Concurrency

- Processes
- Threads
  - Creating Threads
  - sleep/interrupt/join methods
- Synchronization between Threads
  - interference & memory inconsistency
  - happen-before relationship
  - synchronized & volatile
  - liveness
- High-level Concurrency API
- Applications

Core Java Volume 1 – Chapter 12 Concurrency

The Java™ Tutorials – Lesson: Concurrency

Jakob Jenkov, Java Concurrency and Multithreading

장윤기, Practical 모던 자바(2020)

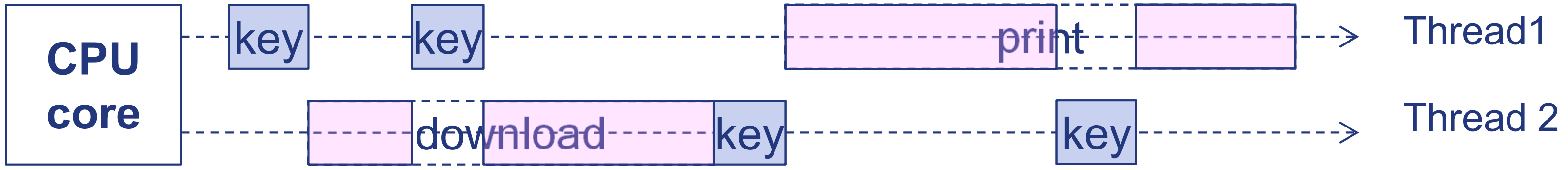
남궁성, 자바의 정석 (2019)

Raoul-Gabriel Urma et. al., Modern Java in Action (2018)

Elliotte Rusty Harold, Java Network Programming (2013)

Concurrency

- Software that can do following things is known as concurrent software.
  - We can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio.
  - **Even the word processor should always be ready to respond to keyboard and mouse events**, no matter how busy it is reformatting text or updating the display. Software that can do such things is known as concurrent software.



- The Java platform is designed from the ground up to support concurrent programming, with **basic concurrency support** in the Java programming language and the Java class libraries.

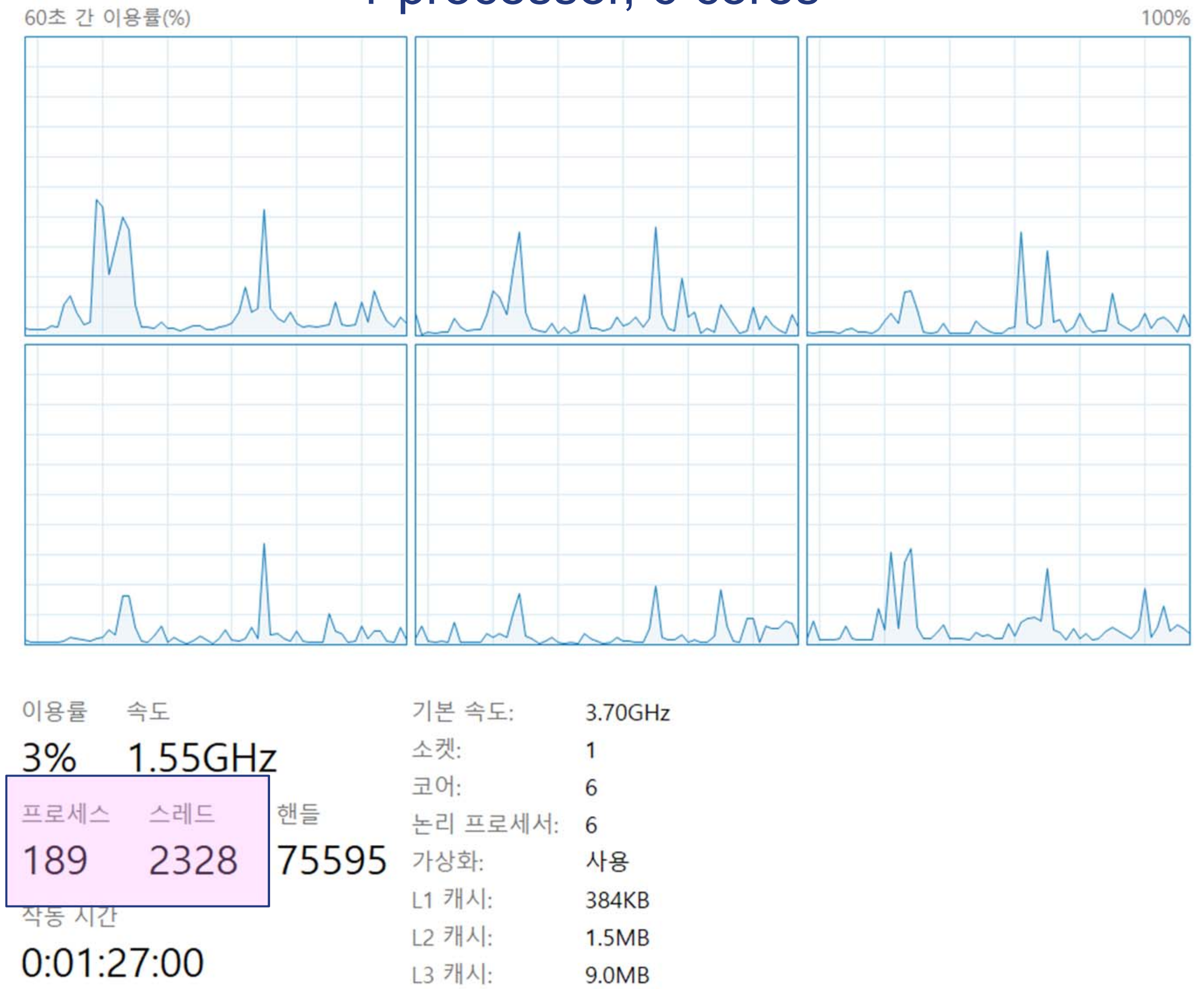
- Since version 5.0, the Java platform has also included **high-level concurrency APIs**.

Processes and Threads (1/2)

- In concurrent programming, there are two basic units of execution: **processes** and **threads**.
  - A computer system normally has many active processes and threads.
  - It's becoming more and more common for computer systems to have **multiple processors or processors with multiple execution cores**.

- A process generally has a complete, private set of basic run-time resource
  - **Most implementations of the Java virtual machine run as a single process.**
  - A Java application can create additional processes using a **ProcessBuilder** object.

