Indiana University
CSCI-C212: Introduction to Software Systems (F25)
Joshua Crotts

Problem Set 11
November 12, 2025

# Interfaces and Inheritance (Part 2)

## Important Dates:

- Assigned: November 12, 2025
- Deadline: November 19, 2025 at 11:59 PM EST

## Objectives:

- Students use abstract and final classes to write a programming language interpreter.
- Students implement a well-known Java data structure for arbitrarily-large integers.
- Students work more with streams to understand interface types and object hierarchies.

## What To Do:

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

## What You Cannot Use:

Anything from Chapter 4 and before is fair game for the chapter. Nothing beyond Chapter 4 is allowed. Problem 2 uses the `assertThrows` and `assertDoesNotThrow` methods to showcase that these methods do not stack overflow. Do not use these OR exceptions in your code. They are present solely for demonstrative purposes. 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.

## Warning:

This problem set is almost assuredly the most difficult problem set of the semester. It is tempting to use generative AI to solve it for you. Do not do this. We can very easily tell, with these harder problem sets, what *you* write versus what generative AI writes. You will receive a 0 for any suspected AI use and expected to orally defend your work. If you do not orally defend your answers, you will be reported for violating our academic integrity policy.

Seriously, just collaborate with other people, ask questions, and go to office hours. It's not that difficult to not cheat!

## Problem 1 (40 points):

In this series of problems, you will design several methods that act on very large/small *integers* resembling the `BigInteger` class. You ***cannot*** use any methods from `BigInteger`, or the `BigInteger` class itself.

(a) Design the `BigInt` class, which has a constructor that receives a string. The `BigInt` class stores a `List<Integer>` as an instance variable, as well as a boolean for keeping track of whether it is negative. You will need to convert the given string into said list of digits. Store the digits in reverse order, i.e., the least-significant digit (the ones digit) of the number is the first element of the list. Leading zeroes should be omitted from the representation.

The possible values for the parameter are an optional sign (either positive **or** negative), followed by one or more digits.

Below are some example inputs.

```
new BigInt("42")         => [2, 4], isNegative = false
new BigInt("0420")       => [0, 2, 4], isNegative = false
new BigInt("-42")        => [2, 4], isNegative = true
new BigInt("0000420000") => [0, 0, 0, 0, 2, 4], isNegative = false
new BigInt("+42")        => [2, 4], isNegative = false;
```

(b) Override the `public boolean equals(Object o)` method to return whether `this` instance represents the same integer as the parameter. If `o` is not an instance of `BigInt`, return `false`.

```
new BigInt("42").equals(new BigInt("42"))      => true
new BigInt("00042").equals(new BigInt("0042")) => true
new BigInt("+42").equals(new BigInt("42"))     => true
new BigInt("42").equals(new BigInt("-42"))     => false
new BigInt("-42").equals(new BigInt("42"))     => false
new BigInt("422").equals(new BigInt("420"))    => false
```

(c) Override the `public String toString()` method to return a stringified version of the number. Remember to include the negative sign where appropriate. If the number is positive, you do not need to include it.

(d) Implement the `Comparable<BigInt>` interface and override the method it provides, namely `public int compareTo(BigInt b2)`. Return the result of comparing `this` instance with the parameter. That is, if $a < b$, return $-1$, if $a > b$, return 1, and otherwise return 0, where $a$ is `this` and $b$ is `b2`.

(e) Design the `BigInt copy()` method, which returns a (deep)-copy of instance representing the same integer as `this` instance of `BigInt`. Do *not* simply copy the reference to the list of digits over to the new `BigInt` (this violates the aliasing principle that we have repeatedly discussed and can be the cause of relentless debugging).

(f) Design the `BigInt negate()` method, which returns a copy of `this` instance of `BigInt`, but negated. Do *not* modify `this` instance.

(g) Design the `private boolean areDifferentSigns(BigInt b)` method, which returns whether `this` instance and `b` have different signs. That is, if one is positive and one is negative, `areDifferentSigns` returns `true`, and false otherwise.

