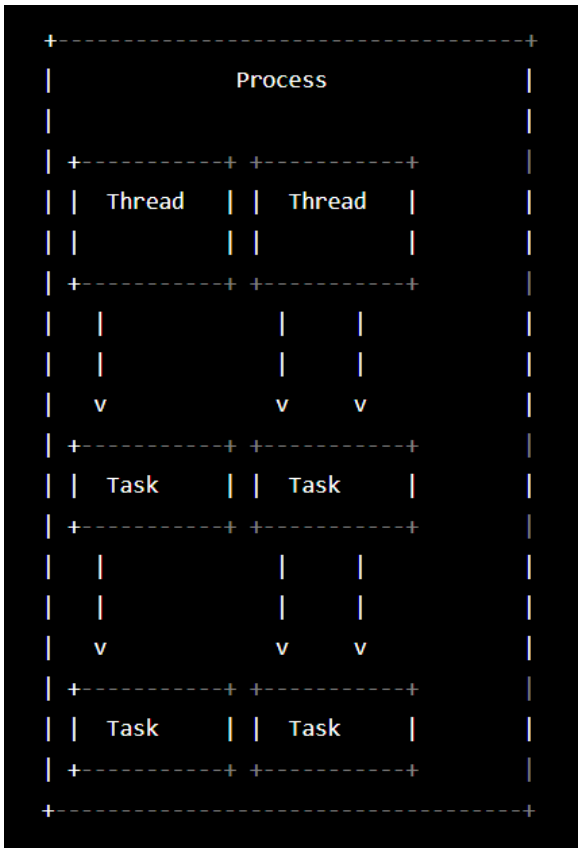
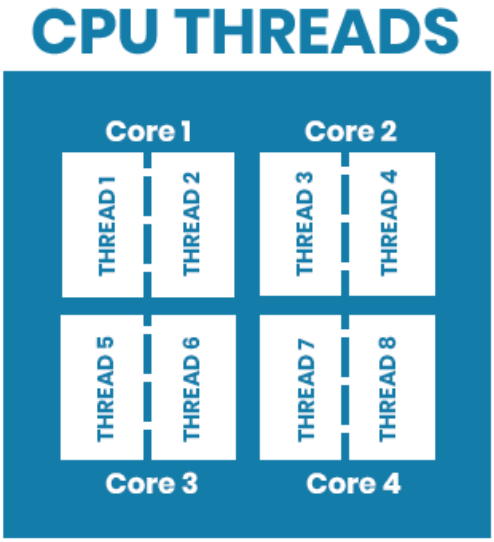
Chapter 13 Concurrency

- A **process** is a program in execution. It's basically the allocated memory for instructions and data.
- A **thread** is a lightweight unit of execution within a process that shares the same memory space and system resources as the process. It's what the CPU actually runs; it's about scheduling access to shared resources.
- A **task** is a logical unit of work that can be executed by one or more threads. It will commonly be implemented as a lambda expression. A thread can complete multiple independent tasks but only one task at a time.



Thread concurrency refers to the ability of multiple threads to run simultaneously and share the same resources, such as memory and CPU time.



One CPU can only execute one instruction at a time, so when multiple threads are running, the CPU will switch between them, giving the illusion of parallelism, as the threads will execute in a seemingly simultaneous manner, but in reality, they are taking turns using the CPU.

Operating systems use a **thread scheduler** to determine which threads should be currently executing. The scheduler will use various algorithms to determine which thread to run next, based on factors such as thread priority, CPU utilization, and system load.

When a thread's allotted time is complete but the thread has not finished processing, a context switch occurs. A **context switch** is the process of storing a thread's current state and later restoring the state of the thread to continue execution.

