Monad

Requirements

- `of` method to initialize the value and side information.
- `flatmap` method to update the value and side information.
- Obey 3 laws:
    1. Identity laws

```
Loggable.of(4).flatMap(x -> incrWithLog(x));
incrWithLog(4); // equivalent statements

// Left identity law
Monad.of(x).flatMap(x -> f(x)) must be the same as f(x)
// Right identity law
monad.flatMap(x -> Monad.of(x)) must be the same as monad
```

    2. Associative law:

```
monad.flatMap(x -> f(x)).flatMap(x -> g(x))// must be the same as
monad.flatMap(x -> f(x).flatMap(y -> g(y)))
```

Functor

1. Preservers Identity:
    - `functor.map(x -> x)` is the same as `functor`
2. Preserves Composition
    - `functor.map(x -> f(x)).map(x -> g(x))` is the same as `functor.map(x -> g(f(x))`.

```
class LazyInt {
    Supplier<Integer> supplier;

    LazyInt(Supplier<Integer> supplier) {
        this.supplier = supplier;
    }

    int get() {
        return supplier.get();
    }
    LazyInt map(Function<? super Integer, Integer> mapper) {
        return new LazyInt(() -> mapper.apply(get()));
    }

    LazyInt flatMap(Function<? super Integer, LazyInt> mapper) {
        return new LazyInt(() -> mapper.apply(get()).get();
    }
}
```

Q: Is `LazyInt` a functor?

> Yes.
>
> - Calling map with identity function gives a `LazyInt` of `identity.apply(supplier.get())` which is just `supplier.get()` – no change.
> - Calling `map(g).map(f)` gives a `LazyInt` of `f.apply(new LazyInt(() -> g.apply(get())).get())`, which is just `f.apply(g.apply(get()))`.

Q: Is `LazyInt` a monad?

> - No, because it does not have a `of` (or unit) method
> - Yes, if `new LazyInt(() -> x)` is assumed to be the of operator for value x, need to further explain what the laws are and linked it to the context of `LazyInt`.

Why is it better to declare the argument to map as Function instead of Function?

> The argument is declared this way so that we can pass in any function that operates on the superclass of `Integer` (such as `Number` and `Object`) (e.g., Function `h = x -> x.hashCode()`) into map

Concurrency & Parallel

- concurrency gives the illusion of subtasks running at the same time,
- parallel computing refers to the scenario where multiple subtasks are truly running at the same time
- parallel program is a subset of concurrent program
- add `parallel()` to the chain of calls in a stream to enable parallel processing
  - parallel() is a lazy operation → can insert the call to parallel() anywhere in the chain
  - sequential() which marks the stream to be process sequentially.
  - If you call both parallel() and sequential() in a stream, the last call determines how the program is processed

```
s.parallel().filter(x -> x < 0).sequential().forEach(..);
// sequential
```

- To ensure that the output of the parallel execution is correct, the stream operations must
  - not interfere with the stream data, and
    - Interference means that one of the stream operations modifies the source of the stream during the execution of the terminal operation.

```
List<String> list = new ArrayList<>(List.of("Luke", "Leia", "Han"));
list.stream()
    .peek(name -> {
        if (name.equals("Han")) {
            list.add("Chewie"); // modifies the source of the stream
        }
    })
    .forEach(i -> {});
// throws ConcurrentModificationException
```

    - stateless most of the time.
        - A stateful lambda is one where the result depends on any state that might change during the execution of the stream.

```
Stream.generate(scanner::nextInt)
    .map(i -> i + scanner.nextInt())
    .forEach(System.out::println)
// generate and map are stateful: depend on the state of the standard input
// Parallelizing this may lead to incorrect output (unordered)
```

    - Side-effects should be kept to a minimum.

```
List<Integer> list = new ArrayList<>(
Arrays.asList(1,3,5,7,9,11,13,15,17,19));
List<Integer> result = new ArrayList<>();
list.parallelStream()
    .filter(x -> isPrime(x))
    .forEach(x -> result.add(x)); //modifies result
// ArrayList is a non-thread-safe data structure.
// If two threads manipulate it at the same time, an incorrect result may result

// To print result in order
// #1 .collect method
list.parallelStream()
    .filter(x -> isPrime(x))
    .collect(Collectors.toList())

//  #2 use a thread-safe data structure.
List<Integer> result = new CopyOnWriteArrayList<>();
list.parallelStream()
    .filter(x -> isPrime(x))
    .forEach(x -> result.add(x));
```

