Lecture 7: Functions

Functions

While we have been using the terms functions and methods (and occasionally, procedure) interchangeably, we will now use the term function to refer to methods with specific properties.

A function, in mathematics, refer to a mapping from a set of inputs (*domain*) to a set of output values (*codomain*). We write . Every input in the domain must map to exactly one output but multiple inputs can map to the same output. Not all values in the codomain needs to be mapped. The set of elements in the codomain that is mapped is called the *image*.

Functions in programming language is the same as functions in mathematics. Given an input, the function computes and returns an output. A *pure* function does nothing else -- it does not print to the screen, write to files, throw exceptions, change other variables, modify the values of the arguments. We say that a pure function does not cause any *side effect*.
Some examples of non-pure functions:

```
int div(int i, int j) {
  return i / j;  // may throw an exception
}
int incrCount(int i) {
  return this.count + i; // assume that count is not final.
                         // this may give diff results for the same i.
}
void incrCount(int i) {
  this.count += i; // does not return a value
                   // and has side effects on count
}
int addToList(ArrayList queue, int i) {
  queue.add(i);  // has side effects on queue
}
```

In fact, in OO paradigm, we commonly need to write methods that update the fields of an instance or compute values using the fields of an instance. Such methods are not pure functions. While the notion of pure functions might seems restrictive, recall how many times your program has a bug that is related to incorrect side effects or unintended side effects? If we design and write our program with pure functions as much as possible, we could significantly reduce the number of bugs.

In mathematics, we say that a mapping is a *partial function* if not all elements in the domain are mapped. A common programming error is to treat a partial function like a function -- for instance, the `div` method above is written as if it is defined for all int values, but it is not defined when `j` is 0.

Mathematically, a function takes in only one value and return one value (e.g., `square` above). In programming, we often need to write functions that takes in more than one arguments (e.g., `add` above). We will see how to reconcile this later.

Let's explore functions in Java 8 by looking at the `Function` interface, it is a generic interface with two type parameters, `Function<T, R>`, `T` is the type of the inpuT, `R` is the type of the Result. It has one abstract method `R apply(T t)` that applies the function to a given argument.

Let's write a class that implements `Function`.

```
class Square implements Function<Integer, Integer> {
  public Integer apply(Integer x) {
    return x*x;
  }
}
```

To use it, we can:

```
int x = new Square().apply(4);
```

So far, everything is as you have seen before, and is significantly more complex than just writing:

```
int x = square(4);
```

So, what is the use of this? Consider now if we have a `List<Integer>` of integers, and we want to return another list where the elements is the square of the first list. We can write a method:

```
List<Integer> squareList(List<Integer> list) {
  List<Integer> newList = new ArrayList<Integer>();
  for (Integer i: list) {
    newList.add(square(i));
  }
  return newList;
}
```

Creating a new list out of an existing list is actually a common pattern. We might want to, say, create a list with the absolute values:

```
List<Integer> negativeList(List<Integer> list) {
  List<Integer> newList = new ArrayList<Integer>();
  for (Integer i: list) {
    newList.add(Math.abs(i));
  }
  return newList;
}
```

This is actually a common pattern. Applying the abstraction principles, we can generalize the method to:

```
List<Integer> applyList(List<Integer> list, Function<Integer,Integer> f) {
  List<Integer> newList = new ArrayList<Integer>();
  for (Integer i: list) {
    newList.add(f.apply(i));
  }
  return newList;
}
```

and call:

```
applyList(list, new Square());
```

to return a list of squares.

If we do not want to create a new class just for this, we can, as before, use an anonymous class:

```
applyList(list, new Function<Integer,Integer>() {
  Integer apply(Integer x) {
    return x * x;
  }
});
```

> **Map**
>
> The `applyList` method above is most commonly refered to as `map`.

Lambda Expression

The code is still pretty ugly, and there is much boiler plate code. The key line is actually Line 3 above, `return x * x`. Fortunately, Java 8 provides a clean way to write this:

```
applyList(list, (Integer x) -> { return x * x; });
applyList(list, x -> { return x * x; });
applyList(list, x -> x * x);
```

The expressions above, including `x -> x * x`, are *lambda expressions*. You can recognize one by the use of `->`. The left hand side lists the arguments (use `()` if there is no argument), while the right hand side is the computation. We do not need the type in cases where Java can refer the type, or need the return statements and the curly brackets.

> **lambda**
>
> Alonzo Church invented lambda calculus (-calculus) in 1936, before electronic computers, as a way to express computation. In -calculus, all functions are anonymous. The term lambda expression originated from there.

We can use lambda expressions just like any other values in Java. We have seen above that we can pass a lambda expression to a method. We can also assign lambda expression to a variable:

```
Function<Integer,Integer> square = x -> x * x;
square.apply(4);
```

Method Reference

We can use lambda expression to implement `applyList` with `abs()` method in `Math`.

```
applyList(list, x -> Math.abs(x));
```

If we look carefully at `abs()`, however, it takes in an `int`, and returns an `int`. So, it already fits the `Function<Integer,Integer>` interface (with autoboxing and unboxing). As such, we can refer to the method with a method reference: `Math::abs`. The code above can be simplified to:

```
applyList(list, Math::abs);
```

Again, we can assign method reference and pass them around like any other objects.

```
Function<Integer,Integer> f = Math::abs;
f.apply(-4);
```

Composing Functions

The `Function` interface has two default methods:

```
default <V> Function<T,V> andThen(Function<? super R,? extends V> after);
default <V> Function<V,R> compose(Function<? super V,? extends T> before);
```

for composing two functions. The term *compose* here is used in the mathematical sense (i.e., the $\circ$ operator in $f \circ g$).

These two methods, `andThen` and `compose`, return another function, and they are generic methods, as they have a type parameter `<V>`. Suppose we want to write a function that returns the square root of the absolute value of an int, we can write:

```
double SquareRootAbs(int x) {
  return Math.sqrt(Math.abs(x));
}
```

or, we can write either

```
Function<Integer,Integer> abs = Math::abs;
Function<Integer,Double> sqrt = Math::sqrt;
abs.andThen(sqrt)   //sqrt(abs(x))
```

or

```
sqrt.compose(abs)    //sqrt(abs(x))
```

But isn't writing the plain old method `SquareRootAbs()` clearer? Why bother with `Function`? The difference is that, `SquareRootAbs()` has to be written before we compile our code, and is fixed once we compile. Using the `Function` interface, we can compose functions at *run time*, dynamically as needed! Here is an example that you might be familiar with, from Lab 5:

```
Function<Customer, Queue> findQueueToSwitchTo;
if (numOfQueue > 1) {
  findQueueToSwitchTo = findShortestQueue.andThen(checkIfFewerInFront);
} else { // only one queue
  findQueueToSwitchTo = Customer::getQueue;  // no need to do anything
}
```

So instead of relying on the logic that the shortest queue is the same as the only queue and there is always the same number of customer in front if the customer is already is in the shortest queue, we just redefine the function that finds the queue to switch to to return the only queue.