Creating a Thread

- Provide a Runnable object or lambda expression to the Thread constructor.
- Create a class that extends Thread and overrides the run() method.

```
@FunctionalInterface
public interface Runnable {
   void run();
}

new Thread(() -> System.out.print("Hello")).start(); // preferred way
System.out.print("World");
```

Order of thread execution

```
Runnable printInventory = () -> System.out.println(TEXT_GREEN + "Printing zoo inventory");
Runnable printRecords = () -> {
    for (int i = 0; i < 3; i++)
        System.out.println(TEXT_BLUE + "Printing record: " + i);
};

System.out.println(TEXT_RED + "begin");
new Thread(printInventory).start();
new Thread(printRecords).start();
new Thread(printInventory).start();
System.out.println(TEXT_YELLOW + "end");
```

```
begin
Printing zoo inventory
end
Printing record: 0
Printing record: 1
Printing record: 2
Printing zoo inventory
```

```
begin
Printing zoo inventory
end
Printing zoo inventory
Printing record: 0
Printing record: 1
Printing record: 2
```

```
begin
end
Printing zoo inventory
Printing zoo inventory
Printing record: 0
Printing record: 1
Printing record: 2
```

The order of thread execution is **indeterminate** once the threads have been started, while the order within a single thread is still **linear**.

!! Calling `run()` on a `Thread` or a `Runnable` does not start a new thread

Thread Types

- A *system thread* is created by the Java Virtual Machine (JVM) and runs in the background of the application. For example, garbage collection is managed by a system thread created by the JVM.
- A *user-defined thread* is one created by the application developer to accomplish a specific task. (at least one thread, the main entry point of the application)
- A *daemon thread* is one that will not prevent the JVM from exiting when the program finishes. System and user-defined threads can both be created as daemon threads.

```
public static void main(String[] args) {
    var job = new Thread(() -> pause());      // Create thread

    job.start();                              // Start thread
    System.out.println(TEXT_GREEN + "Main method finished!");
}

private static void pause() {
    try {
        Thread.sleep(10_000);
    } catch (InterruptedException e) {
    }
    System.out.println(TEXT_RED + "Thread finished!");
}
```

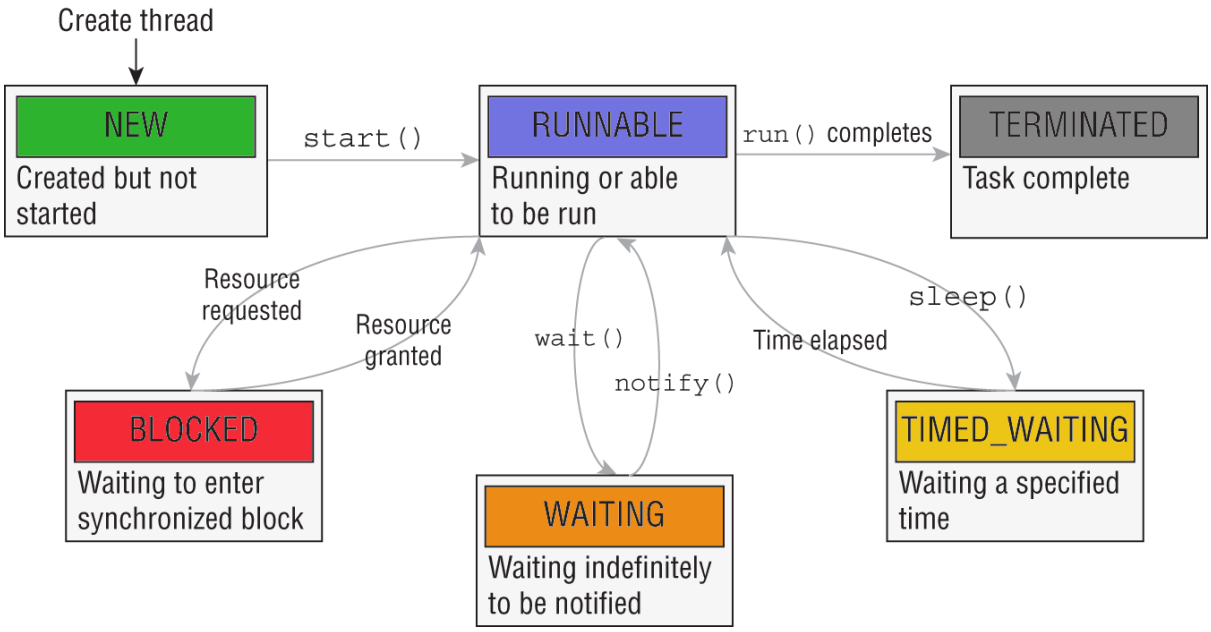
```
PS C:\Users\dmaftei\Projects\other\java-study\src\main\java\org\example> java Main.java |  ForEach-Object { Write-Host "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') $_" }
2023-03-26 15:51:52 Main method finished!
2023-03-26 15:52:02 Thread finished!
PS C:\Users\dmaftei\Projects\other\java-study\src\main\java\org\example>
```