<1 processor, 6 cores>



이용률	속도
3%	1.55GHz

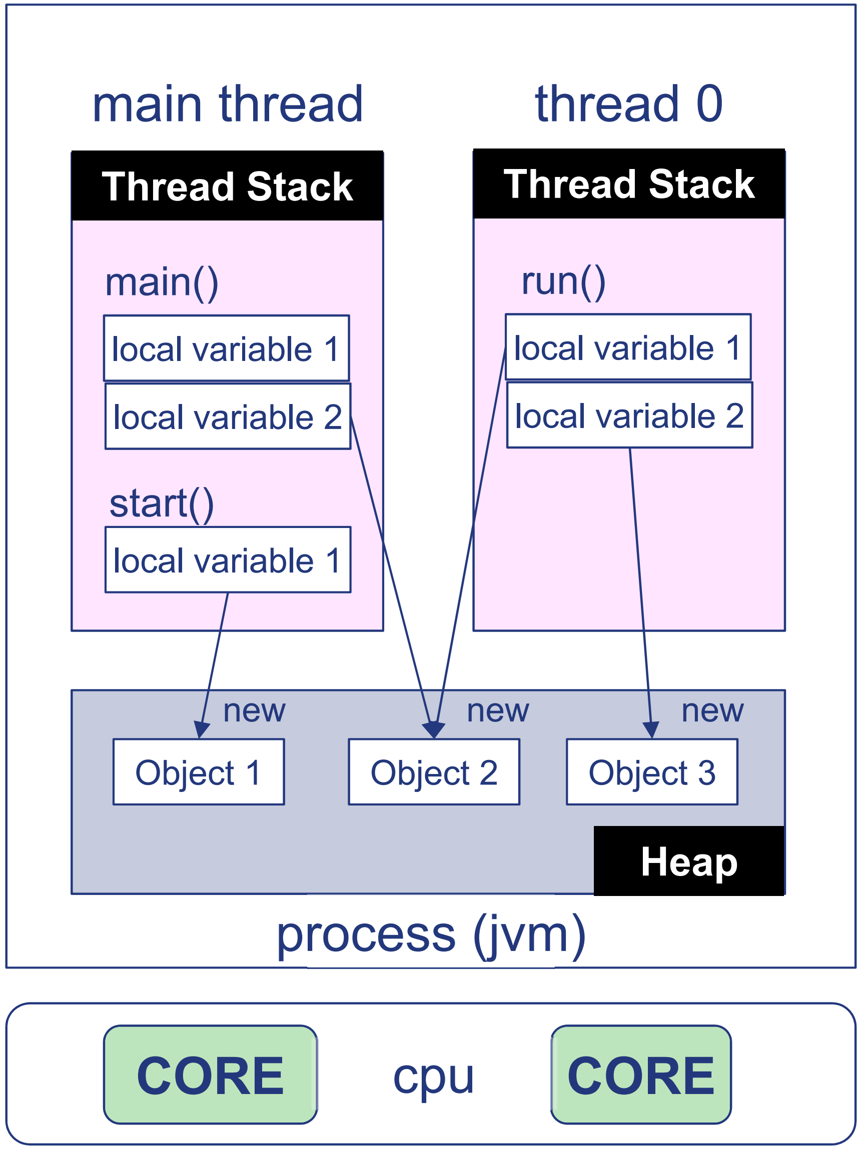
프로세스	스레드	핸들
189	2328	75595

작동 시간
0:01:27:00

기본 속도:	3.70GHz
소켓:	1
코어:	6
논리 프로세서:	6
가상화:	사용
L1 캐시:	384KB
L2 캐시:	1.5MB
L3 캐시:	9.0MB

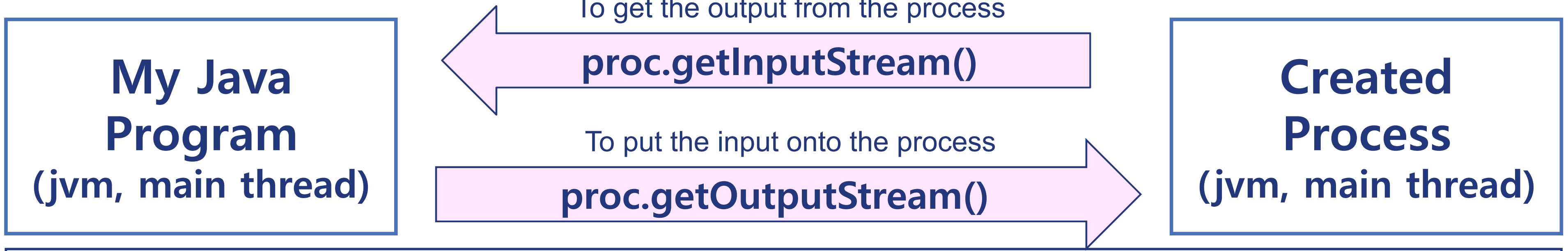
Processes and Threads (2/2)

- Threads are sometimes called lightweight processes.
  - Threads exist within a process.
  - **Every process has at least one.**
  - you start with just one thread, called the **main thread**.
  - **Threads share the process's resources**, including memory and open files.
- Each thread is associated with an instance of the class **Thread**.
  - To directly control thread creation and management, simply **instantiate Thread** each time the application needs to initiate an asynchronous task. (low-level)
  - To abstract thread management from the rest of your application, pass the application's tasks to an **executor**. (high-level)



Creating and Executing Processes

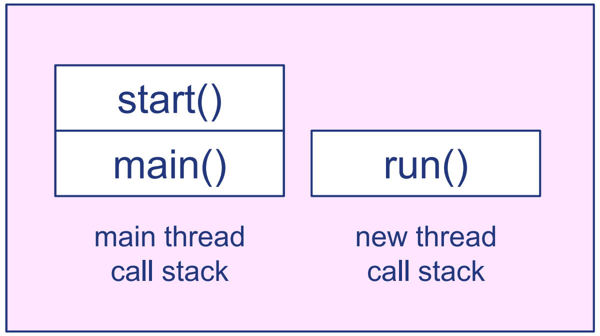
- You can interact with the created process by its standard input, output, and error streams.


To get the output from the process
proc.getInputStream()
My Java Program (jvm, main thread)
To put the input onto the process
proc.getOutputStream()
Created Process (jvm, main thread)

```
public class ProcessBuilderTest {
 public static void main(String args[]) {
   try {
     String param = "C:" + File.separator;  // for Windows
     //Process proc = Runtime.getRuntime().exec("cmd /c dir " + param);
     Process proc = new ProcessBuilder("cmd", "/c", "dir", param).start();
     InputStream in = proc.getInputStream();         // dir I my process
     byte buffer[] = new byte[1024];
     int n = -1;
     while ( (n = in.read(buffer)) != -1 )
        System.out.print(new String(buffer, 0, n));
     in.close();
   } catch(Exception e) { e.printStackTrace(); }
 }
}
```

Creating and Executing Threads

- An application that creates an instance of Thread must provide the code that will run in that thread.
- There are two ways to do this:
  - **Provide a Runnable object**
    - The Runnable interface defines a single method run ()
    - The Runnable object is passed to the Thread constructor
  - **Subclass Thread**
    - The Thread class itself implements Runnable
    - Subclass Thread can provide its own implementation of run
- Employing a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.

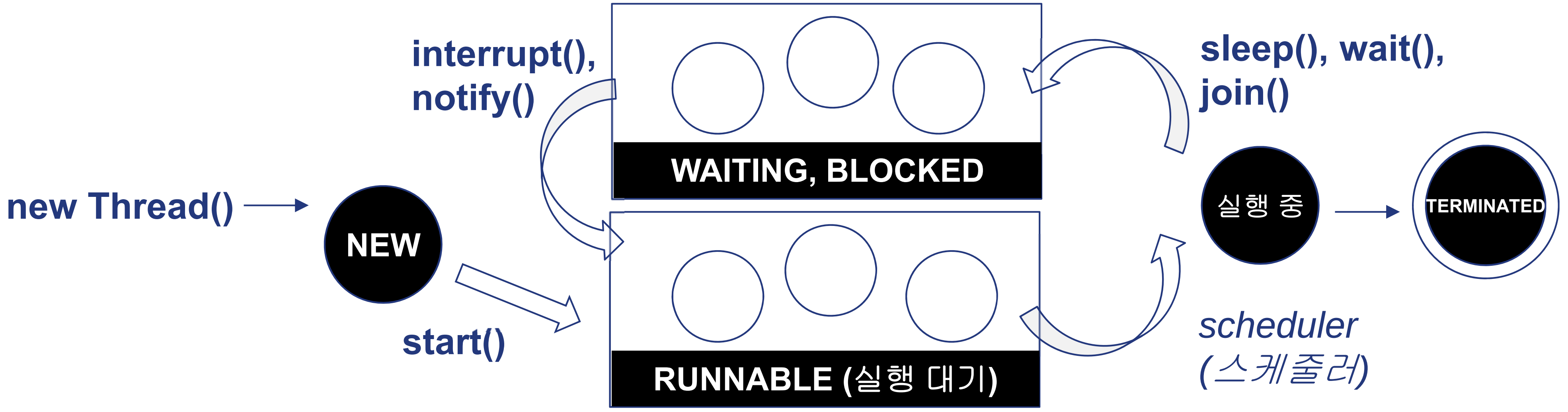


```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Lifecycle and States of Thread

- A thread can be in one of the following states:
  - **NEW** - A thread that has not yet started is in this state.
  - **RUNNABLE** - A thread executing in the Java virtual machine is in this state.
  - **BLOCKED** - A thread that is blocked waiting for a monitor lock is in this state.
  - **WAITING** - A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
  - **TERMINATED** - A thread that has exited is in this state.
- A thread can be **in only one state** at a given point in time.
- There are deprecated methods such as resume(), stop(), suspend().



Pausing Execution with Sleep

- The Thread class defines a number of methods useful for thread management.
- **Thread.sleep()** causes the current thread to suspend execution for a specified period.

