Interfaces and Inheritance (Part 1)

Important Dates:

- Assigned: November 5, 2025
- Deadline: November 12, 2025 at 11:59 PM EST

Objectives:

- Students write and use both regular and functional interfaces to solve problems.
- Students design complex generic methods that leverage lambda expressions.
- Students use inheritance to model hierarchical relationships between objects.

What To Do:

Design classes with the given specification in each problem, along with the appropriate test suite. **Do not round your solutions!**

When writing tests, use exactly one file to test all classes for a problem. For example, for `Ticket`, write a class called `TicketTest` to test `Ticket`, `DiscountedTicket`, and `VirtualTicket`.

What You Cannot Use:

You cannot use any content beyond Chapter 4.4. Namely, do not use abstract classes or anything that trivializes the problem. Please contact a staff member if you are unsure about something you should or should not use. Any use of anything in the above-listed forbidden categories will result in a **zero** (0) on the problem set.

Problem 1:

An *infinite stream* is one that, in theory, produces infinite results! We have illustrated this with Java's `Stream` API, but now we're going to design our own. Consider the `IStream` interface below:

```
interface IStream<T> {
  T next();
}
```

When calling `next` on a stream, we update the contents of the stream and return the next result. We mark this as a generic interface to allow for any desired return type. For instance, below is a stream that produces factorial values:[1]

```
class FactorialStream implements IStream<Integer> {

  private int n;
  private int fact;

  FactorialStream() {
    this.n = 1;
    this.fact = 1;
  }

  @Override
  public Integer next() {
    this.fact *= this.n;
    this.n++;
    return this.fact;
  }
}
```

Testing it with ten calls to `next` yields predictable results.

```
class FactorialStreamTester {

  @Test
  void testFactorialStream() {
    IStream<Integer> FS = new FactorialStream();
    assertEquals(1, FS.next());
    assertEquals(2, FS.next());
    assertEquals(6, FS.next());
    assertEquals(24, FS.next());
    assertEquals(120, FS.next());
  }
}
```

[1] We will ignore the intricacies that come with Java's implementation of the `int` datatype. To make this truly infinite (up to the system's memory limit), we could use `BigInteger`.

Design the `FibonacciStream` class, which implements `IStream<Integer>` and correctly overrides `next` to produce Fibonacci sequence values. You code should *not* use any loops or recursion. Recall that the Fibonacci sequence is defined as $f(n) = f(n-1) + f(n-2)$ for all $n \geq 2$. The base cases are $f(0) = 0$ and $f(1) = 1$.

```
class FibonacciStreamTester {

  @Test
  void testFibonacciStream() {
    IStream<Integer> FS = new FibonacciStream();
    assertEquals(0, FS.next());
    assertEquals(1, FS.next());
    assertEquals(1, FS.next());
    assertEquals(2, FS.next());
    assertEquals(3, FS.next());
    assertEquals(5, FS.next());
  }
}
```

Problem 2:

Design the `CollatzStream` class, which implements `IStream<BigInteger>` and correctly overrides `next` to produce values corresponding to the Collatz numeric sequence. The class should have two constructors: one that receives an `int` and another that receives a `String` representing some arbitrarily large integer. Convert both of these into `BigInteger` values and assign them to instance variables. We use a `BigInteger` because the numbers in the Collatz sequence can grow arbitrarily large. Recall that the Collatz conjecture says that all integers $n \geq 1$ eventually converge to 1 after applying the following formula: if n is odd, then the next number is $3n + 1$, and otherwise $n/2$. You should include a clause to always return 1 if the current value is 1.

