Review Session MT0

(Don’t) study All Night

Major topics

- Types and Variables
- Classes and Objects
- Booleans and Conditionals
- Functions and Recursion
- Lists

Variables

- Variables are **containers** that hold information
- We can give variables names in order to add context to our code
- The **type** of information that we are dealing with dictates the **type** of variable that we declare

Declaring Variables

- Springing our variable into existence

```
let <name>: <type>;
let x: number;
let name: string;
let isWednesday: boolean;
```

Initializing Variables

- Giving our variables an initial value

```
<name> = <value>;
x = 10;
name = “Brooks Townsend”;
isWednesday = false;
```

Declaration and Initialization

- Very common to combine the two steps into one line.

```
let <name>: <type> = <value>;
let x: number = 10;
let name: string = “Brooks Townsend”;
let isWednesday: boolean = false;
```

Basic Types

Type	Values
number	Any number (1, 2, 5.5, 6/7, etc)
string	“Strings” of characters (“hello”, “yes”)
boolean	true and false

- These are your simple value types in TypeScript

Variable tracing

- What gets printed at the end of this code snippet?

```
let x: number = 0;
x = x + 5;
x = x * 2;
let y: number = x;
y = y + 20;
x = x - 1;
print(x);
print(y);
```

9

30

Arithmetic Operators

- Standard +,-,*, / symbols
- Remember your order of operations! PEMDAS
- Can be used in variable initialization
- Ex:
- `let num: number = 15;`
- `let num2: number = (num - 10) * 5;`

Remainder operator - % *modulo*

- Divides numbers, the result is the remainder
- Ex:
- `7 % 2 = 1`
- `8 % 3 = 2`
- `15 % 3 = 0`

- `3 % 5 = 3`

Strings

- Strings of characters
- Special operation called **concatenation** (same operator as addition)
- Ex:

```
let name = await promptString(“What is your name?”);
print(“Hello ” + name + “!”);
```

Concatenation

- **Note**: Whenever you see a + next to a string (if it's a valid statement), then the resulting type is a string.
- Ex. What are the resulting types of the following statements?

```
1. 5 + 3;
2. 5 + “3”;
3. “5” + “3”;
4. “5 + 3”;
5. “Wow” + 3 + “cool” + 5 + “me” + true;
```

Boolean

- True / False value
- Basis for conditional statements and control flow

Assignment vs Equality

- Two similar operators, two totally different functions
- = is different from ===
- = is the **assignment** operator (`let i: number = 0;`)
- === is the **equality** operator
- `“COMP110” === “COMP” + 110;`

Relational Operators

- Relational operators are the basis for conditional statements
- **Always** evaluate to a boolean value
- Ex:
- `5 > 3`
- `6 !== 6`
- `“Yes” === “YES”`

Test	Math	TypeScript Operator
"is greater than?"	$>$	`>`
"is at least?"	$\geq$	`>=`
"is less than?"	$<$	`<`
"is at most?"	$\leq$	`<=`
"is equal to?"	$=$	`===`
"is not equal to?"	$\neq$	`!==`