(h) Design the `private BigInt addPositive(BigInt b)` method, which returns a `BigInt` instance that is the sum of `this` and `b` under the assumption that `this` and `b` are non-negative. We will use $A \oplus B$ to symbolically represent this version of addition. *Be sure you thoroughly test this method!*

(i) Design the `private BigInt subPositive(BigInt b)` method, which returns a `BigInt` instance that is the difference of `this` and `b` under the assumption that `this` and `b` are non-negative, and the minuend (the left-hand operand) is greater than or equal to the subtrahend (the right-hand operand). We will use $A \dot{-} B$ to symbolically represent this version of subtraction. *Be sure you thoroughly test this method!*

(j) Design the `private BigInt mulPositive(BigInt b)` method, which returns a `BigInt` instance that is the product of `this` and `b` under the assumption that `this` and `b` are non-negative. *Be sure you thoroughly test this method!*

(k) Design the `BigInt add(BigInt b)` and `BigInt sub(BigInt b)` method that returns the sum/difference of `this` and `b` respectively. Note that these methods should be a case analysis of the signs of the operands. Use the following equivalences to guide your design. Do *not* over-complicate these methods. Importantly, `sub` can be written in exactly one line!

$$\begin{aligned} A - B &= A + (-B) \\ (-A) + (-B) &= -(A + B). \\ A + (-B) &= A \dot{-} B \text{ if } |A| \geq |B|. \text{ Otherwise, } -(B \dot{-} A). \\ (-A) + B &= B + (-A). \end{aligned}$$

(l) Design the `BigInt mul(BigInt b)` method that returns the product of `this` and `b`. The product of two negative integers is a positive integer, and the product of exactly one positive and exactly one negative is a negative integer.

## Problem 2 (10 points):

In Chapter 2, we discussed tail recursion and an action performed by some programming languages known as tail-call optimization. We know that we can convert any (tail) recursive algorithm into one that uses a loop, and we described said process in the chapter. There is yet another approach that we can mimic in Java with a bit of trickery and interfaces.

The problem with tail recursion (and recursion in general) in Java is the fact that it does not convert tail calls into iteration, which means the stack quickly overflows with activation records. We can make use of a *trampoline* to force the recursion into iteration through *thunks*. In essence, we have a tail recursive method that returns either a value or makes a tail recursive call, such as the factorial example below. Inside our base case, we invoke the `done` method with the accumulator value. Otherwise, we invoke the `call` method containing a lambda expression of no arguments, whose right-hand side is a recursive call to `factTR`. Functions, or lambda expressions, that receive no arguments are called thunks.

```
class FactorialTailRecursiveTest {

  @Test
  void testFactTailRecursiveTrampoline() {
    assertEquals(BigInteger.valueOf(120),
                FactorialTailRecursive.factTailCall(BigInteger.valueOf(5),
                                                    BigInteger.ONE).invoke());
    assertDoesNotThrow(() ->
                     FactorialTailRecursive.factTailCall(BigInteger.valueOf(50000),
                                                         BigInteger.ONE).invoke());
  }
}

class FactorialTailRecursive {

  /**
   * Tail-recursive factorial function. Uses BigInteger to
   * avoid number overflow and thunks to avoid stack overflow.
   * @param n the number to compute the factorial of.
   * @param ac the accumulator.
   * @return a tail call that is either done or not done.
   */
  static ITailCall<BigInteger> factTailCall(BigInteger n, BigInteger ac) {
    if (n.equals(BigInteger.ZERO)) {
      return TailCallUtils.done(ac);
    } else {
      return TailCallUtils.call(() ->
                factTailCall(n.subtract(BigInteger.ONE), ac.multiply(n)));
    }
  }
}
```

The idea is that we have a helper class and method, namely `invoke`, that continuously applies the thunks, **inside a while loop**, until the computation is done. The trampoline analogy is used because we bounce on the trampoline while invoking thunks and jump off when we are "done." The reason why this works, in particular, is because tail calls are the last operations to perform before the method returns. We're effectively trading stack space for heap space; method invocations are made on the procedure call stack, whereas we allocate lambda expressions and objects on the heap.

(a) First, design the generic `ITailCall<T>` interface. It should contain only one (non-default) method: `ITailCall<T> apply()`, which is necessary for the `invoke` method. The remaining methods are all default, meaning they must have a body. Design the `boolean isDone()` method to always return false. Design the `T getValue()` method to simply return `null`. Finally, design the `T invoke()` method that stores a local variable and constantly calls `apply` on it until it is "done."

