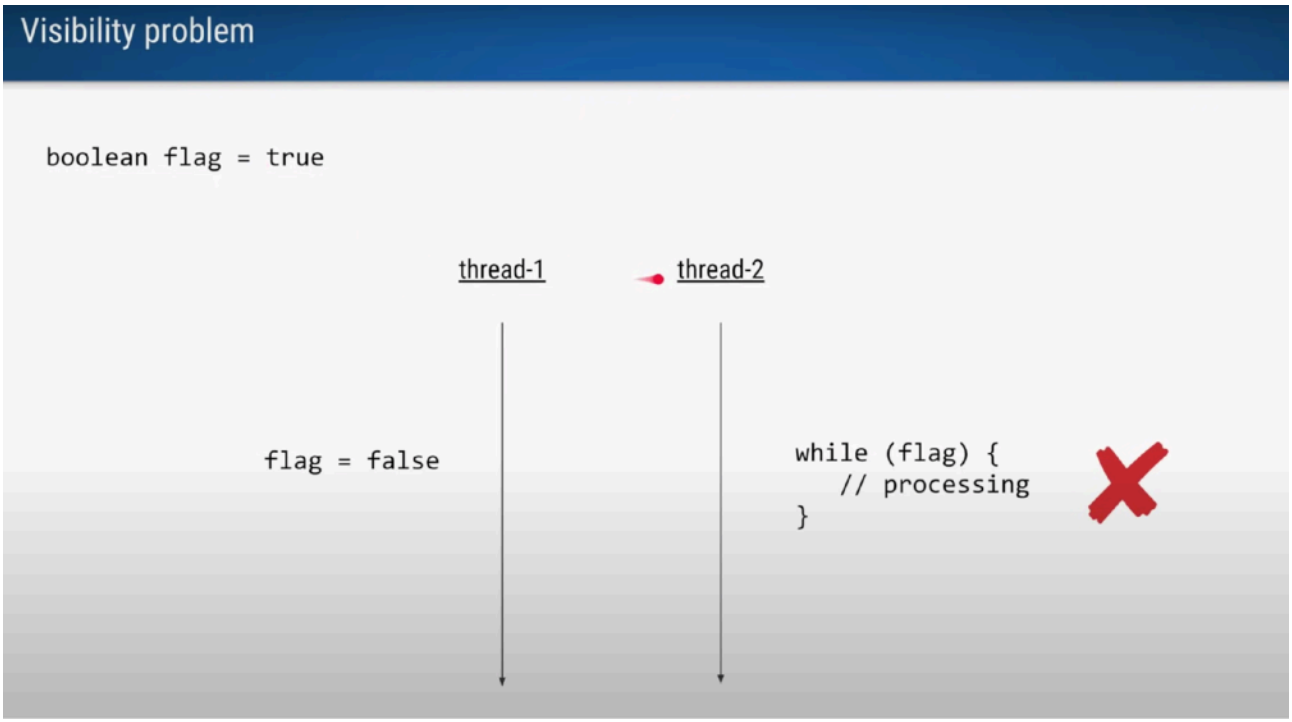
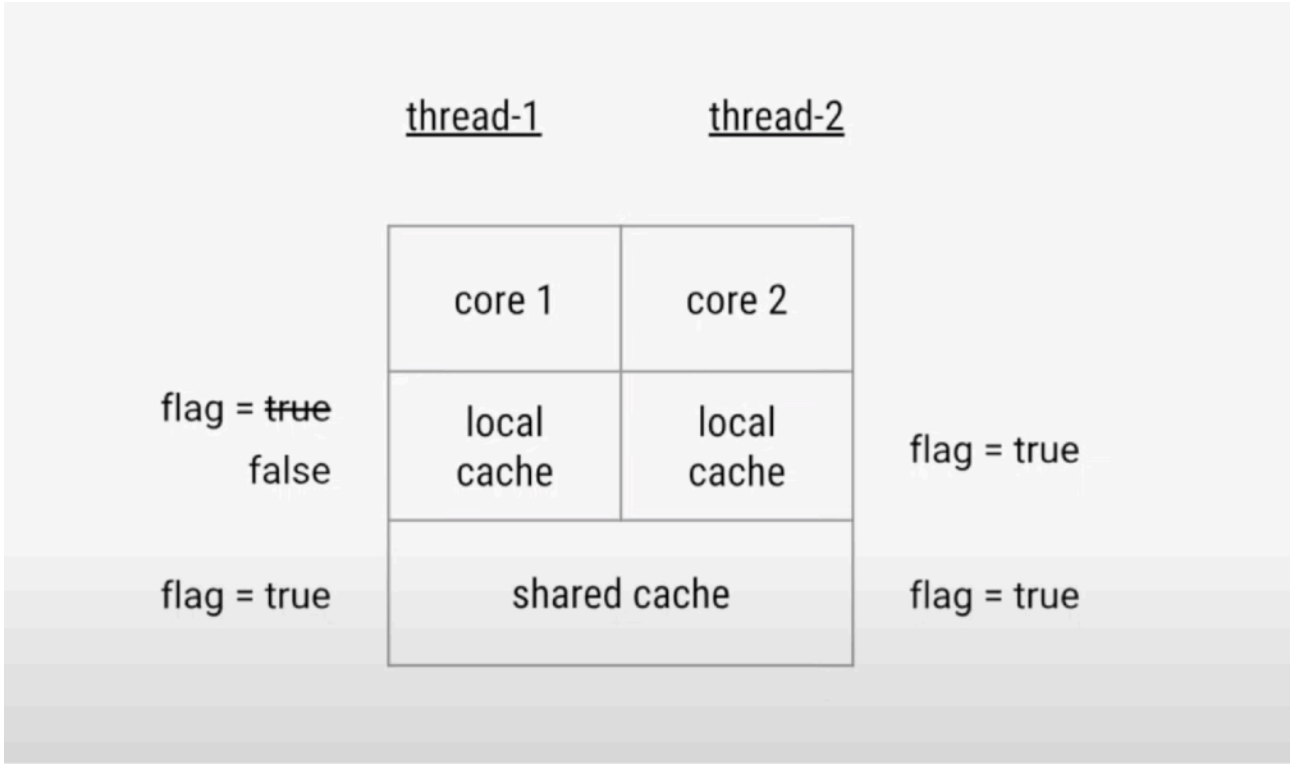
Visibility Problem :