- Associativity

```
Stream.of(1,2,3,4).reduce(1, (x, y) -> x * y, (x, y) -> x * y);
/*
To run reduce in parallel:
- combiner.apply(identity, i) must be equal to i . --> i * 1 equals i

- The combiner and the accumulator must be associative: the order of applying must not matter --> (x * y) * z equals x * (y * z)

- The combiner and the accumulator must be compatible: combiner.apply(u, accumulator.apply(identity, t)) must equal to accumulator.apply(u, t) --> u * (1 * t) equals u * t
*/
```

- Parallelizing a stream does not always improve the performance. Creating a thread to run a task incurs some overhead, and the overhead of creating too many threads might outweigh the benefits of parallelization.

Ordered vs Unordered

- Streams created from `iterate`, ordered collections (e.g., `List` or arrays), from `of`, are ordered.

- Stream created from `generate` or unordered collections (e.g., `Set`) are unordered.
- Stable operations: `distinct` , `sorted`
  - preserve the original order of elements
- The parallel version of `findFirst` , `limit` , and `skip` can be expensive on an ordered stream, since it needs to coordinate between the streams to maintain the order.
- If we have an ordered stream and respecting the original order is not important, we can call `unordered()` as part of the chain command to make the parallel operations much more efficient.

```
Stream.iterate(0, i -> i + 7)
      .parallel()
      .limit(10_000_000)
      .filter(i -> i % 64 == 0)
      .forEachOrdered(i -> { });

Stream.iterate(0, i -> i + 7)
      .parallel()
      .unordered()
      .limit(10_000_000)
      .filter(i -> i % 64 == 0)
      .forEachOrdered(i -> { });
```

Asynchronous Programming

- synchronous: If the method is not done, the execution of our program stalls, waiting for the method to complete its execution. Only after the method returns can the execution of our program continue
  - the method blocks until it returns.

Threads

- A thread is a single ow of execution in a program.
- We can use the instance method `getName()` to find out the name of a thread, and the class method `Thread.currentThread()` to get the reference of the current running thread

```
public static void main(String[] args) {
    // create and run 2 threads
    System.out.println(Thread.currentThread().getName());

    // new Thread(...) constructor takes in a Runnable instance
    // Runnable is a functional interface with only run() method
    new Thread(() -> {
        System.out.print(Thread.currentThread().getName());
        for (int i = 0; i < 10; i += 1) {
            System.out.print("_");
        }
    }).start();

    new Thread(() -> {
        System.out.print(Thread.currentThread().getName());
        for (int i = 0; i < 10; i += 1) {
            System.out.print("*");
```

```
        }
    }).start();
    // start() returns immediately.
    // It does not return only after the given lambda expression completes its execution
}
// main
// Thread-0__________Thread-1**********

Stream.of(1, 2, 3, 4)
                .parallel()
                .reduce(0, (x, y) -> {
                    System.out.println(Thread.currentThread().getName());
                    return x + y;
                });
/*
main
main
main
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-5
*/ // gets a different output each time
If you remove the parallel() call, then only main is printed, showing the reduction being done sequentially in a single thread
```

Sleep

- cause the current execution thread to pause execution immediately for a given period (in milliseconds).
- After the sleep timer is over, the thread is ready to be chosen by the scheduler to run again.

```
Thread findPrime = new Thread(() -> {
    System.out.println(
        Stream.iterate(2, i -> i + 1)
            .filter(i -> isPrime(i))
            .limit(1_000_000L)
            .reduce((x, y) -> y)
            .orElse(null));});

findPrime.start();

while (findPrime.isAlive()) {
// isAlive() to periodically check if another thread is still running.
// The program exits only after all the threads created run to their completion.
    try {
        Thread.sleep(1000);
        System.out.print(".");
    } catch (InterruptedException e) {
        System.out.print("interrupted");
    }
}
```

Limitations of threads

- no methods in Thread that return a value - need the threads to communicate through shared variables.

