CS 5004: Lecture 13

Northeastern University, Spring 2021

At the start of every lecture

1. Pull the latest code from the lecture-code repo
2. Open the Evening_lectures folder
3. Copy this week's folder somewhere else
   - So you can edit it without causing GitHub conflicts
4. Open the code:
   1. Find the build.gradle file in the folder called LectureX
   2. Double click it to open the project

Agenda

Functional programming

- Concepts
- Lambdas
- Using Streams
- Functions as objects

Concepts

Imperative vs. declarative programming

Imperative

- Specify *how* to do something
- Object-oriented

Declarative

- Specifies *what* to do
- Functional

Real world: even when taking an OOD approach, may include *some* functional programming

- lambdas

Immutability

A tenet of functional programming

- An object should be immutable
- Rather than change state, create a new copy with the new state

What is a function?

Think back to 5002...

- All elements in the domain are mapped to a value in the codomain
- There cannot exist elements in the domain with no image in the codomain
- A single element in the domain can only map to one element in the codomain

Functional methods

A method is functional if:

- Does not mutate anything outside the function
- Does not mutate arguments
- Does not throw errors or exceptions
  - Functional programming: no side effects!
- Always returns a value
- When called with the same argument, always returns the same result

Functional Java

Using the `Stream` API

- Start with a **stream** of data e.g. a stream of objects in a collection
  - **Apply** a series of operations/transformation to the stream
  - **Reduce** or **collect** the stream

Familiar stream example: Java IO

Stream oriented

- IO = “input/output”
- I/O stream – a communication channel (“pipe”) between a **source** and a **destination** that allows us to create a flow of **data**
- A stream is a sequence of data.
- Data is read 1+ bytes at a time directly from the stream

I/O Stream

- Input **source**: e.g. a file, another program, device
- Output **destination**: e.g. a file, another program, device
- The ***kind* of data** streamed: e.g. bytes, ints, objects

Reading a file

```
try ( BufferedReader reader =
        new BufferedReader(new FileReader(“filename”))) {
   String line = “”;
   while ((line = reader.readLine()) != null) {
      ...
   }
}
catch…
```

Reading with a Stream…

Related to something we've already seen:

```
Stream<String> lines
      = new BufferedReader(new FileReader(file)).lines();
```

`BufferedReader`'s `lines()` method returns a *stream* of Strings – representing each line in a file

Collections can be streamed

Streams can be **generated by a built-in `Collection` e.g.**

```
List<Integer> nums = new LinkedList();
Stream<Integer> numStream = nums.stream();
```

→returns a *stream* of the Integers in the given linked list

All objects implementing **`Collection`** have this method.

?

Key point: a stream is a ***sequence of data***

```
Stream<String> lines = new BufferedReader(new FileReader(file)).lines();
```

→ a **sequence of Strings**

```
Stream<Integer> numStream = nums.stream();
```

→ a **sequence of Integers**

What can we do with streams?

- Generate
- Iterate
- Filter
- Map input to output
- Count
- Sort
- ...and more

Lambdas

Lambda expressions / anonymous functions

“A function that is not bound to an identifier”

(a function that doesn’t have a name)

- Can be passed as parameters to other functions / methods
- Supported by many languages
- Relatively new to Java

Lambda expressions / anonymous functions

“A function that is not bound to an identifier”

(a function that doesn’t have a name)

- ***Can be passed as parameters to other functions / methods***
- Supported by many languages
- Relatively new to Java

Lambda expressions / anonymous functions

“A function that is not bound to an identifier”

(a function that doesn’t have a name)

- ***Can be passed as parameters to other functions / methods***
- Supported by many languages
- Relatively new to Java

What are they for?