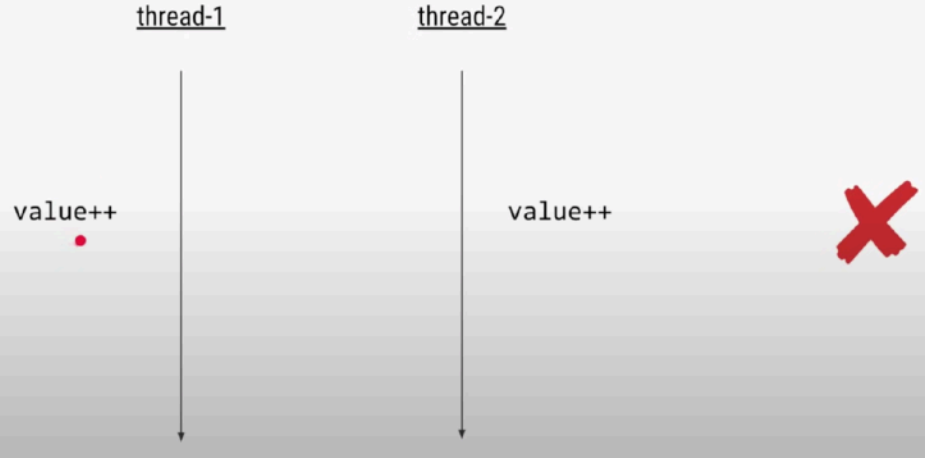


Visibility problem

```
volatile boolean flag = true
```

Synchronization problem

```
int value = 1;
```



Synchronization problem

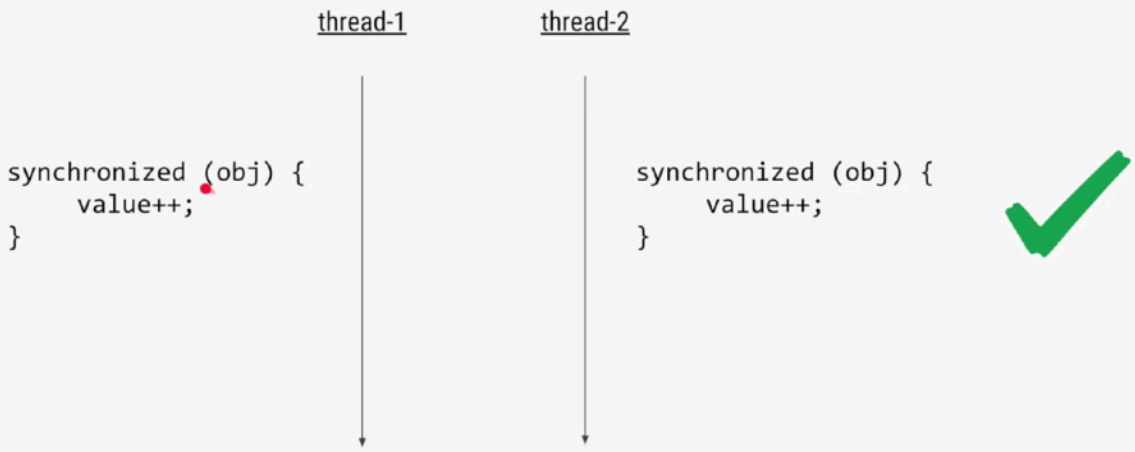
```
volatile int value = 1;
```

Even with volatile

#	Thread-1	Thread-2
1	Read value (=1)	
2		Read value (=1)
3	Add 1 and write (=2)	
4		Add 1 and write (=2)

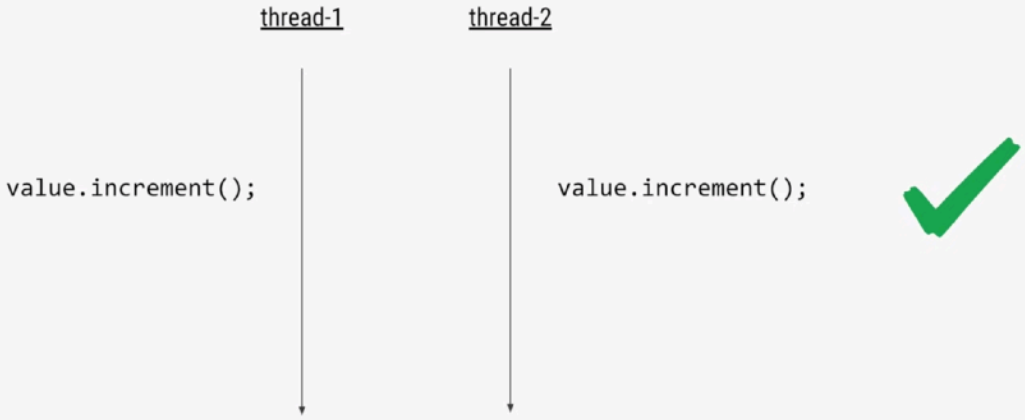
Synchronization solutions - #1

```
volatile int value = 1;
```



Synchronization solutions - #1

```
AtomicInteger value = new AtomicInteger(1);
```