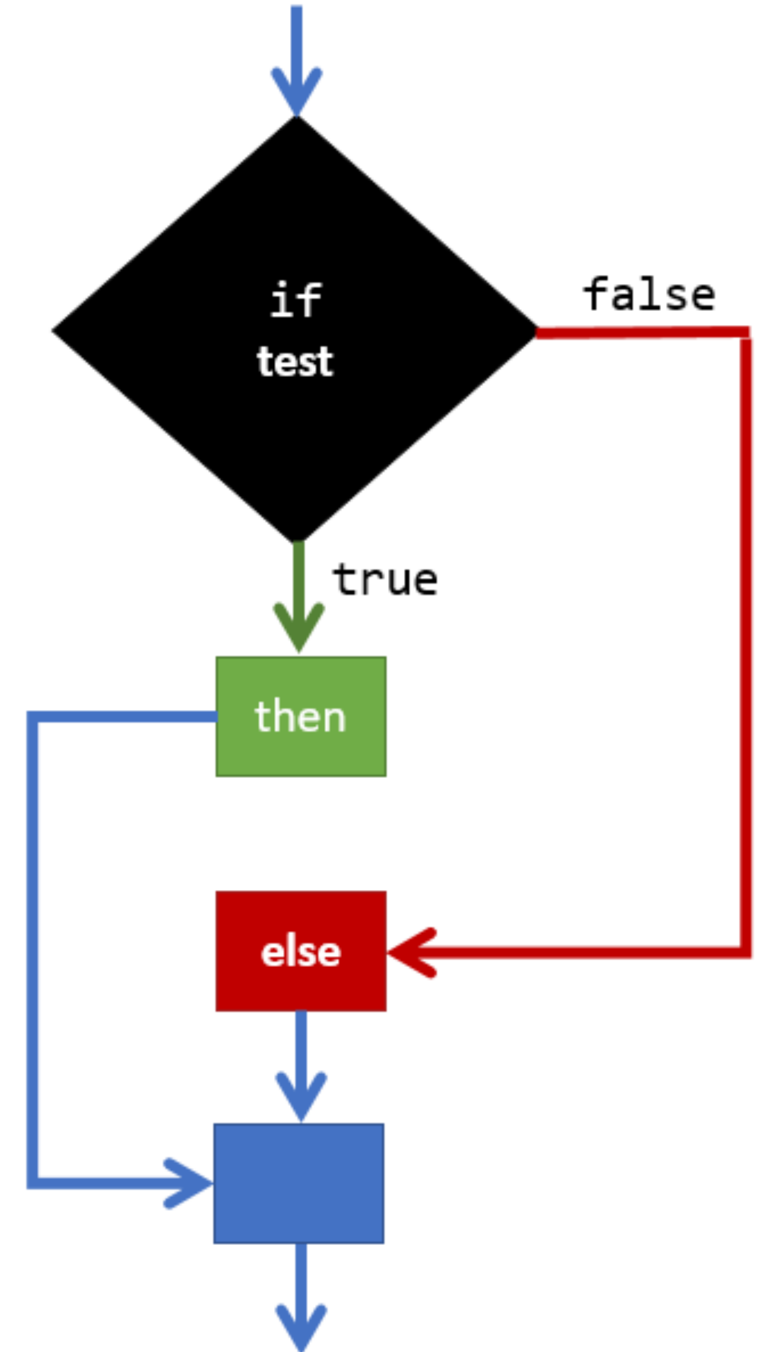
Statements

- Statements are like the sentences of code
- Statements are a composition of expressions
- Expressions are made up of variables and operators
- Ex:
- `1 * 2  * 3;`
- `(x + y) / 100;`

Control flow

- Allows your program to do different things (without your direct interference)

```
if (<condition>) {
   //then run this code
} else {
  //then run this code
}
```



If – then – else

```
let waterIsWet: boolean = await promptBoolean(“?”);

if (waterIsWet) {
     print(“You are correct!”);
} else {
     print(“Fake news”);
}

print(“Don’t forget pop tarts are ravioli too!”);
```

|| and &&

- Two more operators, used in conditional logic
- Or and And
- Ex
- `let num: number = 5;`
- `num === 6 || num > 3;`
- `num >= 5 && num < 100;`
- `true || false;`
- `true && false;`

Else-if

- An easier way to use multiple if statements

```
if (a === b) {
  print("Equal");
} else {
  if (a < b) {
    print("a is less than b");
  } else {
    if (a > b) {
      print("a is greater than b");
    } else {
      print("Can we ever print this?");
    }
  }
}
```

```
if (a === b) {
  print("Equal");
} else if (a < b) {
  print("a is less than b");
} else if (a > b) {
  print("a is greater than b");
} else {
  print("Can we ever print this?");
}
```

Complex types

- We can declare and work with variables that aren’t just strings, numbers, and booleans.
- We have worked with Lists, Movies, Games, Train Cars, etc
- What are these complex types?