- Functions that only need to be used once
- Keeping code short and tidy

Some (non-Java) examples

Things I use anonymous functions for

```
32 df_basic["age"] = df_basic.apply(lambda defendant: normalize_age(defendant, df_basic["age"].max()), axis=1)
```

- Instead of iterating through every row in a large dataset
- Shorter
- Only needs to happen once

Some (non-Java) examples

Things I use anonymous functions for

Event handling, callbacks

- Only needs to happen under specific circumstances
- Doesn’t make sense to be called anywhere else

```
export const sendMessage = (msg, callbackFunc) => {
    socket.emit("new message", msg);

    socket.on("all messages", result => {
        callbackFunc(result);
    })
}
```

When to use lambdas?

…to replace for loops/small helper methods:

- When a particular loop or helper is used in *only one place* e.g. called only from one method
  - If used in multiple places → named method
- Logic is fairly simple and short:
  - Complex/long logic in a lambda is very hard to follow → use named method instead

Java lambda syntax

Anything on the left of the arrow is a **parameter**

```
(int a, int b) -> { return a + b; }
```

Java lambda syntax

Anything on the left of the arrow is a parameter

```
(int a, int b) -> { return a + b; }
```

…passed to the **function** on the right of the arrow

```
(int a, int b) -> { return a + b; }
```

Java lambda syntax

Anything on the left of the arrow is a parameter

```
(int a, int b) -> { return a + b; }
```

…passed to the function on the right of the arrow

```
(int a, int b) -> { return a + b; }
```

Equivalent to:

```
int addNumbers(int a, int b) {
  return a + b;
}
```

Lambda syntax simplified further

When used to do something with items in a collection, parameters don't need types…Java knows what type is in the collection

```
(a, b) -> { return a + b; }
```

Lambda syntax simplified further

(Simple) methods don’t need braces or a return

```
(a, b) -> a + b
```

Lambda syntax simplified further

If there's only one parameter, no need for parentheses

```
a -> a * a
```

Lambda syntax simplified further

If there's only one parameter, no need for parentheses

```
a -> a * a
```

Equivalent to:

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

Using Streams

Functional Java

Uses the `Stream` API

- Start with a **stream** of data e.g. a stream of objects in a collection
  - **Apply** a series of operations/transformation to the stream
  - **Reduce** or **collect** the stream

Start with a Stream…

Create a stream **of known objects e.g.**

```
Stream.of(anObj, anotherObj, obj3);
```

Start with a Stream…

Iteratively generate a stream from some “seed” :

```
Stream.iterate(2, (Integer i) -> i * i);
```

- Generates a stream of Integers where each number is the square of the previous

Start with a Stream…

Iteratively generate a stream from some "seed" :

```
Stream.iterate(2, (Integer i) -> i * i);
```

- Generates a stream of Integers where each number is the square of the previous
- The seed = the object to start with e.g. 2

Start with a Stream…

Iteratively generate a stream from some "seed" :

```
Stream.iterate(2, (Integer i) -> i * i);
```

- Generates a stream of Integers where each number is the square of the previous
- The seed = the object to start with e.g. 2
- For every object in the stream…

Start with a Stream…

Iteratively generate a stream from some "seed" :

```
Stream.iterate(2, (Integer i) -> i * i);
```

- Generates a stream of Integers where each number is the square of the previous
- The seed = the object to start with e.g. 2
- For every object in the stream…pass it to...

Start with a Stream…

Iteratively generate a stream from some "seed" :

```
Stream.iterate(2, (Integer i) -> i * i);
```

- Generates a stream of Integers where each number is the square of the previous
- The seed = the object to start with e.g. 2
- For every object in the stream…pass it to...
- Some operation (e.g. square the number)…returned as next input

Start with a Stream…

Iteratively generate a stream from some "seed" :

```
Stream.iterate(2, (Integer i) -> i * i);
```

• Single line of code → infinite output (without additional methods):

2 4 16 256 65536 4294967296 …. forever…

What can we do with a Stream?

Perform operations on the objects in the stream:

- Intermediate operations – perform actions before further processing
- Terminal operations – “final” operations
- Reductions – make the stream smaller, possibly just one value

What can we do with a Stream?