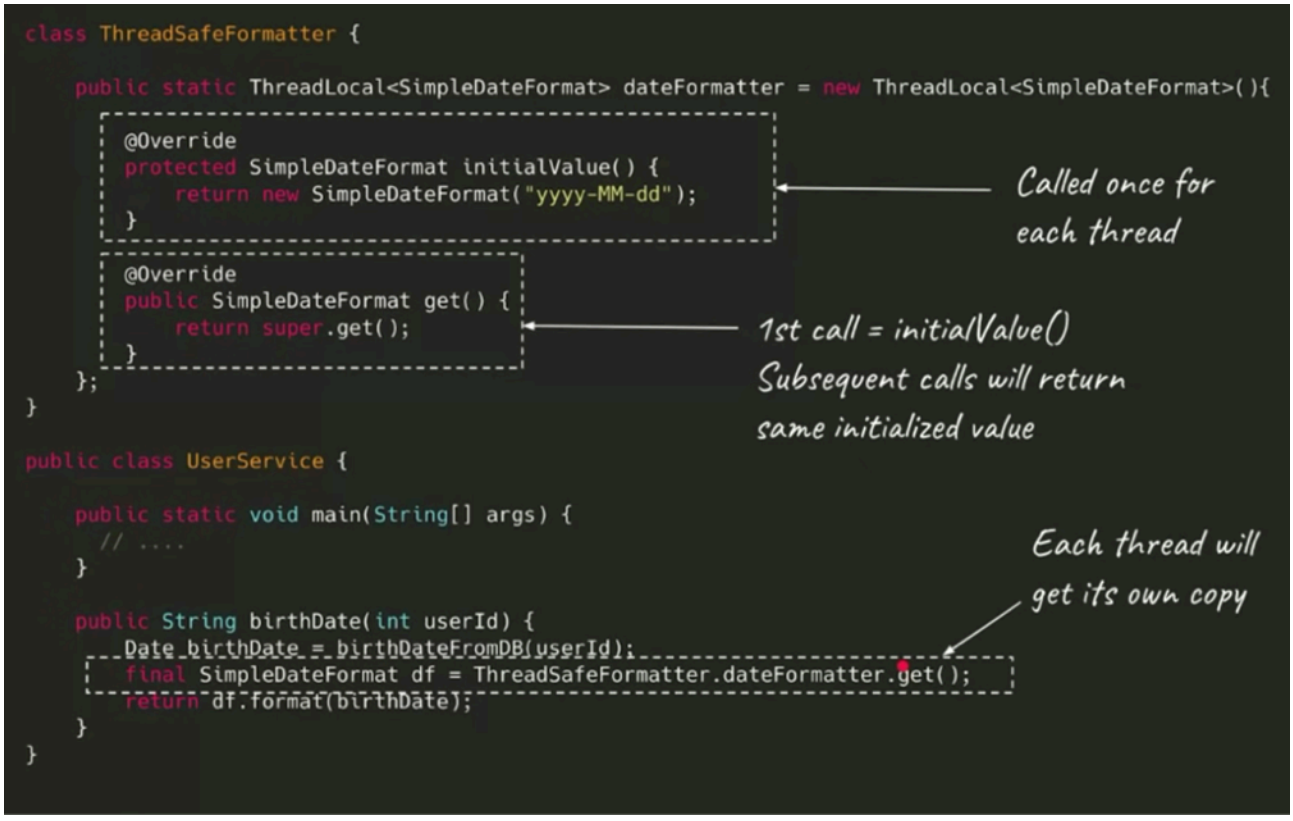
Many methods for various compound operations

- incrementAndGet
- decrementAndGet
- addAndGet (int delta)
- compareAndSet (int expectedValue, int newValue)

Typical Use Cases

Type	Use Case
volatile	Flags
AtomicInteger AtomicLong	Counters
AtomicReference	Caches (building new cache in background and replacing atomically) Used by some internal classes Non-blocking algorithms

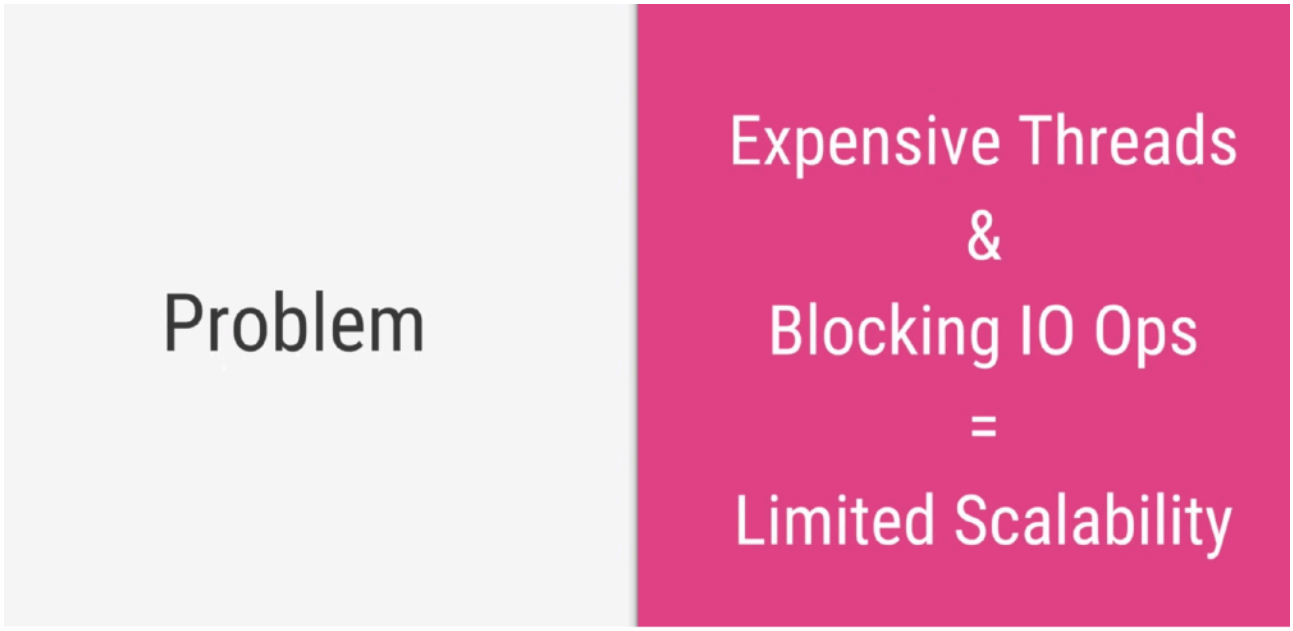
Thread Local:



Async Programming :

Synchronous API

```
for (Integer id : employeeIds) {

    // Step 1: Fetch Employee details from DB
    Future<Employee> future = service.submit(new EmployeeFetcher(id));
    Employee emp = future.get(); // blocking

    // Step 2: Fetch Employee tax rate from REST service
    Future<TaxRate> rateFuture = service.submit(new TaxRateFetcher(emp));
    TaxRate taxRate = rateFuture.get(); // blocking

    // Step 3: Calculate current year tax
    BigDecimal tax = calculateTax(emp, taxRate);

    // Step 4: Send email to employee using REST service
    service.submit(new SendEmail(emp, tax));
}
```