If we add `job.setDaemon(true);`

```
PS C:\Users\dmaftei\Projects\other\java-study\src\main\java\org\example> java Main.java |  ForEach-Object { Write-Host "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') $_" }
2023-03-26 15:52:27 Main method finished!
PS C:\Users\dmaftei\Projects\other\java-study\src\main\java\org\example>
```

❗ For the exam, just remember that by default, user-defined threads are not daemons, and the program will wait for them to finish.

Thread's Life Cycle



- the thread state becomes `TERMINATED` once the work for the thread is completed or an uncaught exception is thrown

Creating Threads with the Concurrency API

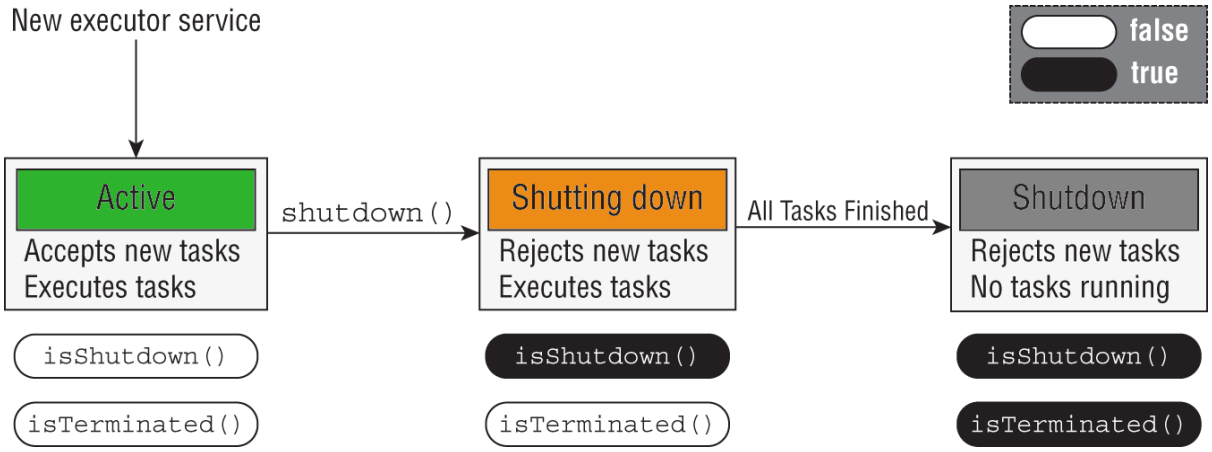
`java.util.concurrent` package

`ExecutorService` interface, which defines services that create and manage threads

```
ExecutorService service = Executors.newSingleThreadExecutor();
try {
    System.out.println(TEXT_RED + "begin");
    service.execute(printInventory); // a void, “fire-and-forget” method
    service.execute(printRecords);
    service.execute(printInventory); // consider using submit()
    System.out.println(TEXT_YELLOW + "end");
} finally {
    service.shutdown(); // always call shutdown() !!!
}
```

- we now have only 2 threads, the main one and the one created by newSingleThreadExecutor(); With a single-thread executor, tasks are guaranteed to be executed sequentially.
- the shutdown process for a thread executor involves first rejecting any new tasks submitted to the thread executor while continuing to execute any previously submitted tasks.
- if a new task is submitted to the thread executor while it is shutting down, a `RejectedExecutionException` will be thrown.

!!A thread executor creates a *non-daemon* thread on the first task that is executed, so failing to call `shutdown()` will result in your application *never terminating. (*`ExecutorService` does not extend `AutoCloseable`)



❗ For the exam, you should be aware that `shutdown()` does not stop any tasks that have already been submitted to the thread executor. (`ExecutorService` provides a method called `shutdownNow()` which *attempts to stop* all running tasks and discards any that have not been started yet, but it is not guaranteed to succeed.

