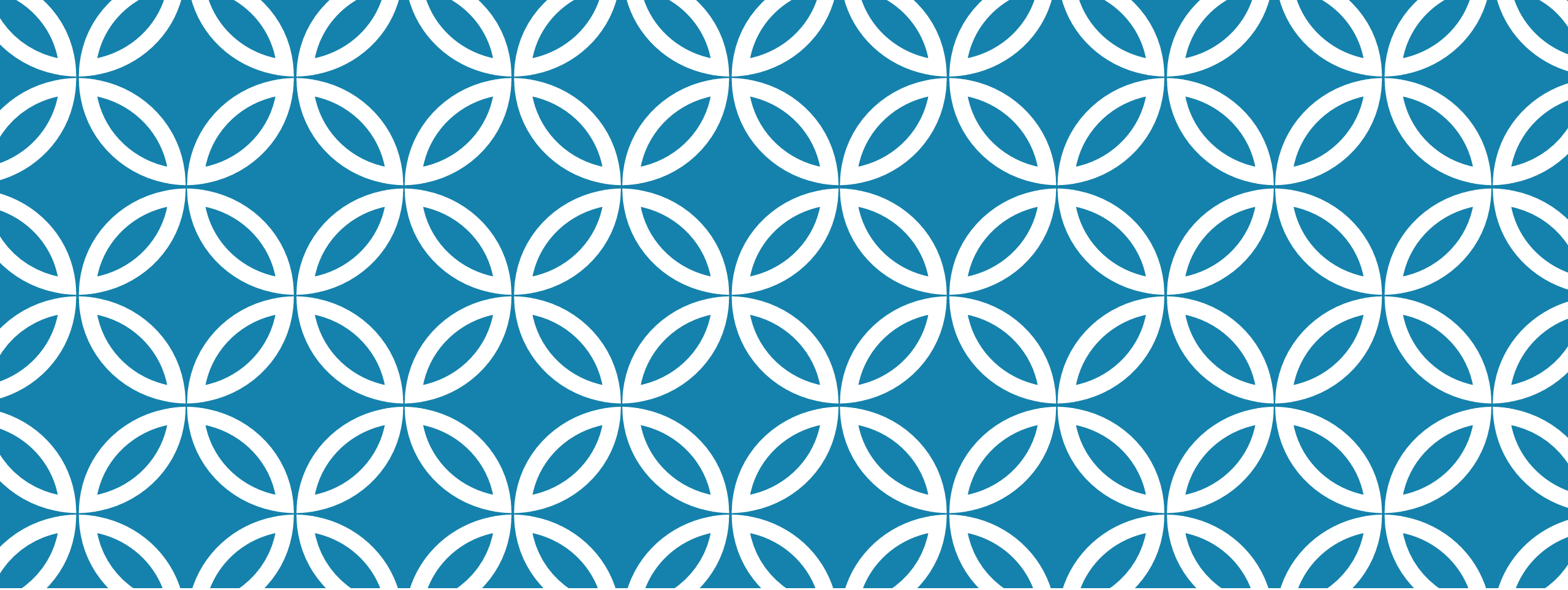
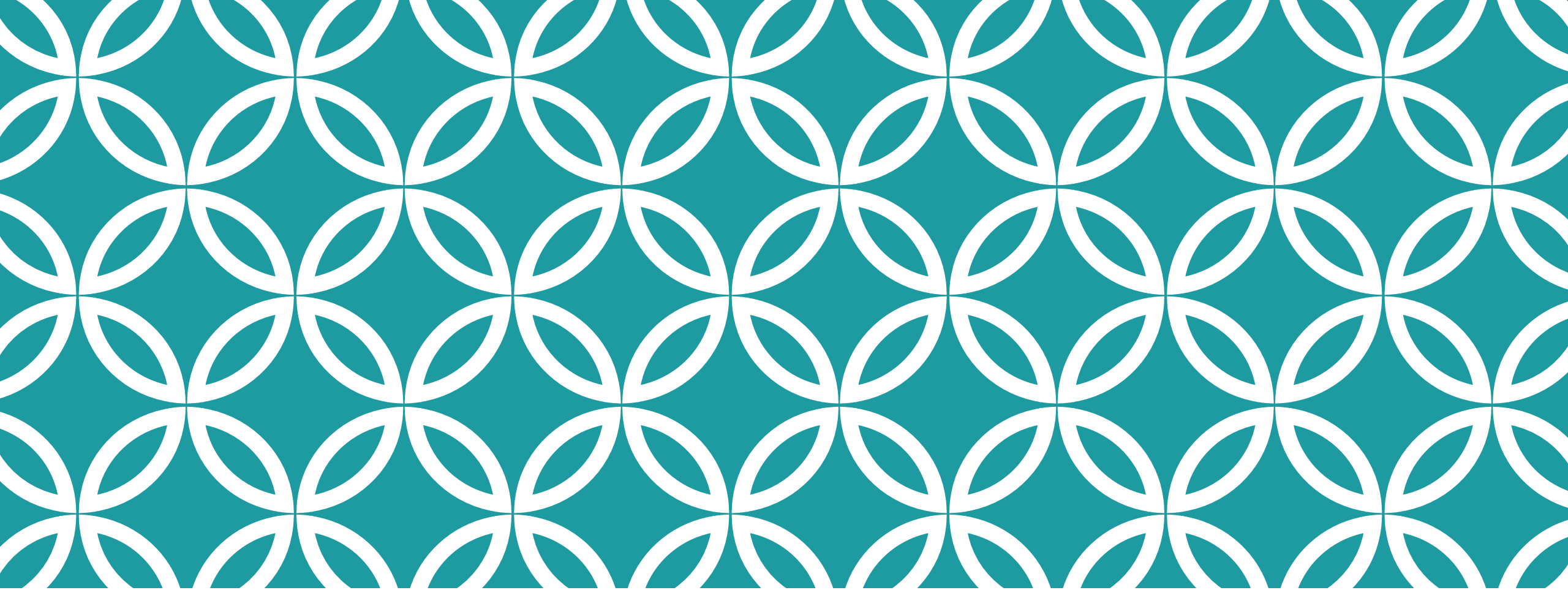