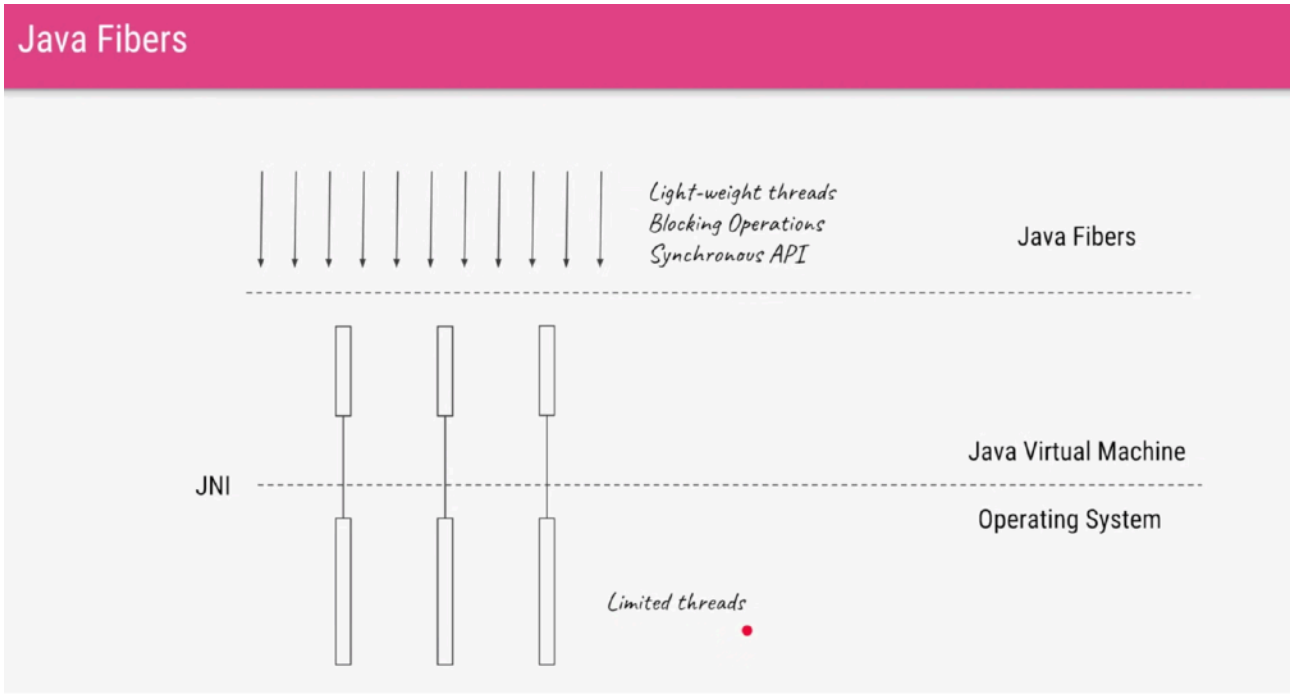
Solution

Non-blocking IO
&
Asynchronous API

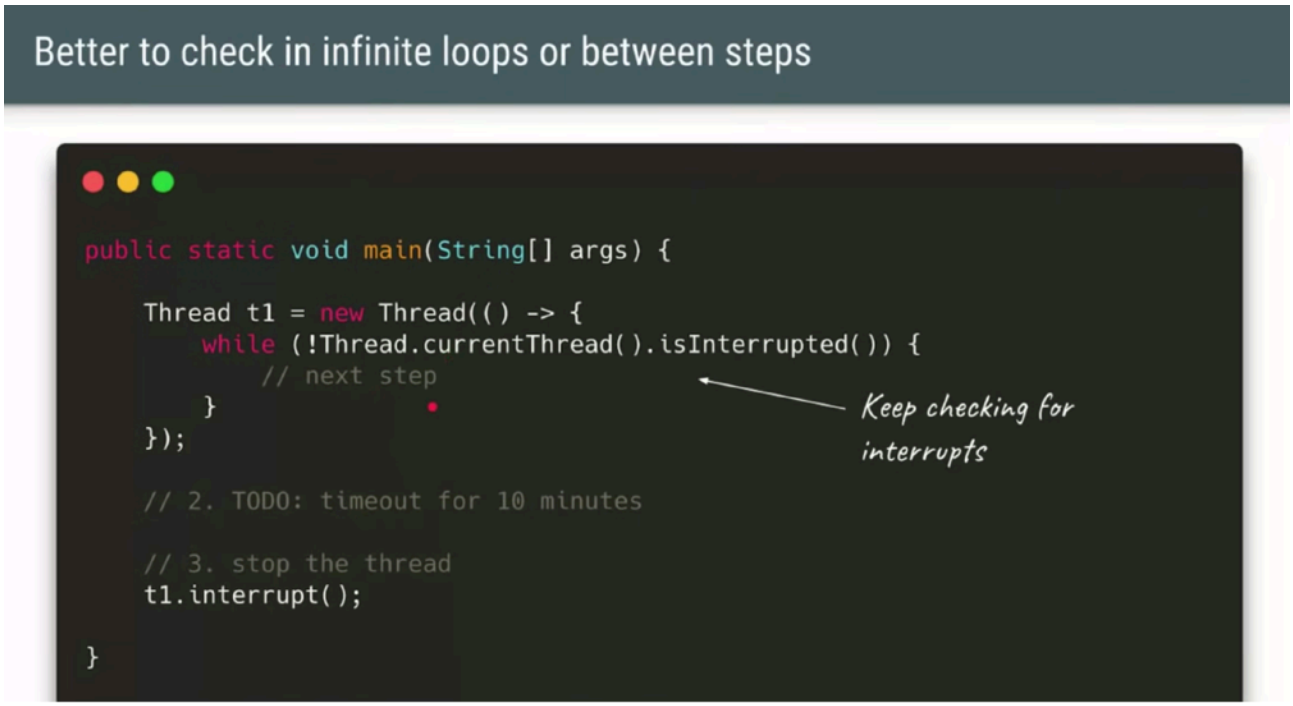
Asynchronous API - Callbacks

```
for (Integer id : employeeIds) {
    CompletableFuture.supplyAsync(() -> fetchEmployee(id))
            .thenApplyAsync(employee -> fetchTaxRate(employee))
            .thenApplyAsync(taxRate -> calculateTax(taxRate))
            .thenAcceptAsync(taxValue -> sendEmail(taxValue));
}
```

Callback chaining (similar to JS)



A task is running separate thread. Stop the task if it exceed 10 minutes :



Interrupt: Works with Futures and Callables as well

```java
public static void main(String[] args) {

    ExecutorService threadPool = Executors.newFixedThreadPool(2);

    Future<?> future = threadPool.submit(() -> {
        while (!Thread.currentThread().isInterrupted()) {
            // next step
        }
    });

    // 2. TODO: timeout for 10 minutes

    // 3. stop the thread
    future.cancel(true);
}
```