```
public class SleepMessages {
      public static void main(String args[]) {
            String messages[] = {
                "1st message", "2nd message", "3rd message", "4th message" } ;
            for ( String message: messages) {
                // Pause for 1 seconds; but not guaranteed !
              try {
                    Thread.sleep(1000);
               } catch (InterruptedException e) { }
                System.out.println(message); // Print a message
            }
      }
}
```

```
1st message
2nd message
3rd message
4th message
```

Interrupts

- An interrupt is an indication to a thread that it should stop what it is doing and do something else.
  - Thread.**interrupt**(); //using an internal flag
- It's up to the programmer to decide exactly how a thread responds to an interrupt, but **it is very common for the thread to terminate**.

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
    //if (Thread.interrupted()) {
    // throw new InterruptedException();
    //}
}
```

```
public class InterruptThreadTest {
  private static class SimpleRunnable
                              implements Runnable {
    public void run() {
      String threadName =
                  Thread.currentThread().getName();
      int i = 0 ;
      while (!Thread.interrupted() ) {
        System.out.printf("%s: %d%n", threadName, i) ;
        i ++ ;
      }
    }
  }

  public static void main(String[] args) {
    Thread thread = new Thread(new SimpleRunnable()) ;
    thread.start();
    Scanner scanner = new Scanner(System.in) ;
    scanner.next() ;
    thread.interrupt() ; // The thread is now interrupted !
  }
}
```

```
Thread-0: 0
Thread-0: 1
Thread-0: 2
...
```

Join

- The join method allows one thread to wait for the completion of another.
- If t is another thread object, invoking t.join() causes **the current thread to pause execution until t's thread terminates**.

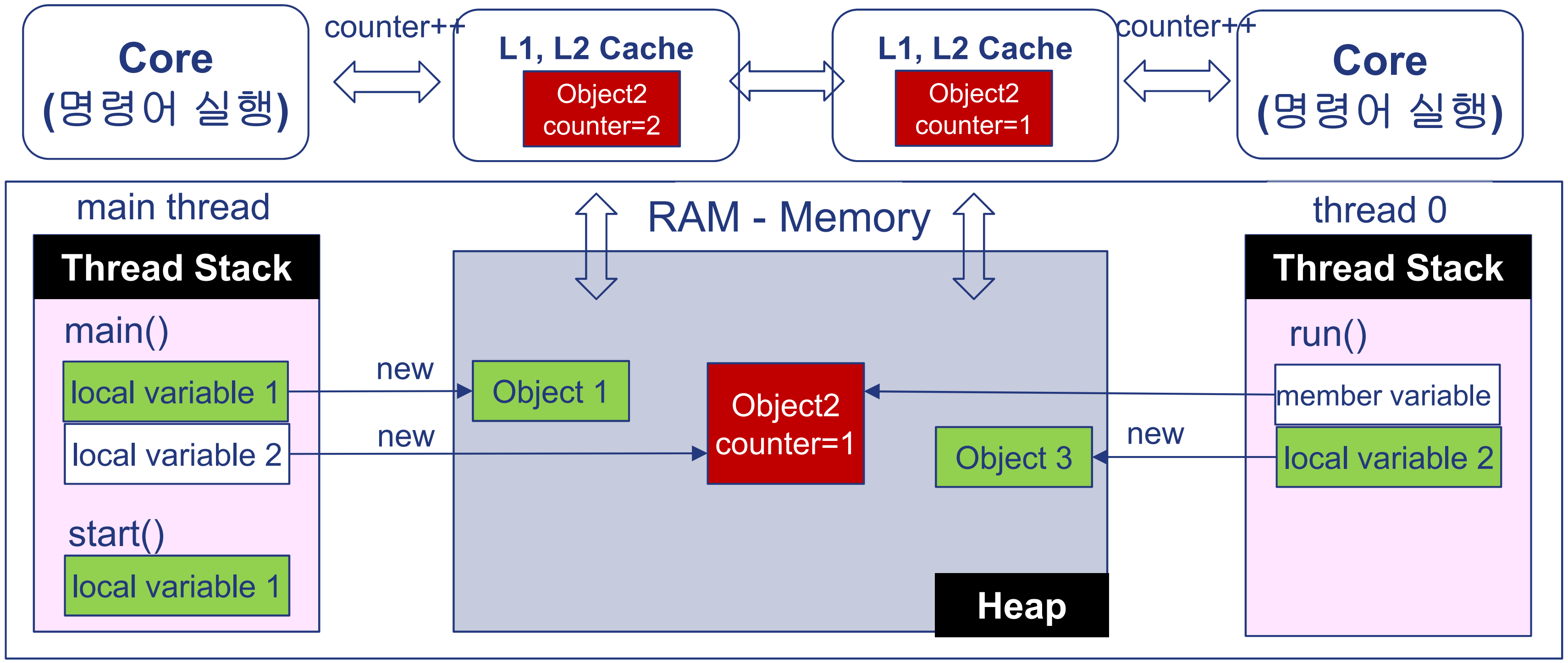
```
public class InterruptThreadTest {
  static void message(String message) {
    String name = Thread.currentThread().getName();
    System.out.format("%s: %s%n", name, message);
  }
  private static class MessageLoop
                             implements Runnable {
    public void run() {
      String importantInfo[] = {"message1",
            "message2", "message3", "message4" };
```

```
      try { for (int i = 0; i < importantInfo.length; i++) {
          Thread.sleep(4000); // Pause for 4 seconds
          message(importantInfo[i]);
        }} catch (InterruptedException e) {
          message("I wasn't done!");
      }}
  }
  public static void main(String args[]) {
    long patience = 1000 * 10; //change 10 to 20
    message("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();
    message("Waiting for MessageLoop thread to finish");
    while (t.isAlive()) {
      message("Still waiting..."); t.join(1000);
      if (((System.currentTimeMillis() - startTime) >
                              patience) && t.isAlive()) {
        message("Tired of waiting!");
        t.interrupt(); t.join();
      }} message("Finally!");
  }}
```

```
main: Still waiting...
Thread-0: message1
main: Still waiting...
...
Thread-0: I wasn't done!
main: Finally!
```

Synchronization

- Threads communicate primarily by **sharing access to fields** and **the objects reference fields refer to**.
- This form of communication is extremely efficient, but makes two kinds of errors possible:
  - **Thread interference**
  - **Memory consistency errors**



Thread Interference

- Counter is designed so that each invocation of increment will add 1 to c, and each invocation of decrement will subtract 1 from c
- However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.
- Because **they are unpredictable, thread interference bugs can be difficult to detect and fix.**

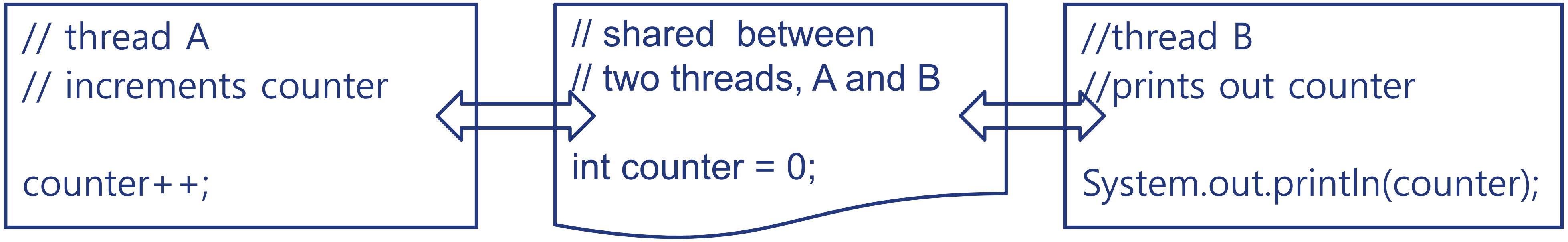
```
public class Actor implements Runnable {
    private String name;
    private Counter counter;
    public List<Integer> getResults() {return results;}
    private List<Integer> results = new ArrayList<>();
    public Actor(String name, Counter counter){
      this.name = name; this.counter = counter;
    }
    public void run() {
        for (int i = 0; i < 10000; i++) {
          counter.increment(); results.add(counter.value());
        }
```

```
public static void main(String[] args) throws InterruptedException {
    Counter counter = new Counter();
    Actor actor1 = new Actor("Actor1", counter);
    Actor actor2 = new Actor("Actor2", counter);
    Thread t1 = new Thread(actor1); Thread t2 = new Thread(actor2);
    t1.start(); t2.start();
    t1.join(); t2.join();
    var stream = actor1.getResults().parallelStream();
    List<Integer> result = Stream.concat(stream, actor2.getResults().stream()).collect(Collectors.toList());
    System.out.println(result);
}
```

```
class Counter {
  private int c = 0;
  public void increment() { c++; }
  public void decrement() { c--; }
  public int value() {return c; }
}
```

Memory Consistency Errors

- Memory consistency errors occur when different threads have inconsistent views of what should be the same data.
- **The causes of memory consistency errors are complex and beyond the scope of our Java curriculum.**
- The key to avoiding memory consistency errors is understanding the **happens-before relationship**.
  - This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement.



```
// thread A
// increments counter

counter++;
```

```
// shared  between
// two threads, A and B

int counter = 0;
```

