To Ponder

Given: roster of students (an array)

Write a JavaScript program that outputs an html list of students (name and midterm score) whose gpa is > 3.0, such that the list is sorted by midterm score

1. Xi Chen (85)
2. Mary Smith (80)
3. Alessandro Reis (74)
JavaScript: Array API

Lecture 16
Arrays: Basics

- Numbered starting at 0
- Indexed with \[ \]
- Property length is \# of elements

```javascript
let sum = 0;
for (let i = 0; i < n.length; i++) {
    sum += n[i];
}
```
Array Instantiation/Initialization

- Instantiate with new
  ```javascript
  let n = new Array(3);
  ```
- Initially, each element is undefined
- Note: Elements can be a mix of types
  ```javascript
  n[0] = 10;
  n[1] = "hi";
  n[2] = new Array(100);
  ```
- Array literals usually preferred
  ```javascript
  let n = [10, 20, 30, 40];
  let m = ["hi", , "world", 3.14];
  [3, "hi", 17, [3, 4]].length == 4
  ```
Dynamic Size

- Arrays can grow

```javascript
let n = ["tree", 6, -2];
n.length == 3 //=> true
n[8] = 17;
n.length == 9 //=> true
```

- Arrays can shrink

```javascript
n.length = 2;
// n is now ["tree", 6]
```
Arrays are Dynamic

```javascript
let n = [];
```
Arrays are Dynamic

```javascript
let n = [];
```
Arrays are Dynamic

\[ n[0] = 4; \]
Arrays are Dynamic
Arrays are Dynamic

\[ n[3] = 3.14; \]
Arrays are Dynamic
Arrays are Dynamic

\[ n[1] = \text{"hi"}; \]
Arrays are Dynamic
Accessors: Searching

- Find occurrence: `indexOf/lastIndexOf`
  - Returns -1 if not found
    - `indexOf(element[, startIndex])`
    - `lastIndexOf(element[, lastIndex])`
  - Optional parameter: start/end index
  - Uses strict equality (===)
    
    ```javascript
    let i = n.indexOf(elt);
    while (i !== -1) {
      report(i);
      i = n.indexOf(elt, i + 1);
    }
    ```
Accessors: Extracting

- None of the following change the array
  - Return a new array/string with result
- Concatenate: `concat`
  ```javascript
  concat(a1, a2, ..., aN)
  let d = n.concat(n);
  ```
- Extract a sub-section: `slice`
  ```javascript
  slice(startIndex, endIndex)
  k = n.slice(1, 3); // k is n[1], n[2]
  ```
- Combine into string: `join`
  ```javascript
  join(separator)
  s = n.join(" "); // default is ","
  ```
Mutators: Growing/Shrinking

- Add/remove from end: `push/pop`
  ```javascript
  let n = [10, 20];
  newLength = n.push(30, 40); //=> 4
  lastValue = n.pop(); //=> 40
  ```

- Add/remove from beginning: `unshift/shift`
  ```javascript
  let n = [10, 20];
  newLength = n.unshift(30, 40); //=> 4
  firstValue = n.shift(); //=> 30
  ```

- Push/shift gives FIFO queue
Push Example

```javascript
function findAll(n, elt) {
    let indices = [];
    let i = n.indexOf(elt);
    while (i != -1) {
        indices.push(i);
        i = n.indexOf(elt, i + 1);
    }
    return indices;
}
```
Mutators: Delete/Insert/Replace

- Delete/insert/replace sub-array: \texttt{splice}

\texttt{splice} \ (\texttt{index}, \texttt{howMany}[, \texttt{e1, e2, ...}, \texttt{eN}])

- Modifies array (cf. \texttt{slice}, an accessor)
- Returns array of removed elements

\begin{verbatim}
let magic = [34, -17, 6, 4];
let removed = magic.splice(2, 0, 13);
// removed is []
// magic is [34, -17, 13, 6, 4]

removed = magic.splice(3, 1, "hi", "yo");
// removed is [6]
// magic is [34, -17, 13, "hi", "yo", 4]
\end{verbatim}
Mutators: Rearrange

- Transpose all elements: `reverse`
  
  ```javascript
  let n = [5, 300, 90];
  n.reverse(); // n is [90, 300, 5]
  ```

- Order all elements: `sort`
  
  ```javascript
  let f = ["blue", "beluga","killer"];
  f.sort(); // f is
  // ["beluga", "blue", "killer"]
  n.sort(); // n is [300, 5, 90]
  ```
Mutators: Rearrange

