①
```
Thread thread = new Thread(
    () -> System.out.println("a")
);
```

②
```
Class MyRunnable implements Runnable {
    @Override
    public void run() {

    }
}
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable);
```

③
```
Class MyThread extends Thread {
    @Override
    public void run() {

    }
}
MyThread thread = new Thread();

thread.start()
thread.join() // need exception handle
```

```
Class Task implements Runnable {
    public void run() {
        try {
            if (Thread.currentThread().isInterrupted())
                return;
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
Thread thread = new Thread(new Task());
thread.start
thread.interrupt() // throw exception if thread sleeping
```

- Object lock1 = new Object
  synchronized (lock1) {
  .....
  }
  public synchronized void foo(){
  .....
  } // use the class obj as lock

  public static synchronized
  void foo() {
  .....
  } // use the class as lock

- Volatile int x; // always read it from mem
  // solve visibility issue across threads due to cache
  var status = new Status();
  thread1.start() // set status.done.
  thread2.start() // wait until status.done.

- obj.wait() // current thread goes to sleep
  obj.notify(), obj.notifyAll()
  // notify all threads sleeping on that object
  // must be in synchronized method or block

- AtomicInteger x = new AtomicInteger();
  x.get(), incrementAndGet(); // ++x;
  getAndIncrement(); // x++;
  CompareAndSwap(expected);

- LongAdder x = new LongAdder();
  x.intValue(), x.doubleValue() .....
  x.increment();
  // work by create slots for each thread.
  // call internal sum() when get value.

- Collections.synchronizedCollection(new ArrayList())
  ------ Map(new HashMap())
  ------ Set(new HashSet())
  List list = ------ List(new ArrayList())
  Thread thread1 = new Thread(() -> {
  list.addAll(Arrays.asList(1,2,3));
  });

- Map<T,T> map = new ConcurrentHashMap();
  map.put(1, "a"); map.get(1); map.remove(1);

- Lock lock = new ReentrantLock();
  lock.lock();
  try {
  .....
  } finally {
  lock.unlock();
  }

- Semaphore sem = new Semaphore(n);
  sem.aquire(n), sem.release(n)
  sem.tryAquire(n, timeout, unit);

- ReentrantReadWriteLock lock = new RRWL();
  readlock = lock.readLock(); wl = lock.writeLock();
  Condition condition = writelock.newCondition();
  condition.await(); condition.signal/signalAll();

- writeLock.lock()
  try {
  while (....) {
  condition.await();
  }
  condition.signal();
  } finally { writeLock.unlock }

```
@Override
public int hashCode() {
    return 31 * x + y;

@Override
public boolean equals(Object other) {
    if (this == other) {true}
    if (!(other instanceof T)) {false}
    ....
}
```

```
Arrays.sort(arr, new Comparator<T>(){
    @Override
    public int compare(T o1, T o2) {
        .....
    }
});
```

```
Arrays.sort(arr, (a,b) -> (a[0] - b[0]));
Collections.sort(collection, .....);
```

```
Random rand = new Random();
rand.nextInt(max - min) + min;
```

```
hashMap.getOrDefault(key, default);
```

```
ArrayList<> list = (ArrayList) Arrays.asList(1,2,3)
Integer[] list = linkedList.toArray(new Integer[0])
```

```
Class::method // pass a function as lambda exp
```

```
LocalTime time = LocalTime.now(); // HH:MM:SS
Instant time = Instant.now();
Duration duration = Duration.between(t1, t2);
```

```
list.stream()
    .map()
    .filter()
    .reduce()
    .collect(Collectors.toList()); // must collect to get result
```

```
executor = Executors.newFixedThreadPool( nThreads );
try {
    executor.submit(Runnable task);
} finally {
    executor.shutdown();
    executor.shutdownNow(); // also kills unfinished tasks
}
```

```
Class MyCallable implements Callable<T> {
    @Override
    public T call() {
        return ...;
    }
}
```

```
Future<T> future = executor.submit(new MyCallable());
try {
    future.get(); // block until the task returns.
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

```
CompletableFuture<void> future
    = CompletableFuture.runAsync(Runnable task, Executor);
// By default ForkJoinPool.commonPool() is used for Executor.
```

```
Supplier<Integer> task = () -> 1;
Class MySupplier implements Supplier<T> {
    @Override
    public T get() {
        return ...;
    }
}
```

```
CompletableFuture<T> future
    = CompletableFuture.supplyAsync(Supplier<T> supplier, exe)
public CompletableFuture<T> sendAsync() {
    return CompletableFuture.supplyAsync(() -> send());
}
// turn a sync method to async
```

- void Consumer<T>.accept(T t) // use previous result to do something

Corsumer<T> Consumer<T>.andThen(Consumer<T> after) // get a composed consumer.

future2 = future1.thenAcceptAsync(Consumer);
future2 = future1.thenAcceptAsync(result -> {});

- V Function<T,R>.apply(T t)

Function<V,T>.compose(Function<V,T> before)
Function<T,U>.andThen(Function<R,U> before)

future3 = future1.thenApplyAsync(Function);
future3 = future1.thenApplyAsync(result -> { return ...; });

- future1<void> = CompletableFuture.runAsync(Runnable); // create initial future with no return
future1<Return Type> = CompletableFuture.supplyAsync(Supplier); // create initial future with return
future2 = future1.thenRunAsync(Runnable/Supplier); // create next future after previous one/no input
future2 = future1.thenAcceptAsync(Consumer); // create next future after previous based on its input
future2 = future1.theApplyAsync(Function); // create next future after previous based on its input

// run after previous future finishes, but async version won't block current thread.

- futureExp = future2.exceptionally(ex -> {return ...});

// handle exception.

- future.thenCompose(Function<T → CompletableStage<U>>);

```
public static CompletableFuture<T> fooAsync(x) {
    return CompletableFuture.supplyAsync(() → x + "a");
}
future.thenCompose(CLASS::fooAsync);
```

- ```
var first = CompletableFuture.supplyAsync(() → 20);
var second = CompletableFuture.supplyAsync(() → 29);
first.thenCombine(second, (price, exchangeRate) → {
                                return price * exchangeRate;
                            })
    .thenAccept(result → System.out.println(result));
```

- ```
CompletableFuture.allOf(first, second, third, ....)
                 .thenRun(() → {
                     first.get();
                 });
```

- ```
CompletableFuture.anyOf(first, second, third, ...)
                 .thenAccept(fastest → {});
```

- future.completeOnTimeOut(default, timeout, unit);
```