Programming Methodology

Lab 10: Midterm 2 Review

Wednesday, November 6th, 2024

Reminders

- HW7 Teams released on CATME, contact teammates to set up first meeting soon after the exam
- Midterm 2 is tonight 7pm
    - ISB 135 (Integrated Sciences Building)
  - Advice:
    - Write legibly (reduces risk of misgrading)
    - Check the interfaces on the front page when you have doubts about typing
    - Write the types of function parameters and return values
    - The order of the questions doesn’t mean anything

Today's Goals

- Review midterm 2 material

Observer Review

- What: A design pattern in which an observable subject automatically notifies dependent observers of any state changes
- Why: It's everywhere. E.g: GUI updates
- How: Reusable class

```
type Observer<T> = (x: T) => any;

class Observable<T> {
  private observers: Observer<T>[] = [];  // Maintain a list of observers

  subscribe(f: Observer<T>) {             // Add an observer to the list
    this.observers.push(f);
  }

  update(x: T) {                          // Notify each observer of update
    this.observers.forEach(f => f(x));
  }
}
```

Exercise 1: Observers

Implement an **observable** class **`PublicFigure`**:

- It has a public object attribute **`profile`** which contains **`name`** and **`status`** attributes
- It has a method **`updateStatus`** which updates the **`profile.status`** attribute and notifies subscribers of the new **`profile`**

Implement another class called **`Follower`**:

- It has attributes **`name`**, **`following`**, and **`followLimit`**
  - **`following`** stores how many **`PublicFigures`** the **`Follower`** is following, and **`followLimit`** denotes the maximum amount the **`Follower`** can follow
- It has a **fluent** method **`follow`** that takes a **`PublicFigure`** and subscribes an **`Observer`** (if the **`followLimit`** has not been met) that prints the status of **`PublicFigure`** every time it updates, along with how many times that **`PublicFigure`** has updated so far

Exercise 1: Solution

```
interface Profile {
  name: string,
  status: string
}

class PublicFigure extends Observable<Profile> {
  profile: Profile;
  constructor(name: string, status: string) {
      super();
      this.profile = { name: name,
                       status: status };
  }

  updateStatus(newStatus: string): void {
      this.profile.status = newStatus;
      this.update(this.profile);
  }
}
```

```
class Follower {
  private following = 0;
  private name: string;
  private followLimit: number
  constructor(name: string, followLimit: number) {
                this.name = name;
                this.followLimit = followLimit;
  }
  follow(publicFigure: PublicFigure): Follower {
      if (this.following < this.followLimit) {
            let updateNum = 0;
            publicFigure.subscribe(profile =>
                console.log(profile.name + " update
                number " + ++updateNum + ": " +
                profile.status));
            ++this.following;
      }
      return this;
  }
}
```

Exercise 1: Solution cont.

```
const pf1 = new PublicFigure("pf1", "Status you won't see because you haven't subscribed yet");
const pf2 = new PublicFigure("pf2", "Status you won't see because you haven't subscribed yet");
const pf3 = new PublicFigure("pf3", "Status you won't see because you haven't subscribed yet");

const f1 = new Follower("f1", 2);
const f2 = new Follower("f2", 2);
f1.follow(pf1).follow(pf2).follow(pf3);
f2.follow(pf2);

pf1.updateStatus("pf1 status update"); //prints “pf1 update number 1: pf1 status update"
pf2.updateStatus("pf2 status update"); //prints "pf2 update number 1: pf2 status update" twice
pf3.updateStatus("f1 won't see this update from pf3 because their follow limit is met");
pf1.updateStatus("pf1 second update"); //prints "pf1 update number 2: pf1 status update"
```