Method name	Description
`void execute(Runnable command)`	Executes `Runnable` task at some point in future.
`Future<?> submit(Runnable task)`	Executes `Runnable` task at some point in future and returns `Future` representing task.
`<T> Future<T> submit(Callable<T> task)`	Executes `Callable` task at some point in future and returns `Future` representing pending results of task.
`<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`	Executes given tasks and waits for all tasks to complete. Returns `List` of `Future` instances in same order in which they were in original collection.
`<T> T invokeAny(Collection<? extends Callable<T>> tasks)`	Executes given tasks and waits for at least one to complete.

Waiting for Results

```
// interface
Future<?> future = service.submit(() -> System.out.println("Hello"));
```

Method name	Description
`boolean isDone()`	Returns `true` if task was completed, threw exception, or was cancelled.
`boolean isCancelled()`	Returns `true` if task was cancelled before it completed normally.
`boolean cancel(boolean mayInterruptIfRunning)`	Attempts to cancel execution of task and returns `true` if it was successfully cancelled or `false` if it could not be cancelled or is complete.
`V get()`	Retrieves result of task, waiting endlessly if it is not yet available.
`V get(long timeout, TimeUnit unit)`	Retrieves result of task, waiting specified amount of time. If result is not ready by time timeout is reached, checked `TimeoutException` will be thrown.

- the `get()` method always returns `null` when working with `Runnable` expressions.

Callable

```
@FunctionalInterface
public interface Callable<V> {
   V call() throws Exception;
}
```

- Runnable and Callable are interchangeable in situations where the lambda does not throw an exception, and there is no return type

Waiting for termination

```
service.awaitTermination(1, TimeUnit.MINUTES);
```

```
// Check whether all tasks are finished
if(service.isTerminated()) System.out.println("Finished!");
else System.out.println("At least one task is still running");
```

Scheduling Tasks

```
ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();
```

Method name	Description
`schedule(Callable<V> callable, long delay, TimeUnit unit)`	Creates and executes `Callable` task after given delay
`schedule(Runnable command, long delay, TimeUnit unit)`	Creates and executes `Runnable` task after given delay
`scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`	Creates and executes `Runnable` task after given initial delay, creating new task every period value that passes
`scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`	Creates and executes `Runnable` task after given initial delay and subsequently with given delay between termination of one execution and commencement of next

- the `scheduleAtFixedRate()` method creates a new task and submits it to the executor every period, regardless of whether the previous task finished
- the `scheduleWithFixedDelay()` method creates a new task only after the previous task has finished