Calls thread.interrupt for thread running the task

Volatile

```java
public void process() {

    // 1. Create a task and submit to a thread
    MyTask task = new MyTask();
    Thread t1 = new Thread(task);
    t1.start();

    // 2. TODO: timeout for 10 minutes

    // 3. ask task to stop using volatile
    task.keepRunning = false;
}

private class MyTask implements Runnable {

    public volatile boolean keepRunning = true;

    @Override
    public void run() {
        while (keepRunning == true) {
            // steps
        }
    }
}
```

Same will work for ThreadPool

AtomicBoolean

```
public void process() {

    // 1. Create a task and submit to a thread
    MyTask task = new MyTask();
    Thread t1 = new Thread(task);
    t1.start();

    // 2. TODO: timeout for 10 minutes

    // 3. stop the thread
    task.stop();
}

private class MyTask implements Runnable {

    public AtomicBoolean keepRunning = new AtomicBoolean(true);

    @Override
    public void run() {
        while (keepRunning.get() == true) {
            // steps
        }
    }

    public void stop() {
        keepRunning.set(false);
    }
}
```

Producer Consumer :


public static void main(String[] args) {
BlockingQueue<Item> queue = new ArrayBlockingQueue<>(10);
Handles concurrent thread access
// Producer
final Runnable producer = () -> {
while (true) {
queue.put(createItem());
}
};
thread blocks if queue full
new Thread(producer).start();
new Thread(producer).start();
// Consumer
final Runnable consumer = () -> {
while (true) {
Item i = queue.take();
process(i);
}
};
thread blocks if queue empty
new Thread(consumer).start();
new Thread(consumer).start();
Thread.sleep(1000);
}
try-catch for take and put skipped for brevity

With Lock:

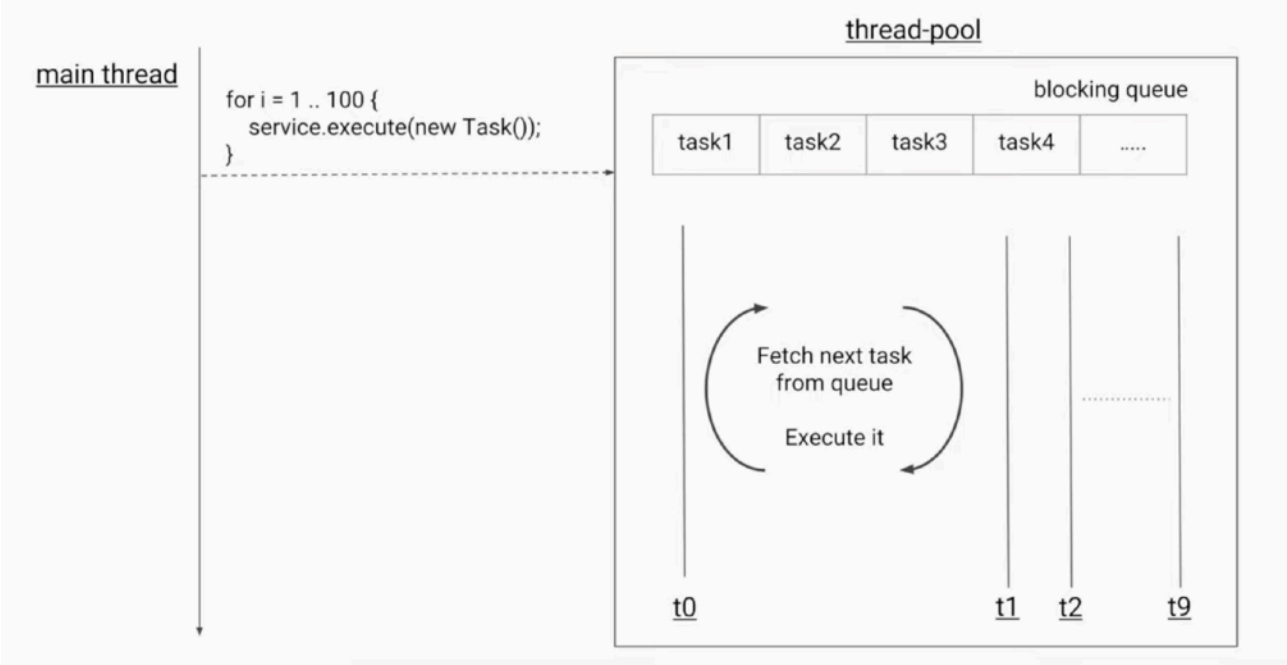
```
private int max;
private Queue<E> queue = new LinkedList<>();
private ReentrantLock lock = new ReentrantLock(true);
private Condition notEmpty = lock.newCondition();
private Condition notFull = lock.newCondition();

public MyBlockingQueue(int size) {
    queue = new LinkedList<>();
    this.max = size;
}

public void put(E e) {
    lock.lock();
    try {
        if (queue.size() == max) {
            notFull.wait();
        }
        queue.add(e);
        notEmpty.signalAll();
    } finally {
        lock.unlock();
    }
}

public E take() {
    lock.lock();
    try {
        if (queue.size() == 0) {
            notEmpty.wait();
        }
        E item = queue.remove();
        notFull.signalAll();
        return item;
    } finally {
        lock.unlock();
    }
}
```

```java
private ReentrantLock lock = new ReentrantLock(true);
private Condition notEmpty = lock.newCondition();
private Condition notFull = lock.newCondition();

public E take() {
    lock.lock();
    try {
        while (queue.size() == 0) {
            notEmpty.await();
        }
        E item = queue.remove();
        notFull.signalAll();
        return item;
    } finally {
        lock.unlock();
    }
}
```

```java
private Object notEmpty = new Object();
private Object notFull = new Object();

public synchronized E take() {
    if (queue.size() == 0) {
        notEmpty.wait();
    }
    E item = queue.remove();
    notFull.notifyAll();
    return item;
}
```

