Java Threads Demystified

An Introduction to Java Concurrency

Maurice Naftalin

Welcome

Practical Matters

– Timetable

– Questions

– Java Version

Course Outline

– Introduction to Java Concurrency

– Writing Thread-safe Code

– Tasks and Task Execution

– Concurrent Utilities

– Virtual Threads

– Course wrap-up and next steps

Introduction to Java Concurrency

Concurrency – What and Why

- Concurrency: multiple tasks running in overlapping time periods
- Processes
  - Run applications “simultaneously”, even on single-CPU machines
  - Word, Excel, IntelliJ, Chrome, etc
- Responsive User Interfaces
  - Don’t freeze for long-running tasks
- Design Convenience
- Resource Utilization
  - Processors are much, *much* faster than any other component
  - Multiple Processors

Implementation – Processes vs. Threads

– Processes

  – Insulated from each other; communicate via files, signals, sockets, pipes, etc

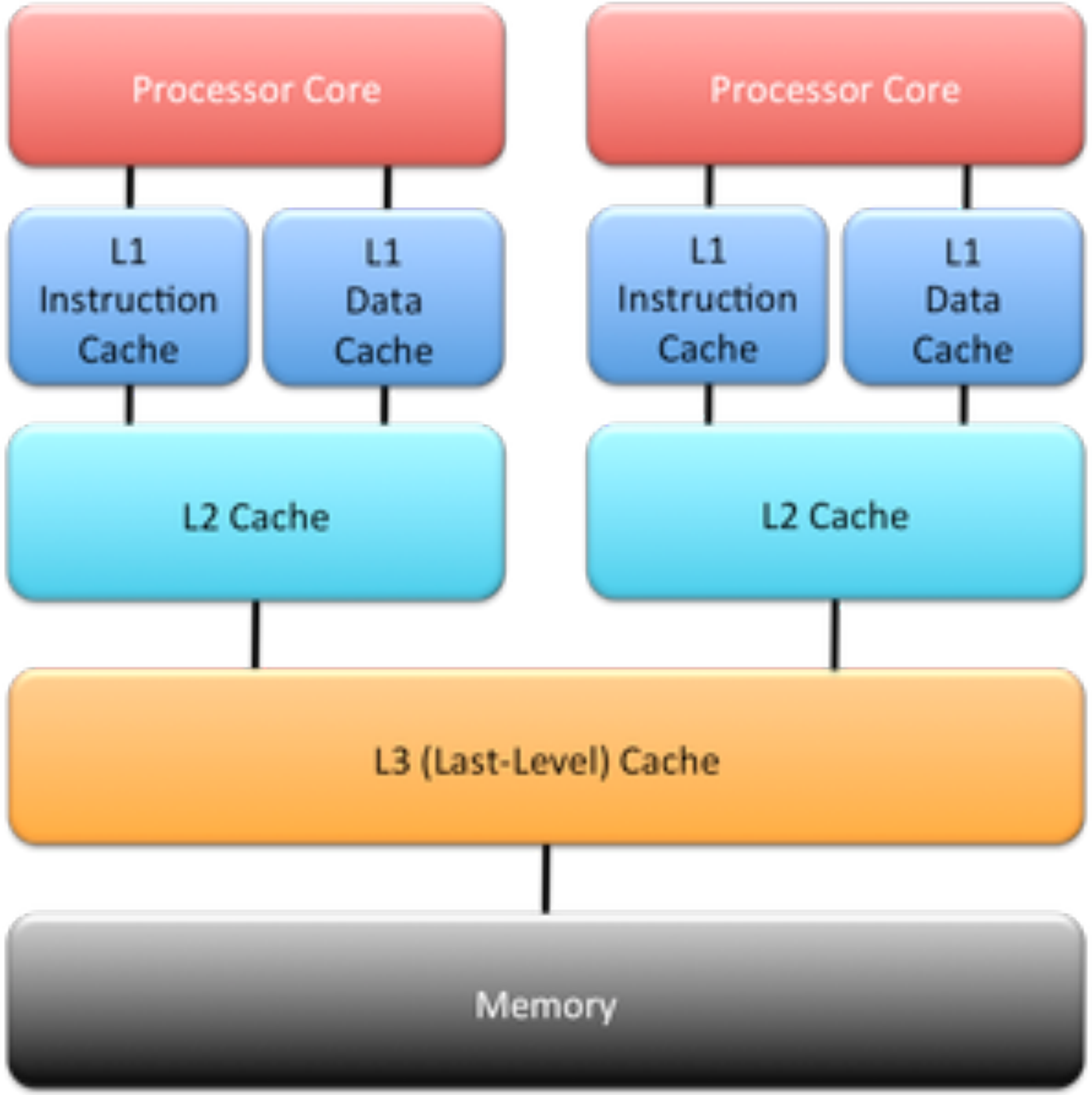
  – No shared memory (by default)

– Threads are *lightweight processes*

  – Run within a single process, share process resources

  – In particular, share memory

Concurrency – What and Why



Clock: ~0.25ns

L1: ~0.5ns

L2: ~10ns

Memory: ~100ns

Benefits of Threads

- Gain concurrency benefits within single process
  - Resource utilisation, responsive interfaces, design convenience
- Simplified handling of asynchronous events
- Divide work of a single process over multiple CPUs

Drawbacks of Threads

- Safety
  - Inadequately synchronized code can yield incorrect results
- Liveness
  - Acquiring locks in the wrong order can deadlock your program
- Performance
  - Overhead of thread creation and switching

Concurrency vs. Parallelism

- Concurrency
  - Multiple tasks running in overlapping time periods
- Parallelism
  - Concurrent tasks (typically the same task) running on different data at the *same* time
- These definitions aren't universally agreed!

Threads and Synchronisation

Java Threads

- Wrapper around native operating system threads
- Conceptually, like a computer
  - The computer's program is a `Runnable`:

```
Runnable r = new Runnable() {
    public void run() {
        // concurrent task
    }
}
```

or

```
Runnable r = (() -> { ... }
```

Java Threads

– The `Runnable` is supplied to the thread at construction time:

```
var t = new Thread(r);
```

