor` to schedule tasks to be executed in the future

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(5);
scheduler.schedule(() -> {
    // Task to execute after a delay
}, 10, TimeUnit.SECONDS); // Execute after 10 seconds
scheduler.shutdown();
```

CachedThreadPool This implementation creates new thread for each task but will reuse previously constructed threads when they are available, it is unbounded, meaning it can grow to accommodate as many threads as needed, suitable for many short lived tasks, where thread per task creation is justified. The cached thread pool is still using pooling however, the pool is unbounded, so unlike the `ThreadPoolExecutor` which has a fixed number of threads that can be used, the `CachedThreadPool` can grow its internal pool of threads

based on throughput, the more tasks that come in the more threads are created, once a task frees up a thread from the cached pool, if within some period of time that thread is not-reused by a new task, or a new task does not come in, the thread is removed/destroyed from the pool. This allows the `CachedThreadPool` to scale the thread count based on the incoming number of tasks, something that the `ThreadPoolExecutor` can not do, since it will always have a fixed number of thread allocated.

```
ExecutorService executor = Executors.newCachedThreadPool();
executor.submit(() -> {
    // Task implementation
});
executor.shutdown();
```

FixedThreadPool This implementation is very similar to the `ThreadPoolExecutor` since it also provides a fixed number of threads in a pool which can execute tasks, however the `FixedThreadPool` provides less flexibility in the way it can be configured compared to the `ThreadPoolExecutor`. The configuration of the `FixedThreadPool` allows one to only provide the number of threads in the pool, and that is about it, while the `ThreadPerTaskExecutor` provides quite a wide variety of configuration options.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(() -> {
    // Task implementation
});
executor.shutdown();
```

SingleThreadExecutor This executor creates a single thread, which is used to execute tasks sequentially, the tasks are queued, and after one finishes the next one is scheduled for execution, useful for scenarios where you need to execute tasks in a particular order, or if the output of one is used as the input of the other, and they need to be chain executed.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    // Task implementation
});
executor.shutdown();
```

ForkJoinPool This executor service implementation is a specialized implementation of the `ExecutorService` , designed with for work-stealing algorithms and tasks. The work-stealing is the primary benefit of using a `ForkJoinPool`. The `ForkJoinPool` is by default created with a fixed number of threads, just as the `ThreadPoolExecutor`, however, it employs work-stealing, the way work-stealing works is that each thread is assigned a `deque/queue` of tasks that will be run, usually assigned to each thread in the pool, in a round robin way. Now if a particular thread finishes all tasks it was assigned to, it can steal tasks from the queue of other threads, which still have pending tasks, usually the work stealing will try to balance pulling tasks from the queue of other threads. The `ForkJoinPool` is very useful for tasks which are recursive, since the tasks can be distributed evenly, and the threads from the pool will not be idle, since they will pull work from other threads, in case the thread finishes all tasks assigned to it. For example algorithms such as merge-sort, quick-sort, and other divide and conquer algorithms which are recursive in nature can greatly benefit from using a `ForkJoinPool`, instead of `ThreadPoolExecutor`, this is due to the fact that the `ThreadPoolExecutor` does not re-distribute tasks among free threads, the only way to make a free thread work, is by issuing a new task, then a free thread from the pool will pick it up, however, in the `ForkJoinPool` there will always be a thread with a running task until there are no longer tasks to run. The `ForkJoinPool` is very useful for a particular task which can be split into multiple stages, such as many algorithms, mentioned above, unlike the `ThreadPoolExecutor` which is tailored more towards completely independent long running tasks.

Future

This interface is very closely related to the `Executor` and `ExecutorService` interfaces, it provides an API wrapping a unit of computation for which the client can obtain the result without blocking the current thread, check if the status of the task is completed, had an exception. In general the methods in the `ExecutorService` return some sort of implementation or sub-interface of `Future` they return a `promise` of sorts, which can be queried, possible results extracted and so on. The `Future` interface and the `Executor` work in tandem to complete the user interaction with asynchronous task management

When one submits a task to an `ExecutorService` using the `submit()` method, it returns a `Future` object that represents the pending result of the task. This allows you to check the status of the task and retrieve the result once it has completed.

FutureTask This implementation of `Future` acts as a wrapper around `Runnable` and `Future`. Basically what it provides is means of creating a self-canceling runnable task. Meaning that while `submit` method on `ExecutorService` does return a `Feature` which can be used to cancel the running task, the task itself can not be canceled from within the task, however the `FutureTask` provides the user with the ability to terminate itself during the execution (e.g. when a given state or result is reached or not reached, the task can stop execution of itself).

