CS2030S PE2

AY24/25 sem 2

github.com/mendax1234

Functional Interface

1. (BooleanCondition<T>):
   - **Lambda example**: `BooleanCondition<Integer> isPositive = x -> x > 0;`
   - **Java equivalent**: `Predicate<T>`.
2. (Producer<T>):
   - **Lambda example**: `Producer<Double> randomValue = () -> Math.random();`
   - **Java equivalent**: `Supplier<T>`.
3. (Consumer<T>):
   - **Lambda example**: `Consumer<String> printUpperCase = s -> System.out.println(s.toUpperCase());`
   - **Java equivalent**: `Consumer<T>`
4. (`Transformer<U, T>`): Tranform a value of type `U` into a value of type `T`.
   - **Lambda example**: `Transfomer<String, Integer> stringLength = s -> s.length();`
   - **Java equivalent**: `Function<U,T>`
5. (`Combiner<S, T, R>`): Combine two values of type `S`, `T` into a value of type `S`.
   - **Lambda example**: `Combiner<Integer, Integer, Integer> multiply = (a, b) -> a * b;`
   - **Java equivalent**: `BiFunction<S, T, R>`.
6. **Functional Interface Example**

```
@FunctionalInterface
public interface Valuable {
  double getPrice();
}
```

List

1. **Create an empty List**:
   - **Create an empty immutable list**: `List<T> l = List.of();`
   - **Create an empty mutable list**: `List<T> l = new ArrayList<>();`
2. **Copy from a List to another**: This is usually used in the **constructor**, use `this.fruits = new ArrayList<>(fruits);`, where `this.fruits` is of type `List<T>` and `fruits` is of type `List<? extends T>` to avoid the type problems. And `import java.util.ArrayList;`
3. **List to stream**: use `list.stream()`
4. **Stream to list**: use `stream.toList()`
5. **isEmpty**: This function can be used to filter out the empty list from the stream.

```
public boolean isEmpty() {
  return this.books.isEmpty();
}
```

`filter()` here will keep the elements that are **not empty**!

```
return bss.stream()
          .map(bs -> Q7.applyDiscount(bs, k))
          .filter(bs -> !bs.isEmpty())
          .toList();
```

Maybe

1. `Maybe.some` always represents a **valid** value inside the wrapper. `Maybe.None` always represents **not found or value DNE**.
2. **map**: if the target is `Maybe.some`, `map` will always add a `Maybe.some` wrapper! (Not calling `Maybe.of`!) **Can think of map() as a method that will return a value**
3. **Use Maybe to rewrite if-else branch**:
   - **Write conditions**: This is usually done by constructing a `Maybe` and using `filter`
   - **Invoke the method in if-branch**: this can be done by using `flatMap`/`map`
     - **To pass several parameters**: Chain `flatMap()` and `map()`, the result after this step is still a `Maybe`.
     - **Change a different variable to use**: This can be done using **flatMap()/map()** also!
     - **ifPresent**: this usually after the previous step, if need further operation on the result, use invoke the method using **ifPresent**. (**Think it as a function that returns void**)
   - **Inovke the method in else-branch**: this can be done using `orElse()`/`orElseGet()`
     - These two methods are usually used to get the value from previous **Maybe** or produce a new value.

```
public void transfer(int from, int to, double amount) {
  Maybe<Account> fromAccount = findAccount(from).filter(
      ac -> !ac.isClosed() && ac.getBalance() > 0);
  Maybe<Account> toAccount = findAccount(to).filter(ac -> !ac.isClosed());
  toAccount.flatMap(tac -> fromAccount.map(f -> f.transferTo(tac, amount)).getFirst()))
           .ifPresent(fac2 -> accounts.put(from, fac2));
  // if (fromAccount != null && toAccount != null && fromAccount.getBalance() > 0 &&
  //     !fromAccount.isClosed() && !toAccount.isClosed()) {
  //   fromAccount.transferTo(toAccount, amount);
  // }
}
```

4. **orElse()/orElseGet() Example**:

```
String foo(Maybe<Object> m) {
  return m.map(obj -> String.valueOf(obj)).orElse(t:"?");
  // if (m.equals(Maybe.none()) {
  //   return "?";
  // }
  // return String.valueOf(m.get());
}
```

5. **Main Method Descriptor**
   - `filter(BooleanCondition<? super T> c)`: Only keep the value in the `Maybe` wrapper if it **pass the test**
   - `map(Transformer<? super T, ? extends U> t)`
   - `flatMap(Transformer<? super T, ? extends Maybe<? extends U>> t)`
   - `orElse(T t)`
   - `orElseGet(Producer<? extends T> p)`
   - `ifPresent(Consumer<? super T> c)`
   - `of(T t)`: if `t == null`, return `Maybe.none()`. Else, return `Maybe.some(t)`.