- no mechanism to specify the execution order and dependencies among them - which thread to start after another thread completes.
- need to consider the possibility of exceptions in each of our tasks.
- overhead -- the creation of Thread instances takes up some resources in Java. As much as possible, we should reuse our Thread instances to run multiple tasks

Fork Join Pool

- Each thread has a queue of tasks.
- When a thread is idle, it checks its queue of tasks.
    - If the queue is not empty, it picks up a task at the head of the queue to execute (e.g., invoke its compute() method).
    - Otherwise, if the queue is empty, it picks up a task from the *tail* of the queue of another thread to run → work stealing.
- When fork() is called, the caller adds itself to the head of the queue of the executing thread. This is done so that the most recently forked task gets executed next, similar to how normal recursive calls. When join() is called, several cases might happen. If the subtask to be joined hasn't been executed, its compute() method is called and the subtask is executed. If the subtask to be joined has been completed (some other thread has stolen this and completed it), then the result is read, and join() returns. If the subtask to be joined has been stolen and is being executed by another thread, then the current thread nds some other tasks to work on either in its local queue or steal another task from another queue.

PYP

```
Integer x = i;
Integer y = i;
// x == y can either be true (<128)/false
// x.equals(y) is always true
```

```
class Element {
    public int row;
    public int col;
    public double value;

    @Override
    public boolean equals(Object o) { .. }

    @Override
    public int hashCode() { .. }
}
```

Show the line of code that declares a field of type `HashMap` in the class Matrix to store the Element objects. Explain your design.

> `HashMap,Element> map = new HashMap<>()`;
>
> Using both row and col as key (as a Pair) allow us to look up efficiently a given row and col, as long as `Pair` computes its `hashCode` using both the row and col value.

Implement `get()`

> `map.get(new Pair(row, col))`.
>
> If result is null, return 0. Otherwise, return the value

Implement `set()`

> `map.put(new Pair(row, col), value)`
>
> - check if row and col are in range of the matrix.
> - check if there is an existing entry, remove it; Otherwise, we do nothing (no point inserting 0 into our sparse matrix).

Should Element be declared as an inner class within Matrix?

> Yes. The class Element is only used within the Matrix class.

```
1   class Fib extends RecursiveTask<Integer> {
2       final int n;
3
4       Fib(int n) { this.n = n; }
5
6       Integer compute() {
7           if (n <= 1) return n;
8
9           Fib f1 = new Fib(n - 1);
10          Fib f2 = new Fib(n - 2);
11          // insert code here
12      }
13  }
```

Q: Which of the following lines of code, when inserted, would compile without error and lead to correct and efficient parallelization of the calculation of Fibonacci number for 10 when `ForkJoinPool.commonPool().invoke(new Fib(10))` is called?

A. `f1.fork(); f2.fork(); return f1.join() + f2.join();`

B. `f1.fork(); f2.fork(); return f2.join() + f1.join();`

C. `f1.fork(); return f1.join() + f2.compute();`

D. `f1.fork(); return f2.compute() + f1.join();`

E. `return f1.compute() + f2.compute();`

> All the choices give correct answers.
>
> - E computes sequentially, so is the least efficient.
> - A and B are the same – they gives up the current thread and launch two new tasks.
>   - From lec notes: Since the most recently forked task is likely to be executed next, we should `join()` the most recent `fork()` task first, i.e. B is slightly more efficient than A
> - C computes sequentially: ensure that `f1` is completed before running `f2`
> - D is the most efficient (slightly better than A and B):

- `f2.compute()` reduces the overhead of interacting with the `ForkJoinPool` and therefore will likely be faster
- if there's only one computation needed, just compute
- if there're two or more, compute one and fork the others

```
public class A {
    static int x;

    static int foo() {
        return 0;
    }

    int bar() {
        return 1;
    }

    static class B {}

    public static void main(String[] args) {
        x = 1;
        A a = new A();
        B b = new B();
        new A().bar();
        new A().foo();
        // not the best way but can still compile: equivalent to new A();
A.foo();
    }
}
```

The interface `TriFunction` is a functional interface for a function that takes in three arguments, of types S, T, and U respectively, and returns a result of type R.

```
interface TriFunction<S,T,U,R> {
    R apply(S s, T t, U u);
}
```