– And the “computer” is then started:

```
t.start();
```

– A modern alternative:

```
var thread3 = Thread.ofPlatform().start(r);
```

– The thread dies when the `run` method exits

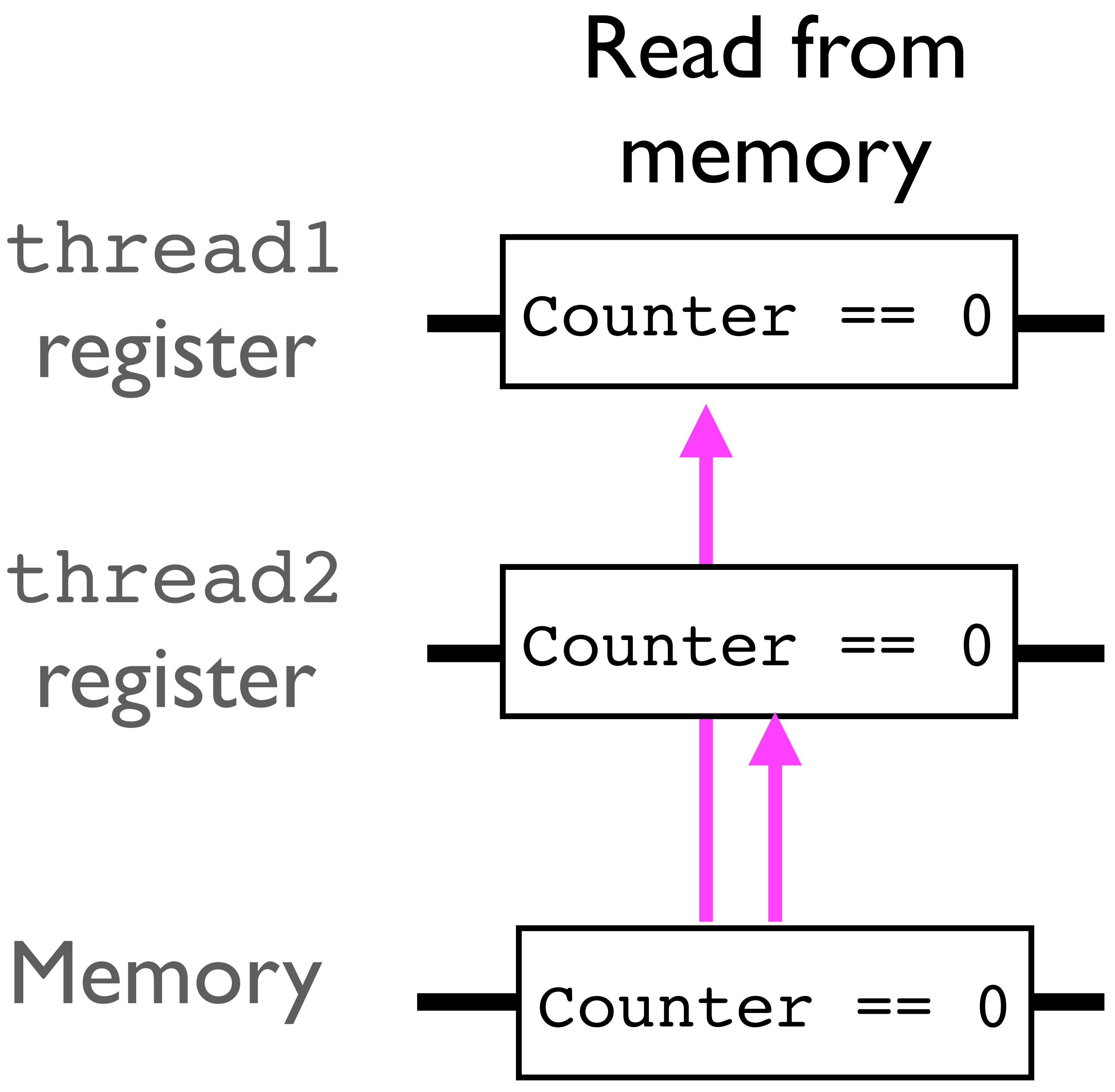
Pitfalls of Multithread Programming

- Multiple threads running on modern hardware can produce unexpected results:
  - caches may duplicate values
  - instructions may execute out of order
  - 64-bit numbers may be processed in two parts by different threads
- Probably most common are *race conditions*
- A race condition occurs when the behaviour of a system depends unpredictably on the timing of uncontrollable events
  - E.g relative CPU timing

Race Condition

```java
Thread thread1 = Thread.ofPlatform().start(this::incrementCounter);
Thread thread2 = Thread.ofPlatform().start(this::incrementCounter);

private void incrementCounter() {
     for (int i = 0; i < 100000; i++) { counter++;}
}
```



Java Memory Model

- The Java Memory Model tells you what you need to do to avoid these pitfalls

- Most important rule (simplified!) is about synchronization:

  - The JMM guarantees that everything done by a thread before it *releases* a monitor lock is visible to another thread after it has *acquired* that monitor lock

  - What does this mean?

Synchronization

– Every Java object is a *monitor*. A monitor contains a lock, which a thread can acquire by entering a critical section

```
synchronized(this) {
    counter++;
}
```

or

```
public synchronized void myMethod() {
    counter++;
}
```

– Code in a critical section can be executed only by one thread at a time

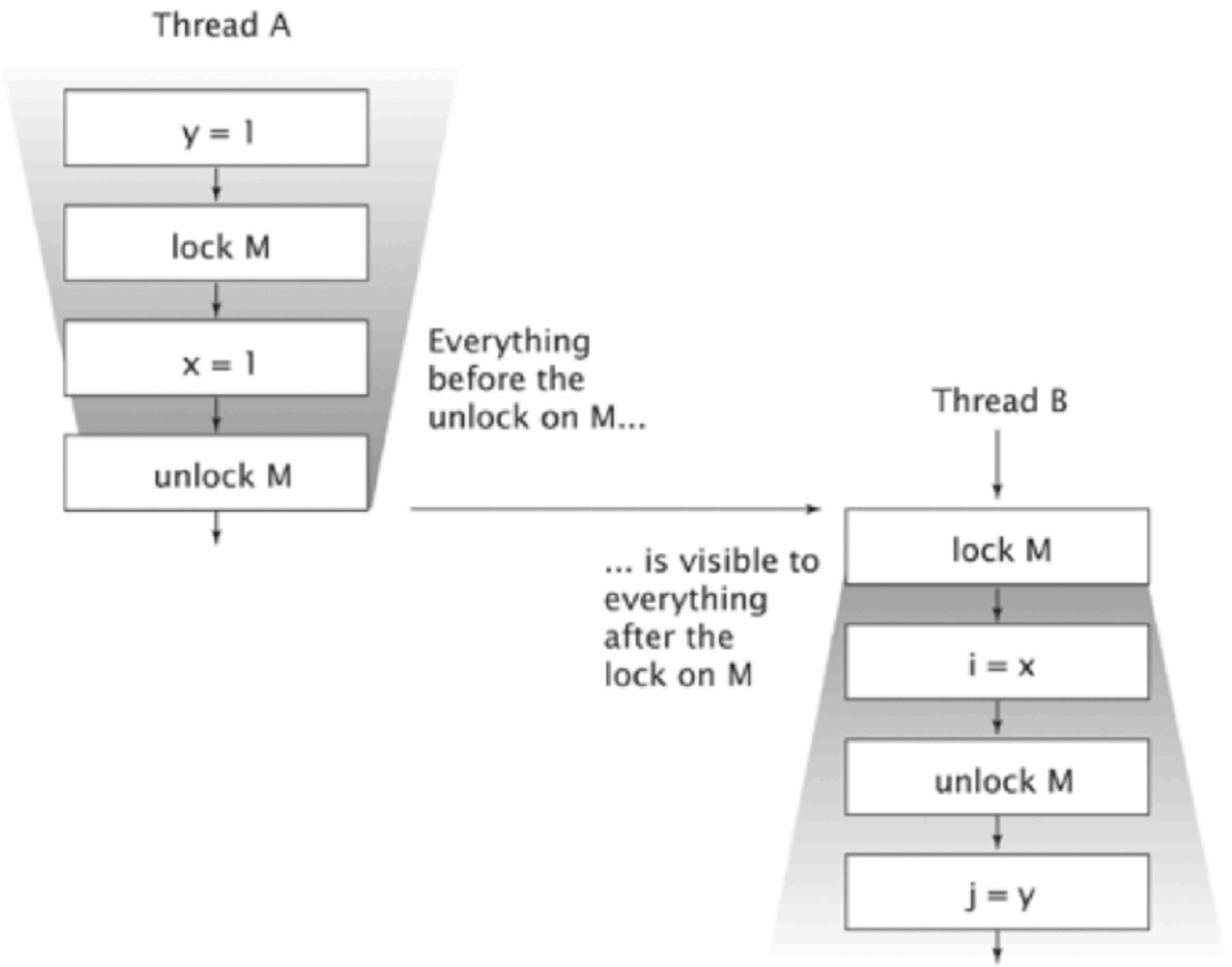
– If a thread is executing the code within a critical section, other threads trying to enter the section are blocked until it exits, releasing the lock