WEEK 8

Brendon Faleiro



PART 1

PARALLELIZING TASKS

CONCURRENCY VS PARALLELIZATION

Concurrency deals with being able to make a system run tasks in parallel on different processors (cores) or switch back and forth between tasks on the same processor.

Parallelization deals with running tasks simultaneously on separate processors.

Parallelization is the process of breaking up large tasks into smaller independent pieces that can be combined for final result.

An important technique for achieving maximal performance in applications is the ability to split intensive tasks into chunks that can be performed in parallel to maximize the use of computational power.

Traditionally tasks were made concurrent by explicitly creating threads using the Runnable interface in java.

AUTOPARALLELIZATION

We now look for parallelization rather than concurrency.

Java 8 lets you achieve parallelization through two mechanisms:

- Java Parallel Streams
- Fork/Join Tasks

JAVA STREAMS

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements.

Stream operations are either intermediate or terminal.

Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons.

Terminal operations are either void or return a non-stream result.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation.

Most of those operations must be both *non-interfering* and *stateless*.

Ref: http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

A function is non-interfering if it does not modify the underlying source of data.

A function is stateless if its execution is deterministic. e.g. lambda expression should not depend on any mutable variables or states from the outer scope which might change during execution.

Q: Write a streamed program to print the elements starting with “c” in ascending order:-

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList .stream()
        .filter(s -> s.startsWith("c"))
        .sorted()
        .forEach(System.out.println);
```

LAZY EVALUATION OF STREAMS

Streams are lazy evaluated.

So your stream code is only run when you have a terminal operation, until then you could imagine that your system is only remembering what all it must do to the input stream, but not actually performing it.

Only when you add the terminal operation will it actually execute all the steps.

What do you think is printed when this is executed?