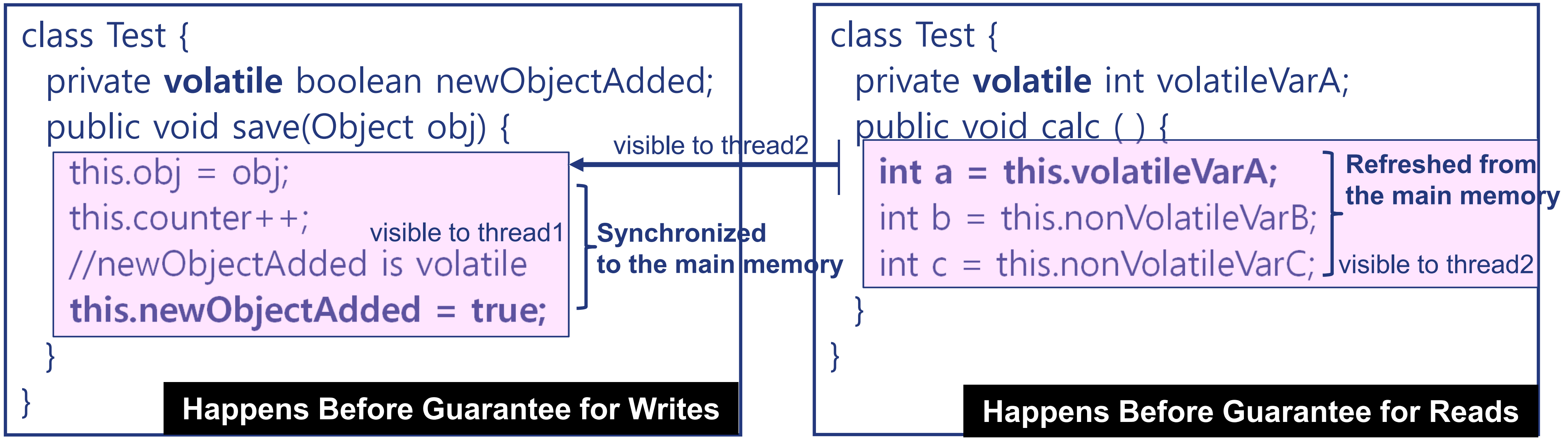
```
//thread B
//prints out counter

System.out.println(counter);
```

  - If the two statements had been executed in the same thread, it would be safe to assume that the value printed out **would be "1"**.
  - But if the two statements are executed in separate threads, the value printed out **might well be "0"**, because **there's no guarantee that thread A's change to counter will be visible to thread B.**

Happens-Before Relationships

- The Happens-before relationship provides some **sort of ordering** and **visibility** guarantee.
- the most important one is if there is a synchronization like a ***synchronized*** block or a ***volatile*** variable then:
  - “A volatile write will happen before another volatile read.”
  - “An unlock on the synchronized block will happen before another lock.”
  - **"All the changes which are visible to T1 before a volatile write or a synchronized unlock will be visible to thread T2 after a volatile read of the same variable or locking on the same monitor."**



```
class Test {
  private volatile boolean newObjectAdded;
  public void save(Object obj) {
    this.obj = obj;
    this.counter++;
    //newObjectAdded is volatile
    this.newObjectAdded = true;
  }
}
```

```
class Test {
  private volatile int volatileVarA;
  public void calc ( ) {
    int a = this.volatileVarA;
    int b = this.nonVolatileVarB;
    int c = this.nonVolatileVarC;
  }
}
```

visible to thread2
visible to thread1
Synchronized to the main memory
Refreshed from the main memory
visible to thread2
Happens Before Guarantee for Writes
Happens Before Guarantee for Reads

synchronized (1/2)

- The Java programming language provides two basic synchronization idioms:
- synchronized methods
  - it is not possible for two invocations of synchronized methods on the same object to interleave.
  - when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object.
- Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors
- This strategy is effective, but can present problems with liveness, as we'll see later in this lesson.

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

synchronized methods

synchronized (2/2)

- Intrinsic lock
  - Synchronization is built around an internal entity known as the intrinsic lock or monitor lock.
  - Every object has an intrinsic lock associated with it.
  - A thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them.
  - When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.
  - when a static synchronized method is invoked the thread acquires the intrinsic lock for the Class object associated with the class.
- Another way to create synchronized code is with synchronized statements.
  - Synchronized statements must specify the object that provides the intrinsic lock.
  - Synchronized statements are also useful for improving concurrency with fine-grained synchronization.
  - Allowing a thread to acquire the same lock more than once enables reentrant synchronization.

```
public class MsLunch {
    private long c1 = 0; private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) { c1++; }
    }
    public void inc2() {
        synchronized(lock2) { c2++; }
    }
}
```

synchronized statements

volatile

- An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all.
- No side effects of an atomic action are visible until the action is complete.
- There are actions you can specify that are atomic:
  - Reads and writes are atomic for reference variables and for most primitive variables.
  - Reads and writes are atomic for all variables declared volatile.
- Atomic actions cannot be interleaved, so they can be used without fear of thread interference.
- However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible.
- Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors.

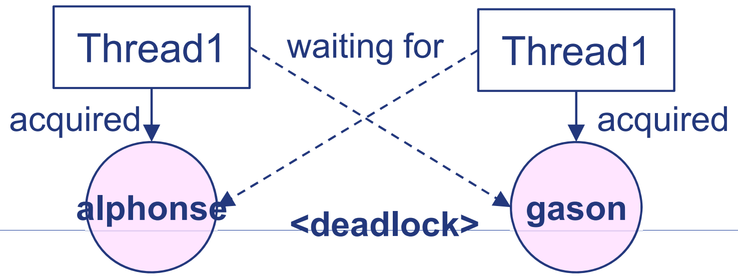
Liveness (1/2)

- A concurrent application's ability to execute in a timely manner is known as its **liveness**.
- **Deadlock** describes a situation where two or more threads are blocked forever, waiting for each other.

```
public class DeadlockTest {
  static class Friend {
    private final String name;
    public Friend(String name) { this.name = name; }
    public String getName() { return this.name; }
    public synchronized void bow(Friend bower) {
      System.out.format("%s: %s has bowed to me!%n",
        this.name, bower.getName());
      bower.bowBack(this);
    }
    public synchronized void bowBack(Friend bower) {
      System.out.format("%s: %s"
        + " has bowed back to me!%n",
        this.name, bower.getName());
    }
  }
```

```
public static void main(String[] args) {
    final Friend alphonse = new Friend("Alphonse");
    final Friend gaston = new Friend("Gaston");
    new Thread(new Runnable() {
      public void run() { alphonse.bow(gaston); }
    }).start();
    new Thread(new Runnable() {
      public void run() { gaston.bow(alphonse); }
    }).start();
  }
}
```

```
Alphonse: Gaston  has bowed to me!
Gaston: Alphonse  has bowed to me!
```

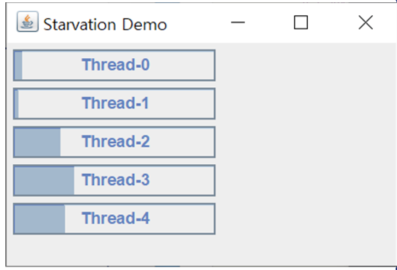


Liveness (2/2)

- **Starvation** and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.
- Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
  - This happens when shared resources are made unavailable for long periods by **"greedy" threads.**

```
public class StarvationTest {
  private static Object sharedObj = new Object();
  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() { createAndShowGUI(); } }); }
  public static void createAndShowGUI () {
    JFrame frame = new JFrame("Starvation Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(new Dimension(300, 200));
    frame.setLayout(new FlowLayout(FlowLayout.LEFT));
    for (int i = 0; i < 5; i++) {
      ProgressThread progressThread = new ProgressThread();
      frame.add(progressThread.getProgressComponent());
      progressThread.start();
    }
    frame.setLocationRelativeTo(null); frame.setVisible(true); }
```

```
private static class ProgressThread extends Thread {
  JProgressBar progressBar;
  ProgressThread () {
    progressBar = new JProgressBar();
    progressBar.setString(this.getName());
    progressBar.setStringPainted(true);
  }
  JComponent getProgressComponent () {
              return progressBar; }
public void run () {
    int c = 0;
    while (true) {
      synchronized (sharedObj) {
        if (c == 100) c = 0;
        progressBar.setValue(++c);
        try { Thread.sleep(100);  //gready thread
        } catch (InterruptedException e) {}}}}}}
```



High Level Concurrency

- So far, this lesson has focused on the low-level APIs that have been part of the Java platform from the very beginning.
- These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks.
- We'll look at some of the **high-level concurrency features introduced with version 5.0** of the Java platform.
- Most of these features are implemented in the new java.util.concurrent packages.
  - **Lock** objects support locking idioms that simplify many concurrent applications.
  - **Executors** define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
  - **Concurrent collections** make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
  - **Atomic variables** have features that minimize synchronization and help avoid memory consistency errors.