Synchronization

- So synchronisation has two important effects:
  - It *prevents threads from interfering with one another* in a critical section
  - It ensures that *changes made by one thread are visible to another* – provided that those changes are made by the first thread before it releases a monitor which the second thread subsequently acquires

Synchronization

– It ensures that *changes made by one thread are visible to another* – provided that those changes are made by the first thread before it releases a monitor which the second thread subsequently acquires:



From *Java Concurrency In Practice,* Goetz et. al., Addison-Wesley 2006

Synchronization

- The `synchronised` feature (“intrinsic locks”) was originally the only way to control thread interference
  - `volatile` does provides the visibility guarantee, but without mutual exclusion
- The platform now offers other ways to synchronise:
  - Explicit locks (`java.util.concurrent.locks.Lock`)
  - Atomic variables

Designing a Thread-Safe Class

Designing a Thread-Safe Class

- Identify the fields that form the object's state
- Identify the invariants that constrain the state fields
- Establish a policy for managing concurrent access to the state

```
@ThreadSafe
public final class Counter {
    @GuardedBy("this") private long value = 0;

    public synchronized long getValue() {
        return value;
    }

    public synchronized long increment() {
        if (value == Long.MAX_VALUE)
            throw new IllegalStateException("counter overflow");
        return ++value;
    }
}
```

Designing a Thread-Safe Class

- This class isn't thread safe
- How can we fix it?

```
public class NaiveVehicleTracker {
    private final Map<Vehicle, MutablePoint> vehicleLocations;
    public NaiveVehicleTracker(Map<Vehicle, MutablePoint> vehicleLocations) {
        this.vehicleLocations = vehicleLocations;
    }
    public Map<Vehicle, MutablePoint> getVehicleLocations() {
        return vehicleLocations;
    }
    public MutablePoint getVehicleLocation(Vehicle v) {
        return vehicleLocations.get(v);
    }
    public void setVehicleLocation(Vehicle v, int x, int y) {
        MutablePoint location = vehicleLocations.get(v);
        location.setX(x);
        location.setY(y);
    }
}
```

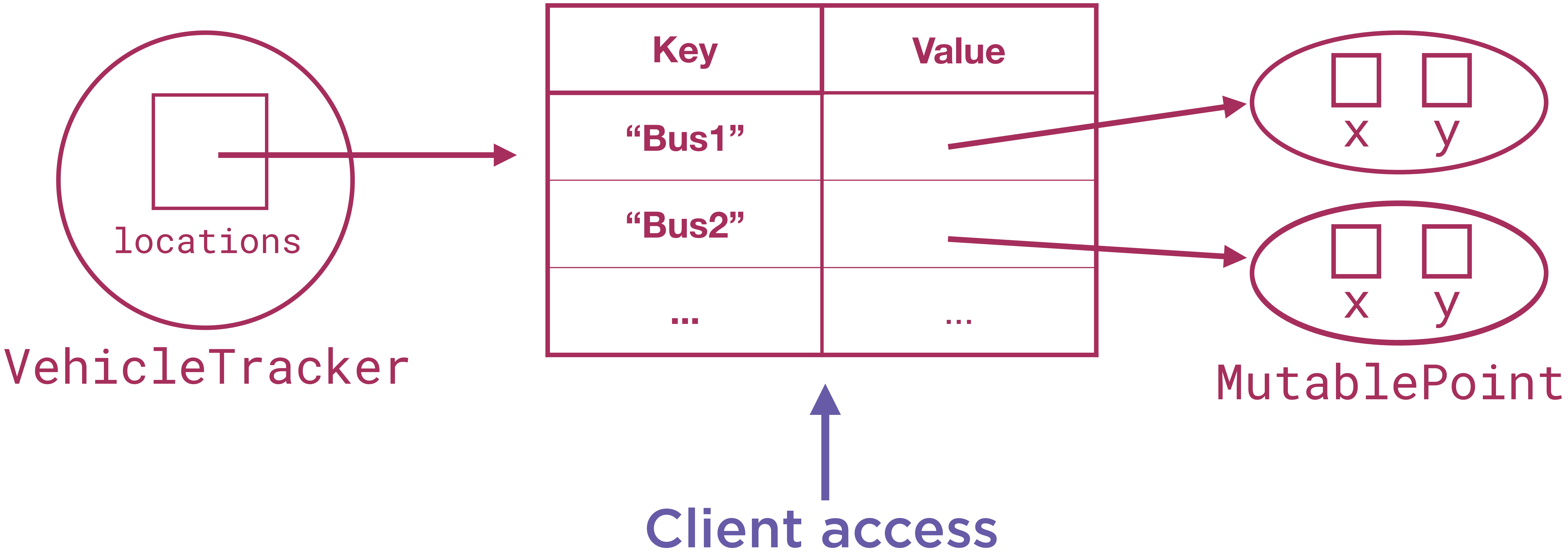
Confinement

- When an object can only ever be accessed by one thread at a time, we say it is *confined*
- Three ways to confine threads:
  - Instance Confinement
    - Depends on encapsulation
  - … two other ways …

