Lecture 7: Functions

Learning Objectives

After this lecture, students should be familiar with:

- the concept of functions as a side-effect free programming constructs and its relation to functions in mathematics.
- the `Function` interface in Java 8, including the methods `apply`, `compose`, and `andThen` methods.
- the interfaces `Predicate`, `Supplier`, `Consumer`, and `BiFunction`.
- the syntax of method reference and lambda expression
- how to write functions with multiple arguments using curried functions

Java 8 introduces several new important new features, including lambda expressions and the stream APIs. We will spend the next few lectures exploring these new features and how it allows us to write more succinct code and hopefully, less buggy code. The stream APIs also makes it possible for us to parallelize our code for execution on multiple cores/processors with ease.

Abstraction Principles Revisited

Let's revisit the abstraction principles we first visited in Lecture 2. It says that *"Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."*

We have seen this principle being applied in several ways.

First, you should be all be familiar with abstracting out code with the same logic but applied to different variables. Consider the following sample code from `RandomGenerator` class:

```
 double genInterArrivalTime() {
   return -Math.log(this.rngArrival.nextDouble()) / this.customerArrivalRate;
 }

 double genServiceTime() {
   return -Math.log(this.rngService.nextDouble()) / this.customerServiceRate;
 }
```

You can see that the two methods above have similar implementation. They all generate an exponentially distributed random number, with a different rate using a different random number generator. One could apply the abstraction principle and write the code as follows:

```
double randomExponentialValue(Random rng, double rate) {
  return -Math.log(rng.nextDouble()) / rate;
}

double generateServiceTime() {
  return randomExponentialValue(this.rngService, this.serviceRate);
}

double generateInterArrivalTime() {
  return randomExponentialValue(this.rngArrival, this.arrivalRate);
}
```

Now, when we need a new exponentially distributed random number, say for rest period (see Lab 5), you can just write a method that calls `randomExponentialValue()` method with the appropriate `Random` object and rate, without worrying about the formula to generate an exponential random variable.

```
double genRestPeriod() {
  return randomExponentialValue(rngRestPeriod, this.serverRestingRate);
}
```

Here, the varying parts that we abstracted out is the variables.

Second, consider the `CircleQueue` and `PointQueue` examples from Lecture 4. The two classes implement similar methods with similar logic. The only difference is the type.

```
class CircleQueue {
  private Circle[] circles;
   :
  public CircleQueue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(Circle c) {...}
  public Circle dequeue() {...}
}

class PointQueue {
  private Point[] points;
   :
  public PointQueue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(Point p) {...}
  public Point dequeue() {...}
}
```

We then replace the two classes (and any future class that we might write to implement such queues) with a generic `Queue<T>`.

```
class Queue<T> {
  private T[] objects;
   :
  public Queue<T>(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(T o) {...}
  public T dequeue() {...}
}
```

Here, the varying parts that we abstracted out is the type of the elements.

Third, consider how we tell `ArrayList.sort()` to sort the items in the array in Lecture 5 [../lec05/index.html]. By passing in `NameComparator`, we can tell `ArrayList` to sort in alphabetical order, either in ascending order or descending order, or by the length of the strings, or any other ways we like. An alternative would be to have our own `StringList` class and implements methods `sortAlphabeticallyAscending()`, `sortAlphabeticallyDescending()`, `sortByLengthAscending()`, `sortByLengthDescending()`, etc. But all of these methods would be implementing the same sorting algorithms, the only part that is different is the comparison method to determine the order of the elements.

```
import java.util.*;

class NameComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    // return (s1.compareTo(s2));
    // return (s2.compareTo(s1));
    return (s2.length() - s1.length());
  }
}

class SortedList {
  public static void main(String[] args) {
    List<String> names = new ArrayList<String>();

    names.add(0, "Joffrey");
    names.add(1, "Cersei");
    names.add(2, "Meryn");
    names.add(3, "Walder");
      :
      :

    names.sort(new NameComparator());
  }
}
```

Here, the varying parts that we abstracted out is a snippet of the code, or functionality, of the methods. This idea is much more powerful than just abstracting out how we compare and sort. We will see how it can lead to a significantly different way of writing code.

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, refers to a mapping from a set of inputs (*domain*) X to a set of output values (*codomain*) Y. We write $f : X \rightarrow Y$. 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 need to be mapped. The set of elements in the codomain that is mapped is called the *image*.

Functions in programming languages 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*.