What is a Class?

- Classes are blue prints for objects
- A class is a set of properties
- In a class definition we have
  - Key word: **class**
  - **Name** for the class
    - Usually starts with a capital letter
  - **Properties**
    - Variables given default values

- Syntax:

```
class Name {
        property1: type1= defaultValue;
        property2: type2= defaultValue;
}
```

- Example:

```
class UNCStudent{
        onyen: string = “onyen”;
        pid: number = 0;
}
```

What is an Object?

- An object is a specific implementation of a class
- We can have many objects of the same class type
- Objects of the same class have the same properties but can have different values for those properties

Creating New Objects

- To create a new object we use the following syntax:

```
let name : ClassName = new ClassName();
```

- Example:

```
let HintonJames: UNCStudent = new UNCStudent();
```

Accessing an Object’s properties

- To change or access the properties of the object, we use the dot operator

```
<objectName>.<property>;
<objectName>.<property> = <value>;

let brooks: UNCStudent = new UNCStudent();
brooks.onyen = “brooksmt”;
print(brooks.onyen);
print(brooks.pid);
```

Classes vs. Objects

Class

- General blue prints
- Ex:

```
class Food {
        name : string = “food name”;
        cal : number = 0;
        healthy : boolean = false;
}
```

Object

- Specific instances
- Ex:

```
let yum: Food = new Food();
yum.name= “pizza”;
yum.cal= 500;
yum.healthy= false;

let meh: Food = new Food();
meh.name= “salad”;
meh.cal=150;
meh.healthy= true;
```

Functions

- Functions are reusable instructions.
  - You can think of functions like recipes
- Can take in values, and return values.
- We can define functions, and we can call functions.

Function Definitions

- Parts of function definitions:
  - **Name:** the name of our function
  - **Parameters:** place holder variables for expected inputs
  - **Return type:** the type of value we want to return from the function
  - **Body:** the statements or lines of code that make up the function
    - This will often include return statements

- We define functions using the following syntax:

```
let <name> = (<parameters>): <return type> => {
    //function body
};

let applePie = (one:string, two:string) : string => {
    return "We combined " + one + " with " + two + " and
now we have apple pie yummm";
};
```

Calling functions

- To run the code in our functions, we call them
- If the function definition has parameters, we must pass values (arguments) into the function call
- Arguments must match parameters in type and order
- To call functions we use the following syntax:

```
<name>(<arguments>);
applePie(“apples”, “sugar”);
```

```
let applePie = (one:string, two:string) : string => {
    return “We combined “ + one + “ with ” + two + “  and now we have apple pie yummm”;
};
```

Function Practice

- Which of the following are valid calls to this function?

```
let funcy = (n: number, s: string): string => {
      return “This: ” + n + “ and that: ” + s;
}
```

```
1. funcy(5, “four”);
2. funcy(“5”, “four”);
3. funcy(6, “” + 5);
4. funcy(17);
```

1. Valid
2. Invalid
3. Valid
4. Invalid

Recursion

- Base case
  - Need to change *something* about an argument to reach the base case
- Recursive case
  - Whenever we call a function inside of itself

Rules of Recursion

1. Test your base case
2. Always change at least one argument when recurring
3. Never make more than one recursive call to a function at a time.
4. For Lists: process the first value, then use recursion to process the rest of the list.

Recursion practice

```
let goodOrNo = (n: number): string => {
    return n * goodOrNo(n - 1);
}
```

No Base Case!!!