Problem 3:

Design the generic `StreamTake` class. Its constructor should receive an `IStream` and an integer *n* denoting how many elements to take, as parameters. Then, write a `List<T> getList()` method, which returns a `List<T>` of *n* elements from the given stream.

```
class StreamTakeTester {

 @Test
 void testStreamTake() {
  StreamTake llt1 = new StreamTake(new FactorialStream(), 8);
  StreamTake llt2 = new StreamTake(new FibonacciStream(), 10);
  assertEquals("[1, 2, 6, 24, 120, 720, 5040, 40320]",
                      llt1.getList().toString());
  assertEquals("[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]",
                      llt2.getList().toString());
 }
}
```

Problem 4:

Java's functional API allows us to pass lambda expressions as arguments to other methods, as well as method references (as we saw in Chapter 3). Design the generic `FunctionalStream` class to implement `IStream`, whose constructor receives a unary function `Function<T, T> f` and an initial value `T t`. Then, override the `next` method to invoke f on the current element of the stream and return the previous. For example, the following test case shows the expected results when creating a stream of infinite positive multiples of three.

```
class FunctionalStreamTester {

  @Test
  void testMultiplesOfThreeStream() {
    IStream<Integer> mtll = new FunctionalStream<>(x -> x + 3, 0);
    assertEquals(0, mtll.next());
    assertEquals(3, mtll.next());
    assertEquals(6, mtll.next());
    assertEquals(9, mtll.next());
    assertEquals(12, mtll.next());
  }
}
```

What's awesome about this exercise is that it allows us to define the elements of the stream as any arbitrary lambda expression, meaning that we could redefine `FactorialStream` and `FibonacciStream` in terms of `FunctionalStream`. We can generate infinitely many ones, squares, triples, or whatever else we desire.

Problem 5:

This exercise has three parts.

(a) Design the `INumberFormat` interface, which contains the `String format(int n)` method.

(b) Design the `DollarFormat` class, which implements `INumberFormat`, and returns a string where the number is prepended with a dollar sign `"$"`.

(c) Design the `CommaFormat` class, which implements `INumberFormat`, and returns a string where the number contains commas where appropriate. For example, `format(4412)` should return `"4,412"`.

(d) Finally, design the `StandardFormat` class, which implements `INumberFormat`, and returns a string where the number is simply returned as a string.

Problem 6:

Design the `ZipWith` class that contains one generic `static <T, U, R> List<R> zipWith(BiFunction<T, U, R> f, List<T> l1, List<U> l2)` method that receives two lists l_1 of type `T` and l_2 of type `U` respectively. It creates a resulting list of type R, which contains the elements of both lists after applying the binary function f. For example, if f is `(a, b) -> a + b`, l_1 is `[1, 2, 3]` and l_2 is `[4, 5, 6]`, then `zipWith` returns `[5, 7, 9]`. The binary function can be anything as long as it receives two parameters of type T and U and returns a type R. Note that T, U, and R do not necessarily need to be distinct. If the two lists are not the same length, use `null` for the pairing item.

Problem 7:

Design the `Ticket` class, which represents a ticket that can be purchased. A ticket has a price and a unique ticket identifier. Each ticket has a method `getPrice` that returns the cost of the ticket, and a method `getId` that returns the ticket's unique identifier. The first ticket's identifier is 0. The ticket identifier should be incremented via a `static` variable that is incremented and then assigned to the instance variable. Override the `hashCode` and `equals` methods as appropriate. The `Ticket` class constructor should only receive a ticket cost (in USD).

Then, design the following concrete subclasses (note that none of these concrete classes should override `hashCode` or `equals`):

- `DiscountedTicket`, which receives both the price and the discount as parameters. The discount should be a value between 0.0 and 1.0. Apply the discount inside an overridden `getPrice` method.
- `VirtualTicket`, which adds a convenience fee of $2.50 on top of whatever that ticket's price is.

Problem 8:

Design the `FoldRight` class, which contains one method: `static <T, U> U foldr(List<T> ls, BiFunction<T, U, U> f, U u)` method that receives a list of values ls, a function f, and an initial value u. The method should return the result of folding the list from the right with the given function and initial value. By "folding," we mean that we apply f to the last element of the list and the initial value, then apply f to the second-to-last element and the result of the previous application, and so forth. To think of this in terms of infix notation over some list, consider the list $[a, b, c, d]$. Folding it over the function $\circ$ and initial value u is $a \circ (b \circ (c \circ (d \circ u)))$. Do *not* use the `reduce` method, as that method solves the problem we want *you* to solve!

