

# Values vs. References

## Exporting and Importing

Lecture 09

# Midterm Prep

- WS3 - Due Monday 9/25
- PS2 - Due Tuesday 9/26
- Once completed: you should work through the study guides and their associated keys posted to [comp110.com](http://comp110.com) home page.
- Review Session is Weds 9/27 from 5-7pm in GSB100

Warm-up: What are the elements of **a**?

```
let a: number[] = [ 2 ]; // Notice initial element 2
let i: number = 0;

while (i < 3) {
    a[a.length] = (i + 1) * 2;
    i++;
}

print(a);
```

Answer: 2, 2, 4, 6

```
let a: number[] = [ 2 ]; // Notice initial element 2
let i: number = 0;

while (i < 3) {
    a[a.length] = (i + 1) * 2;
    i++;
}

print(a);
```

# How do we **append** an element to an array?

- Given an array **a**, what is the **next** index needed to append?
  - When it is **empty**, or has **0 elements**, the next index is **0**
  - When it has **1 element**, the next index is **1**
  - When it has **2 elements**, the next index is **2**
- **Because of 0-based indexing, we can use the # of elements in an array as the index to use to append a value to the array.**
- Append to an array:

**`a[a.length] = <value>;`**

Warm-up:  
What is printed?

```
class Person {  
    name: string;  
}  
  
let a: number = 1;  
let b: number = a;  
a = 2;  
print("b: " + b);  
  
let seanCombs: Person = new Person();  
seanCombs.name = "Sean Combs";  
  
let puffDaddy: Person = seanCombs;  
seanCombs.name = "Diddy";  
  
print("Name: " + puffDaddy.name);
```

Warm-up:  
What is printed?

Answer:  
b: 1  
Name: Diddy

```
class Person {  
    name: string;  
}  
  
let a: number = 1;  
let b: number = a;  
a = 2;  
print("b: " + b);  
  
let seanCombs: Person = new Person();  
seanCombs.name = "Sean Combs";  
  
let puffDaddy: Person = seanCombs;  
seanCombs.name = "Diddy";  
  
print("Name: " + puffDaddy.name);
```

# Value Types Visualized

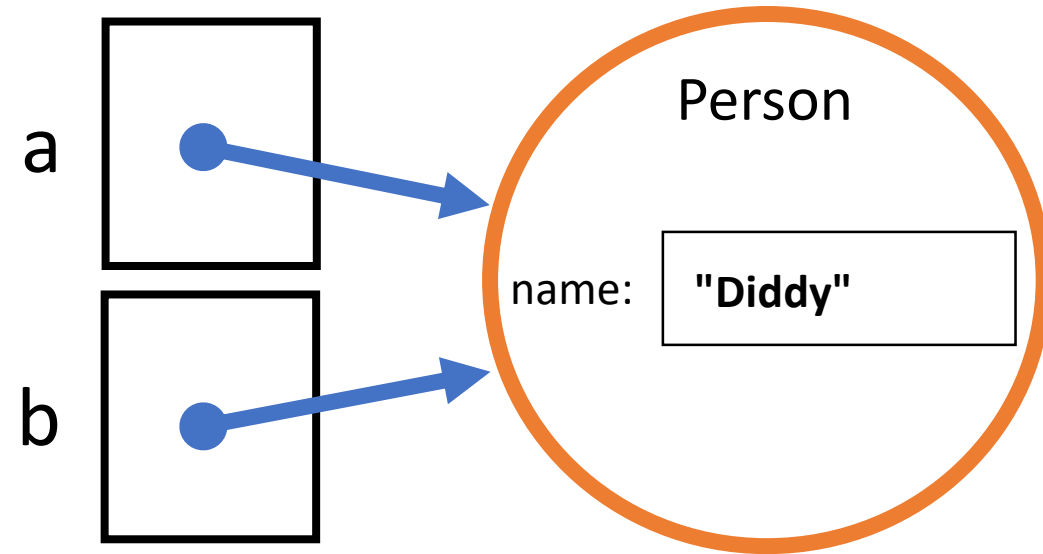
```
let a: string = "hello";  
let b: string = a;  
a = "world";
```



Notice: **b** was assigned **a**. This copied the string **"hello"** to **b**.  
The variable **a** was then assigned the string **"world"**.  
This had no impact on **b** because string variables are value types.