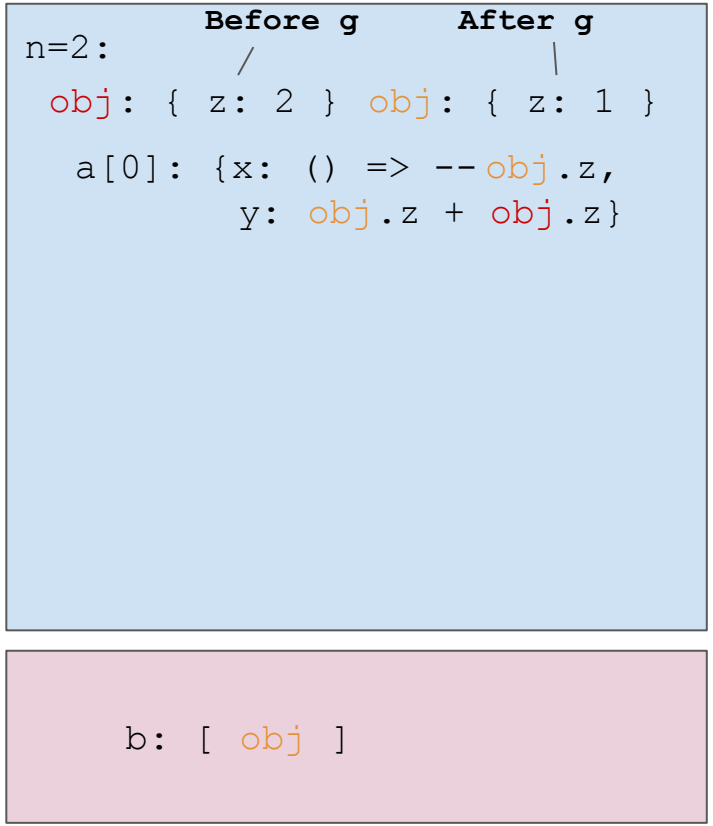
Exercise 2: Mental Models

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15     console.log(a[i].x(), a[i].y + b[i].z)
16  );
```

For each line of the code, explain what objects and closures are created, what values are computed and printed, and when objects are no longer accessible.

Exercise 2: Solution

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15     console.log(a[i].x(), a[i].y + b[i].z)
16  );
```



Exercise 2: Solution

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15     console.log(a[i].x(), a[i].y + b[i].z)
16  );
```

```
n=2:
 obj: { z: 2 } obj: { z: 1 }
  a[0]: {x: () => --obj.z,
         y: obj.z + obj.z}
n=1:
 obj: { z: 1 } obj: { z: 0 }
  a[1]: {x: () => --obj.z,
         y: obj.z + obj.z}
```

```
b: [ obj, obj ]
```

Exercise 2: Solution

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15     console.log(a[i].x(), a[i].y + b[i].z)
16  );
```

```
n=2:
 obj: { z: 2 } obj: { z: 1 }
  a[0]: {x: () => --obj.z,
         y: obj.z + obj.z}
n=1:
 obj: { z: 1 } obj: { z: 0 }
  a[1]: {x: () => --obj.z,
         y: obj.z + obj.z}
n=0:
 obj: { z: 0 } obj: {z: -1}
  a[2]: {x: () => --obj.z,
         y: obj.z + obj.z}
```

```
b: [ obj, obj, obj ]
```

Exercise 2: Solution

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15     console.log(a[i].x(), a[i].y + b[i].z)
16  );
```

```
n=2:
 obj: { z: 2 } obj: { z: 1 }
  a[0]: {x: () => --obj.z,
         y: 1 + 2}
n=1:
 obj: { z: 1 } obj: { z: 0 }
  a[1]: {x: () => --obj.z,
         y: 0 + 1}
n=0:
 obj: { z: 0 } obj: {z: -1}
  a[2]: {x: () => --obj.z,
         y: -1 + 0}
```

```
b: [ obj, obj, obj ]
```

Exercise 2: Solution

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15     console.log(a[i].x(), a[i].y + b[i].z)
16  );
```

Resulting array garbage collected

```
n=2:
 obj: { z: 2 } obj: { z: 3 }
  a[0]: {x: () => --obj.z,
         y: 3}
n=1:
 obj: { z: 1 } obj: { z: 2 }
  a[1]: {x: () => --obj.z,
         y: 1}
n=0:
 obj: { z: 0 } obj: { z: 1 }
  a[2]: {x: () => --obj.z,
         y: -1}
```

```
b: [ obj, obj, obj ]
```

Exercise 2: Solution

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15     console.log(a[i].x(), a[i].y + b[i].z)
16  );
```

```
n=2:
 obj: { z: 2 } obj: { z: 2 }
  a[0]: {x: () => --obj.z,
         y: 3}
n=1:
 obj: { z: 1 } obj: { z: 2 }
  a[1]: {x: () => --obj.z,
         y: 1}
n=0:
 obj: { z: 0 } obj: { z: 1 }
  a[2]: {x: () => --obj.z,
         y: -1}
```