(b) Second, design the `TailCallUtils` final class to contain a private constructor (this class will only utilize and define two static methods). The two methods are as follows:

- `static <T> ITailCall<T> call(ITailCall<T> next)`, which receives and returns the next tail call to apply. This definition should be exactly one line long and as simple as it seems.
- `static <T> ITailCall<T> done(T val)`, which receives the value to return from the trampoline. We need to create an instance of an interface, which sounds bizarre, but is possible only when we provide an implementation of its methods. So, return a `new ITailCall<>()`, and inside its body, override the `isDone` and `getValue` methods with the correct bodies.

Finally, run the factorial test that we provided earlier in its JUnit suite. It should pass and not stack overflow, hence the inclusion of an `assertDoesNotThrow` call.

**Note: when writing tests, place them in the `TailCallTest.java` file.**

## Problem 3 (20 points):

This exercise is multi-part and involves the interpreter we wrote in Chapter 4. The starter code you need has been given to you. When writing tests, write **your** tests in a file called `AsplTest.java`.

(a) First, design the `ProgramNode` class, which allows the user to define a program as a sequence of statements rather than a single expression. It works by evaluating the nodes inside a `ProgramNode` sequentially, then returns the result of the final evaluation.

(b) Design the `DefNode` class, which allows the user to create a global definition. Because we're now working with definitions that do not extend the environment, we should use the set method in environment. When creating a global definition via `DefNode`, we're expressing the idea that, from that point forward, the (root) environment should contain a binding from the identifier to whatever value it binds. Because a `DefNode` is a statement, its evaluation should return `null`.

(c) Design the `FuncNode` node. We will consider a function definition as an abstract tree node that begins with `FuncNode`. This node has two parameters to its constructor: a list of parameter (string) identifiers, and a single abstract syntax tree node representing the body of the function. We will only consider functions that return values; void functions do not exist in this language.

(d) Design the `ApplyNode` class, which applies a function to its arguments. You do not need to consider applications in which the first argument is a non-function.

Calling/Invoking a function is perhaps the hardest part of this exercise. Here's the idea, which is synonymous and shared with almost all programming languages:

(i) First, evaluate each argument of the function call. This will result in several evaluated abstract syntax trees, which should be stored in a list.

(ii) We then want to create an environment in which the formal parameters are bound to their arguments. Overload the extend method in `Environment` to now receive a list of string identifiers and a list of (evaluated) AST arguments. Bind each formal to its corresponding AST, and return the extended environment.

(iii) Evaluate the function abstract syntax tree to get its function definition as an abstract syntax tree.

(iv) Call `eval` on the function body and pass the new (extended) environment.

This seems like a lot of work (because it is), but it means you can write really cool programs, including those that use recursion!

```
new ProgramNode(
  new DefNode("!",
    new FuncNode(
      List.of("n"),
      new IfNode(
        new PrimNode("eq?",
          new VarNode("n"),
          new NumNode(0)),
        new NumNode(1),
        new PrimNode("*",
          new VarNode("n"),
          new ApplyNode(
            new VarNode("!"),
            new PrimNode("-",
              new VarNode("n"),
              new NumNode(1)))))))),
  new ApplyNode(new VarNode("!"), new NumNode(5)))
```

## Problem 4 (10 points):

This exercise involves the interpreter we wrote in Chapter 4, and relies on the addition of `FuncNode` and `ApplyNode`. (So, do Problem 3 before you do this one!) Our current version of the interpreter uses *dynamic scoping*. A dynamically-scoped interpreter is one that uses the value of the closest declaration of a variable. This seems like nonsense without an example, so consider the following code listing.

```
(define f
  (let ([x 10])
    (λ ()
      x)))

(let ([x 3])
  (f))
```

Under dynamic scoping, this program outputs `3`, because the binding of `x` takes on the value `3`. On the other hand, if we were using a *lexically-scoped* interpreter, the program would output `10`, which seems to make more sense due to the binding that exists immediately above the function declaration. The question is: how do we implement lexical scoping into our interpreter? The answer is via *closures*, which are data structures that couple a function definition with an environment. Then, when we apply a closure to an argument (if it exists), we restore the environment that was captured by the closure.

To this end, design the `ClosureNode` class, whose constructor receives a `FuncNode` and an `Environment`. The respective `eval` method returns `this`, but `FuncNode` changes slightly. Rather than returning `this`, we return a `ClosureNode`, which wraps the current `FuncNode` and the environment passed to `eval`. Finally, inside `ApplyNode`, evaluating the function definition should resolve to a closure. Evaluate the arguments to the closure inside the passed environment, but *extend* the captured environment, and bind the formals to the arguments in this extended environment. The body of the closure's function definition is then evaluated inside this new environment.