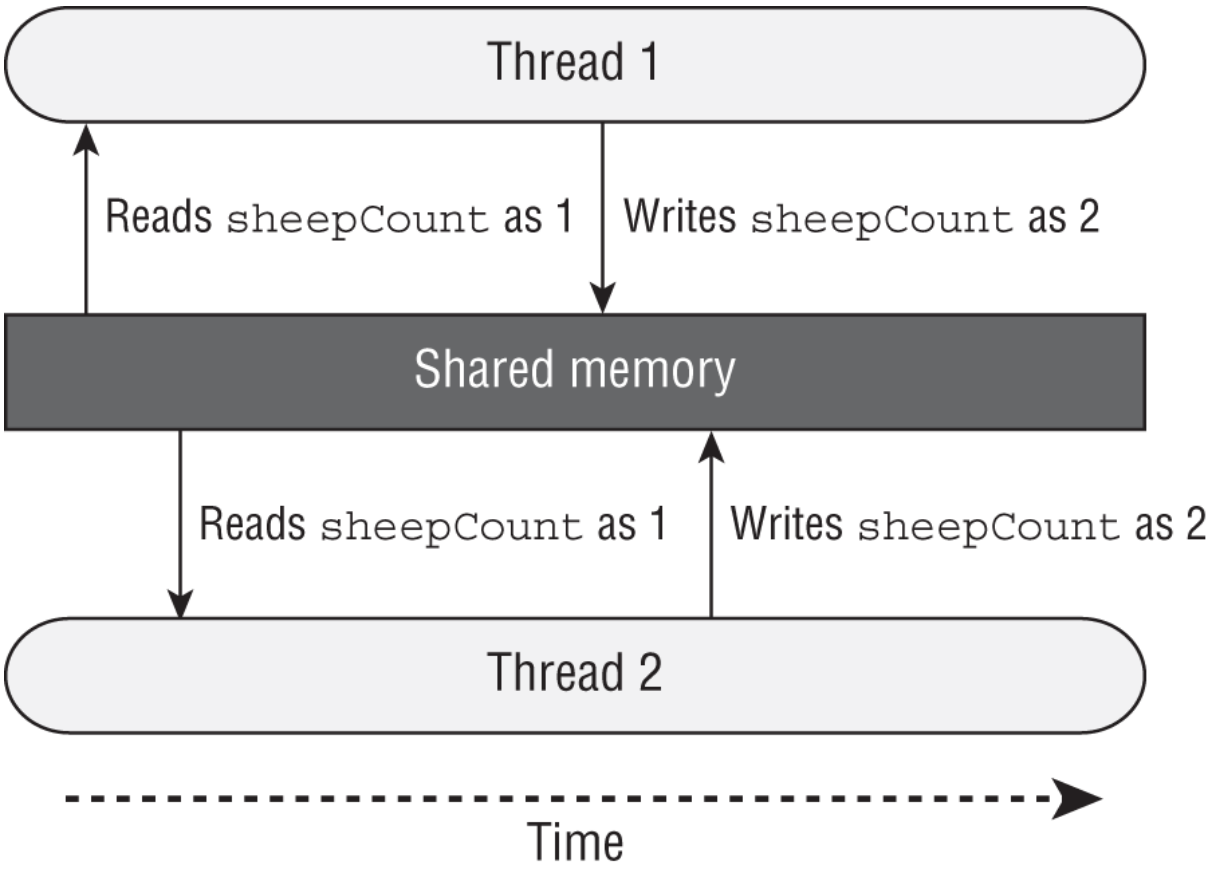
A **thread pool** is a group of pre-instantiated reusable threads that are available to perform a set of arbitrary tasks.

Method name	Description
`ExecutorService newSingleThreadExecutor()`	Creates single-threaded executor that uses single worker thread operating off unbounded queue. Results are processed sequentially in order in which they are submitted.
`ScheduledExecutorService newSingleThreadScheduledExecutor()`	Creates single-threaded executor that can schedule commands to run after given delay or to execute periodically.
`ExecutorService newCachedThreadPool()`	Creates thread pool that creates new threads as needed but reuses previously constructed threads when they are available.
`ExecutorService newFixedThreadPool(int)`	Creates thread pool that reuses fixed number of threads operating off shared unbounded queue.
`ScheduledExecutorService newScheduledThreadPool(int)`	Creates thread pool that can schedule commands to run after given delay or execute periodically.

If the pool runs out of available threads, the task will be queued by the thread executor and wait to be completed.

Thread-safety

- the property of an object that guarantees safe execution by multiple threads at the same time



- the unexpected result of two tasks executing at the same time is referred to as a **race condition**

Volatile

- ensures that only one thread is modifying a variable at one time and that data read among multiple threads is consistent.

```
private volatile int sheepCount = 0;

private void incrementAndReport() {
  System.out.print((++sheepCount)+" "); // still not thread safe
}
```

- if the increment operator represents the expression `sheepCount = sheepCount + 1` then each read and write operation is thread-safe, but the combined operation is not

Atomic Classes

- *Atomic* is the property of an operation to be carried out as a single unit of execution without any interference from another thread.

