Review Session #8

(It’s lit y'all)

Outline

- Arrays
- While Loops
- Loops vs Recursion
- Array Built-in Methods
- Nesting and Chaining Function Calls
- Values vs References

What’s the best way to make more money in the software engineering field?

Ask for arrays!

What's an *Array*?

- An array can be used similarly to a List, but there are some important differences to take note of.

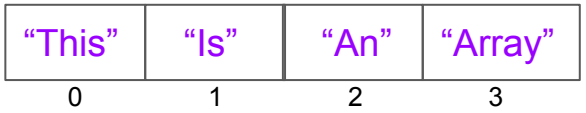
A List:

Ex. "This" → "Is" → "A" → "List" → null

- No inherent indexing system.
- Inherent properties w/ methods to access them (cons, car, rest)
- Recursive structure w/ null terminator
- Can be used with different types, but no single list can by made up of different types.
- The focus of some specific programming languages called "list-processing languages".

An Array:

Ex.



- Built-in indexing system.
- Inherent properties w/ methods to access them (length, map, sort)
- Question: does this have a recursive structure?
- Can be used with different types, but no single array can by made up of different types.
- (Basically) Universally used in CS.

How can we make use of these wondrous objects?

- The process of declaring and instantiating an array isn't that different from creating a list.

To make a List:

```
let disList: List<string> = listify("List!", null);
let emptyList: List<string> = null;
```

To access items in a List:

```
print(first(disList));          // prints: "List!"
print(rest(disList));           // prints:  null
```

To make an Array:

```
let disArray: string [ ] = ["Array!"];
let emptyArray: string [ ] = [ ];
```

To access items in an array:

```
print(disArray[0]);          // prints: "Array!"
print(rest(disArray[1]));    // prints:  undefined
```

More on Arrays

- Just like Lists, Arrays can be made with most types, *but they can only hold items of the same type!*
- Also similar to Lists, each “spot” in an array is reserved for only 1 item.

Ex. Is this allowed?

No ma'am!

“It”	“is”	“so” “very”	“lit”	“rn”

No sir!

“It”	“is”	2	“lit”	“rn”

Lookin good!

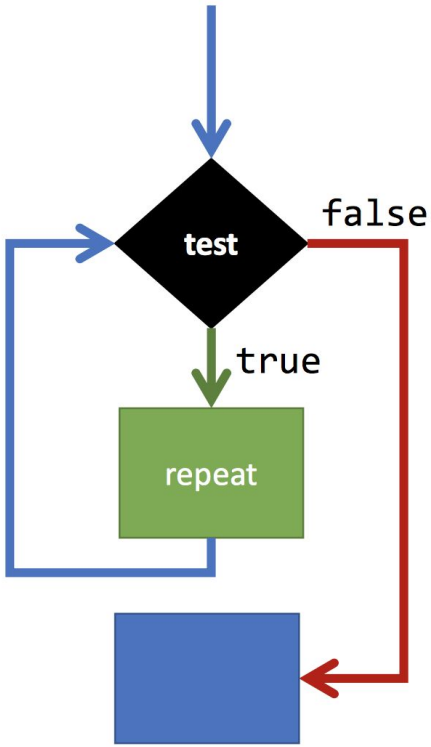
“It”	“is”	“fairly”	“lit”	“rn”

Now… it’s time to talk about the zaniest computer science subject *ever*…

LOOP DE LOOPS

“While-Loops”: They Work For You

- Really good at doing *a ton* of work.
- While-loops will, (in order):
  1. Ask if you want them to do work for you (via boolean test).
  2. True? They will do work for you.
     2.1. Then they jump back to Step 1.
  3. False? They stop doing work for you.



While-Loops *In Action*

- We can do many of the same things with looping that we've done with recursion. (summing, counting, finding minimum or maximum).
- Unlike recursion, where an entire function body is included in the recursive process, a while loop can be localized inside or outside of a function with only a select amount of code to be repeated.
- How do we make a while-loop?

Ex.

```
let i: number = 0;
while (    i < 10    ){

(stuff to be executed)
i++;
}
```

Now we're good!

