UMass Amherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 8: Streams and OOP

Wednesday, October 21st, 2024

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Please sit with your group from last week.
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- Download the starter code.
- Homework 5 is due tonight at 11:59pm
  - Come to office hours for help!
- Homework 6 is available
- Complete the CATME Survey by TBD at midnight
- Midterm 2 is in two weeks!
  - Start studying early.

Today's Goals

- Practice working with the OOP
- Practice working with streams

OOP Review

- One important aspect of OOP and Fluent Design is method chaining.
- By having methods return this, meaning they return a reference to themselves, we can chain method calls.
- We often want to hide properties of objects to prevent unwanted modification.
  - If we know that some property will have to be changed elsewhere, we should implement getter and setter methods that will do this for us instead of changing properties at will.

Exercise 1

Write a class FluentFilter that has a data member which is an array of objects, passed as argument to the constructor. The class has a fluent filter method, with an array of criteria as an argument: filter(Criterion[]): FluentFilter. A criterion is an object with two fields: type Criterion<T> = { name: string, f: T => boolean }. The filter method produces a new FluentFilter object, whose array contains those objects that satisfy all filter criteria: i.e., for each criterion, the object has a property with the given name, and the value of that property satisfies the boolean function f.

For full credit, use a single pass through the array of objects, and do not use loops.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 1 Solution

```
type Criterion<T> = { name: string, f: (g: T) => boolean};
type Data = {[key: string]: any};

class FluentFilter<T> {
  data: Data[];

  constructor (data: Data[]) {
    this.data = data;
  }
}
```

First let's declare data to be a field on the object

Then in the constructor we'll set the data field to hold the data passed in in the argument.

Exercise 1 Solution


```
type Criterion<T> = { name: string, f: (g: T) => boolean};
type Data = {[key: string]: any};

class FluentFilter<T> {
  data: Data[];

  constructor (data: Data[]) {
    this.data = data;
  }

  filter (crit: Criterion<T>[]): FluentFilter<T> {

  }
}
```

Our filter method will need to return a FluentFilter

Exercise 1 Solution

```
type Criterion<T> = { name: string, f: (g: T) => boolean};
type Data = {[key: string]: any};

class FluentFilter<T> {
  data: Data[];

  constructor (data: Data[]) {
    this.data = data;
  }

  filter (crit: Criterion<T>[]): FluentFilter<T> {
    return new FluentFilter<T>(this.data.filter(???));
  }
}
```

But what should our filter condition be?

We can return a new FluentFilter, using the array filter method on our data.

Exercise 1 Solution

```
type Criterion<T> = { name: string, f: (g: T) => boolean};
type Data = {[key: string]: any};

class FluentFilter<T> {
  data: Data[];

  constructor (data: Data[]) {
    this.data = data;
  }

  filter (crit: Criterion<T>[]): FluentFilter<T> {
    function allCrit(o: Data): boolean {

    }
    return new FluentFilter<T>(this.data.filter(allCrit));
  }
}
```

Let's remember that the array filter method takes in a function. We'll explicitly write this function out here.

Exercise 1 Solution

```
type Criterion<T> = { name: string, f: (g: T) => boolean};
type Data = {[key: string]: any};

class FluentFilter<T> {
  data: Data[];

  constructor (data: Data[]) {
    this.data = data;
  }

  filter (crit: Criterion<T>[]): FluentFilter<T> {
    function allCrit(o: Data): boolean {
      return crit.every(c => {

      });
    }
    return new FluentFilter<T>(this.data.filter(allCrit));
  }
}
```

To ensure very criterion in the crit array is met, let’s use the every array method

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 1 Solution

```
type Criterion<T> = { name: string, f: (g: T) => boolean};
type Data = {[key: string]: any};

class FluentFilter<T> {
  data: Data[];

  constructor (data: Data[]) {
    this.data = data;
  }

  filter (crit: Criterion<T>[]): FluentFilter<T> {
    function allCrit(o: Data): boolean {
      return crit.every(c => {
        return (c.name in o) && ???;
      });
    }
    return new FluentFilter<T>(this.data.filter(allCrit));
  }
}
```