Making this alteration not only causes our programs to output the "common sense" result, but also means we can implement recursive functions using *only* a `LetNode`. See if you can figure out how to do this!

## Problem 5 (10 points):

*Finite streams* are streams that represent non-infinite data. In this exercise, you will implement a representation of finite streams. We have provided to you a version of `IStream`, which you should have written in Problem Set 10.

Finite streams work as follows: we can query when they are finished generating, retrieve the next value, or iterate the call to `next` until it is done. Once the finite stream has finished generating data, it will cache the final result into an instance variable, then return that result upon any future calls to `next` or an iteration.

(a) In this first step, you will design the abstract class to represent finite streams.

(i) Design the `FiniteStream<T>` abstract class, which implements `IStream`. Its constructor should receive no arguments.

(ii) Design the `abstract boolean isDone()` method. Any subclass that extends `FiniteStream` will specify when the stream has finished generating data.

(iii) Design the `abstract T produce()` method. This method should be overridden by the subclass to specify how the next element is retrieved. This method is identical in concept to the `IStream` method `next`.

(iv) Override `next` to now be `final`. Its definition should do the following: (1) if the stream is finished generating data, return a cached result. (2) Otherwise, invoke `produce`, cache the result, then return it.

(v) Design the `final T iterate()` method. This method should loop by repeatedly calling `next` until `isDone` returns true. Once so, return the cached result.

(b) Now, let's write a simple class to use `FiniteStream`.

(i) Design the `CountFiniteStream` method, which extends `FiniteStream<Integer>`, and whose constructor receives a positive integer $n > 0$. This stream will count, starting from 0, up to but not including $n$.

(ii) Override `produce` to store the current count in a variable $t$, increment the (instance variable) counter, then return $t$.

(iii) Override `isDone` to return whether the counter equals or exceeds $n$.

**Note: when writing tests, place them in the `FiniteStreamTest.java` file.**

## Problem 6 (10 points):

The Python programming language supports the use of `range(n)`, which returns a generator/stream/lazy list of integers in the interval $[0, n)$. For example, below is a segment of Python code that sums the numbers from 0 to a given value of $n$:

```
def sumNums(n):
  s = 0
  for i in range(n):
    s += i
  return s
```

It also supports a more powerful `range(start, end, step)` generator function, which returns a generator of integers in the interval [`start`, `end`) in changes of `step`.

```
def sumEvenNums(a, b):
  s = 0
  for i in range(a, b + 1, 2)
    s += i
  return s
```

In this exercise you will design the `RangeStream` class, which extends `FiniteStream<Integer>` and implements two interfaces: `Iterator<Integer>` and `Iterable<Integer>`. Chapter 4 showed what `Iterable` is and how it works as an abstraction over `Iterator`.

(a) Design the `RangeStream` class. It should contain two private constructors:

- `private RangeStream(int s, int e, Function<Integer, Integer> f)`, which receives a starting integer `s`, an ending value `e`, and a unary function over integers `f`. The provided function determines how to go from the current number to the next. (This should be eerily similar to your `FunctionalStream` class!)
- `private RangeStream(int n)`, which receives an ending value `n`. Invoking this constructor should call the other constructor with `0, n, x -> x + 1`. In other words, calling the constructor with only one value produces a generator of integers from $[0, n)$ in one-step increments.

Of course, the class should store the necessary and relevant instance variables.

(b) Because your class extends `FiniteStream<Integer>`, you must override `isDone` and `produce`. Override these methods appropriately, as well as the ones from `Iterator`: `next` and `hasNext`.

(c) Because your class implements `Iterable<Integer>`, you must override `public Iterator<Integer> iterator()`. Of course, the `RangeStream` is definitionally an `Iterator<Integer>`, so just return `this`.

(d) Design the following static methods (if you're wondering why we privatize the constructors and instead opt to use static methods, it's because we want to simulate Python's `range` function):

- `static RangeStream range(int s, int e, Function<Integer, Integer> f)` that returns an instance of a new `RangeStream` containing the provided fields.
- `static RangeStream range(int n)`, which returns an instance of a new `RangeStream` containing the provided field.