```
i=0:
    a[0].x(): --3 -> 2
    a[0].y + b[0].z: 3+2=5
```

Exercise 2: Solution

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15     console.log(a[i].x(), a[i].y + b[i].z)
16  );
```

```
n=2:
 obj: { z: 2 } obj: { z: 2 }
  a[0]: {x: () => --obj.z,
         y: 3}
n=1:
 obj: { z: 1 } obj: { z: 1 }
  a[1]: {x: () => --obj.z,
         y: 1}
n=0:
 obj: { z: 0 } obj: { z: 1 }
  a[2]: {x: () => --obj.z,
         y: -1}
```

```
i=1:
    a[1].x(): --2 -> 1
    a[1].y + b[1].z: 1+1=2
```

Exercise 2: Solution

```
 1  const f = obj => () => --obj.z;
 2  const g = (obj, z) => --obj.z + z;
 3
 4  let n = 3;
 5  const a = [];
 6  const b = [];
 7  while (--n >= 0) {
 8    const obj = { z: n };
 9    a.push({ x: f(obj), y: g(obj, obj.z) });
10    b.push(obj);
11  }
12
13  b.map(o => o.z += 2);
14  [0, 1, 2].forEach(i =>
15    console.log(a[i].x(), a[i].y + b[i].z)
16  );
```

Array gets garbage collected!

```
n=2:
 obj: { z: 2 } obj: { z: 2 }
  a[0]: {x: () => --obj.z,
         y: 3}
n=1:
 obj: { z: 1 } obj: { z: 1 }
  a[1]: {x: () => --obj.z,
         y: 1}
n=0:
 obj: { z: 0 } obj: { z: 0 }
  a[2]: {x: () => --obj.z,
         y: -1}
```

```
i=2:
    a[2].x(): --1 -> 0
    a[2].y + b[2].z: -1+0=-1
```

Exercise 3: Property Based Testing

Write property-based tests for function **`partition(arr: number[], p: number)`** with the following specification:

- The array elements and **`p`** are assumed to be integers between 0 and **`arr.length-1`**.
- The function rearranges the array in place, so that: a number with value > **`p`** will not appear before a number with value <= **`p`**, and a number with value < **`p`** will not appear after a number with value >= **`p`**.

Try to write a complete set of tests. You need not write code, but then describe very clearly for each test what you check and how.

Exercise 3: Solution

Tests:

1. Check that the original and resulting array have the same length
2. Check that they contain the same elements with the same frequencies. Together, this can be done by making a copy before partition and comparing:
3. Check that the result has elements less than `p`, then elements equal to `p`, then elements greater than `p` (any of those may be missing).

Exercise 3: Solution

Tests:

1. Check that the original and resulting array have the same length
2. Check that they contain the same elements with the same frequencies. Together, this can be done by making a copy before partition and comparing:
3. Check that the result has elements less than `p`, then elements equal to `p`, then elements greater than `p` (any of those may be missing).

```
function checkSame(a: number[], b: number[]): void
{
  assert(a.length === b.length);
  const inRange = e => 0 <= e && e < a.length;
  let f = new Array<number>(a.length).fill(0);
  b.forEach(e => { assert(inRange(e)); ++f[e]; });
  a.forEach(e => assert(--f[e] >= 0));
}
```

Counts elements

```
function checkPivot(a: number[], p: number): void {
  let i = a.length; if (i === 0) { return; }
  while (i > 0 && a[--i] > p) {} //all > p
  while (i >= 0 && a[i] === p) { --i; } //all = p
  while (i >= 0) { assert(a[i] < p); --i; }
}
```

Exercise 4: Streams

Write a function that takes an infinite stream of numbers `a1,a2,…` and a positive integer `k`, and returns a stream of running averages of `k` values, starting with `average(a1,a2,…,ak)`, `average(a2,a3,…,ak+1)`, etc

Exercise 4: Solution

```
function runavg(s, k) { //assumes s infinite
  const start = s; //save reference to start of stream
  let sum = 0;
  for (let i = k; --i > 0; s = s.tail()) { sum += s.head(); } //k - 1 times
  function avg(p, s, sum) {
    sum += s.head();
    return snode(sum/k, () => avg(p.tail(), s.tail(), sum-p.head()));
  }
  return avg(start, s, sum);
}
```

Remove value of element k-1 elements back in the stream from running sum