Encapsulation

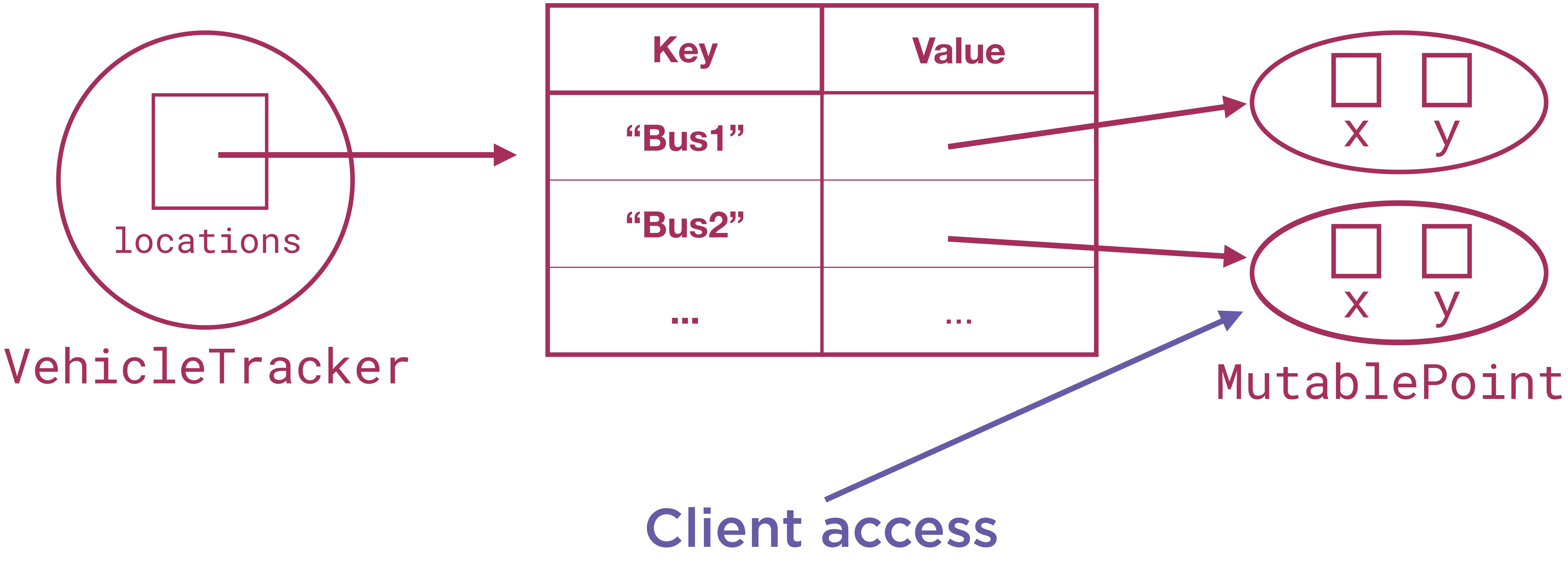
- Encapsulation prevents state of an object from being exposed externally
- No public fields, only getters
- But what *is* the state of an object?
  - Does the state of `VehicleTracker` include the *list* of vehicle locations?
  - Does it include every *individual* location?

Exposing State



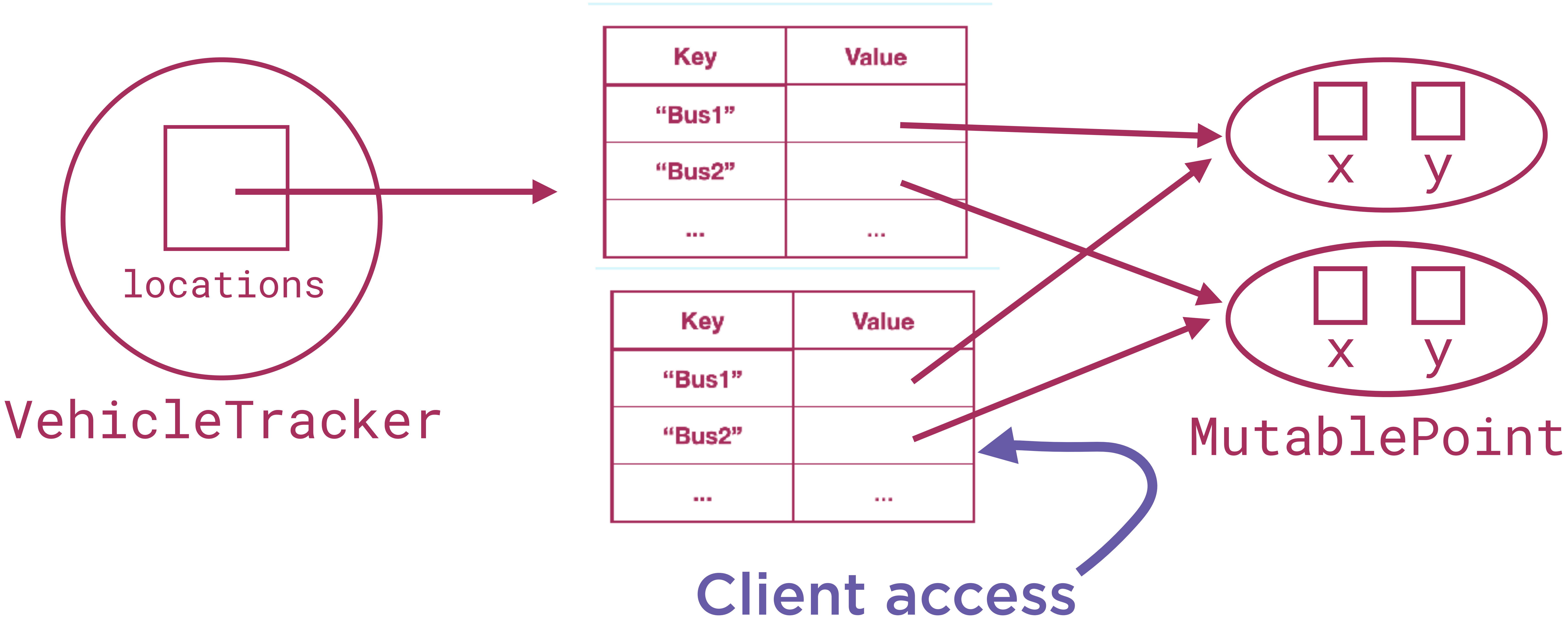

```java
public class NaiveVehicleTracker {
    ...
     public Map<Vehicle, MutablePoint> getVehicleLocations() {
          return vehicleLocations;
     }
    ...
}
```

Unmodifiable Collections



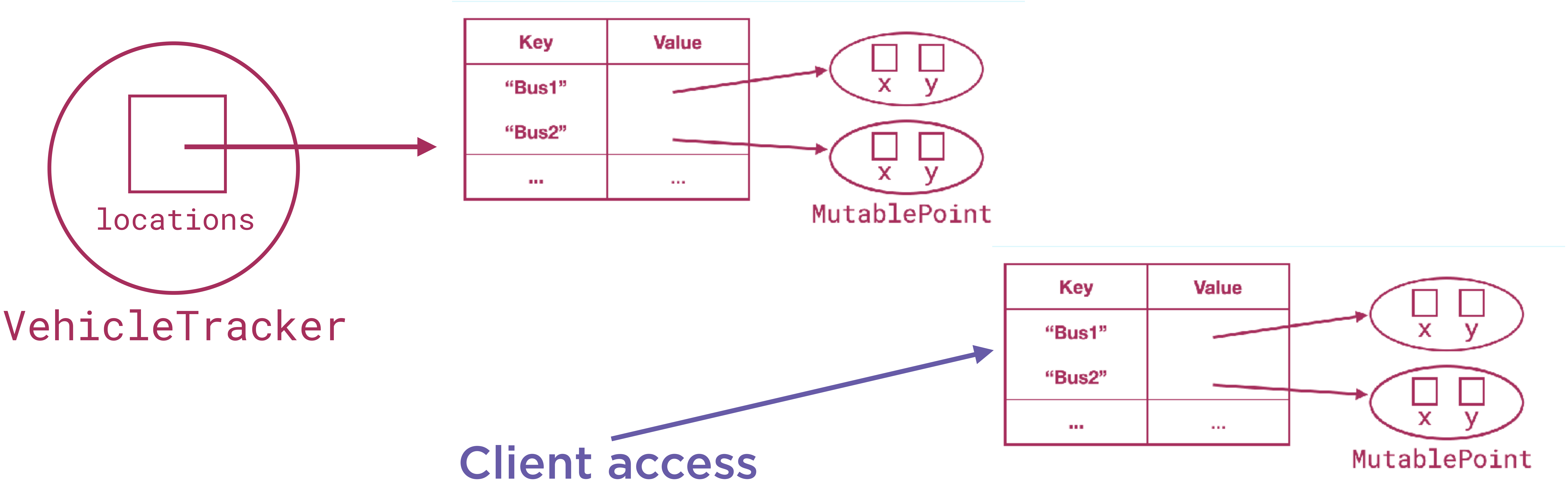

```
public class NaiveVehicleTracker {
    ...
     public Map<Vehicle, MutablePoint> getVehicleLocations() {
         return Collections.unmodifiableMap(vehicleLocations);
     }
    ...
}
```