```
Stream.of("d2", "a2", "b1", "b3", "c")
        .map(s -> { System.out.println("map: " + s);
                    return s.toUpperCase(); })
        .anyMatch(s -> { System.out.println("anyMatch: " + s);
                    return s.startsWith("A"); });
```

// map: d2

// anyMatch: D2

// map: a2

// anyMatch: A2

Now instead of running each stream based operation sequentially, we can run them in parallel.

We could either use “parallel()” operation on an existing stream to parallelize it or we could use “parallelStream()” to create a new stream that is inherently parallelized.

The rest of the working is the same as sequential streams.

FORK/JOIN TASKS

Best used in tasks that can be implemented using “Divide and Conquer” or “Map and Reduce” mechanisms.

Executors ease the management of such concurrent tasks.

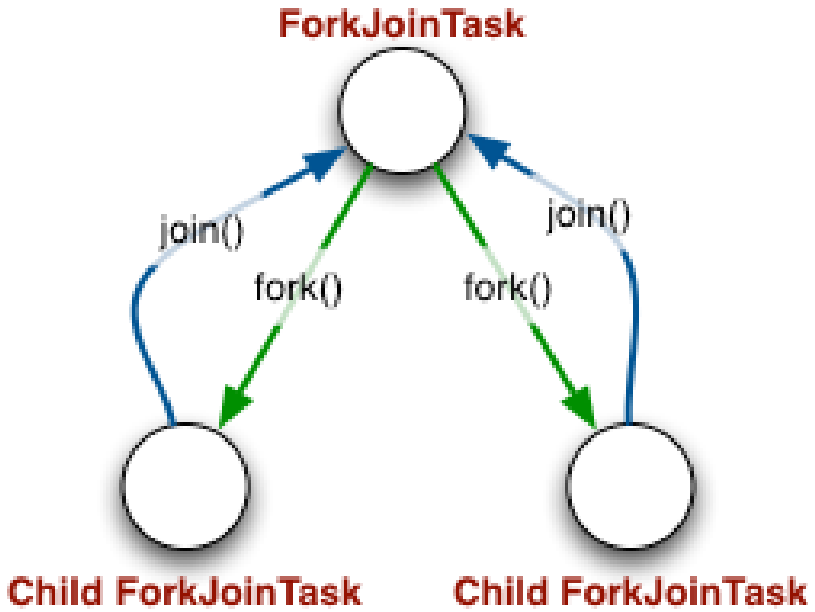
The idea is to split the data space to be processed by an algorithm into smaller, independent chunks. That is the “map” phase.

In turn, once a set of chunks has been processed, partial results can be collected to form the final result. This is the “reduce” phase.

ForkJoinTask objects support the creation of subtasks plus waiting for the subtasks to complete.

ForkJoinTask objects feature two specific methods:

- The fork() method allows a ForkJoinTask to be planned for asynchronous execution. This allows a new ForkJoinTask to be launched from an existing one.
- In turn, the join() method allows a ForkJoinTask to wait for the completion of another one.



PARALLELIZING DIVIDE-AND-CONQUER ALGORITHMS

- The basic strategy:
  - Turn recursive calls into tasks
  - Solve the small instances directly
  - For larger instances requiring recursive calls, create tasks for each recursive call

- Performance tuning
  - Use a larger threshold than that specified in the algorithm for switch to sequential solving
  - Threshold should take account of original problem size, number of CPUs

Divide-and-conquer algorithms take this general form