# Reference Types Visualized

```
let a: Person = new Person();  
a.name = "Sean Combs";  
let b: Person = a;  
a.name = "Diddy";
```



Notice: When the **a** variable is assigned a **new Person** object, it is assigned reference to the Person object.

When **b** is assigned **a**'s value, the **reference** is copied, **not the object**. Variables **a** and **b** now refer to the same object.

Changing property values via *either a* or *b* will be visible from the other because they are aliases of one another.

# Value Types vs. Reference Types

- Primitive types (number, string, boolean) are **value types**
  - Variables hold *copies* of actual values.
  - Assigning one variable to another ***copies the value***.
  - Changing a copied variable's value does not impact original or vice-versa.
- Composite types (classes, arrays) are **reference types**
  - Variables hold *references* to actual values.
  - Assigning one variable to another ***copies the reference***. Both variables now refer to the same value in memory.
  - Modifying a referenced value will impact all references to it.

# When would you ever have multiple variables referring to the same object or array?

- When passing an object or array to a function's parameter!

```
import "intros";

function main(): void {
    let numbers: number[] = [1, 2];
    append(numbers, 3);
    print(numbers); // Elements are now: 1, 2, 3
}

function append(a: number[], n: number): void {
    a[a.length] = n;
}

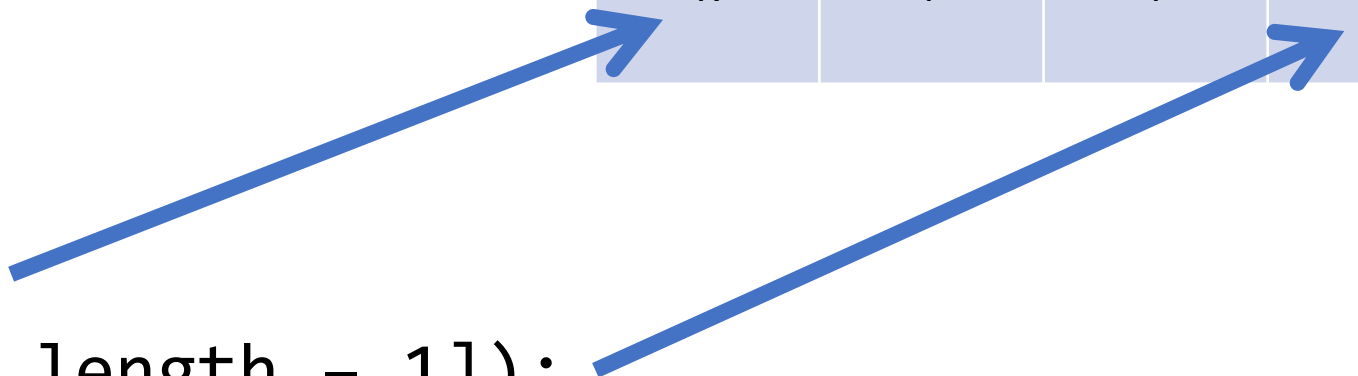
main();
```

Aside: A string is an array of characters!

```
let name: string = "Kris";
```

0	1	2	3
K	r	i	s

```
print(name[0]);  
print(name[name.length - 1]);
```

A diagram illustrating string indexing. A table represents the string 'Kris' with indices 0 to 3. The first row contains the indices '0', '1', '2', and '3'. The second row contains the characters 'K', 'r', 'i', and 's'. A blue arrow points from the code 'name[0]' in the line 'print(name[0]);' to the 'K' in the table. Another blue arrow points from the code 'name[name.length - 1]' in the line 'print(name[name.length - 1]);' to the 's' in the table.

# Breaking a Project into Multiple Files

- You can **export** *functions* and *classes* from one TypeScript file

```
export function (...
```

```
export class {...
```

- And **import** them into *another* TypeScript file

```
import { <names>, <of>, <functions/classes> } from "./<file>";
```

- Example:

```
import { Game, mapPoints, reduceSum } from "./library";
```

# Referencing Imported Files

- Where you see the dot-slash in **"./library"**, this means **"from within the same folder I am in, import the library.ts file"**
  - Note: you do not specify the .ts file extension
- To move up a folder, the dot-dot-slash in **"../super"**, means, **"move up to my parent folder, and import *super.ts* from there"**
- To move up to a parent folder, and back down to a sibling folder: **"../ps01-adventure/index-app"** to reference the file index-app.ts in Problem Set 01.
- No stress: when you need to do imports/exports in problem sets, we'll guide you through it.