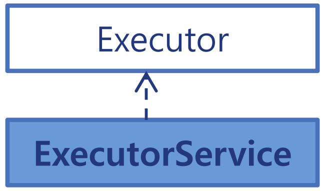
Lock Objects

- Synchronized code relies on a simple kind of reentrant lock **(ReentrantLock, Java 5).**
- This kind of lock is easy to use, but has many limitations.
- More sophisticated locking idioms are supported by the java.util.concurrent.locks package.
- Lock objects work very much like the implicit locks used by synchronized code.
  - As with implicit locks, only one thread can own a Lock object at a time.
  - Lock objects also support a wait/notify mechanism, through their associated Condition objects.
- The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock.
- The tryLock() method backs out if the lock is not available immediately or before a timeout expires (if specified).
- We model this improvement by requiring that our Friend objects must acquire locks for both participants before proceeding with the bow.

```
private final Lock lock = new ReentrantLock();
public boolean impendingBow (Friend bower) {
  Boolean myLock = false;
  Boolean yourLock = false;
  try {
    myLock = lock.tryLock();
    yourLock = bower.lock.tryLock();
  } finally {
    if (! (myLock && yourLock)) {
      if (myLock) { lock.unlock(); }
      if (yourLock) { bower.lock.unlock(); }
    }}
  return myLock && yourLock;
}
```

ExecutorService (Java 5) (1/2)

- In large-scale applications, it makes sense to separate thread management and creation from the rest of the application.
- The java.util.concurrent package defines three executor interfaces:
  - **Executor**, a simple interface that supports launching new tasks.
  - **ExecutorService**, a subinterface of Executor, which adds features that help manage the life cycle, both of the individual tasks and of the executor itself.
  - **ScheduledExecutorService**, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.



```
public class ExecutorTest {
    public static void main(String[] args)
            throws InterruptedException, ExecutionException {
        ExecutorService executor =
                                Executors.newSingleThreadExecutor();
        Future<String> future = executor.submit(() -> {
            TimeUnit.MILLISECONDS.sleep(3000);
            return "Callable task is done!";
        });
        System.out.println(future.get());
        //the program does not stop. why?
        executor.shutdown();
}}
```

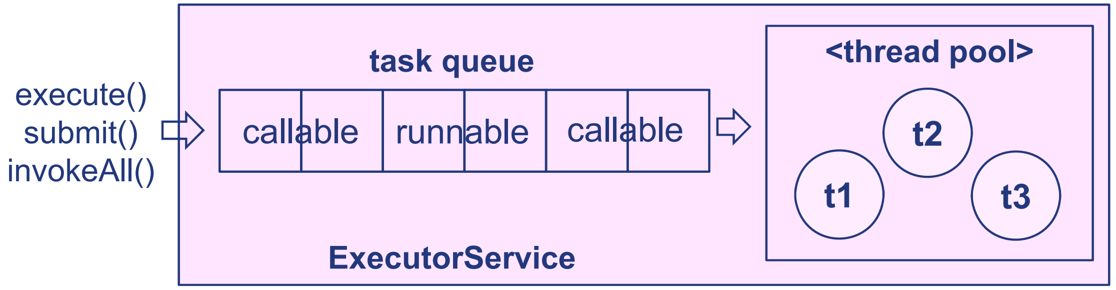
```
main
Hello pool-1-thread-1
```

interface	method	description
Executor	void execute (Runnable command)	Executes the given command at some time in the future.
Callable<T>	V call()	Computes a result, or throws an exception if unable to do so.
Future<V>	V get()	Waits if necessary for the computation to complete, and then retrieves its result.
	<T> Future<T> submit(Callable<T> task)	Submits a value-returning task for execution and returns a Future representing the pending results of the task.
	Future<?> submit(Runnable task)	Submits a Runnable task for execution and returns a Future representing that task.

ExecutorService (2/2)

- The Java ExecutorService is very similar to a thread pool.

newSingleThreadExecutor()	하나의 쓰레드로 등록된 작업을 순차 처리함
newCachedThreadPool()	시스템 자원이 허용하는 만큼의 쓰레드를 생성하여 작업을 동시에 처리함
newFixedThreadPool(n)	동시에 실행할 쓰레드 개수를 지정하여 작업을 동시 처리함



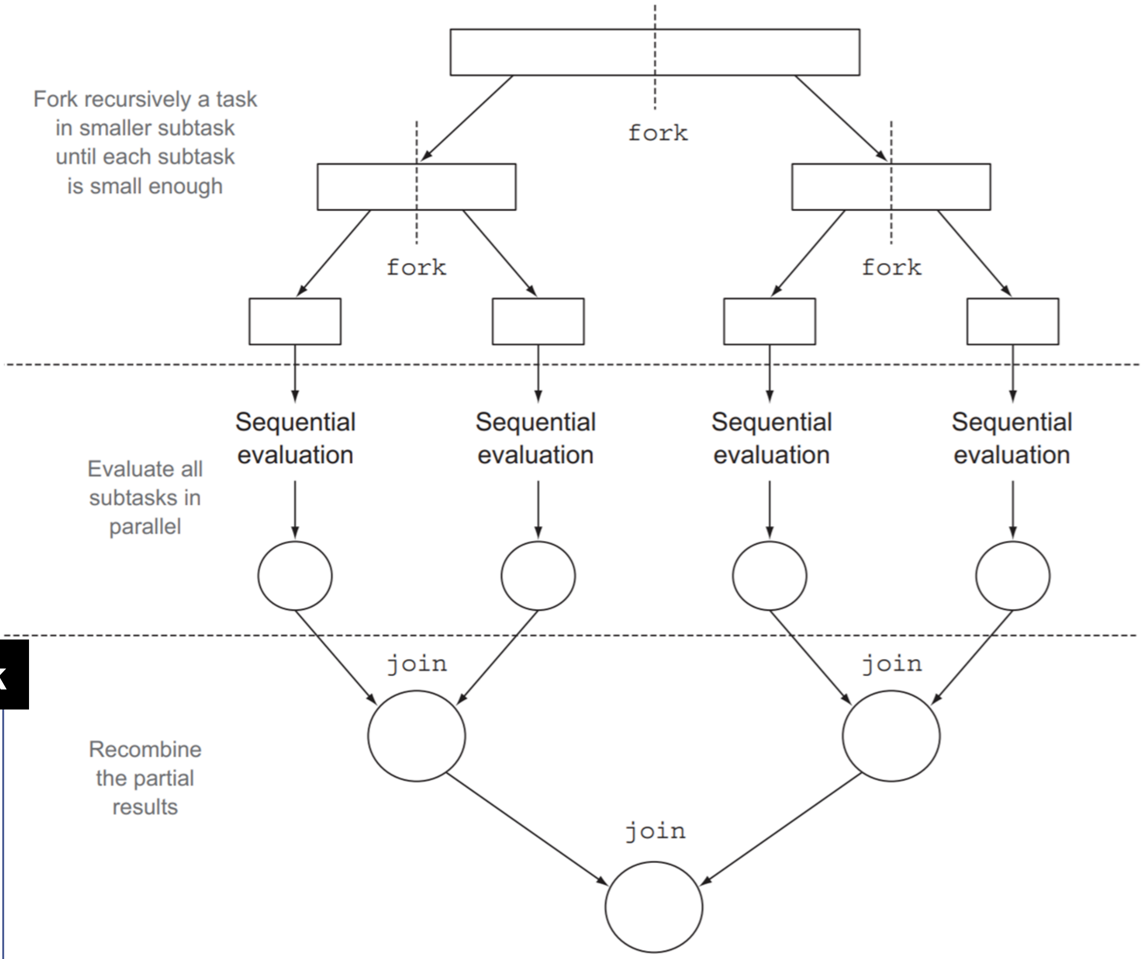
- There are a few different ways to delegate tasks for execution to an ExecutorService:
  - execute(Runnable) - takes a java.lang.Runnable object, and executes it asynchronously
  - submit(Runnable) - takes a Runnable implementation, but returns a Future object. (returns null)
  - submit(Callable) - takes a Java Callable instead of a Runnable
  - invokeAny(...) - takes a collection of Callable objects. Just one of the ones that finish. (cancelled)
  - invokeAll(...) - invokes all of the Callable objects you pass to it in the collection
- The Runnable interface represents a task that can be executed concurrently by a thread or an ExecutorService, but the Callable can only be executed by an ExecutorService.
- If you need to submit a task to a Java ExecutorService and you need a result from the task, then you need to make your task implement the Callable interface.
  - Actions in a thread prior to the submission of a Runnable or Callable task to an ExecutorService happen-before any actions taken by that task, which in turn happen-before the result is retrieved via Future.get().
- To terminate the threads inside the ExecutorService you call its shutdown() method.

fork/join Framework (1/2)

- The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. (Java 7)
- It is designed for work that can be broken into smaller pieces recursively.
- The center of the fork/join framework is the ***ForkJoinPool*** class, an extension of the AbstractExecutorService class.
- ForkJoinPool can execute ***ForkJoinTask*** processes.