Perform operations on the objects in the stream:

- **Intermediate operations – perform actions before further processing**
- Terminal operations – “final” operations
- Reductions – make the stream smaller, possibly just one value

Intermediate operations – some examples

- **`filter()`** – only select some objects in the stream for further processing e.g.

```
IntStream.range(0, 50).filter(x -> x % 2 == 0)
```

…any ideas what that line does?

Intermediate operations – some examples

- **`filter()`** – only select some objects in the stream for further processing e.g.

```
IntStream.range(0, 50).filter(x -> x % 2 == 0)
```

…takes a stream of Integers from 0-49 and selects only the even numbers.

  - Can be used on other types of stream too

Intermediate operations – some examples

- **`limit()`** – Limits the number of items to look at e.g.

```
Stream.iterate(2, i -> i * i).limit(3);
```

… what does this line do?

Intermediate operations – some examples

- **`limit()`** – Limits the number of items to look at e.g.

```
Stream.iterate(2, i -> i * i).limit(3);
```

Stops after 3 items: 2, 4, 16

Intermediate operations – some examples

- **`map()`** – Maps the input to some output by performing an operation e.g.:

```
Stream.of(“a”, “b”, “c”).map(s -> s.toUpperCase());
```

…what is this line doing?

Intermediate operations – some examples

- **`map()`** – Maps the input to some *output* by performing an operation e.g.:

```
Stream.of("a", "b", "c").map(s -> s.toUpperCase());
```

  Converts each String in the Stream to an upper case String: A, B, C

Intermediate operations – some examples

```
Stream.of(“a”, “b”, “c”).map(s -> s.toUpperCase());
```

Equivalent to:

```
// letters is a LinkedList containing “a”, “b”, “c” …
for (String s: letters) {
   s = s.toUpperCase();
}
```

Intermediate operations – some examples

- **`map()`** – Maps the input to some *output* by performing an operation e.g.:

```
Stream.of("a", "b", "c").map(s -> s.toUpperCase());
```

  Converts each String in the Stream to an upper case String: A, B, C

What can we do with a Stream?

Perform operations on the objects in the stream:

- Intermediate operations – perform actions before further processing
- **Terminal operations – "final" operations**
  - **i.e. you can't tack on any additional operations such as filter, map…**
- Reductions – make the stream smaller, possibly just one value

Terminal operations – some examples

- **`forEach()`** – Basically a for-each loop but tidier…

```
Stream.of("a", "b", "c").map(s ->
s.toUpperCase())
    .forEach(s -> System.out.println(s));
```

Converts each String in the Stream to an upper case String then prints it

- Can't print from map() → requires a return

Terminal operations – some examples

- **`forEach()`** – simplified syntax

```
Stream.of("a", "b", "c").map(s ->
s.toUpperCase())
    .forEach(System.out::println);
```

Already knows the parameter—each object in the stream

- Note the ::, not a typo!

Terminal operations – some examples

- **`collect()`** – gather the items into a new collection e.g.

```
List<String> letters = Stream.of("a", "b", "c")
    .map(s -> s.toUpperCase())
    .collect(Collectors.toList());
```

Returns the result of the operation as a new list.

What can we do with a Stream?

Perform operations on the objects in the stream:

- Intermediate operations – perform actions before further processing
- Terminal operations – “final” operations
- **Reductions – make the stream smaller, possibly just one value**

Reductions – some examples

- **`average(), count(), max()`** etc... e.g.

```
Stream.iterate(2, i -> i * i).limit(3).count();
```

Returns the number of items: 3

```
Stream.iterate(2, i -> i * i).limit(3)
                             .mapToInt(i -> i)
                             .sum();
```

Returns the sum of the items: 22

Functional programming: Summary

- Stream gets mapped, filtered, reduced, and collected in some order
- Lambdas: unnamed methods (functions) that can be applied to a stream
- In functional programming, objects are **immutable**:
  - Rather than change an existing object, copy it and return with new state