# Library Files

- As we move further into the semester we will break our projects into library files whose classes and functions can be reused by many different apps.
- Typically these files will not need to import the introcs library, unless your library functions need the print/prompt functions.
- Let's take a look at an example library file in lec09 / library.ts

# Follow-along: Let's export/import `filterByOutcome`

- In `library.ts`, add the **export** keyword before the function `filterByOutcome`

```
export function filterByOutcome(games: Game[], outcome: string): Game[] {
```

- In `02-filter-map-reduce-app.ts`, add the `filterByOutcome` function to the list of items imported from `library.ts`:

```
import {  
  Game,  
  mapPoints,  
  reduceSum,  
  filterByOutcome  
} from "../library";
```

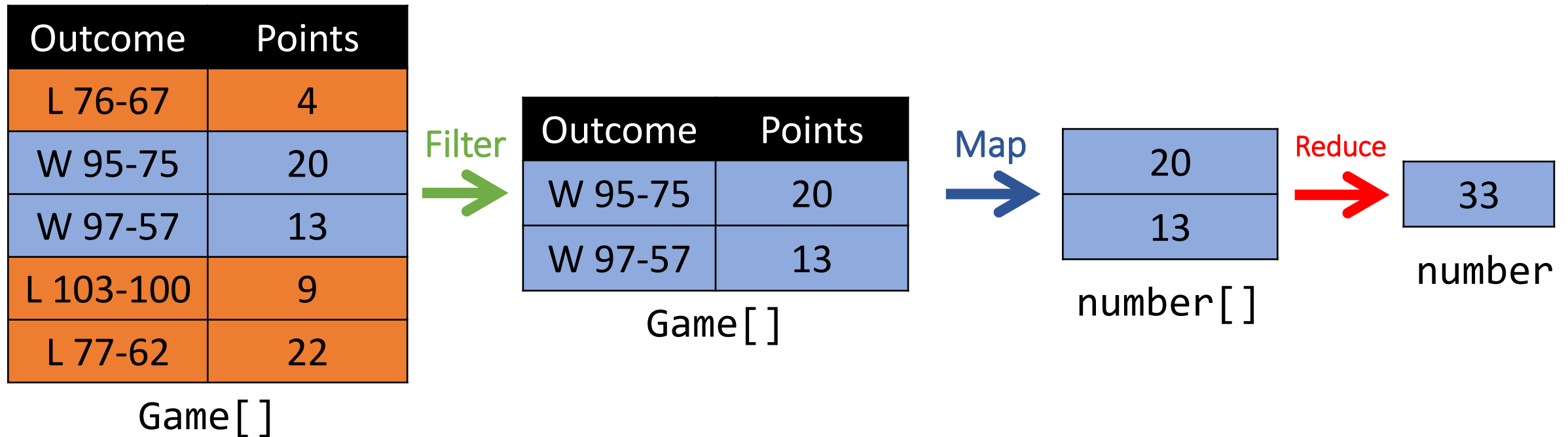
## Back to Working with Data

Warning! What you see past this point is *not* necessary for completing PS02 – Weather Stats.

You'll need these concepts for PS03.

# Filter-Map-Reduce Data Processing Pipeline

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



We will write simple functions for each step in this process.

# Filtering an Array

- Given an input array and selection criteria, **return a new array** containing *only* the elements meeting the criteria.
- For example, filter games by outcomes starting with "W" (wins).