- This is an infinite loop! Much like recursion without a base case, an infinite loop will cause your program to crash.
- Some easy ways to make an infinite loop are:
  - Forgetting to make a counter variable
  - Boolean test is always true
  - Forgetting to increment/decrement your counter

Hands-On: Arrays and While-Loops

- Write a function called countOdds that takes in an array of numbers and returns the number of odd numbers in that array.
- Do not use recursion! Use a while-loop and don't forget to be careful of infinite loops!
- Hint No. 1: The function type is (n: number [ ]): number
- Hint No. 2: To access an element at index i, say n[i]
- Hint No. 3: The method for accessing an array's length is (array name).*length*
- Hint No. 4: To test if num: number is odd, try asking if num % 2 === 1

Hands-On: Arrays and While Loops

```
let countOdds = (n: number[ ]): number=> {
    let i: number = 0;
    let odds: number = 0;
    while(i < n.length) {
        if(n[ i ] % 2 === 1) {
            odds++;
        }
        i++;
    }
    return odds;
};
```

Arrays' Filter/Map/Reduce Methods

- Arrays have built in methods for filter/map/reduce, as opposed to Lists where we had to write the functions and call them ourselves.
- They are applied to arrays in a similar fashion as they are to lists, except that *the methods are called on the arrays themselves* instead of a list being passed in as an argument.
- The methods still perform the same functions:

Filter:	Map:	Reduce:
$T[\,] \rightarrow T[\,]$	$T[\,] \rightarrow U[\,]$	$T[\,] \rightarrow U$
"Filter an array of type T to *become* an array of the same type with certain items removed"	"Map an array of type T to *become* an array where every item is now of type U"	"Reduce an array of type T to *become* a single item."

Let’s Practice:

- What would be printed to the screen for these code snippets:

```
let foo: number = [2, 4, 5, 1, 2].reduce((memo, x) => (x * 2) + memo);

let bar: boolean [ ] = [“Arrays”, “Are”, “Cool”].map((x) => (x[0] === “A”));

print(foo);          // 26

print(bar);          // true, true, false
```

But this is the same stuff we've been doing for weeks!

- Filter/Map/Reduce on arrays *do* indeed accomplish the same things as their List counterparts.
- The most important difference is: the array versions are *methods* being called on array *objects,* rather than *functions* being passed List objects as arguments.

Array:

Array object

```
let sum: number = [ 1 , 2 , 3 , 4 ].reduce((memo, x) => x + memo);
```

List:

List object

Same transformer, same memo

```
let lis: List<number> = listify(1, 2, 3, 4);
let sum: number = reduce(lis, (memo, x) => x + memo, 0);
```

Why bother with this difference?

- Calling filter/map/reduce methods on arrays can lead to some really cool, abstract solutions to problems.
- By chaining method calls, we can drastically manipulate arrays in short amounts of code.
- Sarah is going to speak on this in the second half!

Attendance

https://goo.gl/forms/u5BxzzN679wGwRo63

Nesting Functions

- Where can we call functions? Wherever we expect a value that matches the functions return type
  - Inside print statements

    ```
    print(foo(7));
    ```
  - In a variable assignment

    ```
    let x: number = foo(7);
    ```
  - In some expression

    ```
    if(foo(7)>8){...}
    ```
- Because we can call functions wherever we expect a value that matches the functions return type, we can nest function calls inside of one another

Nesting function example

```
let largest = (x: number, y: number): number => {
    if (x > y) {
        return x;
    } else {
        return y;
    }
};

let smallest = (x: number, y: number): number => {
    if (x < y) {
        return x;
    } else {
        return y;
    }
};
```

```
let bigNumber: number = largest(3, 4);
```
4

```
let biggerNumber: number = largest(3, largest(4, 2));
```
4

4

```
let small: number = smallest(50, largest(3, 70));
```
70

50

```
let inception: number = largest(8, smallest(15, largest(4, smallest(5, 7))));
```
5

5

5

8

Chaining Method Calls

- Similar to how we can call functions as arguments to other functions, we can call methods on other methods if they return an object that has methods
- Example:

```
let strs: string[] = ["It's", "almost", "the", "weekend!"];
let sumOfEvenLengths: number = strs.map((s) => { return s.length; }).filter((x)
=> { return x % 2 === 0; }).reduce((m, s) => { return m + s; }, 0);
```

- Each of these array methods returns an array which we can call another method on