Recursion practice

```
let goodOrNo = (n: number): string => {
      if (n === 0) {
            return “ZERO”;
      } else {
            return goodOrNo(n);
      }
}
```

Making your own Recursive Functions

1. Examine the problem, what is it asking?
    a) Count how many items are in a list of strings
    b) Add up all of the numbers from 0 to n
2. Define your base case
    1. For lists, usually when your list is empty
    2. For numbers, usually when your number is equal to zero
3. Let recursion do the work for you

Let’s do some Factorial.

- The result of multiplying all numbers leading up to a number n.
- Ex: 5! (five factorial) === 5 * 4 * 3 * 2 * 1 === 120

```
let factorial = (n: number): number => {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }

};
```

```
factorial(1) === 1
factorial(2) === 2
factorial(3) === 6
factorial(4) === 24
factorial(5) === 120
.
.
.
```

Lists

- What is a List?
- Ordered collection of values

- Ex: What do I need at the store?
- `“Oranges” -> “Bananas” -> “Milk” -> null`
- `null` means we have reached the end of our list. Done shopping!

Cons

- How do we make a list?
- cons (short for construct) is a **function** that we use to create a List.

- // cons function definition
- let cons = (p1: <T>, p2: List<T>): List<T> => {
      // create a list and return it
- };

Cons

- Usage:
- Let’s make a list of pizza restaurants on Franklin
- We want: “Bennys” -> “IP3” -> “Lotsa” -> null

```
let restaurants: List<string>;
restaurants = cons(“Bennys”, cons(“IP3”, cons(“Lotsa”, null)));
print(restaurants);
```

First

- How do we get individual items out of the list?
- Let’s take the first one item.

```
let studyOrder: List<string> = cons(“BIOL”, cons(“COMP”, null));
let someNumbers: List<number> = cons(12, cons(5, cons(42, null)));
print(first(studyOrder));       ← Prints BIOL
print(first(someNumbers));      ← Prints 12
```

Rest

- What if we don’t want only the first item?
- “Spiderman” “Superman” -> “Wonderwoman” -> null

First

Rest

Rest

- Usage:

```
let roster: List<string>;
roster = cons(“Paige”, cons(“Maye”, cons(“Berry”, null)));

print(first(roster) + “ just went pro!”);
print(rest(roster));
```

How could we get certain list values?

- What if we wanted, say, the third value in a list?
- “Spiderman” | “Superman” -> “Wonderwoman” -> null

First | Rest

“Superman” | “Wonderwoman” -> null

First | Rest

```
first(rest(rest(list)));
```

Generics

- When we want to define a function that can work for all types
- Denoted by a generic T, for type

Recursion Example

- # of occurrences of an item in a list
- Declare a function named “count” that:
  - Takes in 2 parameters. One of type List<number> and one of type number
  - Returns a number, which is the # of times that number occurs in the list
  - Recursively goes through the list and counts how many times a number appears in the list.

Count

```
let count = <T> (input: List<T>, find: T): number => {
    if (input === null) {
        return 0;
    } else if (first(input) === find) {
        return 1 + count(rest(input), find);
    } else {
        return count(rest(input), find);
    }
}
```

Filter, Map, and Reduce

Of games that UNC won, how many points did the player score in total?

Outcome	Points
L 76-67	4
W 95-75	20
W 97-57	13
L 103-100	9
L 77-62	22

List<Game>

Filter →

Outcome	Points
W 95-75	20
W 97-57	13

List<Game>

Map →

20
13

List<number>

Reduce →

33

number

Filter

- Take a list of values, and retain only the values that are relevant
- In this example, take a list of games in UNC's basketball season and only retain the games where we won

Outcome	Points
L 76-67	4
W 95-75	20
W 97-57	13
L 103-100	9
L 77-62	22

List<Game>

Filter →

Outcome	Points
W 95-75	20
W 97-57	13

List<Game>

Map

- Take a list of some value, and map it to a list of another value
- In this example, let's take a list of games where UNC won, and transform it to a list of the amount of points that Joel Berry scored that game.