Class name	Description
`AtomicBoolean`	A `boolean` value that may be updated atomically
`AtomicInteger`	An `int` value that may be updated atomically
`AtomicLong`	A `long` value that may be updated atomically

Method	Description
`get()`	Retrieves current value
`set(`*`type`* `newValue)`	Sets given value, equivalent to assignment `=` operator
`getAndSet(`*`type`* `newValue)`	Atomically sets new value and returns old value
`incrementAndGet()`	For numeric classes, atomic pre-increment operation equivalent to `++value`
`getAndIncrement()`	For numeric classes, atomic post-increment operation equivalent to `value++`
`decrementAndGet()`	For numeric classes, atomic pre-decrement operation equivalent to `--value`
`getAndDecrement()`	For numeric classes, atomic post-decrement operation equivalent to `value--`

- using the atomic classes ensures that the data is consistent between workers and that no values are lost due to concurrent modifications.

Synchronized Blocks

A **monitor**, also called a **lock**, is a structure that supports **mutual exclusion**, which is the property that at most one thread is executing a particular segment of code at a given time.

- In Java, any `Object` can be used as a monitor, along with the `synchronized` keyword

```
var manager = new SheepManager();
synchronized(manager) {
   // Work to be completed by one thread at a time
}

-----
void sing() {
   synchronized(this) {
      System.out.print("La la la!");
   }
}
-----
synchronized void sing() {
   System.out.print("La la la!");
}
```

- You can use `static` synchronization if you need to order thread access across all instances rather than a single instance.

Lock Framework

- Instead of synchronizing on any `Object`, though, we can “lock” only on an object that implements the `Lock` interface

```
Lock lock = new ReentrantLock();
try {
   lock.lock();
   // Protected code
} finally {
   lock.unlock();
}
```

If you attempt to release a lock that you do not have, you will get an exception at runtime.

```
Lock lock = new ReentrantLock();
lock.unlock(); // IllegalMonitorStateException
```

Method	Description
`void lock()`	Requests lock and blocks until lock is acquired.
`void unlock()`	Releases lock.
`boolean tryLock()`	Requests lock and returns immediately. Returns `boolean` indicating whether lock was successfully acquired.
`boolean tryLock(long timeout, TimeUnit unit)`	Requests lock and blocks for specified time or until lock is acquired. Returns `boolean` indicating whether lock was successfully acquired.

!! It is critical that you release a lock the same number of times it is acquired!

CyclicBarrier

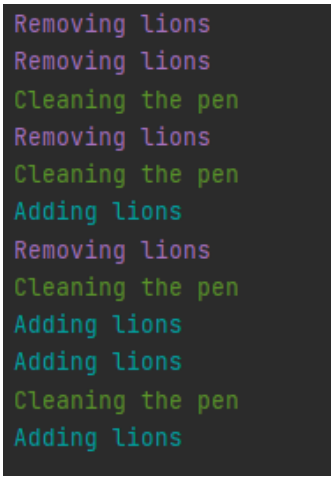
- allows us to perform complex, multithreaded tasks while all threads stop and wait at logical barriers.

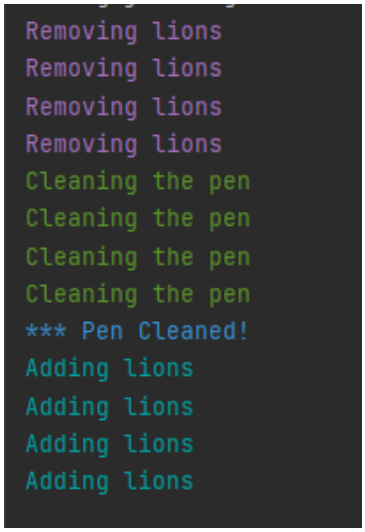
```
private static void removeLions() { System.out.println(TEXT_PURPLE + "Removing lions");   }
private static void cleanPen()    { System.out.println(TEXT_GREEN +  "Cleaning the pen"); }
private static void addLions()    { System.out.println(TEXT_CYAN +   "Adding lions");     }

private static void performTask(CyclicBarrier c1, CyclicBarrier c2) {
      try {
          removeLions();
          c1.await(); // result 1 with this commented out
          cleanPen();
          c2.await(); // result 1 with this commented out
          addLions();
      } catch (InterruptedException | BrokenBarrierException e) {
          // Handle checked exceptions here
      }
}

public static void main(String[] args) {
      var service = Executors.newFixedThreadPool(4);
      try {
          var c1 = new CyclicBarrier(4); // the number of threads to wait for
          var c2 = new CyclicBarrier(4,
                  () -> System.out.println(TEXT_BLUE + "*** Pen Cleaned!"));
          for (int i = 0; i < 4; i++)
              service.submit(() -> performTask(c1, c2));
      } finally {
```

```
            service.shutdown();
        }
}
```