Other Functions

Java 8 package `java.util.function` provides other useful interfaces, including:

- `Predicate<T>` [https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html] with a `boolean test(T t)` method
- `Supplier<T>` [https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html] with a `T get()` method
- `Consumer<T>` [https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html] with a `void accept(T t)` method
- `BiFunction<T,U,R>` [https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html] with a `R apply(T t, U u)` method

Other variations that involves primitive types are also provided.

Curried Functions

Functions have an *arity*. The `Function` interface is for unary functions that take in a single argument; the `BiFunction` inteface for binary functions, with two arguments. But we can have functions that take more than two arguments. We can, however, build functions that take in multiple arguments with only unary functions. Let's look at this mathematically first. Consider a binary function . We can introduce as a set of all functions , and rewrite as , of .

A trivial example for this is the `add` method that adds two `int` values.

```
int add(int x, int y) {
  return x + y;
}
```

This can be written as

```
Function<Integer, Function<Integer, Integer>> add = x -> y -> (x + y);
```

To calcualte 1 + 2, we call

```
add.apply(1).apply(2);
```

Let's break it down a litte, `add` is a function that takes in an `Integer` object and returns a unary `Function` over `Integer`. So `add.apply(1)` returns the function `y -> 1 + y`. We could assign this to a variable:

```
Function<Integer,Integer> incr = add.apply(1);
```

Here is the place where you need to change how you think: `add` is not a function that takes two arguments and return a value. It is a *higher-order function* that takes in a single argument, and return another function.

The technique that translates a general -ary function to a sequence of unary functions is called *currying*. After currying, we have a sequence of *curried* functions.

> **Curry**
>
> Currying is not related to food, but rather is named after computer scientist Haskell Curry, who popularized the technique.

Again, you might question why do we need this? We can simply call `add(1, 1)`, instead of `add.apply(1).apply(1)`? Well, the verbosity is the fault of Java instead of functional programming techniques. Other languages like Haskell or Scala have much simpler syntax (e.g., `add 1 1` or `add(1)(1)`).

If you get past the verbosity, there is another reason why currying is cool. Consider `add(1, 1)` -- we have to have both arguments available at the same time to compute the function. With currying, we no longer have to. We can evaluate the different arguments at different time (as `incr` example above). This feature is useful in cases where some arguments are not available until later. We can *partially apply* a function first. This is also useful if one of the arguments does not change often, or is expensive to compute. We can save the partial results as a function and continue applying later.

Again, using Lab 5 as example, you may have a method `serve` that takes in a `Customer` c and a `Server` s in a `Simulation` class. When a customer is created, you do not know who is the server is yet. You can partially apply the method first with the customer. When the customer is served, you apply it again with the server `s` as argument.

Lecture 8: Lambdas and Streams

Lambdas (continued)

Curried Functions

Functions have an *arity*. The `Function` interface is for unary functions that take in a single argument; the `BiFunction` inteface for binary functions, with two arguments. But we can have functions that take more than two arguments. We can, however, build functions that take in multiple arguments with only unary functions. Let's look at this mathematically first. Consider a binary function . We can introduce as a set of all functions , and rewrite as , of .

If you get past the verbosity, there is another reason why currying is cool. Consider `add(1, 1)` -- we have to have both arguments available at the same time to compute the function. With currying, we no longer have to. We can evaluate the different arguments at different time (as `incr` example above). This feature is useful in cases where some arguments are not available until later. We can *partially apply* a function first. This is also useful if one of the arguments does not change often, or is expensive to compute. We can save the partial results as a function and continue applying later.

Again, using Lab 5 as example, you can have several functions defined:

```
Function<Double, Function<Double,Double>> generateExponetialVariable =
    rate -> randDouble -> -Math.log(randDouble)/rate;
Function<Double,Double> generateServiceTime =
    generateExponentialVariable.apply(param.lambda);
Function<Double,Double> generateInterArrivalTime =
    generateExponentialVariable.apply(param.mu);
  :
```

Instead of keeping around the parameters, you could keep the functions to generate the random time as fields, and invoked them:

```
this.generateServiceTime.apply(rng.nextDouble());
```

Functional Interface

We are not limited to using lambda expression for the interfaces defined in `java.util.function`. We can use lambda expression as a short hand to a class that implements a interface with a single abstract method -- there has to be only one abstract method so that the compiler knows which method the lambda implements. This is more commonly known as SAM interface.
Note that a SAM interface can be mulitple methods, but only one can be abstract (others can have default implementation).

For instance, Java has the following interface:

```
interface Runnable {
  void run();
}
```

There is only one method, and it is abstract (no default implementation). So it is a valid SAM interface.

We can annotate a class with `@FunctionalInterface` to hint our intention to the compiler and to let the compiler helps us to catch unintended error (such as when we add a second abstract method to the interface).

We can define our own interface as well. For instance, in Lab 5, we can define:

```
@FunctionalInterface
interface FindQueueStrategy {
  CustomerQueue findQueue(Shop shop);
}

@FunctionalInterface
interface JoinQueueStrategy {
  void joinQueue(CustomerQueue queue, Customer c);
}
```

Now, we can avoid three `Customer` subclasses. We just need to instantiate a new `Customer` with different strategies:

We can set `FindQueueStrategy` to either `shop -> shop.getShortestQueue()` or `shop -> shop.getRandomQueue()` and set the JoinQueueStrategy to either `(q, customer) -> q.addToBack(customer)` or `(q, customer) -> q.addToFront(customer)`.

Lambda as Closure

Just like a local class an anonymous classes, a lambda can capture the variables of the enclosing scope. For instance, if you do not wish to generate the service time of a customer at the time of arrival, you can pass in a `Supplier` to `Customer` instead:

```
Customer c = new Customer(() -> -Math.log(rng.nextDouble())/rate);
```

Here, `rng` and `rate` are variables captured from the enclosing scope.

And just like in local and anonymous classes, a captured variable must be either explicitly declared as `final` or is effectively final.

A lambda expression therefore store more than just the function to invoke -- it also stores the data from the environment where it is defined. We call such construct which store a function together with the enclosing environments a *closure*.

Function as Delayed Data

Consider a function that produces new value or values. We can consider the function as a promise to give us the given data sometime later, when needed. For instance:

```
() -> -Math.log(rng.nextDouble())/rate)
```

is not the value of a service time, but rather, a supplier of the service time. When we need a service time, we just invoke the supplier.

What's the big deal? Not so much in the simple example above. But consider the case where the function is an expensive one. We can then delay the execution of the expensive function until it is absolutely needed. This allows us to do things that we couldn't before, for instance, create and manipulate an infinite list!

An Infinite List

How can we represent an infinite list? If we store the values of each element in the list, then we will run out of memory pretty soon. If we try to manipulate every element in the list, then we will enter an infinite loop.

The trick to building an infinite list, is to treat the elements in the list as *delayed data*, and store a function that generates the elements, instead of the elements itself.

We can think of an infinite list as two functions, the first is a function that generates the first element, and the second is a function that generates the rest of the list.