Problem 9:

In this exercise you will design a class for storing employees. This relies on having the `Employee` class and its subclasses from the chapter available.

(a) Design the `Job` class, which stores a list of employees `List<Employee>` as an instance variable. Whether you choose to instantiate it as an `ArrayList` or a `LinkedList` is up to you and makes little difference for this particular question. Its constructor should receive no arguments. The instance variable, along with its accessor and mutator, should be named `employees`, `getEmployees`, and `setEmployees` respectively.

(b) Design the `void addEmployee(Employee e)` method, which adds an employee to the `Job`.

(c) Design the `void removeEmployee(Employee e)` method, which removes an employee from the `Job`.

(d) Design the `Optional<Double> computeAverageSalary()` method, which returns the average salary of all employees in the `Job`. If there are no employees, return an empty `Optional`.

(e) Design the `Optional<Employee> highestPaid()` method, which returns the employee whose salary is the highest of all employees in the `Job`. If there are no employees, return an empty `Optional`.

(f) Override the `public String toString()` method to print out the list of employees in the `Job`. To make this easy, you can simply invoke the `toString` method from the `List` implementation.

Extra Credit (20 points):

We have seen and used the `map` method many times by now. Other languages such as Scheme support a multi-argument mapping function. That is, the stream `map` method receives a unary operator and a single list. A multi-argument mapping method would receive n lists, and have an n-argument function. Unfortunately, Java's type system is not powerful enough to support a mechanism for allowing polymorphically many inputs to a `Function<..., ...>`.[2] The next best option is to have the function receive an array of arguments.

For instance, suppose we have a list of unary operators and a list of numbers l. If we want to apply each operator to its corresponding element in l, we would supply a lambda expression (or method reference) that receives an array of values V and applies the first element to the second element.[3]

Another example that we present below is creating a list of lists containing the i^{th} element of each passed list.

[2]This use of polymorphic is distinct from the object-oriented meaning.

[3]When declaring a list of method references, we must initialize an explicit `List<...>` with the type annotation.

```java
class GenericMapTest {

  private static int fact(int n) { /* Omitted. */ }

  private static int fib(int n) { /* Omitted. */ }

  private static int addOne(int n) { /* Omitted. */ }

  private static int subOne(int n) { /* Omitted. */ }

  private static int applyGM(Function<Integer, Integer> f, int x, int y) {
    return f.apply(x) + y;
  }

  private static List<Integer> applyGM2(int x, int y, int z) {
    return List.of(x, y, z);
  }

  @Test
  void testGenericMap001() {
    List<Function<Integer, Integer>> fnList
            = List.of(GenericMapTest::fact, GenericMapTest::fib, GenericMapTest::addOne,
                GenericMapTest::subOne);
    Assertions.assertEquals(List.of(721, 57, 45, 46),
            GenericMap.gMap(V -> applyGM(V.get(0), V.get(1), V.get(2)),
                    fnList,
                    List.of(6, 10, 41, 43),
                    List.of(1, 2, 3, 4)));
  }

  @Test
  void testGenericMap002() {
    Assertions.assertEquals(List.of(List.of(1, 10, 100), List.of(2, 20, 200), List.of(3,
        30, 300)),
            GenericMap.gMap(V -> applyGM2(V.get(0), V.get(1), V.get(2)),
                    List.of(1, 2, 3),
                    List.of(10, 20, 30),
                    List.of(100, 200, 300)));
  }
}
```

Design the `static <T, R> List<R> gMap(Function<List<T>, R> mappingFn, List<? super T>... lists)` method that acts as a polymorphic map function. Remember that the `List<? super T>` acts as a lower bound on the kind of lists that we can pass to `gmap`. Namely, `gmap` receives lists whose type are any superclass of T, or T itself.