Defensive Copying (shallow)



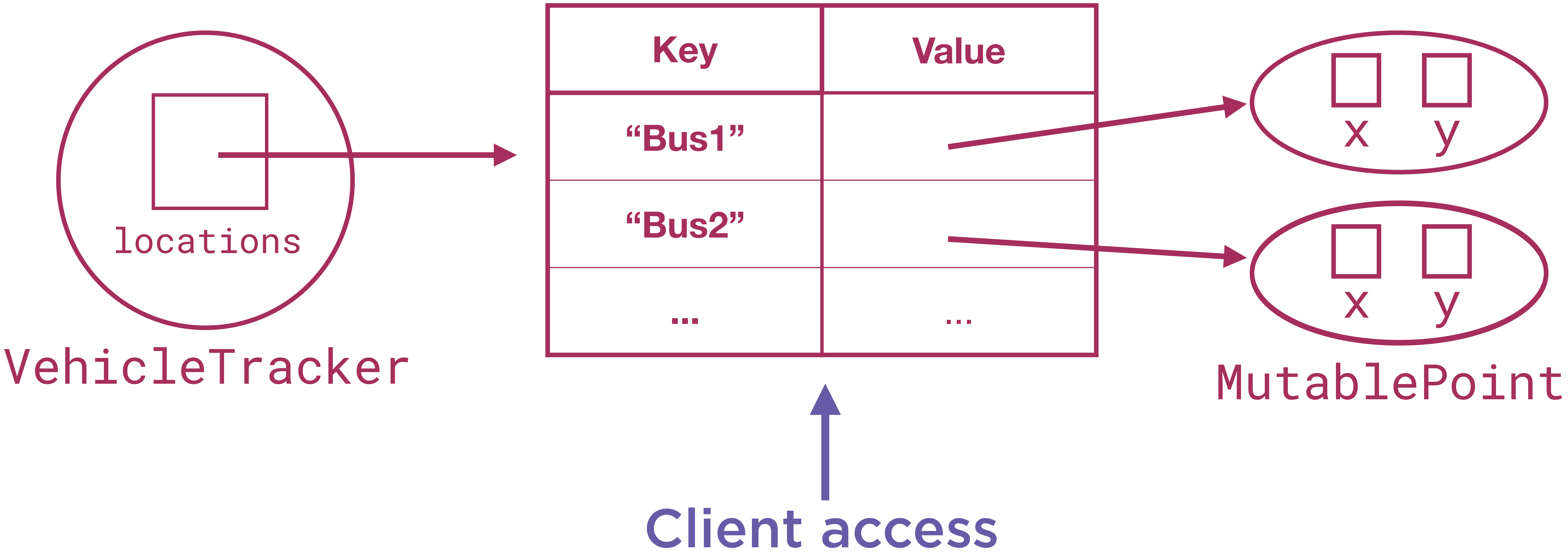

```
public class NaiveVehicleTracker {
    ...
     public Map<Vehicle, MutablePoint> getVehicleLocations() {
         return new HashMap<Vehicle, MutablePoint>(vehicleLocations);
     }
    ...
}
```

Defensive Copying (deep)



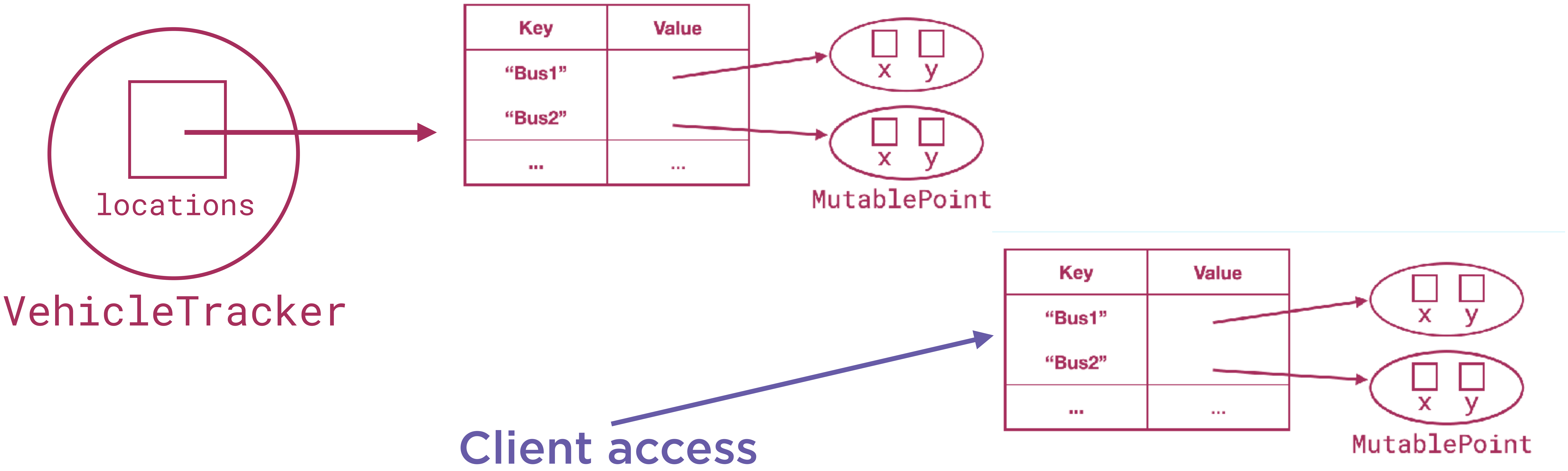

```
public class NaiveVehicleTracker {
    ...
    public Map<Vehicle, MutablePoint> getVehicleLocations() {
        return defensiveCopy(vehicleLocations);
    }
    ...
}
```

Accepting State – :(




```
public class NaiveVehicleTracker {
    ...
    public NaiveVehicleTracker(Map<Vehicle, MutablePoint>
                                                  vehicleLocations) {
       this.vehicleLocations = vehicleLocations;
    }
    ...
}
```