Here are two examples of pure functions:

```
int square(int i) {
  return i * i;
}

int add(int i, int j) {
  return i + j;
}
```

and 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
}
```

A pure function must also be deterministic. Given the same input, the function must produce the same output, *every single time*.

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. How do we write classes with methods that do not have side effects (do not update the fields in the class)? We can do so by making our class *immutable*. Recall that classes such as `String` are immutable. If we want to modify a `String` object, we have to create a new one containing the modified string. Such operations on the `String` object has no side-effect (the original string is still in tact).

In computer science, we refer to the style of programming where we build a program from pure functions as *functional programming* (FP). Many modern programming languages, such as Java, now supports this style of programming. As Java is inherently still an OO language, we cannot build a program from only pure functions. As such, I will refer to this style as *functional-style programming* within an OO language.

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? Or, how much effort is needed to trace through the code in different classes to see what is going on in one line of code. Take the following line, for instance,

```
server.serve(customer);
```

What is updated in the server? How about the customer? Is the variable `numOfServedCustomer` updated? Is a new `DoneEvent` being created and scheduled?

If we design and write our program with pure functions as much as possible, we could significantly reduce the number of bugs.

Michael Feathers tweeted that "(OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.) [https://twitter.com/mfeathers/status/29581296216?lang=en [https://twitter.com/mfeathers/status/29581296216?lang=en]]" The moving parts here refers to changing states. He succintly highlights one of the major differences between OOP and FP.

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. Another common error is that a function may produce a value that is not in the codomain, e.g., `null`.

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 take in more than one arguments (e.g., `add` above). We will see how to reconcile this later.

The `Function` interface in Java 8

Let's explore functions in Java 8 by looking at the `Function` [https://docs.oracle.com/javase/9/docs/api/java/util/function/Function.html] 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 are 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 principle, 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>() {
  public Integer apply(Integer x) {
    return x * x;
  }
});
```

Map
The `applyList` method above is most commonly referred 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 infer the type, nor need the return statements and the curly brackets.

lambda
Alonzo Church invented lambda calculus (λ-calculus) in 1936, before electronic computers, as a way to express computation. In λ-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 a lambda expression to a variable:

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

Method Reference

We can use a 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 $\cdot$ operator in $f \cdot g$).

These two methods, `andThen` and `compose`, return another function, and they are generic methods with 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;
applyList(list, abs.andThen(sqrt))
```

or

```
sqrt.compose(abs)
applyList(list, sqrt.compose(abs))
```

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!

In other languages
Lambda expression and `Function`s are introduced in Java only recently in Java 8, and still relies on classes and interfaces internally to implement them. As such, despite the elegance and beauty of pure functions, the syntax for it in Java is neither elegant nor pretty. Take Haskell, a pure functional programming language, for example. To compose two functions, we can use the `.` operator: `sqrt . abs`

Generics Revisited: PECS

We will see many functional generic interfaces with bounded wildcards, so it is worth to spend a little more time to understand what is going on here. Take `andThen` for example:

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

which is a method in the interface `Function<T,R>`. The method declaration would be clearer if it is written as

```
default <V> Function<T,V> andThen(Function<R, V> after);
```

Here, composing a function $T :\rightarrow R$ followed by $R :\rightarrow V$ gives us a function $T \rightarrow V$. The issue with this `andThen` declaration, is that it is not very general. The argument `after` must be exactly a function with argument type R and return type V.

We can make the method more general, but allowing it to take a function with `R` or any superclass of `R` as input -- surely if the function can take in a superclass of `R`, it can take in `R`. Thus, we can relax input type, or what the function *consumes*, from `R` to `? super R`.

Similarly, if we are expecting the function `after` to return a more general type `V`, it is fine if it returns `V` or a subclass of `V`. Thus, we can relax the return type, or what the function *produces*, from `V` to `? extends V`.

Both are widening type conversions that are safe.

This introduces us to a principle of using generics, with a mnemonic "producer `extends`; consumer `super`", or PECS, for short.

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 involve primitive types are also provided.

Here are some examples of how these types are used:

```
Predicate<Integer> isEven = x -> (x % 2) == 0;

Random rng = new Random(1);
Supplier<Integer> randomInteger = () -> rng.nextInt();

Consumer<Boolean> printer = System.out::println;

printer.accept(isEven.test(randomInteger.get()));
```

Impure Functions
Since we use a random number generator, `randomInteger` is not a pure function -- invoking it changes the internal state of the random number generator, causing it to give us a different number the next time it is invoked. So is the function `printer`, which causes a side effect of having something printed on the standard output. While there are ways to generate random numbers and perform I/O in functional way, I prefer to keep things simple in CS2030 and use random generator and I/O in the traditional way when we explore functional-style programming. We should still isolate these non-pure functions with clear variable names so that the intention of the program is clear.

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 $f : (X, Y) \to Z$. We can introduce F as a set of all functions $f' : Y \to Z$, and rewrite f as $f : X \to F$, of $f : X \to Y \to Z$.

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 + 1, we call

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

Let's break it down a little, `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 returns a value. It is a *higher-order function* that takes in a single argument, and return another function.

The technique that translates a general n-ary function to a sequence of n 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 a 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.

In the former case, we can save the context of a function and carry it around, avoiding the need for multiple parameters, leading to clearer code and fewer states to keep.

Let's go back to the `RandomGenerator` example from your labs below

```
double randomExponentialValue(Random rng, double rate) {
  return -Math.log(rng.nextDouble()) / rate;
}

double generateServiceTime() {
  return randomExponentialValue(this.rngService, this.serviceRate);
}
```

In the `RandomGenerator` class, we need to keep two states, a `Random` object `rngServer`, and a double `serviceRate` to generate the service time required for a customer. These states, however, does not change.

Or we could just keep a field `serviceTimeGenerator` in `RandomGenerator` with the type `Supplier<Double>`, which is a partially applied version of `randomExponentialValue`.