Executor Service

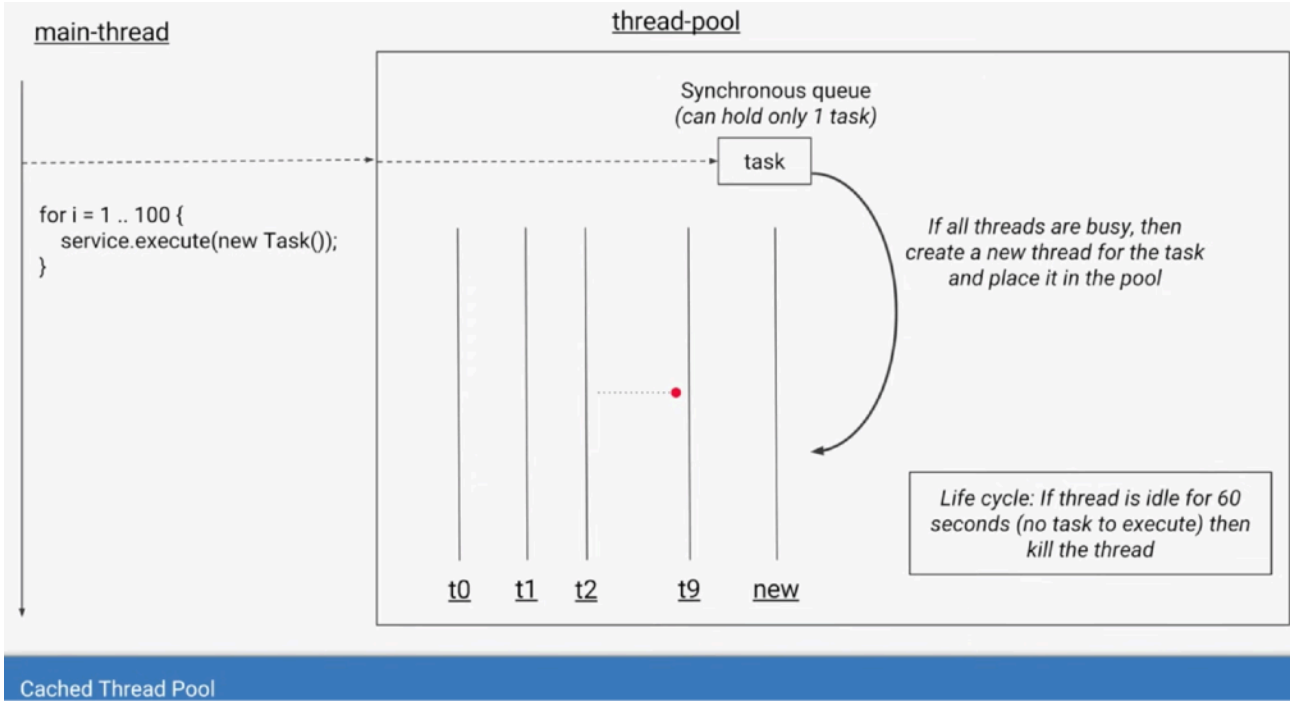
```java
public static void main(String[] args) {

    for (int i = 0; i < 10; i++) {
        Thread thread = new Thread(new Task());
        thread.start();
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}

static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}
```



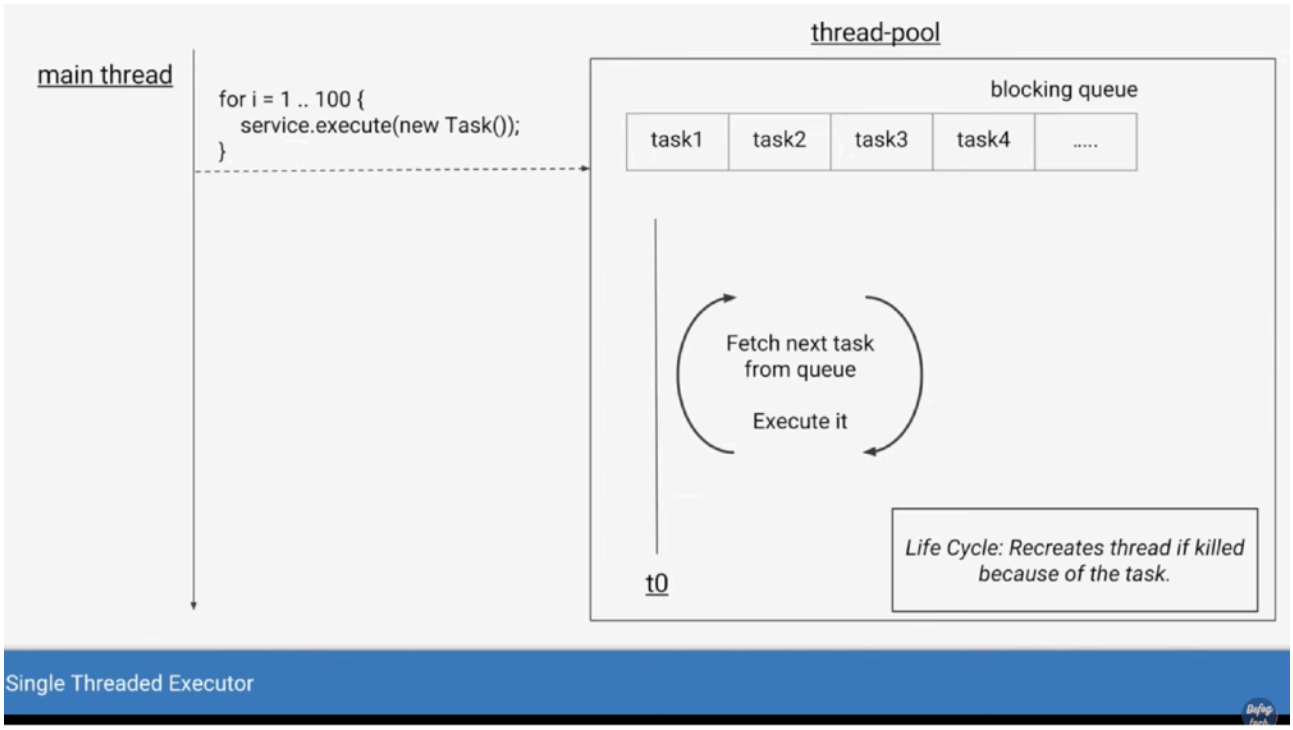
Task Type	Ideal pool size	Considerations
CPU intensive	CPU Core count	How many other applications (or other executors/threads) are running on the same CPU.
IO intensive	High	Exact number will depend on rate of task submissions and average task wait time. Too many threads will increase memory consumption too.

```java
public static void main(String[] args) {

    // get count of available cores
    int coreCount = Runtime.getRuntime().availableProcessors();
    ExecutorService service = Executors.newFixedThreadPool(coreCount);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new CpuIntensiveTask());
    }
}

static class CpuIntensiveTask implements Runnable {
    public void run() {
        // some CPU intensive operations
    }
}
```

