Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: Structuring Applications

("Design Patterns for Parallel Computation")

Christian Kästner **Bogdan Vasilescu**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

Designing Thread-Safe Objects

- Identify variables that represent the object's state
  - may be distributed across multiple objects
- Identify invariants that constraint the state variables
  - important to understand invariants to ensure atomicity of operations
- Establish a policy for managing concurrent access to state

Summary of policies:

- **Thread-confined.** A thread-confined object is owned exclusively by and confined to one thread, and can be modified by its owning thread.
- **Shared read-only.** A shared read-only object can be accessed concurrently by multiple threads without additional synchronization, but cannot be modified by any thread. Shared read-only objects include immutable and effectively immutable objects.
- **Shared thread-safe.** A thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization.
- **Guarded.** A guarded object can be accessed only with a specific lock held. Guarded objects include those that are encapsulated within other thread-safe objects and published objects that are known to be guarded by a specific lock.

Tradeoffs

- Strategies:
  - Don't share the state variable across threads;
  - Make the state variable immutable; or
  - Use synchronization whenever accessing the state variable.
    - Thread-safe vs guarded
    - Coarse-grained vs fine-grained synchronization
- When to choose which strategy?
  - Avoid synchronization if possible
  - Choose simplicity over performance where possible

Documentation

- Document a class's thread safety guarantees for its clients
- Document its synchronization policy for its maintainers.
- @ThreadSafe, @GuardedBy annotations not standard but useful

Intro to Java

Git, CI

UML

Static Analysis

Performance

GUIs

GUIs

More Git

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Design Patterns, Unit Testing	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	Distributed Systems in a Nutshell
	Design for Reuse at Scale: Frameworks and APIs	

REUSE RATHER THAN BUILD: KNOW THE LIBRARIES

Synchronized Collections

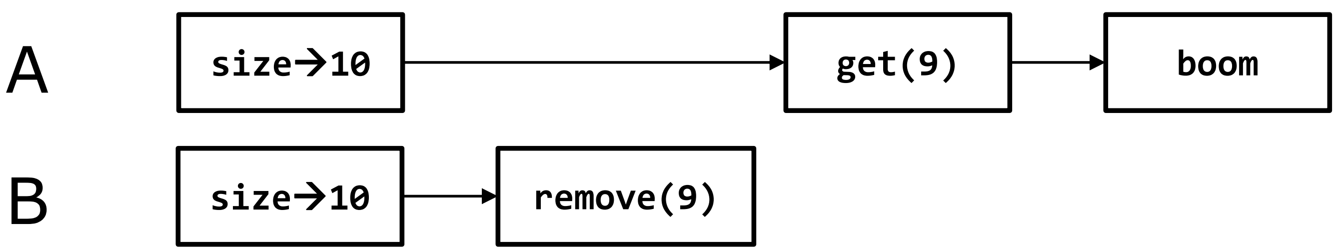
- Are thread safe:
  - Vector
  - Hashtable
  - Collections.synchronizedXXX
- But still require client-side locking to guard compound actions:
  - Iteration: repeatedly fetch elements until collection is exhausted
  - Navigation: find next element after this one according to some order
  - Conditional ops (put-if-absent)

Example

- Both methods are thread safe

```
public static Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}

public static void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```

- Unlucky interleaving that throws `ArrayIndexOutOfBoundsException`



Solution: Compound actions on Vector using client-side locking

- Synchronized collections guard methods with the lock on the collection object itself

```
public static Object getLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

Another Example

- The size of the list might change between a call to `size` and a corresponding call to `get`
  – Will throw `ArrayIndexOutOfBoundsException`

```
for (int i = 0; i < vector.size(); i++)
    doSomething(vector.get(i));
```

- Note: Vector still thread safe:
  – State is valid
  – Exception conforms with specification

Solution: Client-side locking

- Hold the Vector lock for the duration of iteration:
  - No other threads can modify (+)
  - No other threads can access (-)

```
synchronized (vector) {
    for (int i = 0; i < vector.size(); i++)
        doSomething(vector.get(i));
}
```