```
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return problem.solveSequentially();
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(problem.extractLeftHalf());
            right = solve(problem.extractRightHalf());
        }
        return combine(left, right);
    }
}
```

COMPONENTS OF FORK/JOIN FRAMEWORK

Specialized executor class:

ForkJoinPool

- Implements ExecutorService interface
- Uses specialized thread-pool management, work distribution strategies tuned for divide and conquer

Specialized task class: ForkJoinTask<V>

- Implements Future<V> interface
- Has numerous specialized operations
- Two important subclasses

– RecursiveTask: value is returned

– RecursiveAction: no value is returned

FORKJOINPOOL

The ForkJoinPool is the main executor for ForkJoin tasks and limits the number of workers to number of CPUs(default) or the user specified number.

It maintains the threadpool and allocates work to workers.

Workers that are waiting for subtasks to complete are put to work on other subtasks

Work-stealing used to keep workers busy

- Each worker has its own work queue (actually, a work deque)
- When a workers deque is empty, it takes work from another workers deque

Ref: https://www.cs.umd.edu/class/fall2012/cmsc433/0101/lecture-notes/lec20-forkjoin.pdf

IMPORTANT METHODS IN FORKJOINPOOL

ForkJoinTask<V> fork()

- Arranges to asynchronously execute this task

V join()

- Returns the result of the computation when it is done.

V invoke()

- Commences performing this task, awaits its completion if necessary, and returns its result, or throws an (unchecked) RuntimeException or Error if the underlying computation did so.

MORE ON FORK() AND JOIN()

- fork() has effect of submitting task to ForkJoinPool

Task is placed in deque of “parent task” (i.e. one that performed fork())

Task performing fork() keeps executing

- join() has effect like get()

Task performing join() waits until result of subtask is available

While it is waiting it may start work on other tasks in its deque

USING THE FORK JOIN POOL

Make a subclass of RecursiveTask<T>

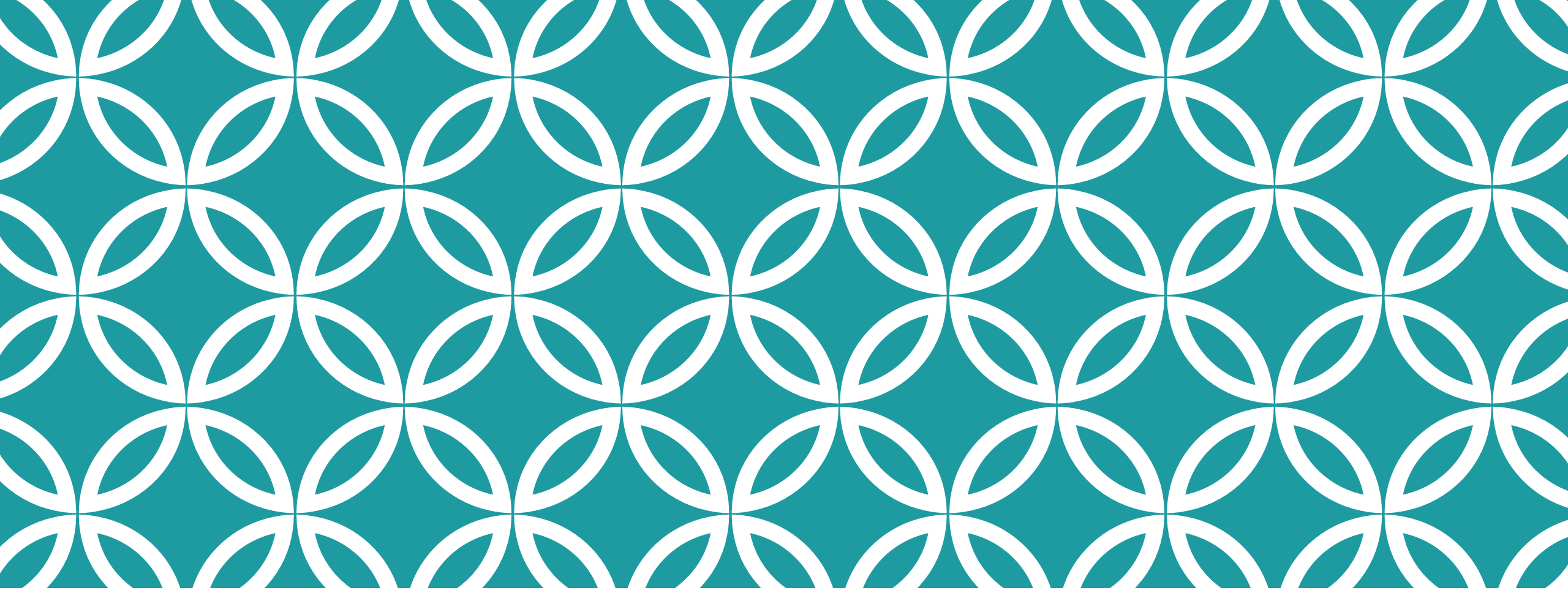
Pass all needed arguments to the constructor of this task

Put all the real work in the compute method.

- If the task is small, calculate the answer and return the answer.
- If task is still large, break it into two subtasks, each of which are new instances of this subsclass.
- Call leftPiece.fork() to start running the leftPiece task.
- Call rightPiece.compute() to start the right piece execution on current thread.
- Call leftPiece.join() to wait for left piece execution to finish.
- Combine the results of the two pieces and return it.

Create an instance of your subclass and call pool.invoke on that instance.