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

Game[ ]



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

Game[ ]

# Filtering Algorithm

1. Setup an empty array to hold matches.
2. Work element-by-element through entire input array.
  - Does this element meet the filtering criteria? Yes? Copy to matches!
3. Return matches array.

input

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

Game[ ]

# Filtering Algorithm

1. **Setup an empty array to hold matches.**
2. Work element-by-element through entire input array.
  - Does this element meet the filtering criteria? Yes? Append it to matches!
3. Return matches array.

input

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

Game[ ]

matches

Outcome	Points
---------	--------

Game[ ]

# Filtering Algorithm

1. Setup an empty array to hold matches.
2. **Work element-by-element through entire input array.**
  - Does this element meet the filtering criteria? Yes? Append it to matches!
3. Return matches array.

input



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

Game[ ]

matches


Outcome	Points
---------	--------

Game[ ]

# Filtering Algorithm

1. Setup an empty array to hold matches.
2. **Work element-by-element through entire input array.**
  - Does this element meet the filtering criteria? Yes? Append it to matches!
3. Return matches array.


input



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

Game[ ]

matches




Outcome	Points
W 95-75	20

Game[ ]

# Filtering Algorithm

1. Setup an empty array to hold matches.
2. **Work element-by-element through entire input array.**
  - Does this element meet the filtering criteria? Yes? Append it to matches!
3. Return matches array.


input



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

Game[ ]

matches




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

Game[ ]

# Filtering Algorithm

1. Setup an empty array to hold matches.
2. **Work element-by-element through entire input array.**
  - Does this element meet the filtering criteria? Yes? Append it to matches!
3. Return matches array.

input



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

Game[ ]

matches

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

Game[ ]

# Filtering Algorithm

1. Setup an empty array to hold matches.
2. **Work element-by-element through entire input array.**
  - Does this element meet the filtering criteria? Yes? Append it to matches!
3. Return matches array.

input

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

Game[ ]

matches

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

Game[ ]

# Filtering Algorithm

1. Setup an empty array to hold matches.
2. Work element-by-element through entire input array.
  - Does this element meet the filtering criteria? Yes? Append it to matches!
3. **Return matches array.**

input

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

Game[ ]

matches

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

Game[ ]



return  
matches

# Follow-along

- In lec09 / **02-filter-map-reduce.ts**'s **process** function, let's declare a variable to hold an array of the Games won, and initialize it by calling the `filterByOutcome` function imported from `library.ts`

```
let filtered: Game[] = filterByOutcome(games, "W");  
print("Filtered: " + filtered.length);
```

- Then, let's correctly implement the `filterByOutcome` function's selection criteria in lec09 / `library.ts`:

```
export function filterByOutcome(games: Game[], outcome: string): Game[] {  
    let matches: Game[] = [];  
    let i: number = 0;  
    while (i < games.length) {  
        if (games[i].outcome[0] === outcome) {  
            matches[matches.length] = games[i];  
        }  
        i++;  
    }  
    return matches;  
}
```

Note: each outcome property is a string that looks like "W 99-97" or "L 90-89" for win/loss

Thus, `games[i].outcome[0]` refers to the letter either "W" or "L".

# Mapping Arrays

- Given an input array, **return a new array** containing *all* of the elements of the input array "transformed" to a different value.
- For example, **map** from an array of Games to an array of points.

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

Game[ ]



20
13

number[ ]

# Mapping Algorithm

1. Setup an empty array to hold the mapped values.
2. Work element-by-element through entire input array.
  - Apply some "transformation" from input element to mapped element.
  - In our example we are "transforming" a Game value to just its points value.
3. Return mapped array.

input

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

Game[ ]