Iterators and `ConcurrentModificationException`

- Iterators returned by the synchronized collections are not designed to deal with concurrent modification → *fail-fast*
- Implementation:
  - Each collection has a modification count
  - If it changes, `hasNext` or `next` throws `ConcurrentModificationException`
- Prevent by locking the collection:
  - Other threads that need to access the collection will block until iteration is complete → starvation
  - Risk factor for deadlock
  - Hurts scalability (remember lock contention in reading)

Alternative to locking the collection during iteration?

Yet Another Example: Is this safe?

```
public class HiddenIterator {
    @GuardedBy("this")
    private final Set<Integer> set = new HashSet<Integer>();

    public synchronized void add(Integer i) { set.add(i); }

    public synchronized void remove(Integer i) { set.remove(i); }

    public void addTenThings() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            add(r.nextInt());
        System.out.println("DEBUG: added ten elements to " + set);
    }
}
```

Hidden Iterator

- Locking can prevent `ConcurrentModificationException`
- But must remember to lock everywhere a shared collection might be iterated

```
public class HiddenIterator {
    @GuardedBy("this")
    private final Set<Integer> set = new HashSet<Integer>();

    public synchronized void add(Integer i) { set.add(i); }

    public synchronized void remove(Integer i) { set.remove(i); }

    public void addTenThings() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            add(r.nextInt());
        System.out.println("DEBUG: added ten elements to " + set);
    }
}
```

Hidden Iterator

```
System.out.println("DEBUG: added ten elements to " + set);
```

- String concatenation
  - → `StringBuilder.append(Object)`
  - → `Set.toString()`
  - → Iterates the collection; calls `toString()` on each element
  - → `addTenThings()` may throw `ConcurrentModificationException`

- **Lesson: Just as encapsulating an object's state makes it easier to preserve its invariants, encapsulating its synchronization makes it easier to enforce its synchronization policy**

Concurrent Collections

- Synchronized collections: thread safety by serializing all access to state
  - Cost: poor concurrency
- Concurrent collections are designed for concurrent access from multiple threads
  - Dramatic scalability improvements

Unsynchronized	Concurrent
`HashMap`	`ConcurrentHashMap`
`HashSet`	`ConcurrentHashSet`
`TreeMap`	`ConcurrentSkipListMap`
`TreeSet`	`ConcurrentSkipListSet`

ConcurrentHashMap

- `HashMap.get`: traversing a hash bucket to find a specific object → calling `equals` on a number of candidate objects
  - Can take a long time if hash function is poor and elements are unevenly distributed
- `ConcurrentHashMap` uses ***lock striping*** (recall reading)
  - Arbitrarily many reading threads can access concurrently
  - Readers can access map concurrently with writers
  - Limited number of writers can modify concurrently
- Tradeoffs:
  - `size` only an estimate
  - Can't lock for exclusive access

You can't exclude concurrent activity from a concurrent collection

- This works for synchronized collections…

```
Map<String, String> syncMap =
    Collections.synchronizedMap(new HashMap<>());
synchronized(syncMap) {
    if (!syncMap.containsKey("foo"))
        syncMap.put("foo", "bar");
}
```

- But ***not*** for concurrent collections
  - They do their own internal synchronization
  - **Never synchronize on a concurrent collection!**

Concurrent collections have prepackaged read-modify-write methods

- `V` **`putIfAbsent`**`(K key, V value)`
- `boolean` **`remove`**`,(Object key, Object value)`
- `V` **`replace`**`(K key, V value)`
- `boolean` **`replace`**`(K key, V oldValue, V newValue)`
- `V` **`compute`**`(K key, BiFunction<...> remappingFn);`
- `V` **`computeIfAbsent`**`(K key, Function<...> mappingFn)`
- `V` **`computeIfPresent`**`(K key, BiFunction<...> remapFn)`
- `V` **`merge`**`(K key, V value, BiFunction<...> remapFn)`