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

  public static InfiniteList<T> generate(Supplier<T> supply) {
    return new InfiniteList(supply,
      () -> InfiniteList.generate(supply));
  }
}
```

There you go! We now have an infinite list defined by the supply function.

A list defined this way is lazily evaluated. We won't get the elements until we need it -- this is in constrast to the eager `LambdaList` you write in Lab 6.

Let's see how to use this list. Consider the `findFirst` method:

```
public T findFirst(Predicate<T> predicate) {
  T first = this.head.get();
  if (predicate.test(first)) {
    return first;
  }
  InfiniteList<T> list = this.tail.get();
  while (true) {
    T next = list.head.get();
    if (predicate.test(next)) {
      return next;
    }
    list = list.tail.get();
  }
}
```

Simpler code

The code shown in the lecture above could be simplified to:

```
public T findFirst(Predicate<T> predicate) {
  InfiniteList<T> list = this;
  while (true) {
    T next = list.head.get();
    if (predicate.test(next)) {
      return next;
    }
    list = list.tail.get();
  }
}
```

In the method above, we repeatedly invoke the supplier, until we find an element that matches the predicate. This way, we never had to generate every element in the list just to find the first element that matches.

iterate

In class, I also showed the `iterate` method to generate a list:

```
public static <T> InfiniteList<T> iterate(T init, Function<T, T> next) {
  return new InfiniteList<T>(
      () -> init,
      () -> InfiniteList.iterate(next.apply(init), next)
    );
}
```

Stream

Such a list, possibly infinite, that is lazily evaluated on demand is also known as a *stream*. Java 8 provides a set of useful and powerful methods on streams, allowing programmers to manipulate data very easily. Java 9 adds a

couple of useful methods, `takeWhile` and `dropWhile`, which is also invaluable. To take full advantage of streams, we will be using Java 9, not Java 8 for the rest of this class.

Stream operations

A few things to note before I show you how to use streams. First, the operations on streams can be classified as either *intermediate* or *terminal*. An *intermediate* operation returns another stream. For instance, `map`, `filter`, `peek` are examples of intermediate operations. An intermediate operation does not cause the stream to be evaluated. A terminal operation, on the other hand, force the streams to be evaluated. It does not return a stream. `reduce`, `findFirst`, `forEach` are examples of terminal operation. A typical way of writing code that operate on streams is to chain a series of intermediate operation together, ending with a terminal operation.

Second, a stream can only be consumed once. We cannot iterate through a stream mulitple times. We have to create the stream again if we want to do that:

```
Stream<Integer> s = Stream.of(1,2,3);
s.count();
s.count(); // <- error
```

In the example above, we use the `of` static method with variable number of arguments to create a stream. We can also create a stream by:

- converting an array to stream using `Arrays.stream` method
- converting a collection to stream using `stream` method
- reading from a file using `Files.lines` method
- using the `generate` method (provide a `Supplier`) or `iterate` method (providing the initial value and incremental operation).

Self-note

```
static <T> Stream<T>    generate(Supplier<T> s)
```

Returns an infinite sequential unordered stream where each element is generated by the provided Supplier.

```
static <T> Stream<T>    iterate(T seed, UnaryOperator<T> f)
```

Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.

You have seen many of the stream operations before, in Lab 6, including `map`, `reduce`, `filter`, `findFirst`, `peek`, and `forEach`. Even though they are in the context of an eagerly evaluated list, the semantics are the same. Here are a few more useful ones.

- `flatMap` is just like `map`, but it takes in a function that produces another stream (instead of another element), and it `flattens` the stream by inserting the elements from the stream produced into the stream.

Let see an example. The lambda below takes a string and return a stream of `Integer` objects:

```
x -> x.chars().boxed()
```

Self-note

`String.chars()` returns `IntStream` as String implements CharSequence. `IntStream.boxed()` converts it into `Stream<Integer>`

We can create a stream of strings using the static `of` method from `Stream`:

```
Stream.of("live", "long", "and", "prosper")
```

If we chain the two together, using `map`, however, we will produce a stream of stream of `Integer`.

```
Steam.of("live", "long", "and", "prosper")
    .map(x -> x.chars().boxed())
```

To produce a stream of `Integer`s, we use `flatMap()`:

```
Stream.of("live", "long", "and", "prosper")
    .flatMap(x -> x.chars().boxed())
```

- `sorted` is an intermediate operation that returns a stream with the elements in the stream sorted. Without argument, it sorts according to the natural order. You can also passed in a `Comparator` to tell `sorted` how to sort.
- `distinct` is another intermediate operation that returns a stream with only distinct elements in the stream.

`distinct` and `sorted` are stateful operations -- it needs to keep track of states in order to perform the operation. `sorted` in particular, need to know every elements in the stream before it can output the result. They are also known as `bounded` operations, since call them on an infinite stream is a very bad idea!

Here is how we print out the unique characters of a given sequence of streams in sorted order

```
Stream.of("live", "long", "and", "prosper")
    .flatMap(x -> x.chars().boxed())
    .distinct()
    .sorted()
    .map(x -> new Character((char)x.intValue()))
    .forEach(System.out::print);
```

There are several intermediate operations that convert from infinite stream to finite stream.

- `limit` takes in an `int` and return a stream containing the first elements of the stream;
- `takeWhile` takes in a predicate and return a stream containing the elements of the stream, until the predicate becomes false. The resulting stream might still be infinite if the predicate never becomes false.

Here are more useful terminal operations:

- `noneMatch` return true if none of the elements pass the given predicate.
- `allMatch` return true if every element passes the given predicate.
- `anyMatch` return true if no elements passes the given predicate.

Example 1: Is this a prime?

Consider the method below, which checks if a given `int` is a prime:

```
1  boolean isPrime(int x) {
2    for (i = 2; i <= x-1; i++) {
3      if (x % i == 0) {
4        return false;
5      }
6    }
7    return true;
8  }
```

The code coudln't be simpler -- or can it? With streams, we can write it as:

```
1  boolean isPrime(int x) {
2    return IntStream.range(2, x)    //exclusive end
3        .matchNone(x % i == 0);
4  }
```

Bug

`IntStream.range(inclusive_start, exclusive_end)` `IntStream.rangeClosed(inclusive_start, inclusive_end)`

What if we want to print out the first 500 prime numbers, starting from 2? Normally, we would do the following:

```
1  void fiveHundredPrime() {
2    int count = 0;
3    int i = 2;
4    while (count < 500) {
5      if (isPrime(i)) {
6        System.out.println(i);
7        count++;
8      }
9      i++;
10   }
11 }
```

The code is still considered simple, and understandable for many, but I am sure some of us will encounter a bug the first time we write this (either forgot to increment counter, or put the increment in the wrong place). If you look at the code, there are a couple of compnents:

- Lines 3 and 9 deal with iterating through different numbers for primality testing
- Line 4 is the test
- Lines 2, 4, and 7, deal with limiting the output to 500 primes
- Line 5 is the action to perform on the prime

With streams, we can write it like the following:

```
1  IntStream.iterate(2, x -> x+1)
2      .filter(x -> isPrime(x))
3        .limit(500)
4        .forEach(System.out::println);
```

Notice how each of the four components matches neatly with one operation on stream!

With stream, we no longer have to write loops -- as you have done in Lab 6, we have move the iterations to within each operation in stream. We no longer need to maintain states and counters, they are done within each operation as needed as well. This has another powerful implication: our code become more *declarative*, we only need to concern about what we want at each step, much less about how to do it.

You should take a look at the methods provided by the `Stream` class, and read through the APIs, a few times, they formed fundamental building blocks for writing data processing code in functional style in Java.

Lecture 9: Functors, Monads, Collectors

Functor

In this lecture, we are going to abstract out some useful patterns that we have seen so far in functional-style programming in Java, and relates it to concepts in functional programming.
Once you see and understand the patterns, hopefully you can reapply the patterns in other context!

Let's start with a simplest one, called *functor*. This funny name originated from a branch of mathematics, called category theory. We can think of a functor as something that takes in a function and returns another functor. If you like, you can think of it as something that implements an interface that looks like:

```
interface Functor<T> {
  public <R> Functor<R> f(Function<T,R> func);
}
```

Wait, that's a recursive definition, and doesn't really explain what is a functor? In OO-speak, a functor can be any class that implements the interface above (or matches the pattern above).

So, is this a functor?