First let's check that c.name is actually a property on the Data object o

Exercise 1 Solution

```
type Criterion<T> = { name: string, f: (g: T) => boolean};
type Data = {[key: string]: any};

class FluentFilter<T> {
  data: Data[];

  constructor (data: Data[]) {
    this.data = data;
  }

  filter (crit: Criterion<T>[]): FluentFilter<T> {
    function allCrit(o: Data): boolean {
      return crit.every(c => {
        return (c.name in o) && c.f(o[c.name]);
      });
    }
    return new FluentFilter<T>(this.data.filter(allCrit));
  }
}
```

Now that we know the property exists, we can apply f to the value stored c.name in o.

Exercise 1 Solution

```
type Criterion<T> = { name: string, f: (g: T) => boolean};
type Data = {[key: string]: any};

class FluentFilter<T> {
  data: Data[];

  constructor (data: Data[]) {
    this.data = data;
  }

  filter (crit: Criterion<T>[]): FluentFilter<T> {
    function allCrit(o: Data): boolean {
      return crit.every(c => {
        return (c.name in o) && c.f(o[c.name]);
      });
    }
    return new FluentFilter<T>(this.data.filter(allCrit));
  }
}
```

Stream Review

- What: A sequence of data made available over time
- Why: Useful abstraction for the paradigm where there's limited random data access and each data record can only be seen once*. E.g: Data reading, signal processing
- How: We implemented stream as a lazily constructed list with memoized tail

```
interface Stream<T> {
  head: () => T;
  tail: () => Stream<T>;
  isEmpty: () => boolean;
  toString: () => string;
  map: <U>(f: (x: T) => U) => Stream<U>;
  filter: (f: (x: T) => boolean) => Stream<T>;
  reduce: <U>(f: (acc: U, e: T) => U, init: U) => Stream<U>; // This is new
}

reduce: (f, init) => snode(init, () => memoizedTail.get().reduce(f, f(init, head)))
```

Exercise 2: Maxima stream (in a previous exam!)

- Implement maxUpTo(s: Stream<number>): Stream<number>
- Input: A stream of numbers a1, a2, a3, ...,
- Output: A stream of maxima of numbers up to the current one:
  a1 => max(a1, a2) => max(a1, a2, a3) =>.... => sempty
- Example:
  Input stream: 1 => 4 => 3 => 2 => 5 => 1 => sempty
  Output stream: 1 => 4 => 4 => 4 => 5 => 5 => sempty

Exercise 2: Solution

```
// Solution 1
function maxUpTo(s: Stream<number>): Stream<number> {
    function maxUpToHelper(s: Stream<number>, prevMax: number): Stream<number> {
        if (s.isEmpty()) {
            return s;
        }
        const curMax = Math.max(prevMax, s.head());
        return snode(curMax, () => maxUpToHelper(s.tail(), curMax));
    }

    return maxUpToHelper(s, -Infinity);
}

// Solution 2
function maxUpTo(s: Stream<number>): Stream<number> {
    let max = -Infinity;
    return s.map(x => max = Math.max(x, max));
}

// Solution 3
function maxUpTo(s: Stream<number>): Stream<number> {
    return s.reduce(Math.max, -Infinity).tail(); // Why .tail()?
}
```

Exercise 3: Closures vs Streams

Implement the following two functions, both take no arguments:

- harmonicClosure: returns a closure where the nth call returns the value of the nth term in the harmonic series
- harmonicStream: returns a stream where the value stored in the nth node is the nth term in the harmonic series

Avoid recomutations.

$$\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots.$$

Exercise 3: Solution

```
function harmonicClosure(): () => number {
  let n = 1;
  let sum = 0;
  return function () {
    sum += 1 / n++;
    return sum;
  };
}
```

```
function harmonicStream(): Stream<number> {
  let n = 1;
  let sum = 0;
  function helper(): Stream<number> {
    sum += 1 / n++;
    return snode(sum, helper);
  }
  return helper();
}
```