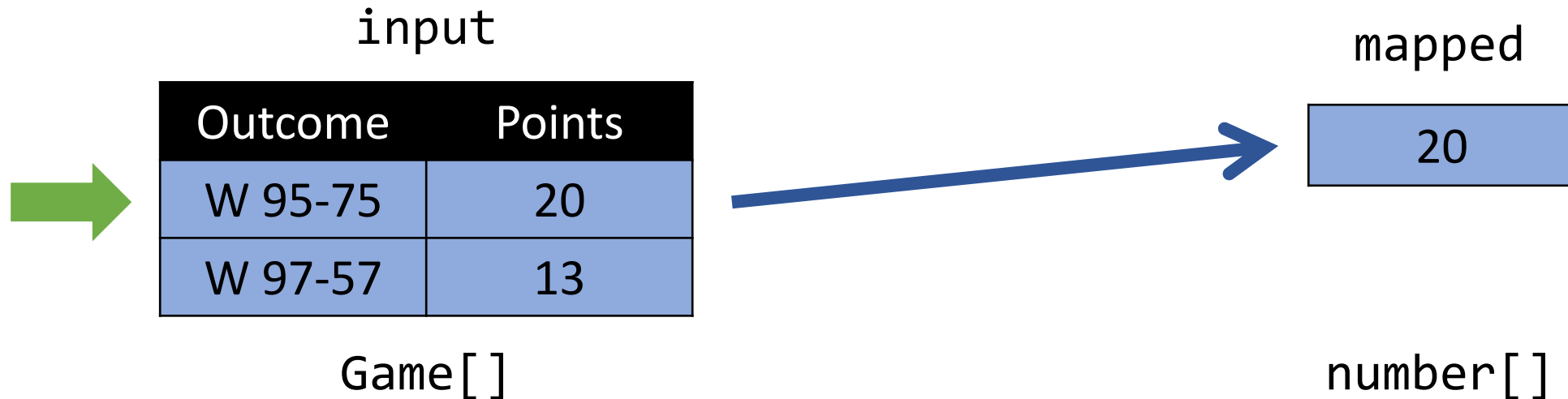
# Mapping Algorithm

1. **Setup an empty array to hold the mapped values.**
2. Work element-by-element through entire input array.
  - Apply some "transformation" from input element to mapped element.
  - In our example we are "transforming" a Game value to just its points value.
3. Return mapped array.

input		mapped	
Outcome	Points		
W 95-75	20		
W 97-57	13		
Game[ ]		number[ ]	

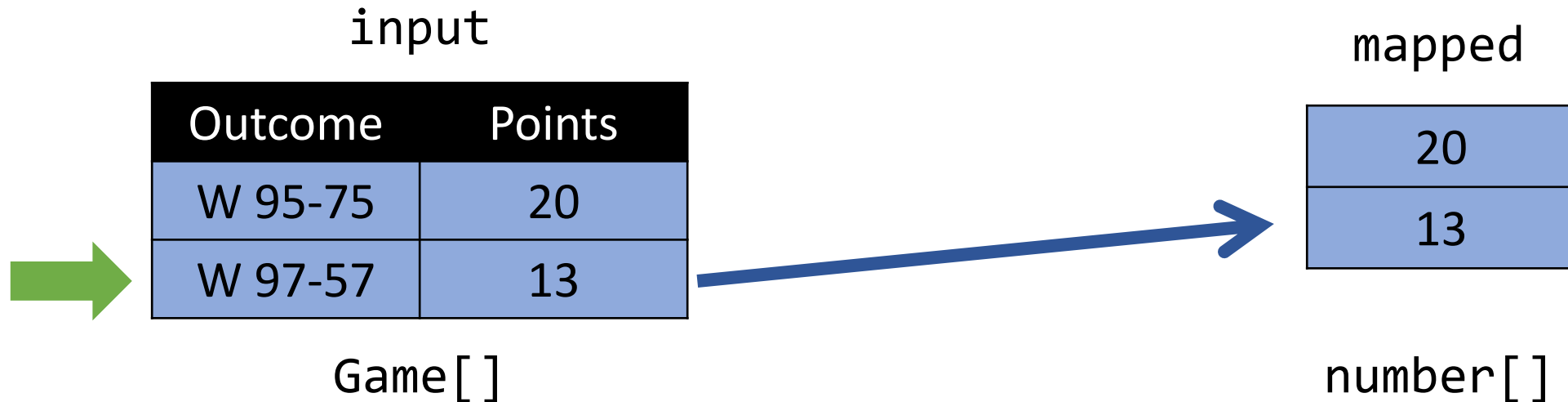
# Mapping Algorithm

1. Setup an empty array to hold the mapped values.
2. **Work element-by-element through entire input array.**
  - Apply some "transformation" from input element to mapped element.
  - In our example we are "transforming" a Game value to just a points value.
3. Return mapped array.



# Mapping Algorithm

1. Setup an empty array to hold the mapped values.
2. **Work element-by-element through entire input array.**
  - Apply some "transformation" from input element to mapped element.
  - In our example we are "transforming" a Game value to just a points value.
3. Return mapped array.