```
class A {
  private int x;

  public A(int i) {
    this.x = i;
  }

  public A f(Function<Integer,Integer> func) {
    if (this.x > 0) {
      return new A(func.apply(x));
    } else {
      return new A(0);
    }
  }

  public boolean isSameAs(A a) {
    return this.x == a.x;
  }
}
```

The class `A` above takes in a function and returns another `A` with `func` applied on the content `x`, if `x` is positive. Otherwise, it returns another `A` with 0.

Despite that it does not implement the interface `Functor`[1][#fn:1], it does match the pattern of having a method that takes in a function and returns itself, it is a special case since both `R` and `T` are `Integer`.

Matching the patterns syntactically, however, is not enough to be a functor. A functor have to semantically obey the functor laws, which are:

- if `func` is an identity function `x -> x`, then it should not change the functor.
- if `func` is a composition of two functions , then the resulting functor should be the same as calling `f` with and then with .

Let's check:

```
A a = new A(-1);
a.isSameAs(a.f(x -> x));
a.f(x -> x + 1).g(x -> x * 2).isSameAs(a.f(x -> (x + 1) * 2);
```

Nope. Our class `A` violates the first functor law and therefore does not qualify to be a functor. A simpler version, without all the weird stuff going on in `f`, is, however, a valid functor.

```
class B {
  private int x;

  public B(int i) {
    this.x = i;
  }

  public B f(Function<Integer,Integer> func) {
    return new B(func.apply(x));
  }

  public boolean isSameAs(B a) {
    return this.x == a.x;
  }
}
```

It is easy to see that if `func` is `x -> x`, then `B(func.apply(x))` is just `B(x)`. Further, if `func` is `g.compose(h)`, then calling `func.apply(x)` is the same as `g.apply(h.apply(x))`.

Another way to think of a functor, in the OO-way, is that that it is a variable wrapped within a class in some context. Instead of manipulating the variable directly, we pass in a function to the class to manipulate the variable. The variable must then interact with the function as if it is not be wrapped. The class should not interfere with the function (as in the class `A`).

You have actually seen several functors before. You might recognize by now that `func` is just our old friend `map`! A `LambdaList` is just a functor with a list of variables stored in an array list. A `Stream` is another functor. So is `InfiniteList`.

> **Self-note**

> In short terms, a `Functor` is something that is `mappable` without any 'harmful effects' and has a function that does `map`.

Once you understand the laws of functor and recognize this pattern, it is easy to learn about new classes -- one just have to tell you that it is a functor, and you will know how the class should behave.

> **Functors in other languages**
>
> Haskell, Scala, Python, Javascript, and other functional languages have functors as well. C++, unfortunately, uses the term functors to mean function object -- a function object is not a functor in the sense of the word in category theory. So, do not get confused between the two.

Optional

Let's see another functor in Java 8: the `Optional` class. Recall that you can wrapped a possibly `null` object in an `Optional` class. It is unfortunate that Java 8 provides a `get()` method to allow retrieval of the object inside -- it is convenient but that defeats the point of `Optional` -- not to mentioned that Java Collections Framework does not support `Optional`. But, since Java's `Optional` is a functor, we can manipulate the value[2] wrapped in an `Optional` with the `map` function, without having to `get()` and put back again!

Let's consider the `Simulator` again. In a better version of Java, we would have a `PriorityQueue<T>` with a `poll` method that returns `Optional<T>`, instead of either an object of type `T` or `null`. Let's pretend that we have a different `poll`, called `optionalPoll` that does that. Now, we can process the event returned (maybe?) in the following way:

```
events.optionalPoll()
    .filter(event -> event.happensBefore(sim.expireTime()))
    .map(event -> sim.handle(event))
    .ifPresent(eventStream -> this.schedule(eventStream));
```

This beats writing code that looks like this:

```
Event event = events.poll();
if (event != null) {
  if (event.happensBefore(sim.expireTire())) {
    Stream<Event> eventStream = sim.handle(event);
    if (eventStream != null) {
      this.schedule(eventStream);
    }
  }
}
```

`Optional` helps us check for `null` and takes care of the "maybe?" for us. In `map`, if `event` is `null`, it does nothing, otherwise it invokes `sim.handle` and returns an `Optional<Steam<Event>>`.

> **In other languages**
>
> Scala has `Option`; Haskell has `Maybe`. If you use Python, check out the `PyMonad` library that supplies various functors and monad, including `Maybe`.

Monad

Monad is another funny name originated from category theory. A monad also takes in a function and returns a monad. But, unlike functor, it takes in a function that returns a monad!

```
interface Monad<T> {
  public <R> Monad<R> f(Function<T,Monad<R>> func);
}
```

Looks complicated? How about now:

```
interface Stream<T> {
  public <R> Stream<R> flatMap(Function<T,Stream<R>> mapper);
}
```

This interface should look familiar to you[3]. We have seen monads before! A Stream is a monad. In contrast, unless you implemented `flatMap` for `InfiniteList` or `LambdaList`, they are not monads.

Just like functors, there are some laws that a monad have to follow:

- there should be an `of` operation that takes an object (or multiple objects) and wrap it/them into a monad. Further,
  - `Monad.of(x).flatMap(f)` should be equal to `f(x)` (called the *left identity* law)
  - `monad.flatMap(x -> Monad.of(x))` should be equal to `monad` (called the *right identity* law)
- the `flatMap` operation should be associative (associative law): `monad.flatMap(f).flatMap(g)` should be equal to `monad.flatMap(x -> f(x).flatMap(g))`

> **In other languages**
>
> The `flatMap` and `of` operations are sometimes known as the `bind` and `unit` operations respectively (e.g., in Haskell).

Knowing what is a monad is useful, since if I tell you something is a monad, you should recognize that it supports a given interface. For instance, I tell you that `Optional` is a monad. You should know that `Optional` supports the `of` and `flatMap` operation (maybe of a different name, but they exists and follows the monad laws).

We won't proof formally that `Optional` follows the laws of monad in this class, but let's explore a bit more to convince ourselves that it does. Let's write the `flatMap` method for `Optional`, which is not that difficult:

```
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
  if (!isPresent()) {
    return empty();
```