Concurrent Collections

- A *memory consistency error* occurs when two threads have inconsistent views of what should be the same data.
- The concurrent classes were created to help avoid common issues in which multiple threads are adding and removing objects from the same collections.
- When two threads try to modify the same nonconcurrent collection, the JVM may throw a `ConcurrentModificationException` at runtime.

```
var foodData = new HashMap<String, Integer>();  // use new ConcurrentHashMap<String, Integer>();
foodData.put("penguin", 1);
foodData.put("flamingo", 2);
for (String key : foodData.keySet())
    foodData.remove(key);
```

- If the collection is immutable (and contains immutable objects), the concurrent collections are not necessary.

Class name	Java Collections interfaces	Sorted?	Blocking
`ConcurrentHashMap`	`Map` `ConcurrentMap`	No	No
`ConcurrentLinkedQueue`	`Queue`	No	No
`ConcurrentSkipListMap`	`Map` `SortedMap` `NavigableMap` `ConcurrentMap` `ConcurrentNavigableMap`	Yes	No
`ConcurrentSkipListSet`	`Set` `SortedSet` `NavigableSet`	Yes	No
`CopyOnWriteArrayList`	`List`	No	No
`CopyOnWriteArraySet`	`Set`	No	No
`LinkedBlockingQueue`	`Queue` `BlockingQueue`	No	Yes

`CopyOnWrite` classes create a copy of the collection any time a reference is added, removed, or changed in the collection and then update the original collection reference to point to the copy.

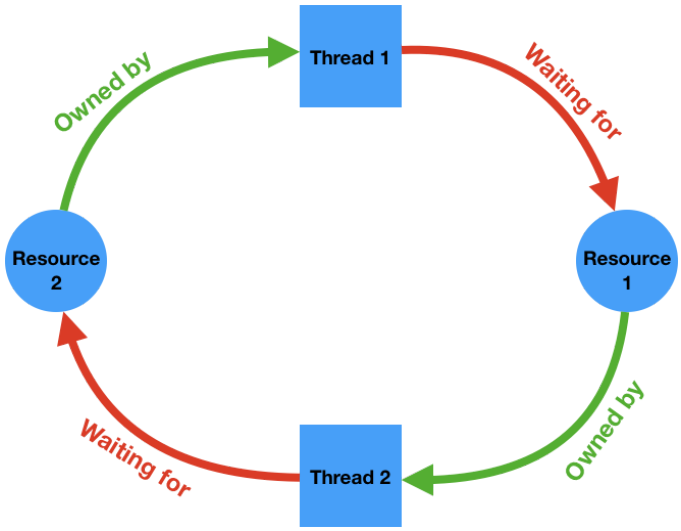
- if you are passed a nonconcurrent collection and need synchronization:

TABLE 13.10 Synchronized `Collections` methods

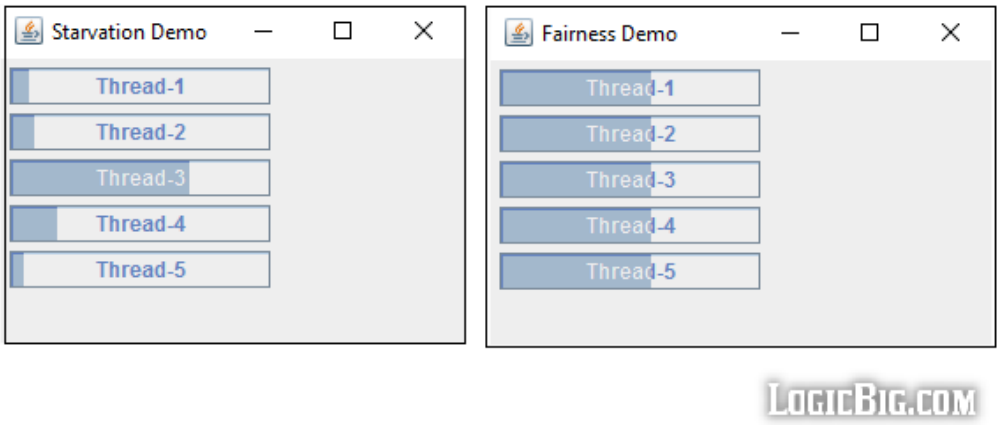
`synchronizedCollection(Collection<T> c)`
`synchronizedList(List<T> list)`
`synchronizedMap(Map<K,V> m)`
`synchronizedNavigableMap(NavigableMap<K,V> m)`
`synchronizedNavigableSet(NavigableSet<T> s)`
`synchronizedSet(Set<T> s)`
`synchronizedSortedMap(SortedMap<K,V> m)`
`synchronizedSortedSet(SortedSet<T> s)`