Example: functionalcountrycodes

New version:

- CountryCodeProcessor.java uses a stream for processing
  - **`processInput`**

processInput()

```
private Map<String, String> processInput(List<String> lines) {
  return lines.stream()
              .map(line -> line.replaceAll("\"",""))
              .collect(
                Collectors.toMap(
                  line -> line.substring(line.lastIndexOf(",")),
                  line -> line.substring(0, line.lastIndexOf(","))
                )
              );
}
```

processInput()

```
private Map<String, String> processInput(List<String> lines) {
  return lines.stream()
                .map(line -> line.replaceAll("\"",""))
                .collect(
                  Collectors.toMap(
                    line -> line.substring(line.lastIndexOf(",")),
                    line -> line.substring(0, line.lastIndexOf(","))
                  )
                );
}
```

Create a stream of Strings

processInput()

```
private Map<String, String> processInput(List<String> lines) {
  return lines.stream()
              .map(line -> line.replaceAll("\"",""))
              .collect(
                Collectors.toMap(
                  line -> line.substring(line.lastIndexOf(",")),
                  line -> line.substring(0, line.lastIndexOf(","))
                )
              );
}
```

Does the replaceAll on every line

processInput()

```
private Map<String, String> processInput(List<String> lines) {
  return lines.stream()
              .map(line -> line.replaceAll("\"",""))
              .collect(
                Collectors.toMap(
                  line -> line.substring(line.lastIndexOf(",")),
                  line -> line.substring(0, line.lastIndexOf(","))
                )
              );
}
```

Terminal operation - Many possible outcomes e.g.

- Collectors.counting() // number of elements in the stream
- Collectors.joining() // join into String
- Collectors.groupingBy() // group by parameter → hash map

processInput()

```
private Map<String, String> processInput(List<String> lines) {
  return lines.stream()
              .map(line -> line.replaceAll("\"",""))
              .collect(
                Collectors.toMap(
                  line -> line.substring(line.lastIndexOf(",")),
                  line -> line.substring(0, line.lastIndexOf(","))
                )
              );
}
```

Produces a map

- Need to define a key and a value

processInput()

```
private Map<String, String> processInput(List<String> lines) {
  return lines.stream()
              .map(line -> line.replaceAll("\"",""))
              .collect(
                Collectors.toMap(
                  line -> line.substring(line.lastIndexOf(",")),
                  line -> line.substring(0, line.lastIndexOf(","))
                )
              );
}
```

Creates the key

- Takes the existing list entry, makes a substring for the key

processInput()

```
private Map<String, String> processInput(List<String> lines) {
  return lines.stream()
              .map(line -> line.replaceAll("\"",""))
              .collect(
                Collectors.toMap(
                  line -> line.substring(line.lastIndexOf(",")),
                  line -> line.substring(0, line.lastIndexOf(","))
                )
              );
}
```

Creates the value

- Takes the existing list entry, makes a substring for the value

Example: functionalcountrycodes

- CountryCodeProcessor.java uses a stream for processing
  - **`processInput`**
- ...but not file reading. Why?
  - **`readFile`**
- Could we do something like this instead?

```
private Stream<String> readFile(String path) {
  try (Stream<String> lines
          = new BufferedReader(new FileReader(path)).lines()) {
    return lines;
  } catch ...
}
private Map<String, String> processInput(Stream<String> lines) { ...
}
```

- ***Try it and see what happens***

Streams are for one-time use only

You probably tried something like this:

```
private Stream<String> readFile(String path) {
    try (Stream<String> lines
      = new BufferedReader(new FileReader(path)).lines()) {
      return lines;
    } catch…{...}
}
private Map<String,String> processInput(Stream<String> lines)
{
   ...
}
```