ForkJoinTask

```
if (my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```



Fork recursively a task in smaller subtask until each subtask is small enough
fork
fork
fork
Sequential evaluation
Sequential evaluation
Sequential evaluation
Sequential evaluation
Evaluate all subtasks in parallel
join
join
Recombine the partial results
join

fork/join Framework (2/2)

- The first step for using the fork/join framework is to write your code that should look similar to the following pseudocode:
- Wrap this code in a ForkJoinTask subclass, typically using one of its more specialized types, either RecursiveTask (which can return a result) or RecursiveAction.
- After your ForkJoinTask subclass is ready, create the object that represents all the work to be done and pass it to the invoke() method of a ForkJoinPool instance.

```
public class ForkJoinSumCalculator extends RecursiveTask<Long> {
    private final long[] numbers;
    private final int start;  private final int end;
    public static final long THRESHOLD = 1000;
    public ForkJoinSumCalculator(long[] numbers) {
        this(numbers, 0, numbers.length);
    }
    private ForkJoinSumCalculator(long[] numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;  this.end = end;
    }
    @Override
    protected Long compute() {
        int length = end - start;
        if (length <= THRESHOLD) {return computeSequentially();}
        ForkJoinSumCalculator leftTask =
                new ForkJoinSumCalculator(numbers, start, start + length/2);
        leftTask.fork();  // async. executes thread of ForkJoinPool
        ForkJoinSumCalculator rightTask =
                new ForkJoinSumCalculator(numbers, start + length/2, end);
        Long rightResult = rightTask.compute(); // sync executes
        Long leftResult = leftTask.join();  //reads the result (waiting)
        return leftResult + rightResult;
}}
```

```
private long computeSequentially() {
  long sum = 0;
  for (int i = start; i < end; i++)
    sum += numbers[i];
  return sum;}}
```

```
public class ForkJoinPoolTest {
  public static void main(String[] args) {
    long[] numbers =
       LongStream.rangeClosed(1, 100_000).toArray();
    ForkJoinTask<Long> task =
       new ForkJoinSumCalculator(numbers);
    System.out.println((new ForkJoinPool()).invoke(task));
  }}
```

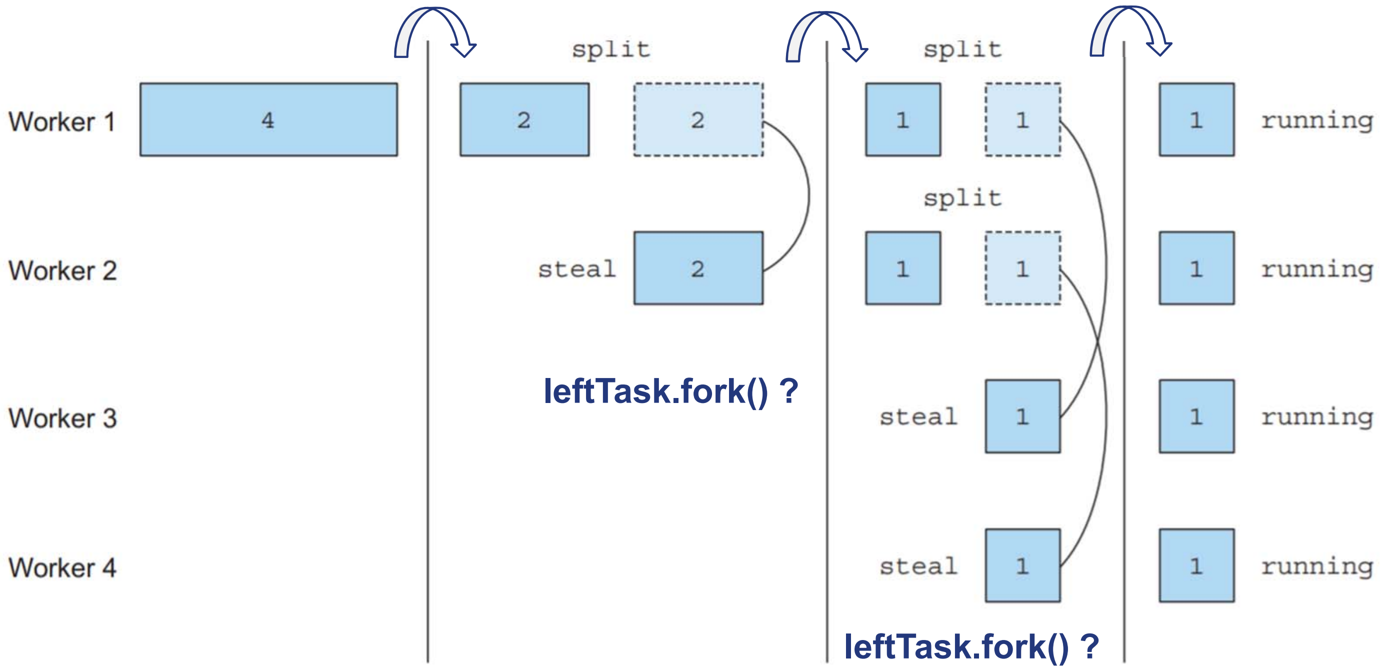
sequential code should use

Best Practices for using the fork/join Framework

- Even though the fork/join framework is relatively **easy to use**, unfortunately it's also **easy to misuse**.
- Invoking the **join()** method on a task blocks the caller until the result produced by that task is ready.
  - For this reason, it's necessary to call it after the computation of both subtasks has been started.
- The **invoke()** method of a ***ForkJoinPool*** shouldn't be used from within a ***RecursiveTask***. Instead, you should always call the methods **compute()** or **fork()** directly.
  - Only sequential code should use **invoke()** to begin parallel computation.
- Calling the **fork()** method on a subtask is the way to schedule it on the ***ForkJoinPool***.
  - Doing this allows you to reuse the same thread for one of the two subtasks and avoid the overhead caused by the unnecessary allocation of a further task on the pool.
- Debugging a parallel computation using the fork/join framework can be difficult.
- You should never take for granted that a computation using the fork/join framework on a multicore processor is faster than the sequential counterpart.

Work-Stealing Algorithm & *Spliterator* class

- fork/join framework works around this problem with a technique called **work stealing**.
  - For the reasons, one thread might complete all the tasks assigned to it much faster than the others, which means its queue will become empty while the other threads are still pretty busy.
  - Instead of becoming idle, **the thread randomly chooses a queue of a different thread and "steals" a task**, taking it from the tail of the queue.



- You must choose the criteria used to decide if a given subtask should be further split or is small enough to be evaluated sequentially.
  - **The criteria is an arbitrary choice**, but in most cases it's difficult to find a good heuristic, other than trying to optimize it by making several attempts with different inputs.
  - Forking a quite large number of **fine-grained tasks** is in general a winning choice, but the time taken by each subtask can dramatically vary either due to the use of an inefficient partition strategy or because of unpredictable causes like slow access to the disk.
- There must be an automatic mechanism splitting the stream for you (***Spliterator*** class)

Atomic Variables

- The *java.util.concurrent.atomic* **package** defines classes that support atomic operations on single variables.
- All classes have get and set methods that work like reads and writes on ***volatile*** variables. That is, a set has a **happens-before relationship** with any subsequent get on the same variable.
- One way to make Counter safe from thread interference is to make its methods ***synchronized***, but for a more complicated class, we might want to avoid the **liveness** impact of unnecessary synchronization.
- Replacing the ***int*** field with an ***AtomicInteger*** allows us to prevent thread interference without resorting to synchronization.