See ForkJoinExample.java



PART 2

Prolog

PROGRAMMING LANGUAGES (TYPE 3)

Types of languages:

1. Functional Languages
2. Imperative Languages
3. Logical/ Declarative languages

We saw Functional Languages (OCaml) and Imperative Languages (Java). We will now see Logical Languages (Prolog)

LOGICAL PROGRAMMING

Looks at what must be done and not how to do it.

It uses first order logic as the universal language.

You give the system a bunch of constraints, and ask the system to find an answer that satisfies the constraints.

Challenge: Huge search space!

Applications:

- Database querying
- Big Data Analytics

PROLOG : ACCORDING TO WIKIPEDIA

Prolog is a general-purpose logic programming language.

Prolog has its roots in first-order logic, and unlike many other programming languages, Prolog is declarative: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a *query* over these relations.

The language has been used for theorem proving, expert systems, type inference systems, and automated planning, as well as its original intended field of use, natural language processing.

Modern Prolog environments support creating graphical user interfaces, as well as administrative and networked applications.

Prolog is well-suited for specific tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates.

PROLOG

Prolog is a logical language.

It consists of predicates and uninterpreted constants.

Uninterpreted constants are basically text strings that have absolutely no meaning to the system.

Predicates are like questions that can be resolved to true or false.

A predicate fed to prolog is a fact.

A set of facts is called a knowledge base or a database.

Once you build your knowledge base, you can ask it questions.

Besides stating facts, you can also write rules.

Rules can build new facts!

Rules can be recursive to achieve multiple level facts.

NOTE: We will use GNUProlog.

WORKING WITH LISTS

Ocaml code for size of list:

```
let rec size l =
        match l with
        | [ ] -> 0
        | h::t -> let N = size(t) in
                  N+1;;
```

```
size [1;2;3;4];;
```

Prolog code:

```
size([ ], 0 ).
size([H|T], N) :- size(T , N1), N is N1+1.

size([1,2,3,4], N).
```

Q. What would happen if size([H|T], N) :- N is N1+1, size(T, N1).
Ans: Instantiation error on (is)/2

Q. What would happen if size([H|T], N) :- size(T, N1), N=N1+1.
Ans: N = 0+1+1+1+1

WORKING WITH LISTS

Prolog code:

```
ismember([X|_], X).

ismember([H|T], X):- member(T, X).
```

What does this code do?

Returns true for every time X is found in list.

If I don't want to look for all instances, but just for one instance, I need to stop looking the moment I have looked once.

We will look at how Prolog achieves its search in the next class.