THE PRODUCER-CONSUMER DESIGN PATTERN

Pattern Idea

- Decouple dependency of concurrent producer and consumer of some data
- Effects:
  - Removes code dependencies between producers and consumers
  - Decouples activities that may produce or consume data at different rates

Blocking Queues

- Provide blocking `put` and `take` methods
  – If queue full, put blocks until space becomes available
  – If queue empty, take blocks until element is available

- Can also be bounded: throttle activities that threaten to produce more work than can be handled

Example: Desktop Search (1)

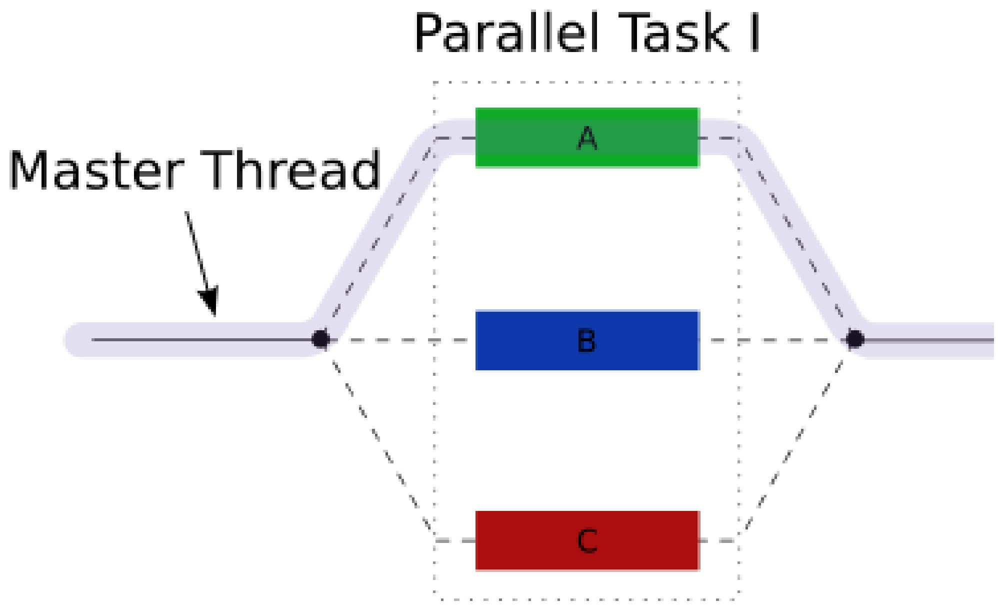
```
public class FileCrawler implements Runnable {
    private final BlockingQueue<File> fileQueue;
    private final FileFilter fileFilter;
    private final File root;

    ...
    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void crawl(File root) throws InterruptedException {
        File[] entries = root.listFiles(fileFilter);
        if (entries != null) {
            for (File entry : entries)
                if (entry.isDirectory())
                    crawl(entry);
                else if (!alreadyIndexed(entry))
                    fileQueue.put(entry);
        }
    }
}
```

Example: Desktop Search (2)

```
public class Indexer implements Runnable {
    private final BlockingQueue<File> queue;

    public Indexer(BlockingQueue<File> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true)
                indexFile(queue.take());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public void indexFile(File file) {
        // Index the file...
    };
}
```

THE FORK-JOIN DESIGN PATTERN

Pattern Idea



- Pseudocode (parallel version of the divide and conquer paradigm)

```
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
```


Image from: Wikipedia

THE MEMBRANE DESIGN PATTERN

Pattern Idea

Multiple rounds of fork-join that need to wait for previous round to complete.