Streams are for one-time use only

You probably tried something like this:

```
private Stream<String> readFile(String path) {
    try (Stream<String> lines
      = new BufferedReader(new FileReader(path)).lines()) {
      return lines;
    } catch…{...}
}
private Map<String,String> processInput(Stream<String> lines)
{
   ...
}
```

“Stream has already been operated upon or closed”

Streams are for one-time use only

You probably tried something like this:

```
private Stream<String> readFile(String path) {
    try (Stream<String> lines
      = new BufferedReader(new FileReader(path)).lines()) {
      return lines;
    } catch…{...}
}
private Map<String,String> processInput(Stream<String> lines)
{
    ...
}
```

“Stream has already been operated upon or closed”

→ streams need to be used when they’re created

A solution: read and process at the same time (has it’s own problems)

Exercise 1: streams & lambdas

In functional package, FunctionalPractice class, main method:

- Use a List to store multiple Books
- Use streams & lambdas on the List to:
  - make a new List containing only books published before a certain date
  - make a new List containing only books in a particular price range
  - get the total price of all books in the list
  - get the average price of all books in the list

- Refer to the Java 11 Streams docs
  - Google ”Java 11 Streams”

Break (10 mins)

Filtering by year, price

```
List<Book> oldBooks = books.stream()
                        .filter(book -> book.getYear() < 1995)
                        .collect(Collectors.toList());

List<Book> midPrice = books.stream()
                        .filter(book -> book.getPrice() >= 5 &&
                                       book.getPrice() <= 10)
                        .collect(Collectors.toList());
```

Total, average price

```
double total = books.stream()
                    .mapToDouble(book -> book.getPrice())
                    .sum();

OptionalDouble avg = books.stream()
                    .mapToDouble(book -> book.getPrice())
                    .average();
```

Total, average price

```
double total = books.stream()
                    .mapToDouble(book -> book.getPrice())
                    .sum();

OptionalDouble avg = books.stream()
                    .mapToDouble(book -> book.getPrice())
                    .average();
```

In case the stream is empty

Things to remember about streams

Built-in collections can be turned in to **streams** using **`stream()`**:

```
public static List<Book> publishedBefore(List<Book> books, int year) {
  return books.stream()
              .filter(b -> b.getYear() < year)
              .collect(Collectors.toList());
}
```

Things to remember about streams

Use **`filter()`** to select only the items that meet some condition:

```
public static List<Book> publishedBefore(List<Book> books, int year) {
  return books.stream()
              .filter(b -> b.getYear() < year)
              .collect(Collectors.toList());
}
```

Things to remember about streams

`filter()` takes a **lambda** expression:

```
public static List<Book> publishedBefore(List<Book> books, int
year) {
  return books.stream()
              .filter(b -> b.getYear() < year)
              .collect(Collectors.toList());
}
```

Things to remember about streams

`filter()` takes a **lambda** expression:

```
public static List<Book> publishedBefore(List<Book> books, int
year) {
  return books.stream()
                  .filter(b -> b.getYear() < year)
                  .collect(Collectors.toList());
}
```

Parameter passed to right side of lambda

- Represents each object in the stream (a Book in this example)

Things to remember about streams

`filter()` takes a **lambda** expression:

```
public static List<Book> publishedBefore(List<Book> books, int
year) {
  return books.stream()
                  .filter(b -> b.getYear() < year)
                  .collect(Collectors.toList());
}
```

Boolean condition

- Only stream objects that return true will be considered in the next operation
- i.e. Books whose year field is less than year

Things to remember about streams

`collect(Collectors.tolist())` returns the remaining items as a new list:

```
public static List<Book> publishedBefore(List<Book> books, int
year) {
  return books.stream()
                  .filter(b -> b.getYear() < year)
                  .collect(Collectors.toList());
}
```

Things to remember about streams

Use **`map()`** to *transform* each object in stream:

```
List<Book> saleBooks = books.stream().filter(b -> b.getYear() <
year)
        .map(b -> {
              b.setPrice(b.getPrice * 0.9f);
              return b;
        })
        .collect(Collectors.toList());
```