Outcome	Points
W 95-75	20
W 97-57	13

List<Game>

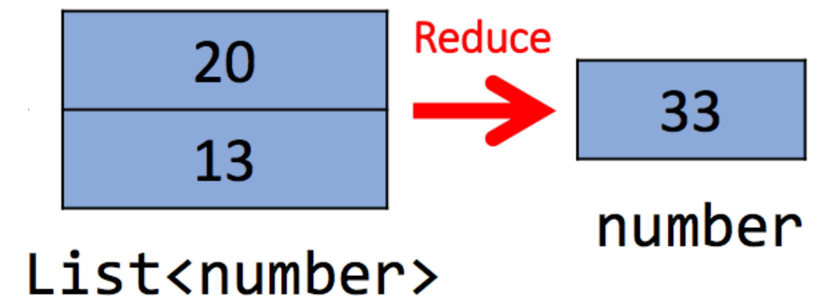
Map

20
13

List<number>

Reduce

- Take a list of values and reduce it to just one value
- Does not have to be the same value as the List (ex. List of strings can be reduced to a single number, like the length of all the strings)
- In this example, reduce a list of Joel Berry II's points to the total amount of points that he scored.



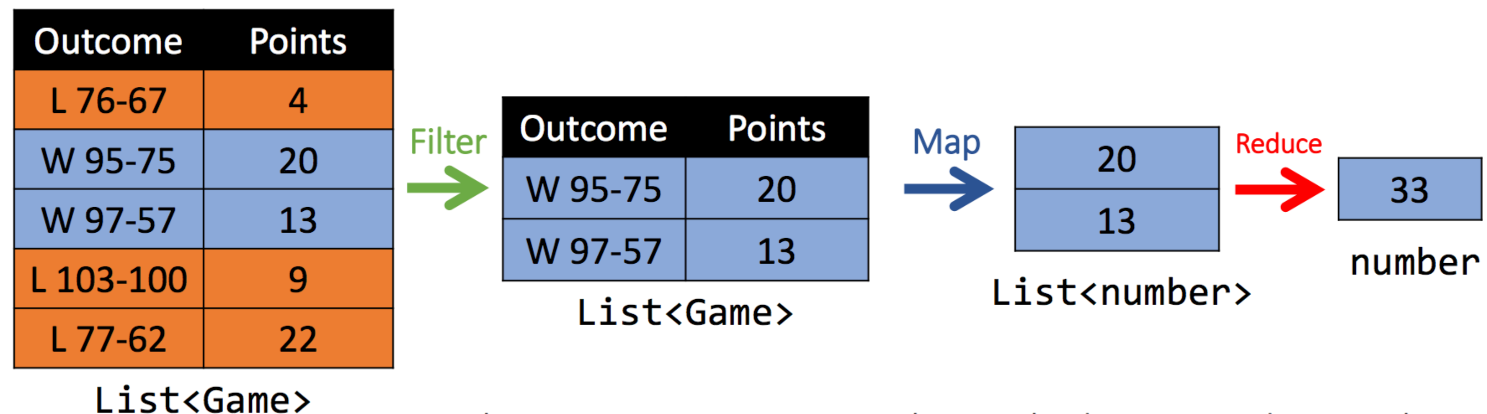
Coming back to Filter

- When you filter recursively, make sure to follow these steps.

1. Check your base case
2. Process the first item in the list
3. Depending on your condition (your filtering condition) either:
    1. cons the first item of the list with the recursive result of the rest of the list
    2. return the recursive result of the rest of the list

List filtering

- Let's follow the pipeline we've been looking at.
- How could we code up this first step?
- Base case?
- Recursive case?



Example

```
let uncWins = (games: List<Game>): List<Game> => {
      if (games === null) {
            return null;
      } else {
            let firstGame: Game = first(games);
            if (firstGame.uncPoints > firstGame.opponentPoints) {
                  return cons(firstGame, uncWins(rest(list)));
            } else {
                  return uncWins(rest(games));
            }
};
```