Types

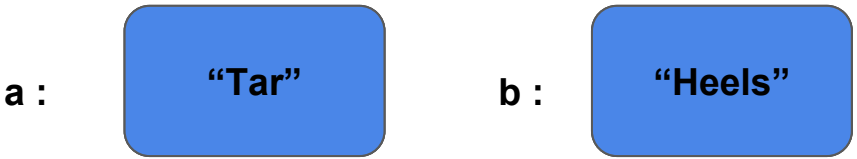
- So far we have talked about two general categories of types: **value** types and **reference** types
  - Value types hold simple values
    - string
    - number
    - boolean
  - Reference types hold complex values that are usually compositions of various values
    - Class types
    - Interfaces
    - Arrays
- These two categories of types behave differently in some cases

Value Types

- What is a value type?
  - **primitive** type that hold simple values
- Examples:
  - string
  - number
  - boolean
- Special behaviors:
  - When you re-assign value types, the **value is copied**
  - Value type variables are **independent** of one another

Value Type Example

```
let a: string = "Assigning one variable to another copies the value";
print("The value of a is: " + a);
let b: string = "Tar";
print("The value of b is: " + b);
a = b;
print("The value of a is: " + a);
b = "Heels!";
print("The value of b is: " + b);
print("The value of a is: " + a);
```



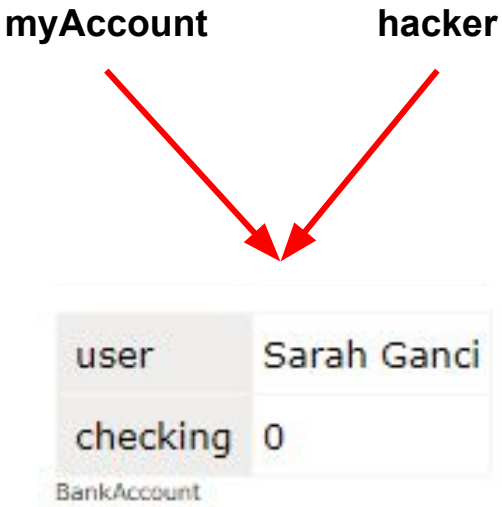
Notice that when we change the value of b, that doesn’t affect the value of a!!!

References

- What is a reference type?
  - **composite** type that holds a reference to an actual value
- Examples:
  - Classes
  - Arrays
  - Functional interfaces
- Special behaviors:
  - When you assign reference variables to one another, the **reference to that value is copied**
    - Copying over a reference variable doesn't create a copy of the actual thing, it just gives that thing an extra name.
  - Modifying a referenced value will affect all references to it.

Reference Type Example

```
let myAccount: BankAccount = new BankAccount();
myAccount.user = "Sarah Ganci";
myAccount.checking = 5;
let hacker: BankAccount = myAccount;
hacker.checking = hacker.checking - 5;
```



Both myAccount and hacker **refer** to the same account! Any change to one will affect the other

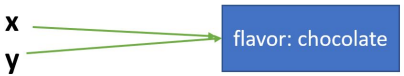
Value vs. Reference Types

Value Types

- Composite or complex types
  - Ex: class types, interface types, arrays
- Hold a reference to a particular object or function
- In assignment, reference to an object is copied over
- Example:

```
class IceCream {
   flavor: string = "none";
}

let x: IceCream = new IceCream();
x.flavor = "chocolate";
let y: IceCream = x;
```



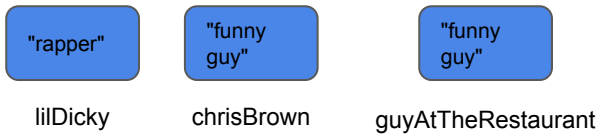
Reference Types

- Primitive or simple value types
  - Ex: string, boolean, number
- Hold a simple value
- In assignment, simple values are copied over
- Example:

```
let x: number = 7;
let y: number = x;
```

Value vs. Reference Hands on

```
/**After this code runs,
 * what is stored in lilDicky,
 * chrisBrown, and guyAtTheRestaurant
*/
let lilDicky: string = "funny guy";
let chrisBrown: string = "rapper";
let guyAtTheRestuarant: string;
guyAtTheRestuarant = lilDicky;
lilDicky = chrisBrown;
chrisBrown = guyAtTheRestuarant;
```




```
//What are the values stored in secret, message, and mystery?
let secret: string[] = ["comp", "110", "is", "cool"];
let message: string[] = ["this", "is", "fun"];
let mystery: string[] = message;
secret[3] = mystery[2];
mystery[0] = secret[0];
mystery[2] = "cool";
```