Things to remember about streams

`map()` takes a **lambda** expression:

```
List<Book> saleBooks = books.stream().filter(b -> b.getYear() <
year)
        .map(b -> {
              b.setPrice(b.getPrice * 0.9f);
              return b;
        })
        .collect(Collectors.toList());
```

Things to remember about streams

`map()` takes a **lambda** expression:

```
List<Book> saleBooks = books.stream().filter(b -> b.getYear() <
year)
        .map(b -> {
              b.setPrice(b.getPrice * 0.9f);
              return b;
        })
        .collect(Collectors.toList());
```

Use { } if the lambda function contains multiple lines

Things to remember about streams

`map()` takes a **lambda** expression:

```
List<Book> saleBooks = books.stream().filter(b -> b.getYear() <
year)
        .map(b -> {
              b.setPrice(b.getPrice * 0.9f);
              return b;
        })
        .collect(Collectors.toList());
```

The lambda function must return an Object

Things to remember about streams

`map()` takes a **lambda** expression:

```
List<String> saleBooks = books.stream().filter(b -> b.getYear() < year)
                              .map(b -> b.toString())
                              .collect(Collectors.toList());
```

…`map()` can also be used to convert stream elements to a different type

Things to remember about streams

Streams of Integers or Doubles can use `average()`, `min()`, and `max()`:

```
Integer oldestPubDate = books.stream().mapToInt(b -> b.getYear()).min();
```

Things to remember about streams

Streams of Integers or Doubles can use `average()`, `min()`, and `max()`:

```
Integer oldestPubDate = books.stream().mapToInt(b -> b.getYear()).min();
```

To map an Object to Integer or Double:

- `mapToInt(…)`
- `mapToDouble(…)`

map() vs. flatMap()

What does list contain?

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any other words"};
Object list = Stream.of(someStrings)
                    .map(line -> splitAtSpaces.splitAsStream(line))
                    .collect(Collectors.toList());
```

map() vs. flatMap()

What does list contain? ...a list of lists of Strings

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any other words"};
Object list = Stream.of(someStrings)
                    .map(line -> splitAtSpaces.splitAsStream(line))
                    .collect(Collectors.toList());
```

map() vs. flatMap()

To get a list of words, use flatMap()

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any other words"};
Object list = Stream.of(someStrings)
                    .flatMap(line -> splitAtSpaces.splitAsStream(line))
                    .collect(Collectors.toList());
```

reduce()

Reduce the elements in a stream to a single value

```
double total = books.stream()
                    .mapToDouble(b -> b.getPrice())
                    .reduce(0, (p1, p2) -> p1 + p2);
```

reduce()

Reduce the elements in a stream to a single value

```
double total = books.stream()
                    .mapToDouble(b -> b.getPrice())
                    .reduce(0, (p1, p2) -> p1 + p2);
```

the lambda takes two parameters

reduce()

Reduce the elements in a stream to a single value

```
double total = books.stream()
                    .mapToDouble(b -> b.getPrice())
                    .reduce(0, (p1, p2) -> p1 + p2);
```

the starting point, passed in along with the first element in the stream

reduce()

Reduce the elements in a stream to a single value

```
double total = books.stream()
                    .mapToDouble(b -> b.getPrice())
                    .reduce(0, (p1, p2) -> p1 + p2);
```

This example does the same as:

```
double total = books.stream()
                    .mapToDouble(b -> b.getPrice())
                    .sum();
```

Stream API

- We’ve seen just a small subset of the API!
- Read the docs for more
- (We’ll see some more variations in the practice finals)

Functions as objects

Functions as Objects

Functions can be saved as variables

```
Function<T, R> functionName = t -> operation returning an R;
```

Must specify T & R

- T = type of object passed to `functionName`
- R = type of object returned by `functionName`

Functions as Objects

Functions can be saved as variables

