Java Multithreading

(1) Creating threads:

Method 1: Implementing Runnable

Method 2: Extending Thread.

* Both would require you to override the run() method & call thread.start() to start the actual execution.

- Setting thread priority will not mean that a higher priority thread gets scheduled first. It depends on the underlying platform.

(2) Types of threads:

- User threads: High priority. JVM waits for them to complete their task before terminating a program.
- Daemon threads: low-priority & mostly responsible for providing service to user threads. They won't prevent program from exiting when all user threads have completed. Should not be used for I/O tasks. Can be used for garbage collection, clearing cache entries etc.

thread.setDaemon(true).

(3) Executor Service to create thread pool:

* ExecutorService executor = Executors.newSingleThreadExecutor()
* ExecutorService executor = Executors.newFixedThreadPool(pass #threads

executor.execute(new Class()).

~~* ExecutorService executor = Executors.scheduleAtFixedRate(~~ obj,

* ExecutorService executor = newScheduledThreadPool(#threads

executor.scheduleAtFixedRate(obj, delay, period)

↓ or task.

executor.shutDown().

5 (*) Callable:

Used to return results in the future.

Will return Future<T>

```
public class Tests implements Callable<Integer> {
    private int id=0;
    @Override
    public Integer call() throws Exception {
        :
    }
}.

class Main {
  p.s.v.m() {
    ExecutorService executor = Executors.new CachedThreadPool();
    List<Future<Integer>> results = new ArrayList<>();
    for (int i=0; i<5; ++i)
    { results.add(executor.submit(new Tests())); }

    for (Future<Integer> r: results)
    { s.o.p (r.get()) }.

  }
```

return value is Integer → or newFixedThreadPool()

r.get()

r.get (timeout, unit)

r.isDone() → non-blocking to check if thread is done.

5) Synchronized (method):

- Using synchronized in func declaration will make operations within it thread safe.

public synchronize boolean dec() {...}

(6) Synchronized (Object):

* Synchronized (this)

holds lock within a block of code so can block for less time than when added to the method.

```
public boolean dec()
{  synchronize (this) {
      --number;
   }
}
```

} lock for only this block.

* Synchronized (obj);

private final Object lockObject = new Object().

```
public boolean dec()
{  synchronized (lockObject) {
      --number;
   }
}
```

→ lockObject can be used by subclasses as well.

(7) ReentrantLock(). (2)

- Similar to synchronized.

```
ReentrantLock lock = new ReentrantLock()
lock.lock()
   :
lock.unlock().
```

(8) ReentrantReadWriteLock:

- readlock() → will allow others to read when acquired but prevents writes
- writelock() → will prevent all read/write when acquired.

(9) Atomic operations:

Works on java primitives: Long, Boolean, Integer etc.

Used machine instructions like compare & swap

AtomicInteger counter = new AtomicInteger (10)

counter.get() / set() / compareAndSet().

- volatile: similar to helping operations atomic + make sure updated value is visible to all threads.
- Can be used with only one variable.

Java thread safe data structures.

- LinkedBlockingQueue., PriorityBlockingQueue.
- ConcurrentHashMap; Set<Integer> set = new ConcurrentHashMap.newKeySet().
- ConcurrentLinkedQueue
- Collections.synchronizedList(new ArrayList<>());
- Collections.synchronizedMap()
- Collections.synchronizedSortedMap() → TreeMap.
- Collections.synchronizedSet()