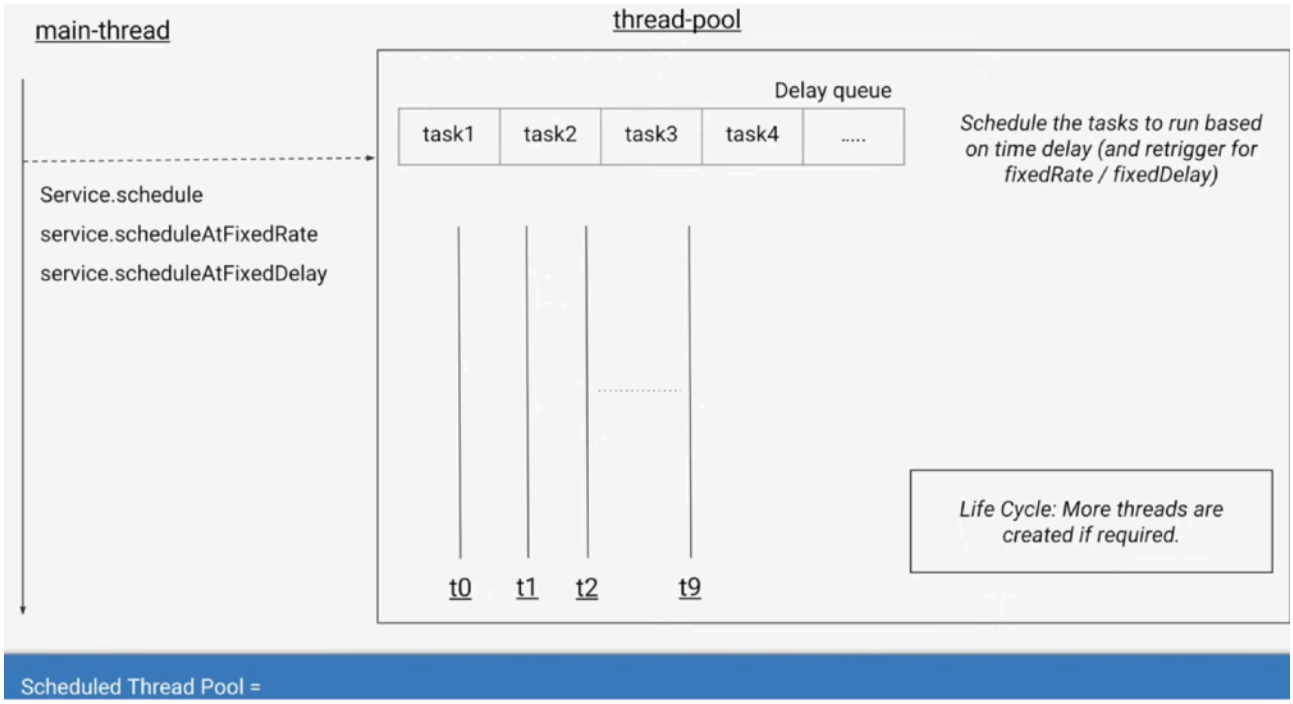
1. FixedThreadPool
2. CachedThreadPool
3. ScheduledThreadPool
4. SingleThreadedExecutor



Cached Thread Pool



Single Threaded Executor



Scheduled Thread Pool =

```java
// for scheduling of tasks
ScheduledExecutorService service = Executors.newScheduledThreadPool( corePoolSize: 10);

// task to run after 10 second delay
service.schedule(new Task(), delay: 10, SECONDS);

// task to run repeatedly every 10 seconds
service.scheduleAtFixedRate(new Task(), initialDelay: 15, period: 10, SECONDS);

// task to run repeatedly 10 seconds after previous task completes
service.scheduleWithFixedDelay(new Task(), initialDelay: 15, delay: 10, TimeUnit.SECONDS);


atic class Task implements Runnable {
  public void run() {
      // task that needs to run
      // based on schedule
  }
```

Constructor Parameters

Parameter	Type	Meaning
corePoolSize	int	Minimum/Base size of the pool
maxPoolSize	int	Maximum size of the pool
keepAliveTime + unit	long	Time to keep an idle thread alive (after which it is killed)
workQueue	BlockingQueue	Queue to store the tasks from which threads fetch them
threadFactory	ThreadFactory	The factory to use to create new threads
handler	RejectedExecutionHandler	Callback to use when tasks submitted are rejected

Parameter	FixedThreadPool	CachedThreadPool	ScheduledThreadPool	SingleThreaded
corePoolSize	constructor-arg	0	constructor-arg	1
maxPoolSize	same as corePoolSize	Integer.MAX_VALUE	Integer.MAX_VALUE	1
keepAliveTime	0 seconds	60 seconds	60 seconds	0 seconds