Lazy

1. **Basic**: Lazy can store either an **already existed value** (Wrapped by `Maybe.some`) or a **producer which will be used to produce a value**.
2. **The memoization thinking** in `Lazy::get()`, also an example for multiple lines lambda

```
public T get() {
  return this.value.orElseGet(
      () -> {
        T compute = this.producer.produce();
        this.value = Maybe.some(compute);
        return compute;
      });
}
```

3. **Lazy List Example**:

```
public static <T> LazyList<T> generate(
  int n, T seed, Transformer<T, T> f) {
  List<Lazy<T>> lazyList = new ArrayList<>();
  Lazy<T> curr = Lazy.of(seed);
  for (int i = 0; i < n; i++) {
    lazyList.add(curr);
    curr = curr.map(f);
  }
  return new LazyList<>(lazyList);
}
```

Infinite List

1. **Traverse through the infinite list**:

```
public void forEach(Consumer<? super T> action) {
  InfiniteList<T> currList = this;
  while (!currList.isSentinel()) {
    // Consume the head
    currList.head.get().ifPresent(action);
    // Shrink the sublist
    currList = currList.tail.get();
  }
}
```

Stream

1. When using the **intermediate operation** on stream, use the good practice to write lambda as `each element -> what operation`. e.g. `fruits.stream().map(fruit -> String.format("- %s", fruit));`
2. **reduce()**: It's workflow is like that **result = identity → for each element in the stream; result = accumulator.apply(result, element); return result**
   - `reduce(identity, accumulator)`: the `identity` and `element in the stream` must be of the **same** type!

```
return sortedAccounts.stream()
                     .filter(ac -> !ac.isClosed())
                     .map(ac -> ac.toString())
                     .reduce("", (a, b) -> String.format("%s\n", a + b));
// for (Account account : sortedAccounts) {
//   if (!account.isClosed()) {
//     s += account + "\n";
//   }
// }
// return s;
```

   - `reduce(identity, accumulator, combiner)`: the **identity may not be the same type** as the **element in the stream**
     - **Combiner** is `(a,b) -> a + b`, this is usually used for the summation of the stream elements.
     - **Combiner** is `(a,b) -> a * b`, this is usually used for the product of the stream elements.
     - The other Combiner like `(a,b) -> a - b` and `(a,b) -> a / b` are **not safe**! Don't use!

```
/**
 * Return the details of all accounts in the bank as a string.
 */
public String allAccounts() {
  return getAccountStream()
    .filter(a -> !a.isClosed())
    .sorted((x, y) -> Double.compare(y.getBalance(), x.getBalance()))
    .reduce("", (s, a) -> s + a + "\n", (s, a) -> s + a);
}
```

   - **Explicitly cast the return type**: This can be done by changing the type of **identity** in `reduce`

```
public class Q6 {
  public static <T extends Book> List<BookShop<Book>>
    consolidateShopsbyGenre(List<? extends BookShop<? extends T>> bss) {
    return Q5.findUniqueGenres(bss)
             .stream()
             .map(genre ->
               bss.stream()
                  .map(bs -> new BookShop<Book>(bs.findBooksByGenre(genre)))
                  .reduce(new BookShop<Book>(), (a, b) -> Q4.mergeShops(a, b)))
             .toList();
  }
}
```

3. **map()**: transforms **each element** in the stream by **applying a function (transformer)** to it, producing a new stream of the transformed elements with a **one-to-one relationship**.
   For example, here the **name** and the **cost after reducing** has a **one-to-one** relationship

```
public static Stream<String> totaledCustomerCosts(
  Map<String, List<Integer>> customerTable,
  Map<Integer, Double> salesTable) {
  return customerTable.keySet().stream()
          .map(name ->
            name + ": "
                 + Query.getCostsFromName(customerTable, salesTable, name)
                        .reduce(identity:0.0, (a, b) -> a + b));
}
```

4. **flatMap()**: transforms **each element** into a **stream** and then **flattens** all resulting streams into a single stream, useful for working **with nested collections** or **when one element should produce multiple output elements**.
   For example, here the **name** and the **cost** has a **one-to-multiple** relationship

```
public static Stream<String> allCustomerCosts(
  Map<String, List<Integer>> customerTable,
  Map<Integer, Double> salesTable) {
  return customerTable.keySet().stream()
                      .flatMap(name ->
                        Query.getCostsFromName(customerTable, salesTable, name)
                             .map(x -> name + ": " + x));
}
```