```
    } else {
      return mapper.apply(value);
    }
  }
```

Let check:

- Left identity law: `Optional.of(1).flatMap(f)` will return `f.apply(1)` (i.e.,).
- Right identity law: `opt.flatMap(x -> Optional.of(x)` will apply `x -> Optional.of(x)` on the value of `opt`, if it exists, resulting in `Optional.of(value)`, which is `opt`. If the value does not exist (`Optional` is empty), then `flatMap` will not apply the lambda, instead it will return `empty()` right away. So it obeys the law.
- Associative law: `opt.flatMap(f).flatMap(g)` is the same as `f.apply(value).flatMap(g)`; `opt.flatMap(x -> f(x).flatMap(g))` will apply the lambda to `value`, so we get `f.apply(x).flatMap(g)`. They are the same. If `opt` is empty, then `flatMap` returns `empty` for both cases.

So, despite the complicated-sounding laws, they are actually easy to obey!

Collectors

Going back to streams now. We previously have seen several terminal operations of streams which are useful and general, such as `reduce` and `forEach`. Java 8, however, provide something more powerful called `collect`, which you can think of as `reduce` on steroids! You have seen this used to convert a stream into a `List` collection using `.collect(Collectors.toList())`. Here are a few, self-explanatory examples using `collect` and predefined `Collector`.

```
Map<Server, List<Customer>> byServer = customers.stream()
    .collect(Collectors.groupingBy(c -> c.servedBy());
```

The code above put the list of customers into a map collection, with server as the key. We can further find out all customers that ever been served by a given server.

```
Map<Boolean, List<Customer>> byServiceTime = customer.stream()
    .collection(Collectors.partitionBy(c -> c.getServiceTime() < 1));
```

The code above partition the customer into two, those that require less than 1 unit time of service that those that require more.

And how about this one that computes the average waiting time:

```
double avgWaitingTime = customer.stream()
    .collection(Collectors.averagingDouble(c -> c.getWaitingTime()));
```

You can take a look at the list of predefined `Collectors` [https://docs.oracle.com/javase/9/docs/api/java/util/stream/Collectors.html] in Java documentation to see what is available.

The Collector Interface

Let's delve deeper and try to understand what exactly is a `Collector`? A `Collector` is defined as follows:

```
interface Collector<T, A, R> {
  BiConsumer<A, T> accumulator();
  BinaryOperator<A> combiner();
  Function<A, R> finisher();
  Supplier<A> supplier();
   :
}
```

It is helpful to understand the type first:

- `T` is the type of the elements we are collecting
- `R` is the type of the result of the collection
- `A` is the type of the partial result from the accumulator (ala reduction)

Each of these methods shown above in the `Collector` is returning a function that will be invoked by the `collect` method. Except combiner, the other three methods are rather straightforward: the `supplier` supplies a "container" of type `A` for the `accumulator` to accumulate into, and finally the `finisher` converts the container into the result of type `R`.

Let's see the `Collector` in action by rewriting the `Collectors.toList()` constructor:

```
class ListCollector<T> implement Collector<T, List<T>, List<T>> {
  Supplier<List<T>> supplier() {
    return () -> new ArrayList<T>();
  }

  BiConsumer<List<T>, T> accumulator() {
    return (list, item) -> list.add(item);
  }

  Function<List<T>,List<T>> finisher() {
    return Function.identity();
  }

   :
}
```

Now, let's discuss what `combiner` does. A `combiner` is actually required by the `Stream` `reduce` method as well:

```
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator,
```

```
    BinaryOperator<U> combiner)
```

But, in our `LambdaList` and `InfiniteList`, we have not been using a `combiner`.

A `combiner` is useful in the context of parallel processing -- which we will cover more in the coming lectures. The short version of it is that, a stream can be broken up into substreams and process independently (e.g., reduced or collected independently). After these independent processing, we will need to combined their results back together. This is where combiner comes in -- it specifies how to combined partial results back into first result.

Let's see two examples: In the case of our `ListCollector`, we will get one `List<T>` each from each substream after collection. We just need to combine both into the first list:

```
BinaryOperator<List<T>> combiner() {
  (list1, list2) -> {
    list1.addAll(list2);
    return list1;
  }
}
```

In the case of reducing a stream, suppose we want to find the product of all numbers:

```
Stream.of(1,2,3,4).reduce(1, (x,y)->x*y, (x,y)->x*y);
```

We include the lambda `(x,y)->x*y` twice, the second one is a `combiner` that combines two partial product into one, by multiplying them. Here is one that count the number of elements:

```
Stream.of(1,2,3,4).reduce(0, (x,y)->x+1, (x,y)->x+y);
```

To combine two partial counter, we add them with `(x,y)->x+y`.

Another way we can create a customized collector is to pass lambdas into `collect` method of `Stream` directly.

```
s.collect(()->new LinkedList<Integer>(),
    (l,i)->l.add(i),
    (l1, l2) -> l1.addAll(l2));
```

The `collect` method takes in a `Supplier<R>` supplier, `BiConsumer<R,T>` accumulator, and a `BiConsumer<R,R>` combiner. Note that `combiner` here is a `BiConsumer` which expects the results to be combined into the first argument, and is different from the the combiner of `Collector` interface, which is a `BiOperator`. Further, there is no finisher so we can't specify anything that requires a more complex finisher (example, an averaging collector).

---

1. In fact, no functors in Java 8 does, since this is the interface I created just to explain the pattern of a functor.
2. To be more precise, create a new `Optional` with the manipulated value.

Lecture 10: Parallel Streams

Parallel and Concurrent Programming

So far, the programs that we have written in CS2030 run *sequentially*. What this means is that at any one time, there is only one instruction of the program running on a processor.

What is concurrency?

A single core processor can only execute one instruction at one time -- this means that only one *process* (or less precisely speaking, one application) can run at any one time. Yet, when we use the computer, it *feels* as if we are running multiple processes at the same time. The operating system, behind the scene, is actually switching between the different processes, to give the user an illusion that they are running at the same time.

We can write a program so that it runs concurrently -- by dividing the computation into subtasks called *threads*. The operating system, behind the scene, can switch between the different threads, to give the user an illusion that the threads are running at the same time. Such multi-threads programs are useful in two ways: (i) it allows us, the programmers, to separate the unrelated tasks into threads, and write each thread separately; (ii) it improves the utilization of the processor. For instance, if I/O is in one thread, and UI rendering is in another, then when the processor is waiting for I/O to complete, it can switch to the rendering thread to make sure that the slow I/O does not affect the responsiveness of UI.

What is parallelism?

While 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 -- either we have a processor that is capable of running multiple instructions at the same time, or we have multiple cores / processors and dispatch the instructions to the cores / processors so that they are executed at the same time.

All parallel programs are concurrent, but not all concurrent programs are parallel.

Modern computers have more than one cores / processors [1][#fn:1]. As such, the line between parallelism and concurrency is blurred.

Parallel computing

Parallel computing is one of the major topics in computer science. One can teach a whole module (or a focus area) on this topic alone. The goal of this lecture is not to cover it in depth, but is to expose students in CS2030 to the concept of parallel computing in relation to the stream abstraction in Java 8.

Parallel Stream

We have seen that Java `Stream` class is a powerful and useful class for processing data in declarative style. But, we have not fully unleash the power of `Stream`. The neatest thing about `Stream` is that it allows parallel operations on the elements of the stream in one single line of code.

Let's consider the following program that prints out all the prime numbers between 1 and 999,999.