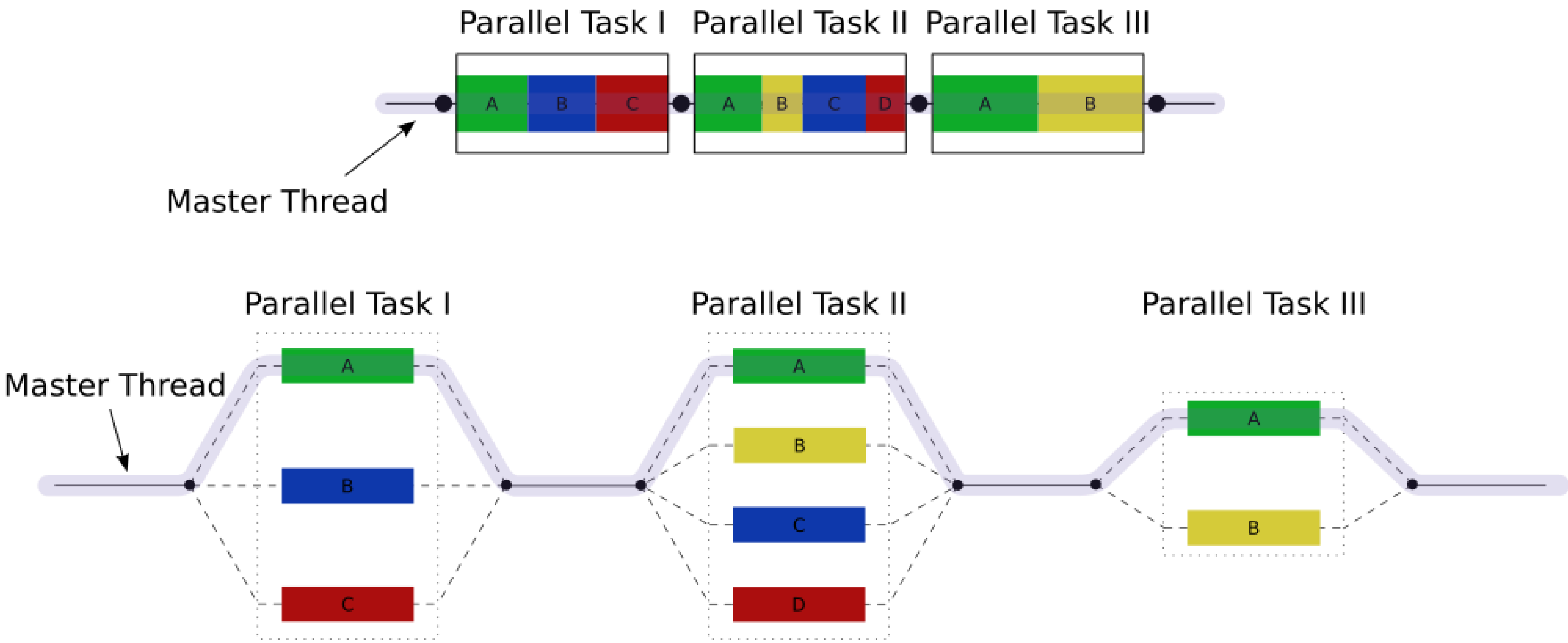


Image from: Wikipedia

TASKS AND THREADS

Executing tasks in threads

- Organize program around task execution
  - Identify *task boundaries*; ideally, tasks are *independent*
- Typical requirements for server applications:
  - Good throughput
  - Good responsiveness
  - Graceful degradation
- Choosing good task boundaries + a sensible *task execution policy* can help
  - Natural choice of task boundary: individual client requests

Executing tasks sequentially

```
public class SingleThreadWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }

    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```

- Can only handle one request at a time
- Main thread alternates between accepting connections and processing the requests

Explicitly creating threads for tasks

```
public class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() { handleRequest(connection); }
            };
            new Thread(task).start();
        }
    }
    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```

- Main thread still alternates between accepting connections and dispatching requests
- But each request is processed in a separate thread