```
class Counter {
  private int c = 0;
  public void increment() { c++; }
  public void decrement() { c--; }
  public int value() {return c; }
}
```

Synchronization Errors

```
class SynchronizedCounter {
  private int c = 0;
  public synchronized void increment() {c++;}
  public synchronized void decrement() {c--;}
  public synchronized int value() {return c;}
}
```

Liveness Issues

```
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() { c.incrementAndGet();}
    public void decrement() {c.decrementAndGet();}
    public int value() {return c.get();}
}
```

Thread Safety

- In multithreaded environments, we need to write implementations in a **thread-safe** way.
- This means that different threads can access the same resources without exposing **erroneous behavior** or producing **unpredictable results**.

<Approaches>

- **Stateless Implementations** - a method neither relies on external state nor maintains state at all (the simplest way to achieve thread-safety)
- **Immutable Implementations** - a class instance is immutable when its internal state can't be modified after it has been constructed.
  - The easiest way to create an immutable class in Java is by declaring **all the fields private and final and not providing setters.**
- **Synchronized Collections**
  - We can use one of these synchronization wrappers to create a thread-safe collection.
  - Synchronized collections use **intrinsic locking** in each method.
- **Concurrent Collections**
  - Concurrent collections achieve **thread-safety** by dividing their data into segments.
  - Synchronized and concurrent collections only make the collection itself thread-safe and **not the contents.**

https://www.baeldung.com/java-thread-safety

Immutable/Synchronized/Concurrent Collections

```
// Collections.unmodifiableList()
List<String> list = new ArrayList<>(Arrays.asList("one", "two", "three"));
List<String> unmodifiableList = Collections.unmodifiableList(list);
//unmodifiableList.add("four");  //UnsupportedOperationException
// Java 9
final List<String> unmodifiableList2 = List.of(list.toArray(new String[]{}));
//unmodifiableList2.add("four");  //UnsupportedOperationException
// Google Core Libraries for Java (Guava)
List<String> unmodifiableList3 = ImmutableList.copyOf(list);
//ImmutableList<String> unmodifiableList3 = ImmutableList.<String>builder().addAll(list).build();
//unmodifiableList3.add("four");  //UnsupportedOperationException
// CopyOnWriteArrayList
CopyOnWriteArrayList<Integer> numbers = new CopyOnWriteArrayList<>();
Iterator<Integer> iterator = numbers.iterator();  // get an immutable snapshot
numbers.add(10); //add an element after Arrays.copyOf()
// Collections.synchronizedList()
List<String> syncList = Collections.synchronizedList(list);
syncList.add("four");
// ConcurrentHashMap
Map<String,String> concurrentMap = new ConcurrentHashMap<>();
concurrentMap.put("1", "one"); concurrentMap.put("2", "two"); concurrentMap.put("3", "three");
```

"Synchronized collections achieve thread-safety through intrinsic locking, and the entire collections are locked."

"Concurrent collections (e.g. ConcurrentHashMap), achieve thread-safety by dividing their data into segments."

https://www.baeldung.com/java-immutable-list
https://www.baeldung.com/java-synchronized-collections

Parallel streams (1/2)

- It's possible to turn a collection into a parallel stream by invoking the method **parallelStream()** on the collection source.
- A **parallel stream** is a stream that splits its elements into multiple chunks, processing each chunk with a different thread.
- Thus, you can **automatically** partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy.

```
public class ParallelStreamTest {
    public static void main(String[] args) {
        System.out.println(iterativeSum(100_000));
        System.out.println(sequentialSum(100_000));
        System.out.println(parallelSum(100_000));
    }
    public static long iterativeSum(long n) {
        long result = 0;
        for (long i = 1L; i <= n; i++) {
            result += i;
        }
        return result;
    }
```

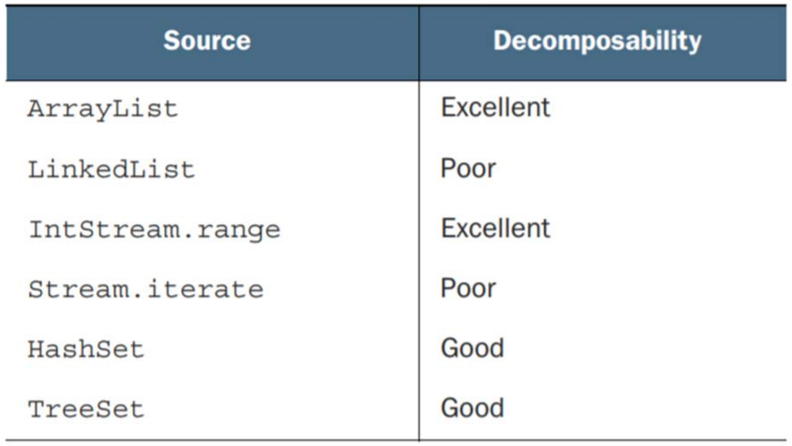
```
  public static long sequentialSum(long n) {
        return Stream.iterate(1L, i -> i + 1)
                   .limit(n).reduce(0L, Long::sum);
      //return LongStream.rangeClosed(1, n)
      //         .reduce(0L, Long::sum);
    }
    public static long parallelSum(long n) {
        return Stream.iterate(1L, i -> i + 1)
                   .limit(n).parallel()
                   .reduce(0L, Long::sum);
      //return LongStream.rangeClosed(1, N)
      //         .parallel() .reduce(0L, Long::sum);
    }
}
```

Parallel streams (2/2)

- The main cause of **errors** generated by misuse of parallel streams is the use of algorithms that mutate **some shared state**.

- When deciding whether it makes sense to use a parallel stream in a certain situation:
  - **If in doubt, measure.** Turning a sequential stream into a parallel one is trivial but not always the right thing to do. (JMH)
  - **Watch out for boxing.** Automatic boxing and unboxing operations can dramatically hurt performance.
  - **Some operations** naturally perform worse on a parallel stream than on a sequential stream. (rely on the **order** of the elements)
  - Take into account how well the **data structure** underlying the stream decomposes.

```
public class ParallelStreamTest {
    public static void main(String[] args) {
        System.out.println(sideEffectSum(100_000));
    }
    public static long sideEffectSum(long n) {
        Accumulator accumulator = new Accumulator();
        LongStream.rangeClosed(1, n)
                        .parallel().forEach(accumulator::add);
        return accumulator.total;
    }
    private static class Accumulator {
        public long total = 0;
        public void add(long value) { total += value; }
    }
}
```



Source	Decomposability
`ArrayList`	Excellent
`LinkedList`	Poor
`IntStream.range`	Excellent
`Stream.iterate`	Poor
`HashSet`	Good
`TreeSet`	Good

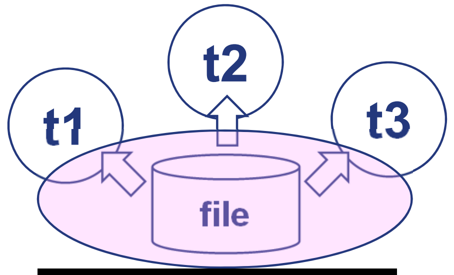
Concurrent IO (1/2)

- We assume that the standard implementation of IO is thread-safe.
  - Multiple threads can safely read from or write to the same IO stream, but the Java class library doccumentation is not explicit on the issue.

- For instance, a cryptographic hash function (CHF) is a mathematical algorithm that maps data of an arbitrary size to a bit array of a fixed size
  - "hash value", "hash", or "**message digest**"

```
private static class SHA256DigestR implements Runnable {
  private String filename;
  private Callback<String> callback;
  public SHA256DigestR(String filename, Callback<String> callback) {
    this.filename = filename; this.callback = callback; }
  public void run() {
    final MessageDigest sha256 = getSHA256Digest();
    try (FileInputStream fis=new FileInputStream(filename);
      DigestInputStream dis=new DigestInputStream(fis,sha256);
      BufferedInputStream bis=new BufferedInputStream(dis);){
      byte[] buffer = new byte[1024];
      while (bis.read(buffer)!=-1){;}
      BigInteger biginteger = new BigInteger(1, sha256.digest());
      String hash = biginteger.toString(16);
      String zeros = String.format("%064d", 0); //32 bytes
      hash = zeros.substring(hash.length()) + hash;
      System.out.print(hash);
      //callback.call(hash);
    }
   // catch statements
}
```