```
    IntStream.range(1, 1_000_000)
        .filter(x -> isPrime(x))
        .forEach(System.out::println);
```

We can parallelize the code by adding the call `parallel()` into the stream.

```
    IntStream.range(1, 1_000_000)
        .filter(x -> isPrime(x))
        .parallel()
        .forEach(System.out::println);
```

You may observe that the output has been reordered, although the same set of numbers are still being produced. This is because `Stream` has broken down the numbers into subsequences, and run `filter` and `forEach` for each subsequences in parallel. Since there is no coordination among the parallel tasks on the order of the printing, whichever parallel tasks that complete first will output the result to screen first, causing the sequence of numbers to be reordered.

If you want to produce the output in the order of input, use `forEachOrdered` instead of `forEach`, we will loose some benefits of parallelization because of this.

Suppose now that we want to compute the number of primes below 1,000,000. We can run:

```
    IntStream.range(1, 1_000_000)
        .filter(x -> isPrime(x))
        .parallel()
        .count()
```

The code above produce the same output regardless of it is being parallelized or not.

Note that the task above is stateless and does not produce any side effect. Furthermore, each element is processed individually without depending on other elements. Such computation is sometimes known as *embarrassingly parallel*. The only communication needed for each of the parallel subtask is to combine the result of `count()` from the subtasks into the final count (which has been implemented in `Stream` for us).

How to parallelize a stream

You have seen that adding `parallel()` to the chain of calls in a stream enables parallel processing of the stream. Note that `parallel()` is a lazy operation -- it merely marks the stream to be process in parallel. As such, you can insert the call to `parallel()` anywhere in the chain.

> **sequential()**
>
> There is a method `sequential()` which marks the stream to be process sequentially. If you call both `parallel()` and `sequential()` in a stream, the last call "wins". The example below processes the stream sequentially:
>
> ```
> s.parallel().filter(x -> x < 0).sequential().forEach(..);
> ```

Another way to create a parallel stream is to call the method `parallelStream()` instead of `stream()` of the `Collector` class. Doing so would create a stream that will be processed in parallel from the collection.

What can be parallelized?

To ensure that the output of the parallel execution is correct, the stream operations must not *interfere* with the stream data, and most of time must be *stateless*. Side-effects should be kept to minimum.

Interference

Interference means that one of the stream operation modifies the source of the stream during the execution of the terminal operation. For instance:

```
List<String> list = new ArrayList<>(Arrays.asList("Luke", "Leia", "Han"));
list.stream()
    .peek(name -> {
         if (name.equals("Han")) {
           list.add("Chewie"); // they belong together
         }
      })
    .forEach(i -> {});
```

Would cause `ConcurrentModificationException` to be thrown. Note that this non-interference rule applies even if we are using `stream()` instead of `parallelStream()`.

Stateless

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

For instance, the `generate` and `map` operations below are stateful, since they depend on the events in the queue and the states of the shops. Parallelizing this may lead to incorrect output. To ensure that the output is correct, additional work needs to be done to ensure that state updates are visible to all parallel subtasks.

```
Stream.generate(this.events::poll)
    .takeWhile(event -> event != null)
    .filter(event -> event.happensBefore(sim.expireTime()))
    .peek(event -> event.log())
    .map(event -> sim.handle(event))
    .forEach(eventStream -> this.schedule(eventStream));
```

Side Effects

Side-effects can lead to incorrect results in parallel execution. Consider the following code:

```
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));
```

The `forEach` lambda generates a side effect -- it modifies `result`. `ArrayList` is what we call a non-thread-safe data structure. If two threads manipulate it at the same time, incorrect result may result.

If we use `Collectors.toList()` instead, we can achieve the intended result without visible side effects.

Associativity

The `reduce` operation is inherently parallelizable, as we can easily reduce each sub-streams and then use the `combiner` to combine the results together. Recall this example from Lecture 9:

```
Stream.of(1,2,3,4).reduce(1, (x,y)->x*y, (x,y)->x*y);
```

There are several rules that the `identity`, the `accumulator` and the `combiner` must follow:

- `combiner.apply(identity, i)` must be equal to `i`.
- The `combiner` and the `accumulator` must be associative -- the order of applying must not matter.
- The `combiner` and the `accumulator` must be compatible -- `combiner.apply(u, accumulator.apply(identity, t))` must equal to `accumulator.apply(u, t)`

The multiplication example above meetings the three rules:

- `i * 1` equals `i`
- `(x * y) * z` equals `x * (y * z)`
- `u * (1 * t)` equals `u * t`

Performance of Parallel Stream

Let's go back to:

```
IntStream.range(1, 1_000_000)
    .filter(x -> isPrime(x))
    .parallel()
    .count()
```

How much time can we save by parallelizing the code above?

Let's use the `Instant` [https://docs.oracle.com/javase/9/docs/api/java/time/Instant.html] and `Duration` [https://docs.oracle.com/javase/9/docs/api/java/time/Duration.html] class from Java to help us:

```
    Instant start = Instant.now();
    long howMany = IntStream.range(1,1000000)
        .filter(x -> isPrime(x))
        .parallel()
        .count();
    Instant stop = Instant.now();
    System.out.println(howMany + " " + Duration.between(start,stop).toMillis() + " ms");
```

The code above measures roughly the time it takes to count the number of primes below 1,000,000. On my iMac, it takes about 300-320 ms. If I remove `parallel()`, it takes about 500 ms. So we gain about 36 - 40% performance.

Can we parallelize some more? Remember how we implement `isPrime` [2] [#fn:2]

```
  boolean isPrime(int n) {
    return IntStream.range(2, (int)Math.sqrt(n) + 1)
        .noneMatch(x -> n % x == 0);
  }
```

Let's parallelize this to make this even faster!

```
  boolean isPrime(int n) {
    return IntStream.range(2, (int)Math.sqrt(n) + 1)
        .parallel()
        .noneMatch(x -> n % x == 0);
  }
```

If you run the code above, however, you will find that the code is not as fast as we expect. On my iMac, it takes about 12.7s, about 25 times slower!

Parallelizing a stream does not always improve the performance.

What is going on? To understand this, we have to delve a bit deeper into how Java implements the parallel streams.

Thread Pools and Fork/Join

Internally, Java maintains pool of *worker threads*. A worker thread is an abstraction for running a task. We can submit a task to the pool for execution, the task will join queue. The worker thread can pick a task from the queue to execute. When it is done, it pick another task, if one exists in the queue, and so on -- not unlike our `Server` (worker thread) and `Customer` (task).

A `ForkJoinPool` is a class the implements a thread pool with a particular semantic --- the task that the worker runs must specify `fork` -- how to create subtasks, and `join` -- how to merge the results from the subtasks.

In the case of a parallel stream, `fork` will create subtasks running the same chain of operations on sub-streams, and when done, run `join` to combine the results (e.g., `combiner` for `reduce` is run in `join`). `fork` and `join` can be recursive -- for instance, a `fork` operation can split the stream into two subtasks. The subtasks can further split the sub-streams into four smaller sub-streams, and so on, until the size of the sub-stream is small enough that the task is actually invoked.

To define a task, we subclass from `RecursiveTask<T>` (if the task returns a value of type `T`) or `RecursiveAction` (if the task does not return a value).

Here is an example task that we can submit to ForkJoinPool:

```
1   static class BinSearch extends RecursiveTask<Boolean> {
2     final int FORK_THRESHOLD = 2;
3     int low;
4     int high;
5     int toFind;
6     int[] array;
7
8     BinSearch(int low, int high, int toFind, int[] array) {
9       this.low = low;
10      this.high = high;
11      this.toFind = toFind;
12      this.array = array;
13    }
14
15    @Override
16    protected Boolean compute() {
17      // stop splitting into subtask if array is already small.
18      if (high - low < FORK_THRESHOLD) {
19        for (int i = low; i < high; i++) {
20          if (array[i] == toFind) {
21            return true;
22          }
23        }
24        return false;
25      }
26
27      int middle = (low + high)/2;
28      BinSearch left = new BinSearch(low, middle, toFind, array);
29      BinSearch right = new BinSearch(middle, high, toFind, array);
30      left.fork();
31      return right.compute() || left.join();
32    }
33  }
```

To run the task, we call the `invoke` method of `ForkJoinPool`, which executes the given task immediately and return the result.

```
1     BinSearch searchTask = new BinSearch(0, array.length, 12, array);
2     boolean found = ForkJoinPool.commonPool().invoke(searchTask)
```

The task above recursively search for an element in the left half and right half of the array. Note that this is similar to, but is NOT binary search. Binary search of course just search in either the left or the right side, depending on the middle value, and is not parallel.

ForkJoinPool overhead

In the example above, you can see that creating subtasks incur some overhead (new task objects, copying of parameters into objects, etc). In the `isPrime` example above, the task is trivial (checking `n % x == 0`), and so, by parallelizing it, we are actually creating more work for Java to do! It is much more efficient if we simply check for `n % x == 0` sequentially.

The moral of the story is, parallelization is worthwhile if the task is complex enough that the benefit of parallelization outweighs the overhead. While we discuss this in the context of parallel streams, this principle holds for all parallel and concurrent programs.

Ordered vs. Unordered Source

Whether or not the stream elements are *ordered* or *unordered* also plays a role in the performance of parallel stream operations. A stream may define an *encounter order*. 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.

Some stream operations respect the encounter order. For instance, both `distinct` and `sorted` preserve the original order of elements (if ordering is preserved, we say that an operation is *stable*).

The parallel version of `findFirst`, `limit`, and `skip` can be expensive on ordered stream.

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.

The following, for example, takes about 700 ms on my iMac:

```
1     Stream.iterate(0, i -> i + 7)
```

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

But, with `unordered()` inserted, it takes about 350ms, a 2x speed up!

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

Collectors

To wrap up, we will revisit the `Collector` class. Recall that `collect` is a mutable version of `reduce` on `Stream`. Being mutable, parallelizing it is tricky. Luckily, we only need to provide hints to the Collector class, which will optimize the implementation for us. The hint is given as part of the `characteristics()` method (the last method that we did not cover last week).

The `characteristics()` method returns a `Set` that contains a combination of three enums:

- `CONCURRENT` to indicate that the container that the supplier created can support accumulator function being called concurrently from multiple threads,
- `IDENTITY_FINISH` to indicate the the finisher function is the identity function, and can be skipped.
- `UNORDERED` to indicate that the collection operation does not necessary preserve the encounter order of the elements.

The operation `collect` will only be parallelized if the following three conditions hold:

- The stream is parallel
- The collector has characteristic `CONCURRENT`
- The stream is unordered or has characteristic `UNORDERED`

Of course, if we tell the collector has the characteristic `CONCURRENT`, the container that we use must actually supports that! Luckily for us, Java `java.util.concurrent` package provides many collections that support concurrency, including `CopyOnWriteArrayList`, `ConcurrentHashMap`, etc. Obviously these are more expensive. For instance, `CopyOnWriteArrayList` creates a fresh copy of the underlying array whenever there is a mutative operation (e.g., `add`, `set`, etc), not unlike your `LambdaList`.

Again, this is the overhead cost of parallelization -- the cost that might not outweigh the benefit of parallelization, and should be considered carefully.

---

11. Asynchronous Programming

Learning Objectives

Synchronous vs. Asynchronous

In synchronous programming, when we call a method, we expect the method to be executed, and when the method returns, the result of the method is available.

```
int multiple(int x, int y) {
    return x * y;
}

int z = multiple(3, 4);
```

In the simple example above, our code continues executing after, and only after `add()` completes.

If a method takes a long time to run, however, the execution will delay the execution of subsequent methods, and maybe undesirable.

Asynchronous call to a method allows execution to continue immediately after calling the method, so that we can continue executing the rest of our code, while the long-running method is off doing its job.

You have seen examples of asynchronous calls:

```
    task = new MatrixMultiplyerTask(m1, m2);
    task.fork();
```

The call above returns immediately even before the matrix multiplication is complete. We can later return to this task, and call `task.join()` to get the result (waiting for it if necessary).

A `RecursiveTask` also has a `isDone()` method that it implements as part of the `Future` interface. Now, we can do something like this:

```
    task = new MatrixMultiplyerTask(m1, m2);
    task.fork();
    while (!task.isDone()) {
        System.out.print(".");
        Thread.sleep(1000);
    }
    System.out.print("done");
```

So, while the task is running, we can print out a series of "."s to feedback to the users to indicate that it is running.

`Thread.sleep(1000)` cause the current running thread to sleep for 1s. It might throw an `InterruptedException`, if the user interrupts the program (by Control-C). To complete the snippet, we should catch the exception and cancel the task.

```
    task = new MatrixMultiplyerTask(m1, m2);
    task.fork();
    try {
        while (!task.isDone()) {
            System.out.print(".");
            Thread.sleep(1000);
        }
        System.out.println("done");
    } catch (InterruptedException e) {
        task.cancel();
        System.out.println("cancelled");
    }
```

Future

Let's look at the `Future` interface a bit more. `Future<T>` represents the result (of type `T`) of an asynchronous task that may not be available yet. It has five simple operations:

- `get()` returns the result of the computation (waiting for it if needed).
- `get(timeout, unit)` returns the result of the computation (waiting for up to the timeout period if needed).
- `cancel(interrupt)` tries to cancel the task -- if `interrupt` is true, cancel even if the task has started. Otherwise, cancel only if the task is still waiting to get started.
- `isCancelled()` returns `true` of the task has been cancelled.
- `isDone()` returns `true` if the task has been completed.

Both `RecursiveTask` and `RecursiveAction` implements the `Future` interface, so you can use the above methods on your tasks.

> **In Other Languages**
>
> Scala's `Future` is more powerful -- it allows us to specify what to do when the task completes, and it hands abnormal completions (e.g., exceptions). Python 3.2 supports `Future` through `concurrent.futures` [https://docs.python.org/3/library/concurrent.futures.html] module. C++11 supports `std::future`](http://en.cppreference.com/w/cpp/thread/future [http://en.cppreference.com/w/cpp/thread/future]) as well.

CompletableFuture

The example code above tries every second to see if task is done. For some applications, the response time is critical, and we would like to know as soon as a task is done. For instance, response time is important in stock trading applications and web services.

One way to do so, is to sleep for a shorter duration. Or even not sleeping all together:

```
    task.fork();
    while (!task.isDone()) {
        System.out.print(".");
    }
    System.out.print("done");
```

This is problematic in many ways, besides printing out too many dots:

- this is known as *busy waiting* -- and it occupies the CPU while doing nothing. Such code should be avoided at all cost.
- we may want to continue doing other things besides printing out "."s, so the code won't be a simple for loop anymore. We can do something like this instead:

```
    task.fork();
    if (!task.isDone()) {
        // do something
    } else {
        task.join();
    }
    if (!task.isDone()) {
        // do something else
    } else {
        task.join();
    }
    if (!task.isDone()) {
        // do yet something else
    } else {
        task.join();
    }
```

You can see that the code gets out of hand quickly, and this is only if we have one asynchronous call!

What we need is have a way to specify a *callback*. A callback is basically a method that will be executed when a certain event happens. In this case, we need to specify a callback when an asynchronous task is complete. This way, we can just call an asynchronous task, specify what to do when the task is completed, and forget about it. We do not need to check again and again if the task is done.

Java 8 introduces the class `CompletableFuture`, which implements the `Future` interface, and allows us to specify an asynchronous task, and an action to perform when the task completes.

To create a `CompletableFuture` object, we can call one of its static method. For instance, `supplyAsync` takes in a `Supplier`:

```
CompletableFuture<Matrix> future = CompletableFuture.supplyAsync(() -> m1.multiply(m2));
```

To specify the callback, we can use the `thenAccept` method, which takes in a consumer:

```
future.thenAccept(System.out::println);
```

Or, you can use the oneliner:

```
CompletableFuture
    .supplyAsync(() -> m1.multiply(m2))
    .thenAccept(System.out::println);
```

CompletableFuture is a Functor / Monad

`CompletableFuture` is a functor. Recall that a functor, in OO-speak, is a class that implements a (hypothetical) interface that looks like the following:

```
interface Functor<T> {
  public <R> Functor<R> f(Function<T,R> func);
}
```

In `CompletableFuture`, the method that makes `CompletableFuture` a functor is the `thenApply` method:

```
<U> CompletableFuture<U> thenApply(Function<? super T,? extends U> func)
```

The method `thenApply` is similar to `thenAccept`, except that instead of a `Consumer`, the callback that gets invoked when the asynchronous task completes is a `Function.

There are other variations:

- `thenRun`, which takes a `Runnable`,
- `thenAcceptBoth`, which takes a `BiConsumer` and another `CompletableFuture`
- `thenCombine`, which takes a `BiFunction` and another `CompletableFuture`
- `thenCompose`, which takes in a `Function` `fn`, which instead of returning a "plain" type, `fn` returns a `CompletableFuture`.

All the methods above return a `CompletableFuture`.

BTW, `CompletableFuture` is a monad too! The `thenCompose` method is analougous to the `flatMap` method of `Stream` and `Optional`.

This also means that `CompletableFuture` satisfies the monad laws, one of which is that there is a method to wrap a value around with a `CompletableFuture`. We call this the `of` method in the context of `Stream` and `Optional`, but in `CompletableFuture`, it is called `completedFuture`.

Being a functor and a monad, `CompletableFuture` objects can be chained together, just like `Stream` and `Optional`. We can write code like this:

```
CompletableFuture
    .completedFuture(Matrix.generate(nRows, nCols, rng::nextDouble))
    .thenApply(m -> m.multiply(m1))
    .thenApply(m -> m.add(m2))
    .thenApply(m -> m.transpose)
    .thenAccept(System.out::println);
```

Another example:

```
CompletableFuture left = CompletableFuture
    .supplyAsync(() -> a1.multiply(b1));
CompletableFuture right = CompletableFuture
    .supplyAsync(() -> a2.multiply(b2))
    .thenCombine(left, (m1, m2) -> m1.add(m2));
    .thenAccept(System.out::println);
```

Similar to `Stream`, some of the methods are terminal (e.g., `thenRun`, `thenAccept`), and some are intermediate (`thenApply`).

Variations

- There are variations of methods with name containing the word `Either` or `Both`, taking in another `CompletableFuture`. These methods invoke the given `Function` / `Runnable` / `Consumer` when either one (for `Either`) or both (for `Both`) of the `CompletableFuture` completes.
- There are variations of methods with name ending with the word `Async`. These methods are called asynchronously in another thread

For example, `runAfterBothAsync(future, task)` would run `task` only after `this` and given `future` is completed.

Other features of `CompletableFuture` include:

- Some methods take in additional `Executor` parameter, for cases where running in the default `ForkJoinPool` is not good enough.
- Some methods takes in additional `Throwable` parameter, for cases where earlier calls might throw an exception.

The table [http://www.codebulb.ch/2015/07/completablefuture-clean-callbacks-with-java-8s-promises-part-4.html#api] by Nicolas Hofstetter neatly summarizes all the methods available. As you can see, the API is quite extensive (bloated?).

Handling Exceptions

Handling exceptions is non-trivial for asynchronous methods. Remember that, in synchronous method calls, the exceptions are repeatedly thrown to the caller up the call stack, until someone catches the exception. For asynchronous calls, it is not so obvious. For instance, should we put a catch around `fork()` or around `join()`? A `ForkJoinTask` doesn't handle exception with catch, but instead requires us to check for `isCompletedAbnormally` and then call `getException` to get the exception thrown.

As `CompletableFuture` allows chaining, it provides a cleaner way to pass exceptions from one call to the next. The terminal operation `whenComplete` takes in a `BiConsumer` as parameter -- the first argument to the `BiConsumer` is the result from previous chain (or `null` if exception thrown); the second argument is an exception (null if completes normally).

!!! Self-note: Javascript's `callback(data, err)`

```
CompletableFuture
    .completedFuture(Matrix.generate(nRows, nCols, rng::nextDouble))
    .thenApply(m -> m.multiply(m))
    .whenComplete((result, exception) -> {
        if (exception) {
            System.err.println(exception);
        } else {
            System.out.print(result);
        }
    }
```

`whenComplete` returns a CompletableFuture, surprisingly, despite it taking in a `BiConsumer` -- in a sense, `whenComplete` is more similar to `peek` rather than `forEach`.

`handle` is similar to `whenComplete`, but takes in a `BiFunction` instead of a `BiConsumer`, thus allowing the result or exception to be transformed.

Finally, `exceptionally` handles exception by replacing a thrown exception with a value, similar to `orElse` in `Optional`.

```
CompletableFuture
    .completedFuture(Matrix.generate(nRows, nCols, rng::nextDouble))
    .thenApply(m -> m.multiply(m))
    .exceptionally(Matrix.generate(nRows, nCols, ()->0);
```

Promise

`CompletableFuture` is similar to `Promise` in other languages, notably JavaScript and C++ (`std::promise`).

CompletionStage

In Java, `CompletableFuture` also implements a `CompletionStage` interface. Thus, you will find references to this interface in many places in the Java documentation. I find this name unintuitive and makes an already-confusing java documentation even harder to read.