Still, what's wrong?

```
public class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() { handleRequest(connection); }
            };
            new Thread(task).start();
        }
    }
    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```

Disadvantages of unbounded thread creation

- Thread lifecycle overhead
  - Thread creation and teardown are not free
- Resource consumption
  - When there are more runnable threads than available processors, threads sit idle
  - Many idle threads can tie up a lot of memory
- Stability
  - There is a limit to how many threads can be created (varies by platform)
    - `OutOfMemory` error

THE THREAD POOL DESIGN PATTERN

Pattern Idea

- A thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program
  - Tightly bound to a *work queue*
- Advantages:
  - Reusing an existing thread instead of creating a new one
    - Amortizes thread creation/teardown over multiple requests
    - Thread creation latency does not delay task execution
  - Tune size of thread pool
    - Enough threads to keep processors busy while not having too many to run out of memory

EXECUTOR SERVICES

The Executor framework

- Recall: *bounded queues* prevent an overloaded application from running out of memory
- *Thread pools* offer the same benefit for thread management
  - Thread pool implementation part of the `Executor` framework in `java.util.concurrent`
  - Primary abstraction is `Executor`, not `Thread`

```
public interface Executor {
    void execute(Runnable command);
}
```

  - Using an `Executor` is usually the easiest way to implement a *producer-consumer* design

Executors – your one-stop shop for executor services

- `Executors.new`**`SingleThreadExecutor`**`()`
  – A single background thread
- `new`**`FixedThreadPool`**`(int nThreads)`
  – A fixed number of background threads
- `Executors.new`**`CachedThreadPool`**`()`
  – Grows in response to demand

Web server using Executor

```
public class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
            = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }

    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```

Easy to specify / change execution policy

- Thread-per-task server:

```
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    };
}
```

- Single thread server:

```
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    };
}
```

Execution policies

- Decoupling submission from execution
- Specify:
  - In what thread will tasks be executed?
  - In what order (FIFO, LIFO, …)?
  - How many tasks may execute concurrently?
  - How many tasks may be queued pending execution?
  - …
- Notice the strategy/template method pattern: general mechanism but highly customizable

Task granularity and structure

- Maximize parallelism
  - The smaller the task, the more opportunities for parallelism → better CPU utilization, load balancing, locality, scalability; greater throughput
- Minimize overhead
  - Intrinsically more costly to create and use task objects than stack-frames → coarse-grained tasks
- Minimize contention
  - Maintain as much independence as possible between tasks → ideally, no shared resources, global (static) variables, locks
  - Some synchronization is unavoidable in fork/join designs
- Maximize locality
  - When parallel tasks all access different parts of a data set (e.g., different regions of a matrix), use partitioning strategies that reduce the need to coordinate across

Finding exploitable parallelism

- `Executor` framework makes it easy to specify an execution policy if you can describe your task as a `Runnable`
  - A single client request is a natural task boundary in server applications
- Task boundaries are not always obvious

Example: HTML page renderer

```
void renderPage(CharSequence source) {
    renderText(source);
    List<ImageData> imageData = new ArrayList<ImageData>();
    for (ImageInfo imageInfo : scanForImageInfo(source))
        imageData.add(imageInfo.downloadImage());
    for (ImageData data : imageData)
        renderImage(data);
}
```

- Issues:
  - Underutilize CPU while waiting for I/O
  - User waits long time for page to finish loading

Result bearing tasks: Callable and Future

- `Runnable.run` cannot return value or throw checked exceptions (although it can have side effects)
- Many tasks are deferred computations (e.g., fetching a resource over a network) → `Callable` is a better abstraction
  - `Callable.call` will return a value and anticipates that it might throw an exception
- Runnable and Callable describe *abstact* computational tasks
- Future represents the *lifecycle* of a task (created, submitted, started, completed)