```
Function<T, R> functionName = t -> operation returning an R;
```

Must specify T & R

- T = type of object passed to `functionName`
- R = type of object returned by `functionName`

Functions as Objects

Different syntax, same outcome:

```
Function<T, R> functionName
    = t -> operation returning
      an R;
```

```
R functionName(T t) {
    operation…
    return something of type R
}
```

Functions as Objects

Different syntax, same outcome:

```
Function<T, R> functionName
    = t -> operation returning
      an R;
```

```
R functionName(T t) {
    operation…
    return something of type R
}
```

T = type of parameter passed to `functionName`

- Only one parameter
- Must inherit Object

Functions as Objects

Different syntax, same outcome:

```
Function<T, R> functionName
    = t -> operation returning
      an R;
```

```
R functionName(T t) {
    operation…
    return something of type R
}
```

t = parameter name

- Call it whatever you want (doesn't have to be t)

Functions as Objects

Different syntax, same outcome:

```
Function<T, R> functionName
    = t -> operation returning
      an R;
```

```
R functionName(T t) {
    operation…
    return something of type R
}
```

=, ->, and ; instead of ()

- Single-line operations don’t need { }
- Multi-line operations do
- Don’t always need return

Calling a function object

```
T oldObj = new T();
R newObj = functionName.apply(oldObj);
```

Calling a function object

```
T oldObj = new T();
R newObj = functionName.apply(oldObj);
```

function variable name

Calling a function object

```
T oldObj = new T();
R newObj = functionName.apply(oldObj);
```

Use Function.apply() to call the function

Calling a function object

```
T oldObj = new T();
R newObj = functionName.apply(oldObj);
```

pass in the required parameter

Example

```
Function<Book, Book> discountBook = b -> {
  b.setPrice(b.getPrice() * 0.9f);
  return b;
};

// aBook is $10
discountBook.apply(aBook);
// aBook is $9
```

Does this comply with the functional paradigm?

```
Function<Book, Book> discountBook = b -> {
  b.setPrice(b.getPrice() * 0.9f);
  return b;
};

// aBook is $10
discountBook.apply(aBook);
// aBook is $9
```

Does this comply with the functional paradigm?

```
Function<Book, Book> discountBook = b -> {
  b.setPrice(b.getPrice() * 0.9f);
  return b;
};

// aBook is $10
discountBook.apply(aBook);
// aBook is $9
```

No. The argument is modified
→ create a new book instead

Uses of Functions as Objects

- Tidying up stream operations
- If you want a method to accept a function as a parameter
- If you want a function to be accessible to only one object
  - e.g. event listeners
- *a design choice*

```
private void transform(Shape s,
    Function<Shape, Shape> change) {
  change.apply(s);
  redraw();
}


transform(myShape, scaleUp);
transform(myShape, scaleDown);
transform(myShape, moveLeft);
```

Exercise 2: Functions as parameter practice

Add two function objects to the FunctionalPractice class:

- Discount a book by 10% (same as prev slide)
- Double the price of the book

Add a static method to the FunctionalPractice class:

- **`Book adjustPrice(Book book, Function<Book, Book> adjuster)`**
- Could be used to either decrease or increase the price

Exercise 2: Functions as parameter practice

```
private static Function<Book, Book> discountBook = b -> {
  Book newBook = new Book(b.getTitle(),b.getAuthor(),b.getYear(),  price: b.getPrice() * 0.9f);
  return newBook;
};

private static Function<Book, Book> doublePrice = b -> {
  Book newBook = new Book(b.getTitle(),b.getAuthor(),b.getYear(),  price: b.getPrice() * 2);
  return newBook;
};

private static Book adjustPrice(Book book, Function<Book, Book> adjuster) {
  return adjuster.apply(book);
}
```

```
Book cheaper = FunctionalBook.adjustPrice(b1, discountBook);
```