```
0df8c5a5a5381fe1f1202fc037165c92c2cf65cdfa29a4a63e26d1a56173214
```



Thread-Safety ?

Concurrent IO (2/2)

```
private static class SHA256DigestC implements Callable<String> {
  private String filename;
  public SHA256DigestC(String filename) { this.filename = filename; }
  public String call () {
    final MessageDigest sha256 = getSHA256Digest();
    try (FileInputStream fis=new FileInputStream(filename);
      DigestInputStream dis=new DigestInputStream(fis,sha256);
      BufferedInputStream bis=new BufferedInputStream(dis);){
      byte[] buffer = new byte[1024];
      while (bis.read(buffer)!=-1){;}
      BigInteger biginteger = new BigInteger(1, sha256.digest());
      String hash = biginteger.toString(16);
      String zeros = String.format("%064d", 0); //32 bytes
      hash = zeros.substring(hash.length()) + hash;
    }
   // catch statements
   return hash; }
}
```

```
private static class LogFile {
  private Writer out;
  public LogFile(String filename) throws IOException {
      FileWriter fw = new FileWriter(filename);
      this.out = new BufferedWriter(fw);
  }
  public void write(String log) throws IOException {
      Date d = new Date(); out.write(d.toString());
      out.write("₩t"); out.write(log); out.write("₩t");
      out.write("₩r₩n");
  }
  public void close() throws IOException {
      out.flush(); out.close();
  }
}
```

Synchronization

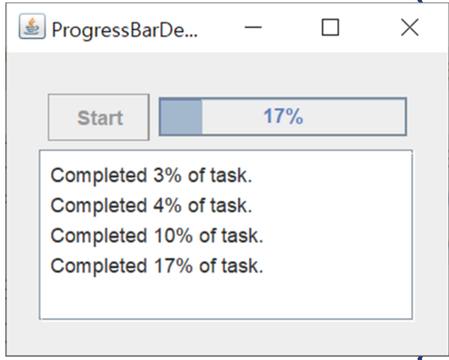
```
0df8c5a5a5381fe1f1202fc037165c92c2cf65cdfa29a4a63e26d1a56173214
true
0df8c5a5a5381fe1f1202fc037165c92c2cf65cdfa29a4a63e26d1a56173214
0df8c5a5a5381fe1f1202fc037165c92c2cf65cdfa29a4a63e26d1a56173214
0df8c5a5a5381fe1f1202fc037165c92c2cf65cdfa29a4a63e26d1a56173214
0df8c5a5a5381fe1f1202fc037165c92c2cf65cdfa29a4a63e26d1a56173214
```

```
public static void main(String[] args) throws
            ExecutionException, InterruptedException {
  ExecutorService pool = Executors.newFixedThreadPool(10);
  SHA256DigestR digestRunnable =
            new SHA256DigestR("build.gradle", s-> System.out.println(s));
  pool.submit(digestRunnable); // 5 times
  SHA256DigestC digestCallable = new SHA256DigestC ("build.gradle");
  Future<String> digest = pool.submit(digestCallable);
  System.out.println(CHECKSUM.equals(digest.get()));
  pool.shutdown(); }
```

SwingWorker (1/2)

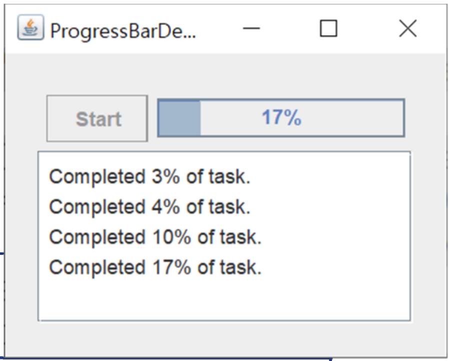
- An ***SwingWoker*** class to perform lengthy GUI-interaction tasks in a background thread.
- When writing a multi-threaded application using Swing, there are two constraints to keep in mind:
  - **Time-consuming tasks** should not be run on the Event Dispatch Thread. Otherwise the application becomes unresponsive.
  - Swing components should be accessed on the Event Dispatch Thread only.
- There are three threads involved in the life cycle of a SwingWorker :
  - **Current thread**: The execute() method is called on this thread. It schedules ***SwingWorker*** for the execution on a worker thread and returns immediately.
  - **Worker thread**: The doInBackground() method is called on this thread. By default there are two bound properties available: **state** and **progress**.
  - **Event Dispatch Thread**: All Swing related activities occur on this thread. ***SwingWorker*** invokes the process and **done()** methods and notifies any **PropertyChangeListeners** on this thread.

```
class Task extends SwingWorker<Void, Void> {
  public Void doInBackground() {  // executed in worker thread
    Random random = new Random();
    int progress = 0;
    setProgress(0);  // Initialize progress property.
    while (progress < 100) {
      // Sleep for up to one second.
      try {
        Thread.sleep(random.nextInt(1000));
       } catch (InterruptedException e) { }
       // Make random progress
       progress += random.nextInt(10);
       setProgress(Math.min(progress, 100));
    }
    return null;
  }
  public void done() {  // executed in Event Dispatch Thread
    Toolkit.getDefaultToolkit().beep();
    startButton.setEnabled(true);
    setCursor(null); // turn off the wait cursor
    taskOutput.append("Done!₩n");
  }
}
```



```
Completed 3% of task.
Completed 4% of task.
Completed 10% of task.
Completed 17% of task.
```

SwingWorker (2/2)



```
Completed 3% of task.
Completed 4% of task.
Completed 10% of task.
Completed 17% of task.
```

```
public class **ProgressBarView** extends JPanel
      implements ActionListener, PropertyChangeListener {
  private JProgressBar progressBar;
  private JButton startButton;
  private JTextArea taskOutput;
  private Task task;
  public ProgressBarView() {
    super(new BorderLayout());
    startButton = new JButton("Start");
    startButton.setActionCommand("start");
    startButton.addActionListener(this);
    progressBar = new JProgressBar(0, 100);
    progressBar.setValue(0);
    progressBar.setStringPainted(true);
    taskOutput = new JTextArea(5, 20);
    taskOutput.setMargin(new Insets(5,5,5,5));
    taskOutput.setEditable(false);
    JPanel panel = new JPanel();
    panel.add(startButton); panel.add(progressBar);
    add(panel, BorderLayout.PAGE_START);
    add(new JScrollPane(taskOutput),BorderLayout.CENTER);
    setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
  }
```

```
public void actionPerformed(ActionEvent evt) {
  startButton.setEnabled(false);
  setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
  task = new Task();
  task.addPropertyChangeListener(this); task.execute();
}
public void propertyChange(PropertyChangeEvent evt) {
  if ("progress" == evt.getPropertyName()) {
    int progress = (Integer) evt.getNewValue();
    progressBar.setValue(progress);
    taskOutput.append(
      String.format("Completed %d%% of task.₩n",
                    task.getProgress()));
  }}
```

```
public class ProgressBarTest {
  ...
  private static void createAndShowGUI() {
  JFrame frame = new JFrame("ProgressBarDemo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JComponent newContentPane = new ProgressBarView();
    newContentPane.setOpaque(true);
    frame.setContentPane(newContentPane);
    frame.pack();
    frame.setVisible(true);
  }}
```

Multi-threaded HTTPServer

- Management of threads can be done external to this object by providing a Executor object. If none is provided a default implementation is used.
  - setExecutor() - sets this server's Executor object.
  - start() - Starts this server in a new background thread.

```
class ServerImpl implements TimeSource {
  private Object lolock = new Object();
  private volatile boolean finished = false;
  private volatile boolean terminating = false;
public void start () {
  dispatcherThread = new Thread(null, dispatcher,
          "HTTP-Dispatcher", 0, false);
  started = true;
  dispatcherThread.start();
}
public final boolean isFinishing() { return finished; }
public void stop (int delay) {
  terminating = true;
  finished = true;
  for (HttpConnection c : allConnections)  c.close();
  dispatcherThread.join();
}
```

```
class Dispatcher implements Runnable {
  public void run() {
    while (!finished) {
      List<Event> list = null;
      synchronized (lolock) {
        if (events.size() > 0) {
          list = events;
          events = new LinkedList<Event>();
        }
      }
      if (list != null) {
        for (Event r: list) {
          handleEvent (r);
        }
      }
```

```
private static class DefaultExecutor implements Executor {
  public void execute (Runnable task) { task.run(); }
}
```

```
public class HttpServerImpl
        extends HttpServer {
  ServerImpl server;
  public void start () {
    server.start();
  }
  public void stop (int delay) {
    server.stop (delay);
  }}
```

Q&A