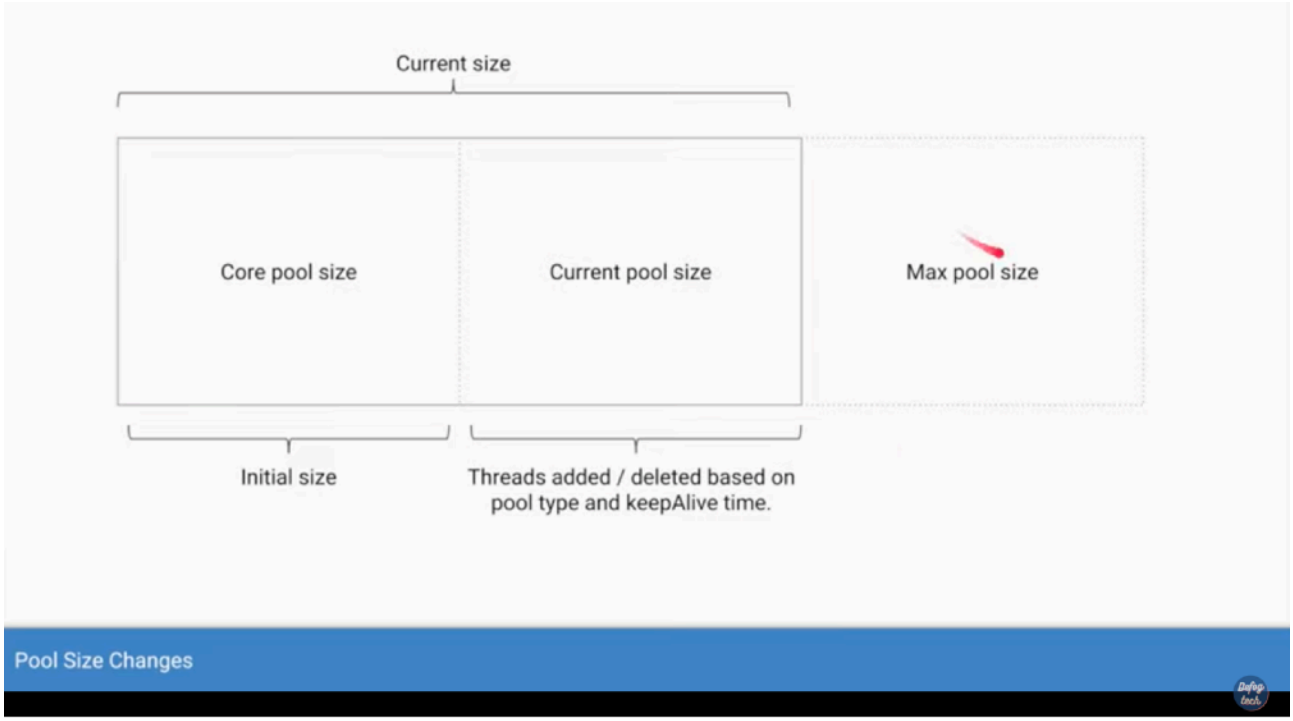
Note: Core pool threads are never killed unless allowCoreThreadTimeOut(boolean value) is set to true.

```
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);
```

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  keepAliveTime: 0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
```

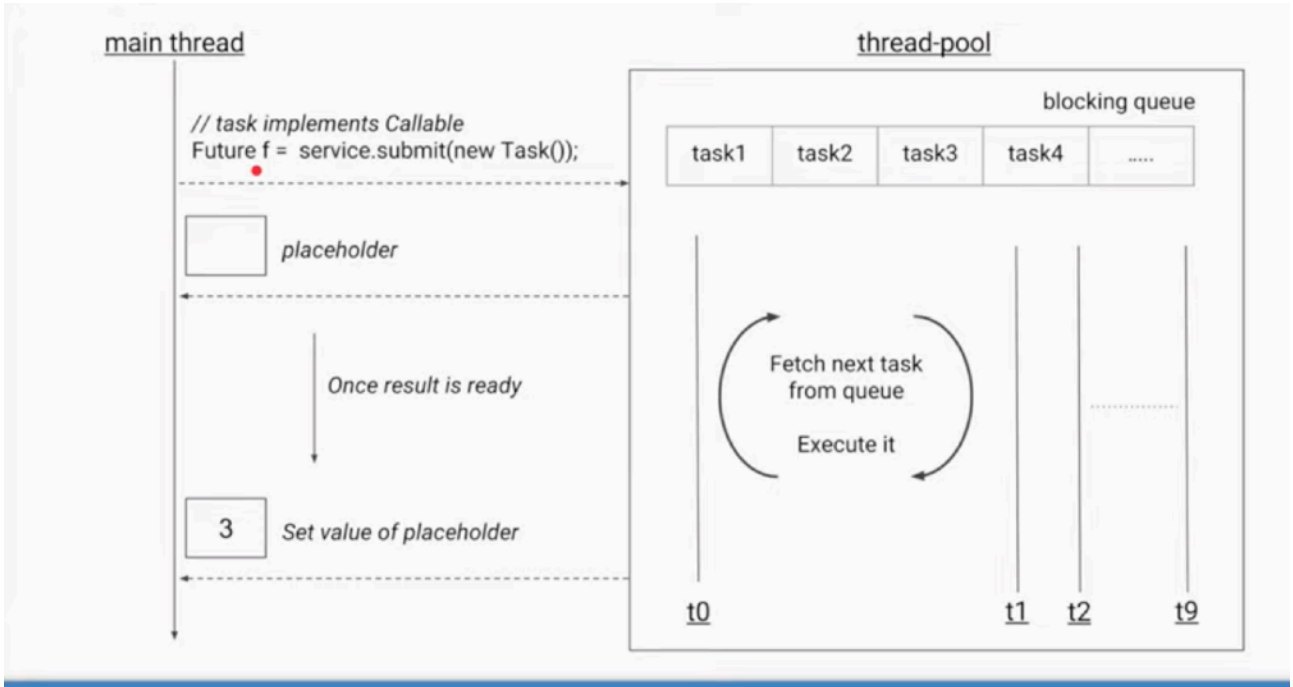
ThreadPoolExecutor constructor



Pool	Queue Type	Why?
FixedThreadPool	LinkedBlockingQueue	Threads are limited, thus unbounded queue to store all tasks. *Note: Since queue can never become full, new threads are never created.*
SingleThreadExecutor	LinkedBlockingQueue	
CachedThreadPool	SynchronousQueue	Threads are unbounded, thus no need to store the tasks. Synchronous queue is a queue with single slot
ScheduledThreadPool	DelayedWorkQueue	Special queue that deals with schedules/time-delays
Custom	ArrayBlockingQueue	Bounded queue to store the tasks. If queue gets full, new thread is created (as long as count is less than maxPoolSize).

```java
public static void main(String[] args) throws ExecutionException,
InterruptedException {

    // create the pool
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    // submit the tasks for execution
    Future<Integer> future = service.submit(new Task());

    // perform some unrelated operations

    // 1 sec
    Integer result = future.get(); // blocking

    System.out.println("Thread Name: " + Thread.currentThread().getName());
}

static class Task implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        Thread.sleep( millis: 3000);
        return new Random().nextInt();
    }
```

```java
// create the pool
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

// submit the tasks for execution
List<Future> allFutures = new ArrayList<>();
for (int i = 0; i < 100; i++) {
    Future<Integer> future = service.submit(new Task());
    allFutures.add(future);
}

// 100 futures, with 100 placeholders.

// perform some unrelated operations

// 1 sec
try {
    Integer result = future.get(); // blocking
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```



Callable

```
try {
    Integer result = future.get( timeout: 1, TimeUnit.SECONDS);
    System.out.println("Result from the task is " + result);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    System.out.println("Couldn't complete task before timeout");
}
```

Visibility problem

```
boolean flag = true
```