Accepting State :)




```
public class NaiveVehicleTracker {
    ...
    public NaiveVehicleTracker(Map<Vehicle, MutablePoint>
                                             vehicleLocations) {
      this.vehicleLocations = defensiveCopy(vehicleLocations);;
    }
    ...
}
```

Confinement

- When an object can only ever be accessed by one thread at a time, we say it is *confined*
- Three ways to confine objects:
  - Instance Confinement
    - Supported by encapsulation
  - Thread Confinement
    - e.g. Swing
    - No need for synchronization
    - ThreadLocal
  - Stack Confinement
    - local variables

Immutability

– Immutable objects are always thread-safe

– An immutable object is one whose state *graph* can't *observably* change after construction

– Java objects *can* be immutable. One way is to define their class such that it

- Has no mutators (setter methods, methods that change the state)
- Has only `private final` fields
- Has exclusive access to any mutable components

Immutability is the simplest way to achieve thread safety

Exercise

Either

- Fix the class `RaceConditionDemo` to be thread-safe in two ways:
  - Guard the `counter` variable with monitor locks, either on the `RaceConditionDemo` instance or (better) on a private object
  - Using an atomic variable

Or

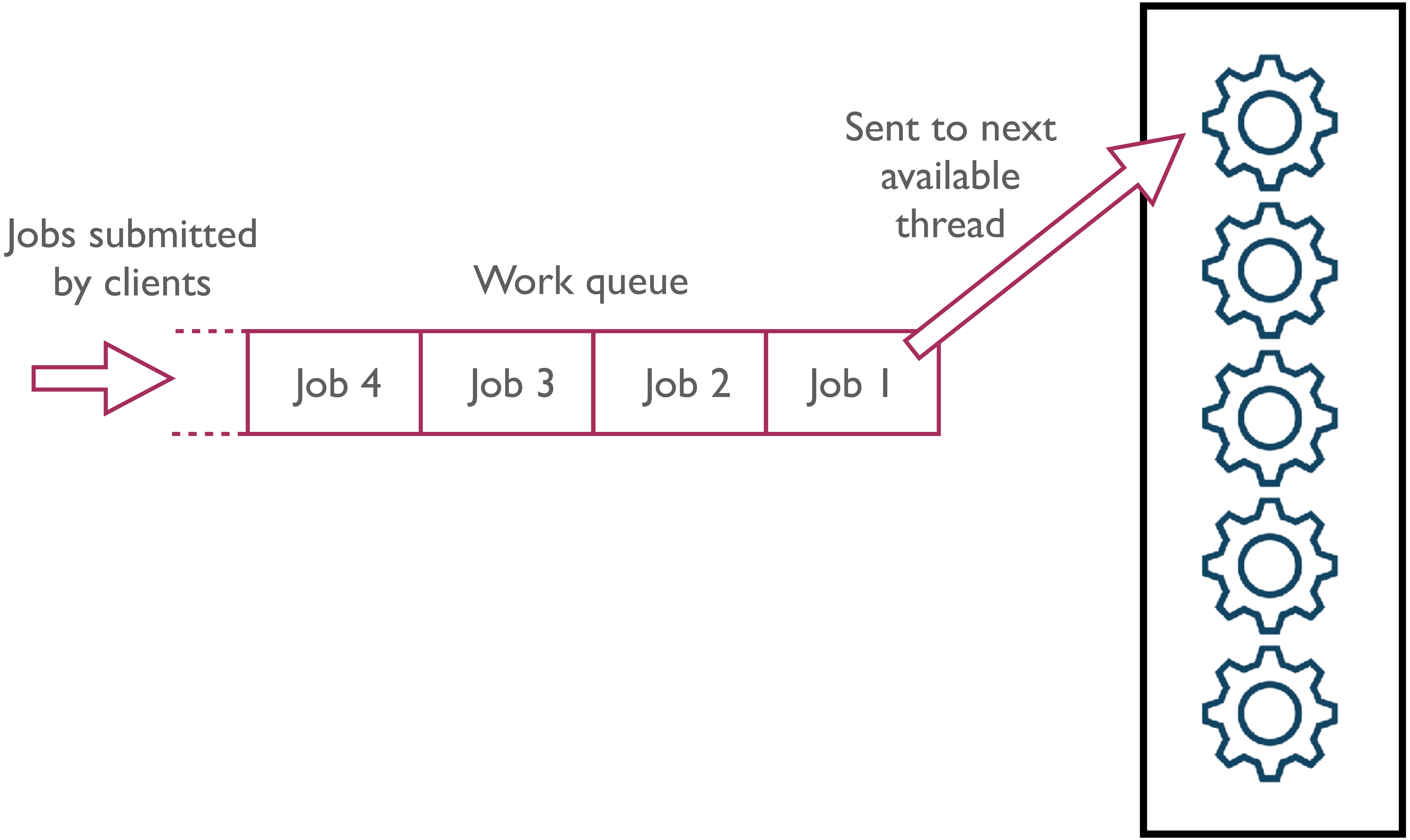
- Fix the `NaiveVehicleTracker` so it no longer exposes its state

Tasks and Task Execution

Per-thread vs. Pooling

- Starting a new thread has a significant overhead (~1MB, 1ms)
  - For small tasks, the overhead may predominate
  - Too many requests being concurrently handled ⇒ too many threads!
- For resources that are expensive to create, pooling is the best answer
  - Can limit the number of requests being handled by queuing them up
  - Pooled threads can be reused indefinitely

Thread Pools



Tasks and Task Execution

- A task is a unit of work
- Ideally, tasks should be
  - independent of the state of other tasks
  - fine-grained (relatively small fraction of application's computation requirement)
- For example, single client request to a server application
- In designing a concurrent system, think about tasks, not threads!

Tasks and Task Execution

– Sample task: handling requests to a web server

– We could start a new thread for each request:

```
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 = () -> handleRequest(connection);
            new Thread(task).start();
        }
    }

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

The `Executor` framework

– The abstraction for task execution is `Executor`;

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

The `Executor` framework

Using the `Executor` framework:

```
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 = () -> handleRequest(connection);
            exec.execute(task);
        }
    }

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

`Executor` implementations

- Create with `java.util.concurrent.Executors`
  - `Executors.newFixedThreadPool()`
    - Keeps a permanent total number of threads
  - `Executors.newCachedThreadPool()`
    - Pool size varies with demand
  - `Executors.newSingleThreadExecutor()`
    - Replaces the single thread if it dies unexpectedly
  - `Executors.newScheduledThreadPool()`
    - Similar to java.util.Timer, but better

The `Executor` lifecycle

When shutting things down, we always have to think about unfinished work…

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination (long timeout, TimeUnit unit)
            throws InterruptedException;
    // ... additional convenience methods for task submission
}
```

Callable

– `Runnables` can't return a result. What if we want to collect together the results of concurrently executing tasks?

– Result-bearing version of `Runnable` is `Callable<V>`:

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

– Example: retrieving images to render a web page

```
final Callable<ImageData> imageDataCallable =
    () -> imageInfo.downloadImage();
```

`Future<V>`

- You may want to interact with a task after it's been submitted for execution.
  - Is it completed?
  - Has it been cancelled?
  - You want to cancel it!
  - Get the result (waiting if necessary)

- `Future<V>` allows you to observe and manage tasks after submission

Future<V>

– Future<V> allows you to observe and manage tasks after submission:

```
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;
}
```

`Future<V>`

- Where do `Futures` come from?
- They're created by `ExecutorService` and `CompletionService` (coming next)
  - You submit a `Callable<V>` to get back a `Future<V>`
  - You submit a `Runnable` to get back a `Future<?>`

CompletionService<V>

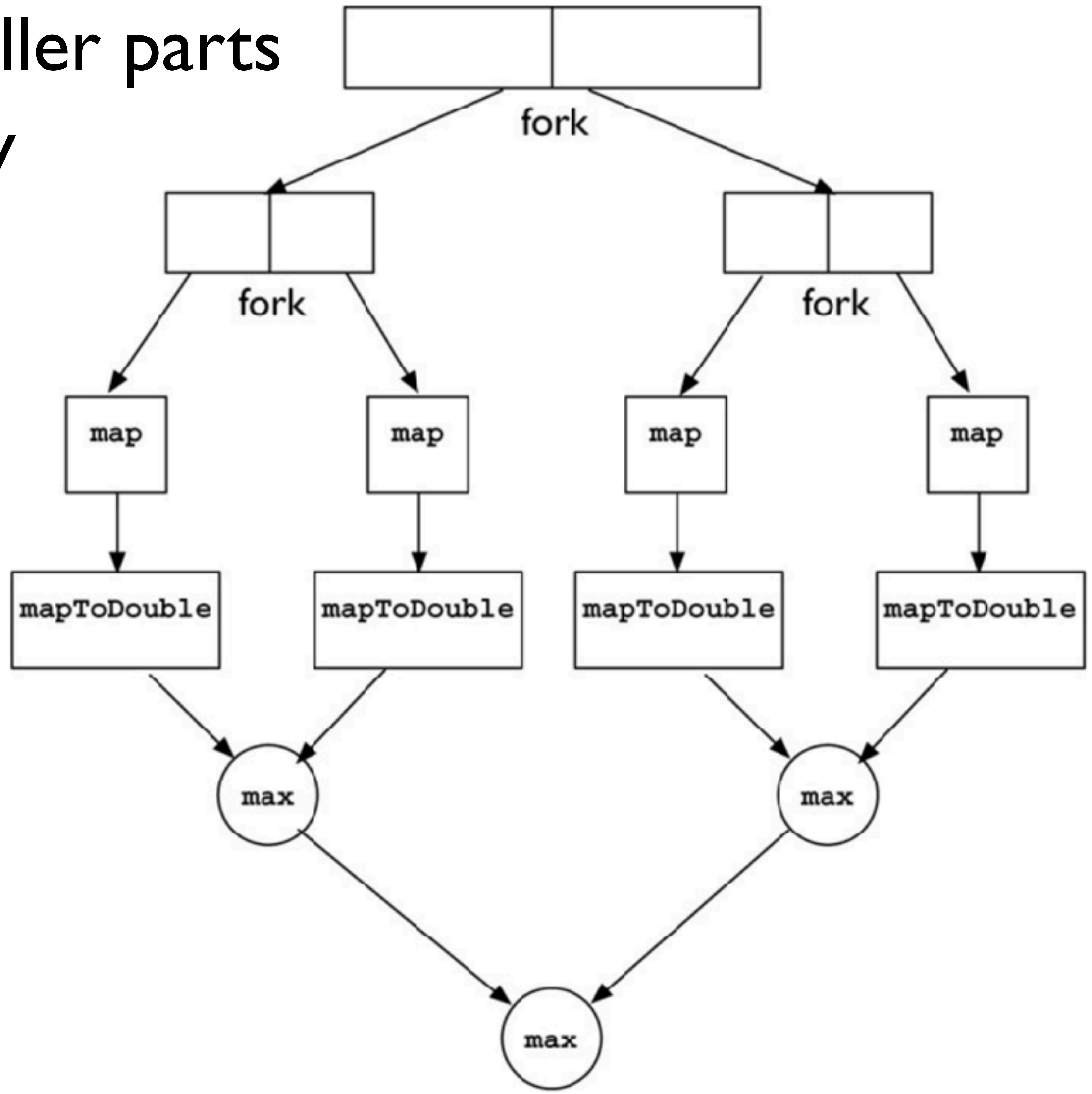
– Allows you to submit `Callables` and get them back in order of completion:

```
void renderPage(CharSequence source) {
      final List<ImageInfo> info = ... ; // List of image URLs
      CompletionService<ImageData> completionService =
              new ExecutorCompletionService<>(executor);
      info.forEach(imageInfo -> completionService.submit(imageInfo::downloadImage));
      renderText(source);
      try {
          for (int t = 0, t < info.size(); t++) {
              Future<ImageData> f = completionService.take();
              renderImage(f.get());
          }
      } catch (InterruptedException e) { ...
      } catch (ExecutionException e) { ...
      }
}
```

Any Executor

The Fork/Join Framework

- Fork-Join framework implements *recursive decomposition*
  - Repeatedly dividing a large task into smaller parts
  - Processing each small part independently
  - Joining the results together



The Fork/Join Framework

- To use the FJ Framework
  - Define a `RecursiveAction` (for void tasks) or a `RecursiveTask<V>` (for result-bearing tasks)
  - Template for `RecursiveAction`:

```
protected void compute() {
    if (workload.length() > THRESHOLD) {
        ForkJoinTask.invokeAll(createSubtasks());
    } else {
        process(workload);
    }
}
```

  - Submit your `RecursiveAction` or `RecursiveTask<V>` to the common Fork/Join pool, or one that you have created.

Parallel Streams

– A much easier way to get FJ functionality is to use parallel streams
– Only need to insert a call to `parallel()`

```
final long[] array1 = Arrays.stream(array)
          .parallel()
          .map(i -> new PrimeHelper().sumOfFirstNPrimes(i))
          .toArray();
```

– But conditions have to be right
  – The workload should be computationally intensive
    – And per-element independent
  – The total time to execute the sequential version should be around 50 microseconds or greater (N * Q >= 10000)
  – The stream source should be efficiently splittable
  – There should be enough cores to outweigh the overhead

CompletableFuture

– A Future with additional completion logic

– Flexible way to model complex concurrent workflows

– CompletableFuture.supplyAsync(Supplier<U>)

  – Factory method: creates a CompletableFuture that completes with the value obtained by calling the Supplier asynchronously

– CompletableFuture<Void> thenAccept(Consumer<T>)

  – Instance method: creates a new CompletableFuture that, when this future completes, executes with its result

CompletableFuture

– To asynchronously download the images and render each one:

```
public void renderPage(CharSequence source) {
    List<ImageInfo> info = scanForImageInfo(source);
    info.forEach(imageInfo ->
            CompletableFuture
                    .supplyAsync(imageInfo::downloadImage)
                    .thenAccept(this::renderImage));
    renderText(source);
}
```

CompletableFuture

- Very flexible way to model complex concurrent workflows
- API supports the composition of CompletableFutures in a variety of ways:
  - `acceptEither`, `applyToEither`, `runAfterBoth`, `runAfterEither`, `thenAccept`, `thenAcceptBoth`, `thenApply`, `thenCombine`, `thenCompose`, `thenRun`, `whenComplete`, etc
- Every method has variants to support running tasks synchronously or asynchronously
  - Asynchronous variants allow for running on the default Fork/Join pool, or a supplied pool

CompletableFuture

– Reactive frameworks like this are very effective at avoiding thread blocking

– But hard to program:

  – Handing lambdas to the framework (to manage blocking calls as callbacks)

  – Difficult to follow program flow

  – Problems with error handling, debugging, testing, profiling

Exercise

- Modify `FjArrayInitializer` to write the sum of the first N primes (using the `PrimeHelper` class) each element of the array, where N is the element's index
- Modify `FjArraySumCalculator` to return the maximum element of the array
  - And both the maximum and the minimum
- Implement these solutions using parallel streams

Concurrent Utilities

Concurrent Utilities

– Storage Collections
  - `List, Set, Map`
– Queues
– Synchronizers

Storage Collections – `List,Set,Map`

– Original (Java 2) Collections Framework implementations *not* thread-safe

  – `ArrayList, HashSet, HashMap`

  – “Why pay for thread safety if you don’t need it?”

  – Can use with *client-side locking*

  – Or with synchronized wrappers

  – Look out for `ConcurrentModificationException`

Concurrent Storage Collections

- Java 5 introduced concurrent storage structures
- Most important is `ConcurrentHashMap`
  - Supports concurrent read operations
  - Thread-safe and atomic operations
  - Can iterate over the collection while accessing it
    - Result not fully deterministic

ConcurrentHashMap

- Client-side locking not possible
- But also not necessary, because of these atomic actions:

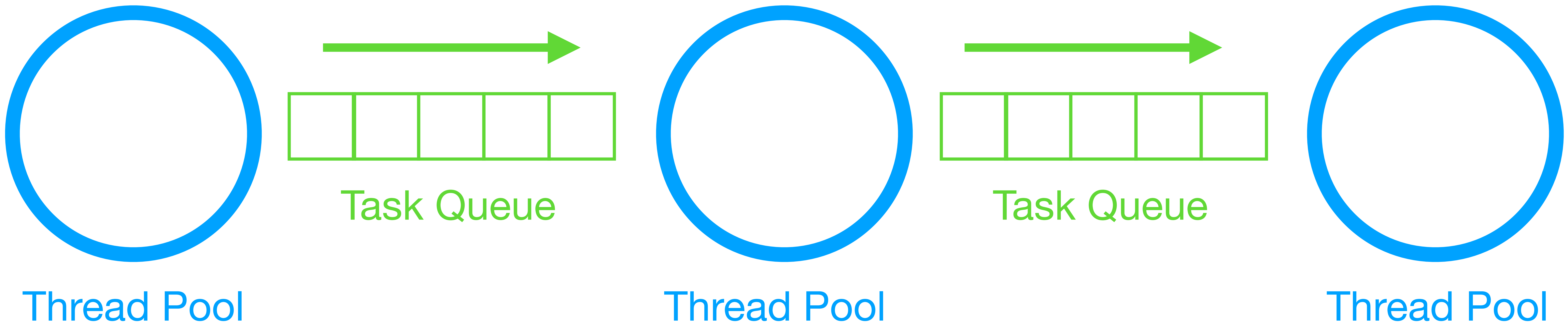
```
public class ConcurrentHashMap<K,V> extends Map<K,V> {
    V putIfAbsent(K key, V value) { ... }
    boolean remove(Object key, Object value) { ... }
    boolean replace(K key, V oldValue, V newValue) { ... }
    V replace(K key, V value) { ... }
    V computeIfPresent(K key, BiFunction<K,V,V> remappingFunction) { ... }
    V compute(K key, BiFunction<K,V,V> remappingFunction) { ... }
    V merge(K key, V value, BiFunction<V,V,V> remappingFunction) { ... }
     ...
}
```

Concurrent Storage Collections

- Nothing similar for `List`
  - For `Set`: `ConcurrentHashMap.newKeySet()`
- `List,Set`: `CopyOnWriteArrayXxx` classes
  - Useful for supporting many concurrent read operations, not useful if you have many concurrent writes

Blocking Queues

- Blocking Queues
  - Designed for workflow scenarios
  - Producer-consumer pattern
  - Queues can be bounded or unbounded



BlockingQueue

– Blocking queues force consumers to wait if there are no tasks available
– Bounded blocking queues force producers to wait if the queue is full

```
public interface BlockingQueue<E> extends Queue<E> {
    void put(E e) throws InterruptedException;
    boolean offer(E e, long timeout, TimeUnit unit)
            throws InterruptedException;
    E take() throws InterruptedException;
    E poll(long timeout, TimeUnit unit)
            throws InterruptedException;

    ... // other (non-blocking) operations
}
```

Concurrent Utilities

- Synchronizers: Utilities that co-ordinate control flow of threads
  - Semaphore
    - Manages a finite set of *permits*
    - Threads wanting to access a resource must wait for a permit
  - CountDownLatch – gate that can block threads
    - Closed when created
    - Maintains a count of unavailable resources
    - When count has reached zero, gate opens
  - CyclicBarrier, Phaser, Exchanger, SynchronousQueue

Virtual Threads

Virtual Threads

- The reactive programming model has been very successful
- It handles the problem of IO blocking
  - The cost is a very difficult programming model
- Virtual threads (finalised in Java 21) are a different approach
  - Same API as `Thread`, but very lightweight (0.001x)
    - Feasible to create millions of virtual threads
  - Very low cost of blocking
  - Implementation uses small number of platform threads
    - Called "carrier threads"

Virtual Threads – Scheduling

- Virtual threads scheduled differently from platform threads
  - OS schedulers are *pre-emptive*
    - Allow each thread a time-slice, then do a context switch
  - Virtual threads are schedule non-preemptively
    - Can run as long as they want – or until they block
- So only useful for blocking code

Structured Concurrency

- A `StructuredTaskScope` defines subtasks that can be executed asynchronously
  - Like the Fork/Join framework, but subtasks can be heterogenous
  - Like `CompletableFuture`, but with a much easier programming model
- New feature, ***in preview in Java 21***

```
Callable<String> task1 = ...
Callable<Integer> task2 = ...

try (var scope = new StructuredTaskScope<Object>()) {
    Subtask<String> subtask1 = scope.fork(task1);
    Subtask<Integer> subtask2 = scope.fork(task2);
    scope.join();
    ... process results/exceptions ...
} // close
```

Conclusion

Conclusion – where to go next

- There is much more to learn!
- O'Reilly on-demand courses:
  - *Java Multithreading and Parallel Programming Masterclass* (Cosmin Ionita)
  - *Java Concurrency, 2/e* (Douglas Schmidt)
- Java Memory Model – JSR 133
- *Java Concurrency in Practice* (Brian Goetz)
- *Mastering Lambdas* (Maurice Naftalin) – parallel stream performance

Subscribe to Heinz's monthly newsletter about live Java courses on the O'Reilly platform: https://javaspecialists.eu/courses/oreilly/

The End – Thank you!

Contact me at
maurice@morninglight.co.uk