```
Callable<String> callableTask = () -> {
    // Simulate a long-running task
    Thread.sleep(2000);
    return "Task Completed";
};

FutureTask<String> futureTask = new FutureTask<>(callableTask);
ExecutorService executor = Executors.newFixedThreadPool(1);
executor.submit(futureTask);

// Retrieve the result
try {
    String result = futureTask.get(); // This will block until the result is
       available
    System.out.println(result);
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
} finally {
    executor.shutdown();
}
```

CompletableFuture This particular implementation of the Future interface, allows one to chain and transform computation operations which are going to be run in an executor. By default the `CompletableFuture` uses a `ForkJoinPool`, this implementation is done in the `CompletableFuture` interface. Similarly to `Streams` the `CompletableFutures` have two types of operations

Finalizing and composing. The finalizing operations are such that they will block the current thread until the future completes The composing operations, similarly to streams will chain the future, with other futures, it is important to note that upon registering an composing operation the operation will be schedule for running on the executor, if further chaining is done through composing operations, the are done by chaining on the previous instance, each new chaining operation (thenCompose, thenApply, thenCombine etc) will either

register a new task in the executor if the future from which it is being chained from is not complete or immediately start executing if the future is complete.

`The chaining pattern used in CompletableFuture is very similar to how Stream chaining work , however unlike streams processing begins immediately as soon as the previous future in the chain has completed/has result, instead of when calling a terminating operation with Streams`

Finalizing operations

Method	Description
get()	Waits for the future to complete and retrieves its result, blocking the calling thread if necessary.
join()	Similar to `get()`, but throws an unchecked `CompletionException` if the computation fails.
thenAccept(Consumer)	Consumes the result without returning a new `CompletableFuture`, useful for performing side effects.
thenRun(Runnable)	Runs a `Runnable` after the computation is complete, regardless of the result.
whenComplete(BiConsumer)	Executes a `BiConsumer` after the completion of the future, providing both the result and exception.
handle(BiFunction)	Similar to `whenComplete`, but allows for recovery or transformation of the result if an exception occurs.

Composing operations

Method	Description
thenApply(Function)	Applies a function to the result of the future, returning a new `CompletableFuture` with the transformed result.
thenCompose(Function)	Chains another `CompletableFuture` that depends on the result of the current one, allowing for dependent async tasks.
thenCombine(CompletableFuture, BiFunction)	Combines the results of two `CompletableFutures` when both complete, using a function to produce a final result.
thenAcceptBoth(CompletableFuture, BiConsumer)	Applies a `BiConsumer` to the results of two futures without returning a result.
applyToEither(CompletableFuture, Function)	Applies a function to whichever of two futures completes first.
acceptEither(CompletableFuture, Consumer)	Similar to `applyToEither`, but applies a `Consumer` to the first result without returning a new future.
exceptionally(Function)	Returns a new `CompletableFuture` that handles exceptions and can provide a fallback value.

Factory operations

- `supplyAsync` - Starts execution of a task that produces a result, has a generic return type of T
- `runAsync` - Starts execution of a task that does not produce a result i.e. has a void return type
- `completedFuture` - Returns a future that is already completed with a specified value. This works as if a future was created, task was run, completed and a result was produced.

- `newIncompleteFuture` - Creates a new complete-able future, without a task, use `composing` and `finalizing` operations to complete it

`Note, the factory operations above provide means of creating a completeable future, in different states, there are mostly two - already running a given task, or one that is not initialized with any task and has to be registered into an executor service to begin execution.`

In the example below one can see how multiple `CompletableFuture` are created, each of which depends on the previous, using compose (`thenCompose`) the result of a previous future is used as an input to the next, however they are executed asynchronously but in order. When one calls `supplyAsync` on a future instance, the task is `immediately` scheduled for execution, it returns a new instance of `CompletableFuture` which can be further chained with more `supplyAsync` calls, the next call to `supplyAsync` will either schedule a task to be run after the first future completes / has result, or if the first future is already finished, the second call to `supplyAsync` will be immediately scheduled for execution.

Note that `get` will block the current thread until the task is finished, so it is best to execute some other long running task between the calls to `thenAccept` and `get` to make sure there is no idle current thread time

`It is important to remember that chained calls to supplyAsync, will start the task as soon as possible only if the result of the previous future is complete, that is not applicable only for the very first future in the chain, it does not need to wait for result, it can start immediately`

```
public static void main(String[] args) {
    OrderProcessing orderProcessing = new OrderProcessing();
    orderProcessing.processOrder(123);
}

public void processOrder(int orderId) {
    CompletableFuture<User> userFuture = fetchUserDetails(orderId);
    CompletableFuture<Payment> paymentFuture = userFuture.thenCompose(user ->
        processPayment(user));
    CompletableFuture<Void> confirmationFuture = paymentFuture.thenAccept(
       payment -> sendConfirmation(payment));

    // after calling `supplyAsync`, the future tasks are already being
       executed, inside the thread executor pool
    // therefore the current thread is not blocked and more tasks can be run
       at this point while the result of the
    // `confirmationFuture` is being computed

    try {
        confirmationFuture.get(); // obtain the final result
        System.out.println("Order processed successfully.");
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

private CompletableFuture<User> fetchUserDetails(int orderId) {
    return CompletableFuture.supplyAsync(() -> {
        sleep(1); // Simulate delay
```

```
        System.out.println("Fetched user details for order " + orderId);
        return new User("John Doe", "john.doe@example.com");
    });
}

private CompletableFuture<Payment> processPayment(User user) {
    return CompletableFuture.supplyAsync(() -> {
        sleep(2); // Simulate delay
        System.out.println("Processed payment for user: " + user.getName());
        return new Payment("12345", 99.99); // Simulate a successful payment
    });
}

private void sendConfirmation(Payment payment) {
    sleep(1); // Simulate delay
    System.out.println("Sent confirmation email for payment ID: " + payment.
       getPaymentId());
}

private void sleep(int seconds) {
    try {
        TimeUnit.SECONDS.sleep(seconds);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
```