Threading Problems

- *Liveness* is the ability of an application to be able to execute in a timely manner. Liveness problems are often the result of a thread entering a `BLOCKING` or `WAITING` state forever, or repeatedly entering/exiting these states.
- For the exam, there are three types of liveness issues with which you should be familiar: `deadlock`, `starvation` and `livelock`
- **Deadlock** occurs when two or more threads are blocked forever, each waiting on the other.



- **Starvation** occurs when a single thread is perpetually denied access to a shared resource or lock.



- **Livelock** occurs when two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task. Two or more threads actively try to acquire a set of locks, are unable to do so, and restart part of the process. It is the result of a failed deadlock recovery.

Parallel Streams

- The number of threads available in a parallel stream is proportional to the number of available CPUs in your environment.

```
Collection<Integer> collection = List.of(1,2);

Stream<Integer> p1 = collection.stream().parallel();
Stream<Integer> p2 = collection.parallelStream();
```

- The `Stream` interface includes a method `isParallel()`

```
long start = System.currentTimeMillis();
List.of(1, 2, 3, 4, 5)
        .stream() // replace with parallelStream()
        .map(w -> doWork(w)) // do work just includes a Thread.sleep(5000);
        .forEach(s -> System.out.print(s + " ")); // if you require ordering, use forEachOrdered()

System.out.println();
var timeTaken = (System.currentTimeMillis() - start) / 1000;
System.out.println("Time: " + timeTaken + " seconds"); //25 seconds with serialization, 5 in parallel
```

Parallel Reductions

- Since order is not guaranteed with parallel streams, methods such as `findAny()` on parallel streams may result in unexpected behavior
- For methods such as `findFirst()`, `limit()`, and `skip()` order is still preserved, but performance may *suffer* on a parallel stream
- On parallel streams, the `reduce()` method works by applying the reduction to pairs of elements within the stream to create intermediate values and then combining those intermediate values to produce a final result.
  - it is recommended that you use the three-argument version of `reduce()` when working with parallel streams, by providing an explicit combiner
  - if order is important: make sure that the accumulator and combiner produce the same result regardless of the order they are called in.

```
System.out.println(List.of(1,2,3,4,5,6)
   .parallelStream()
   .reduce(0, (a, b) -> (a - b))); // it may output -21, 3, or some other value.

System.out.println(List.of("w","o","l","f")
   .parallelStream()
   .reduce("X", String::concat)); // serial = Xwolf, parallel = XwXoXlXf.
```

- similar for `collect()`

Parallel Reduction on a Collector

Every `Collector` instance defines a `characteristics()` method that returns a set of `Collector.Characteristics` attributes.

Requirements for Parallel Reduction with *collect()*

- The stream is parallel.
- The parameter of the `collect()` operation has the `Characteristics.CONCURRENT` characteristic.

- Either the stream is unordered or the collector has the characteristic `Characteristics.UNORDERED`.

```
parallelStream.collect(Collectors.toSet());  // Not a parallel reduction

parallelStream.collect(Collectors.toConcurrentMap(String::length,
      k -> k,
      (s1, s2) -> s1 + "," + s2)); // parallel

parallelStream().collect(
   Collectors.groupingByConcurrent(String::length)); // parallel
```