# Mapping Algorithm

1. Setup an empty array to hold the mapped values.
2. Work element-by-element through entire input array.
  - Apply some "transformation" from input element to mapped element.
  - In our example we are "transforming" a Game value to just a points value.
3. **Return mapped array.**

input

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

Game[ ]

mapped

20
13

number[ ]

  
return  
mapped

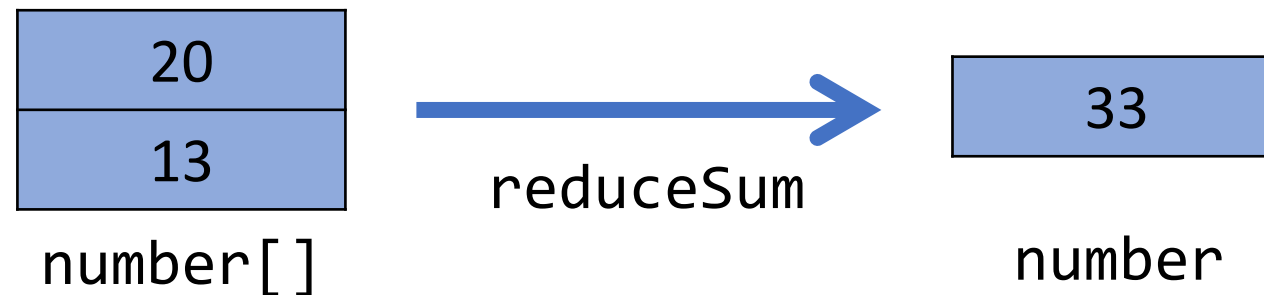
# Hands-on: Call `mapPoints`

- From `lec09 / 02-filter-map-reduce.ts`
1. In the **`process function`**, declare a variable named **`mapped`** of type **`number[]`**
  2. Assign its initial value to be the result of calling the **`mapPoints`** function defined in **`library.ts`** using the **`filtered`** array as an argument.
  3. Print the **`mapped`** array. It should look something like...  
23,18,23,2,8,24...
  4. Check-in on [PollEv.com/comp110](https://pollev.com/comp110) once complete

```
let mapped: number[] = mapPoints(filtered);  
print(mapped);
```

# Reducing Arrays

- Given an input array, reduce it to a single value.
- For example, reduce an array of numbers to their sum.



- We've written reducing functions in the past two lectures!

# Follow-along: Reducing points to a total.

- In lec09 / 02-filter-map-reduce.ts's process function
- Declare a variable of type number to hold the "reduced" total and initialize it to the result of calling **reduceSum** with the mapped array.

```
let reduced: number = reduceSum(mapped);  
print(reduced); // Total Points
```

# Filter-Map-Reduce Data Processing Pipeline

Of games that UNC won, that UNC lost, with 3+ assists, with a block, etc, what was the points, total, assists, average, fouls, min, blocks, max, etc, etc

Filter  
Game[]  $\longrightarrow$  Game[]

Map  
Game[]  $\rightarrow$  number[]

Reduce  
number[]  $\rightarrow$  number

Big idea: We can **select any combo** of a filter, map, and reduce sequence.  
Result: (# **Filters**) x (# **Maps**) x (# **Reduces**) different combinations.

# Follow-along: How many assists in games won?

- In library.ts, let's implement the mapAssists function:

```
export function mapAssists(games: Game[]): number[] {  
    let assists: number[] = [];  
    let i: number = 0;  
    while (i < games.length) {  
        assists[i] = games[i].assists;  
        i++;  
    }  
    return assists;  
}
```

- Then, add it to the import list of 02-filter-map-reduce-app.ts
- Finally, try calling mapAssists rather than mapPoints

```
let mapped: number[] = mapAssists(filtered);
```

# Follow-along: reduceAvg

- Let's introduce another reducing function in library.ts ...

```
export function reduceAvg(a: number[]): number {  
    return reduceSum(a) / a.length;  
}
```

- ... and import it in 02-filter-map-reduce-app.ts
- And swap it out for reduceSum....

```
let reduced: number = reduceAvg(mapped);  
print(reduced);
```

# 110 Words of Wisdom

- **Write simple functions that have *one job*.**
- Functions that do only one of *filter*, *map*, **or** *reduce* are better than single mega-functions that do all three at once.
- Why?
  - simpler to **compose** new functionality
  - easier to **reuse** code you've already written
  - **fewer errors** not having to reimplement algorithms
- Are there potential downsides to small functions versus a mega-functions?
  - Potentially a slightly less optimal run-time, but only matters once you're working with data sets well into the millions of data points. Even then, always start with small, simple, tried, and true... optimize once you need to.