Suppose we want to write a method curry that takes in a `TriFunction` and returns a curried version of the method.

```
.. curry(TriFunction<S, T, U, R> lambda) {
// missing line
}
/*
For instance, calling curry on (x, y, z) -> x + y * z
returns x -> y -> z -> x + y * z
*/
```

return type should be `Function<S, Function <T, Function<U, R>>>`

body should be `return x -> y -> z -> lambda.apply(x, y, z);`

```
class CannotUndoException extends RuntimeException {
}
```

```
3
4   class Undoable<T> {
5       T value;
6       Deque<Object> history;
7
8       Undoable(T t, Deque<Object> history) {
9           this.value = t;
10          this.history = history;
11      }
12
13      static <T> Undoable<T> of(T t) {
14          return new Undoable<T>(t, new LinkedList<Object>());
15      }
16
17      public <R> Undoable<R> flatMap(Function<T, Undoable<R>> mapper) {
18          // fill in the blank
19          Undoable<R> r = mapper.apply(value);
20          Deque<Object> newHistory = new LinkedList<>();
21          newHistory.addAll(history);
22          newHistory.addAll(r.history);
23          return new Undoable<R>(r.value, newHistory);
24      }
25      public <R> Undoable<R> undo() {
26          Deque<Object> newHistory = new LinkedList<>(this.history);
27          R r;
28
29          try {
30              r = (R)newHistory.removeLast(); // Line A
31          } catch (NoSuchElementException e) {
32              // Missing line B
33              throw new CannotUndoException();
34          }
35          return new Undoable<R>(r, newHistory);
36      }
37
38      Undoable<Integer> length(String s) {
39          Deque<Object> history;
40          history = new LinkedList<>();
41          history.add(s);
42          return new Undoable<Integer>(s.length(), history);
43      }
44  }
```

Why does line A lead to a compiler warning of unchecked cast?

> This is a narrowing type conversion, from R to Object.
>
> But since R is erased during compile time, the runtime system cannot safely check the type to make sure that it matches.

```
1   // What would happen if we do the following?
2   Undoable<Integer> i = Undoable.of("hello").flatMap(s -> length(s));
3   Undoable<Double> d = i.undo(); // becomes Undoable during runtime
```

> - The code runs without error. Even though we assign an `Undoable<String>` to `Undoable<Double>`, during runtime, it is stored as an Object reference, and the

reference can refer to String.

- An error would occur only if we try to apply function that operates on `Double` to the `Undoable`, in which case it will throw a `ClassCastException`.

```
class LazyList<T> {
    private Supplier<T> head;
    private Supplier<LazyList<T>> tail;

    public LazyList(Supplier<T> head, Supplier<LazyList<T>> tail) {
        this.head = head;
        this.tail = tail;
    }

    public static <T> LazyList<T> iterate(T init, Predicate<T> cond,
    UnaryOperator<T> op) {
        if (!cond.test(init)) {
            return LazyList.empty();
        } else {
            return new LazyList<T>(() -> init,
                                   () -> iterate(op.apply(init), cond, op));
        }
    }

    public <R> LazyList<R> map(Function<T, R> mapper) {
        return new LazyList<R>(() -> mapper.apply(head.get()),
                               () -> tail.get().map(mapper));
    }

    public T forEach(Consumer<T> consumer) {
        LazyList<T> list = this;

        while (!list.isEmpty()) {
            cosumer.accept(list.head.get());
            list = list.tail.get();
        }
    }
}

```

```
// Suppose we call
LazyList.iterate(0, i -> i < 2, i -> i + 1).map(f).map(g).forEach(c)
// where f and g are lambda expressions of type Function and c is a lambda
expression of type Consumer. Let e be the lambda expression i -> i + 1 passed to
iterate.
Sequence of which the lambda expressions are evaluated: fgce fgce (must repeat
twice)
```

```java
/* The method concat takes in two LazyList objects, l1 and l2, and creates a new
LazyList whose elements are all the elements of the first list l1 followed by all
the elements of the second list l2 */
public static <T> LazyList<T> concat(LazyList<T> l1, LazyList<T> l2) {
    if (l1.isEmpty()) {
        return l2;
    } else {
        return new LazyList<T>(l1.head, () -> concat(l1.tail.get(), l2));
    }
}
```