Callable and Future interfaces

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

public interface Future<V> {
   boolean cancel(boolean mayInterruptIfRunning);
   boolean isCancelled();
   boolean isDone();
   V get() throws InterruptedException,
      ExecutionException, CancellationException;
   V get(long timeout, TimeUnit unit)
      throws InterruptedException, ExecutionException,
      CancellationException, TimeoutException;
}
```

Creating a Future to describe a task

- `submit` a `Runnable` or `Callable` to an executor and get back a `Future` that can be used to retrieve the result or cancel the task
- Explicitly instantiate a FutureTask for a given `Runnable` or `Callable`

Example: Page renderer with Future

- Divide into two tasks
  - Render text (CPU-bound)
  - Download all images (I/O-bound)
- Steps:
  - Create a `Callable` for download subtask
  - Submit `Callable` to `ExecutorService`
  - `ExecutorService` returns `Future` describing the task's execution
  - When main task reaches point where it needs the images, it waits for the result by calling `Future.get`
    - If lucky, images already downloaded
    - If not, at least we got a head start

Future renderer (1)

```
public abstract class FutureRenderer {
    private final ExecutorService executor = ...;

    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        Callable<List<ImageData>> task =
                new Callable<List<ImageData>>() {
                    public List<ImageData> call() {
                        List<ImageData> result = new ArrayList<ImageData>();
                        for (ImageInfo imageInfo : imageInfos)
                            result.add(imageInfo.downloadImage());
                        return result;
                    }
                };

        Future<List<ImageData>> future = executor.submit(task);
        renderText(source);

                // Continued below
```

Future renderer (2)

```
public abstract class FutureRenderer {
            ...

      try {
         List<ImageData> imageData = future.get();
         for (ImageData data : imageData)
            renderImage(data);

      } catch (InterruptedException e) {
         // Re-assert the thread's interrupted status
         Thread.currentThread().interrupt();
         // We don't need the result, so cancel the task too
         future.cancel(true);
      } catch (ExecutionException e) {
         throw launderThrowable(e.getCause());
      }
   }
}
```

Future renderer analysis

- Allows text to be rendered concurrently with downloading data
- When all images are downloaded, they are rendered onto the page

- Can we do better?

Limitations of parallelizing heterogeneous tasks

- We tried to execute two different types of tasks in parallel—downloading images, rendering page
- Does not scale well
  – How can we use more than two threads?
  – Tasks may have disparate sizes
    - If rendering text is much faster than downloading images, performance is not much different from sequential version
- Lesson: real performance payoff of dividing a program's workload into tasks comes when there are many independent, *homogeneous* tasks that can be processed concurrently

Example: Page renderer with CompletionService

- `CompletionService` combines the functionality of an `Executor` and a `BlockingQueue`
  - `submit` `Callable` tasks to `CompletionService`
  - use queue-like methods `take` and `poll` to retrieve completed results, packaged as `Futures`, as they become available

Page renderer with CompletionService
Download images in parallel (1)

```
public abstract class Renderer {
  private final ExecutorService executor;

  ...

  void renderPage(CharSequence source) {
    final List<ImageInfo> info = scanForImageInfo(source);

    CompletionService<ImageData> completionService =
        new ExecutorCompletionService<ImageData>(executor);

    for (final ImageInfo imageInfo : info)
      completionService.submit(new Callable<ImageData>() {
        public ImageData call() {
          return imageInfo.downloadImage();
        }
      });

    renderText(source);
    // Continued below
```

Page renderer with CompletionService
Download images in parallel (2)

```
public abstract class Renderer {
    ...

    try {
      for (int t = 0, n = info.size(); t < n; t++) {
        Future<ImageData> f = completionService.take();
        ImageData imageData = f.get();
        renderImage(imageData);
      }

    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    } catch (ExecutionException e) {
      throw launderThrowable(e.getCause());
    }
  }
}
```

Summary

- Structuring applications around the execution of tasks can simplify development and facilitate concurrency
- The Executor framework permits you to decouple task submission from execution policy
- To maximize benefit of decomposing an application into tasks, identify sensible task boundaries
  - Not always obvious

Recommended Readings

- Goetz et al. Java Concurrency In Practice. Pearson Education, 2006, Chapters 5-6
- Lea, Douglas. Concurrent programming in Java: design principles and patterns. Addison-Wesley Professional, 2000, Chapter 4.4