```
Function<Random, Function<Double, Supplier<Double>>> randomExponentialValue =
    rng -> rate -> () -> -Math.log(rng.nextDouble()) / rate;
Supplier< Double> serviceTimeGenerator = randomExponentialValue.apply(rngService).apply(rate);
```

We can then call `RandomGenerator.serviceTimeGenerator.get()` to get the next service time.

Exercise

1. Which of the following are pure functions?

```
int fff(int i) {
  if (i < 0) {
    throw new IllegalArgumentException();
  } else {
    return i + 1;
}

int ggg(int i) {
  System.out.println(i);
  return i + 1;
}

```

```
int hhh(int i) {
  return new Random().nextInt() + i;
}

int jjj(int i) {
  return Math.abs(i);
}
```

2. The method `and` below takes in two `Predicate` objects `p1` and `p2` and returns a new `Predicate` that evaluates to `true` if and only if both `p1` and `p2` evaluate to `true`.

```
Predicate<T> and(Predicate<T> p1, Predicate<T> p2) {
  // TODO
}
```

Fill in the body of the method `and`

3. Java implements lambda expressions as anonymous classes. Suppose we have the following lambda expression `Function<String,Integer>`:

```
str -> str.indexOf(' ')
```

Write the equivalent anonymous class for the expression above.

4. Consider the lambda expression:

```
x -> y -> z -> f(x,y,z)
```

where `x`, `y`, `z` are of some type `T` and `f` returns a value of type `R`.

(a) What is the type of the lambda expression above?

(b) Suppose that the lambda expression above is assigned to a variable `exp`. Given three variables `x`, `y`, and `z`, show how you can evaluate the lambda expression with `x`, `y`, `z` to obtain `f(x,y,z)`.

5. Write a class `LambdaList<T>` that is *immutable* and supports `generate`, `map`, `filter`, `reduce`, and `forEach`, methods. The skeleton is given below. The `map` method, similar to what you see in the lecture, is given. The static method `of` can be used to build the class and is given as well.

`of` constructs the list from some number of arguments. We use the Java *varargs* construct here `T... varargs`, which is a short cut for creating and passing in an array.

```
LambdaList.of(1, 3, 4);
LambdaList.of("one", "three", "four");
```

`generate` is a static method that returns a new list with `count` elements, where each element in the list is generated with a given Supplier `s`. For example

```
LambdaList.generate(4, rng::nextInt);
```

`filter` returns a new list, containing only elements in the list that pass the predicate test (i.e., the predicate returns true). Example:

```
LambdaList<String> list = LambdaList.of("show", "me", "my", "place", "in", "all", "this");
list.filter(x -> x.length() == 2); // returns [me, my, in]
```

`reduce` takes in a `BiFunction` that is called the *accumulator* -- it basically goes through the list, and accumulate all the values into one. The accumulator requires an initial value to start with. This initial value is the *identity* of the `BiFunction` (in mathematical notation, for identity i, $f(i, x) = x$ for any x).

Example:

```
LambdaList<Integer> list = LambdaList.of(4, 3, 2, 1);
list.reduce(1, (prod, x) -> prod * x); // returns 24
```

`forEach` consumes each element in the list with a consumer. Example:

```
LambdaList<Integer> list = LambdaList.of(4, 3, 2, 1);
list.forEach(System.out::println);
```

The skeleton code is given:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.BiFunction;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Supplier;

class LambdaList<T> {
  List<T> list;

  public static <T> LambdaList<T> of(T... varargs) {
    List<T> list = new ArrayList<>();
    for (T e : varargs) {
      list.add(e);
    }
    return new LambdaList<T>(list);
  }

  private LambdaList(List<T> list) {
    this.list = list;
  }

  public static <T> LambdaList<T> generate(int count, Supplier<T> s) {
    // TODO
    return null;
  }

  public <V> LambdaList<V> map(Function<? super T, ? extends V> f) {
    List<V> newList = new ArrayList<V>();
    for (T i: list) {
      newList.add(f.apply(i));
    }
    return new LambdaList<V>(newList);
  }

  public <U> U reduce(U identity, BiFunction<? super U, ? super T, ? extends U> accumulator) {
    // TODO
    return null;
  }
```

```
41
42    public LambdaList<T> filter(Predicate<? super T> predicate) {
43      // TODO
44      return null;
45    }
46
47    public void forEach(Consumer<? super T> action) {
48      // TODO
49    }
50
51    public String toString() {
52      return list.toString();
53    }
54  }
```