5. **filter()**: Creates a new stream that keeps **only the elements that passed the test** in `Predicate`.
6. **none/any/allMatch()**: a **boolean** method. It returns `true` only if **no/any/all** element in the stream **passes the predicate test**.
7. **sorted()**: elements according to **natural order** (small to big or ascending order) or a **provided comparator**
   The following provides an example on `sorted` as well as how to use `reduce` to find maximum/minimum value in a stream

```
class Q8 {
  static class ShopComparator<T extends Book>
  implements Comparator<BookShop<? extends T>> {
    public int compare(
      BookShop<? extends T> stall1, BookShop<? extends T> stall2) {
      return (int) (stall1.getBooks().stream()
          .map(Book::getPrice)
          .reduce(Double.MAX_VALUE, Math::min)
          - stall2.getBooks().stream()
          .map(Book::getPrice)
          .reduce(Double.MAX_VALUE, Math::min));
    }
  }

  public static <T extends Book> List<BookShop<Book>> searchForBook(
    List<? extends BookShop<? extends T>> bss, String title) {
    return bss.stream()
              .filter(bs -> bs.hasBook(title))
              .map(bs -> new BookShop<Book>(
                bs.getBooks().stream()
                  .filter(book -> book.hasTitle(title))
                  .toList()))
              .sorted(new ShopComparator<>())
              .map(bs -> new BookShop<Book>(bs.getBooks()))
              .toList();
    }
}
```

Monad and Functors

1. **Monad Laws**:
   - **The left identity law**: `Monad.of(x).flatMap(x -> f(x))` must be the same as `f(x)`
   - **The right identity law**: `monad.flatMap(x -> Monad.of(x))` must be the same as `monad`
   - **The associative law**: `monad.flatMap(x -> f(x)).flatMap(x -> g(x))` must be the same as `monad.flatMap(x -> f(x).flatMap(y -> g(y)))`
2. **Functor Laws**:
   - **Identity Law**: `functor.map(x -> x)` is the same as `functor`
   - **Composition Law**: `functor.map(x -> f(x)).map(x -> g(x))` is the same as `functor.map(x -> g(f(x)))`.

Miscs

1. (**Bounded type parameter**): A type parameter, like `T` can have a bound! e.g., `T extends Fruit` means the type parameter `T` must be a subtype of `Fruit`.
2. **Make your method as flexible as possible!**: Usually, we use **producer extends** more.
3. (**Immutable class**):
   - When you are designing a class with a `List` as its member and you may use stream later, think about making it final to make the class immutable.
   - For Java **arrays** and **String**, just add keyword `final`.
   - When return a **non-primitive** type, **make a hard copy**! For example, to make a hard copy of `List`, the first argument is your List object, the second element is the length of your List.

```
public Customer[] getOwners() {
  return Arrays.copyOf(this.owners, this.owners.length);
}
```

4. **General advice for writing one-liner**
   - Start by considering the condition to use (This should be a object of type `Maybe`)
   - End by using `orElse()/orElseGet()`.
     - Anything between the start and the end is your **if branch**.
     - The "placeholder" in your end (in the `orElse()/orElseGet()`) is the **else branch**.
5. **Print Modifier**:

Specifier	Data Type
$\%d$	Integer (decimal)
$\%f$	Floating point (decimal)
$\%s$	String
$\%c$	Character
$\%b$	Boolean
$\%\%$	Literal % sign

To print the decimal point, we use `%.2f`(2 decimal places)

6. **Write a class with one static method**: In case need other methods, can also write like below

```
class Q7 {
  static class DiscountedBook extends Book {
    public DiscountedBook(Book b, double discount) {
      super(
        b.getTitle(), b.getGenre(),
        () -> b.getPrice() * (100 - discount) / 100
      );
    }
  }

  private static <T extends Book> BookShop<DiscountedBook>
    applyDiscount(BookShop<? extends T> bs, int k) {
      int bookNum = bs.getBooks()
                      .stream()
                      .reduce(
                        identity:0, (a, b) -> a + 1,
                        (a, b) -> a + b);
      if (bookNum < k) {
        return new BookShop<DiscountedBook>(
          bs.getBooks().stream()
                       .map(book ->
                         new DiscountedBook(book, discount:50))
                       .toList());
      }
      return new BookShop<DiscountedBook>();
  }

  public static <T extends Book> List<BookShop<DiscountedBook>>
    discountRemainingStock
      List<? extends BookShop<? extends T>> bss, int k) {
      return bss.stream()
                .map(bs -> Q7.applyDiscount(bs, k))
                .filter(bs -> !bs.isEmpty())
                .toList();
  }
}
```