- Transpose all elements: `reverse`
  
  ```javascript
  let n = [5, 300, 90];
  n.reverse(); // n is [90, 300, 5]
  ```

- Order all elements: `sort`
  
  ```javascript
  let f = ["blue", "beluga", "killer"];
  f.sort(); // f is
  // ["beluga", "blue", "killer"]
  n.sort(); // n is [300, 5, 90]
  ```

- Problem: Default ordering is based on string representation (lexicographic)
- Solution: Use a function that compares
Sorting with Comparator

- A comparator \((a, b)\) returns a number
  - \(< 0\) iff \(a\) is *smaller than* \(b\)
  - \(== 0\) iff \(a\) is *same size as* \(b\)
  - \(> 0\) iff \(a\) is *greater than* \(b\)

- Examples

  ```javascript
  function lenOrder(a, b) {
    return a.length - b.length;
  }

  function compareNumbers(a, b) {
    return a - b;
  }
  ```
Sorting with Comparator

- Optional argument to sort
  
  ```
  sort([[compareFunction]])
  ```

- Example
  ```
  names.sort(lenOrder);
  n.sort(compareNumbers);
  ```

  ```
  n.sort(function(a, b) {
    return a - b;
  });
  ```

  ```
  n.sort((a, b) => a - b);
  ```
Iteration: Logical Quantification

```javascript
let isBig = (elt, index, array) => {
    return (elt >= 10);
}
```

- Universal quantification: `every`
  
  
  ```javascript
  [5, 8, 13, 44].every(isBig); // false
  [51, 18, 13, 44].every(isBig); // true
  ```

- Existential quantification: `some`
  
  ```javascript
  [5, 8, 13, 44].some(isBig); // true
  [5, 8, 1, 4].some(isBig); // false
  ```

- Neither modifies original array
Iteration: Filter

- Pare down an array based on a condition: *filter*
  
  \[
  \text{filter}(\text{predicate})
  \]
  
  \[
  \text{predicate(\text{element, index, array})}
  \]

- Returns a new array, with elements that satisfied the predicate
  
  - Does not modify the original array

- Example
  
  \[
  t = [12, 5, 8, 13, 44].\text{filter}(\text{isBig});
  \]
Iteration: Map

- Transform an array into a new array, element by element: map
  - E.g. an array of strings into an array of their lengths
  - `"hi", "there", "world"` → `[2, 5, 5]`
    
    ```javascript
    map(callback)
    callback(element, index, array)
    ```

- Examples
  ```javascript
  len = names.map(function(elt, i, a) {
    return elt.length
  });
  
  names.map(w => w[0].toUpperCase());
  ```
Recall: Ruby Map

- Transform an array into a new array, element by element
- Uses block to calculate each new value

```
a.map { |item| block }
```
Iteration: For Each

- Similar to map, but preferred for side-effects and changing an array in place
  
  ```javascript
  forEach(callback)
  callback(element, index, array)
  ```

- Example
  
  ```javascript
  let logArrayElts = (elt, i, arr) => {
    console.log("[" + i + "] = " + elt);
  }

  [2, 5, 9].forEach(logArrayElts);
  ```
Iteration: Reduce

- Applies a binary operator between all the elements of the array
  - E.g., to sum the elements of an array
  - \([15, 10, 8] \rightarrow 0 + 15 + 10 + 8 \rightarrow 33\)
  - `reduce(callback[, initialValue])`
    - `callback(previous, elt, index, array)`

- Examples
  ```javascript
  let sum = (a, b) => a + b;
  let acc = (a, b) => a + 2 * b;
  [2, 3, 7, 1].reduce(sum) //=> ?
  [2, 3, 7, 1].reduce(sum, 0) //=> ?
  [2, 3, "7", 1].reduce(sum) //=> ?
  [2, 3, 7, 1].reduce(acc) //=> ?
  [2, 3, 7, 1].reduce(acc, 0) //=> ?
  ```
Iteration: Reduce

- Applies a binary operator between all the elements of the array
  - E.g., to sum the elements of an array
  - \([15, 10, 8] \rightarrow 0 + 15 + 10 + 8 \rightarrow 33\)
  - `reduce(callback[, initialValue])`
    - `callback(previous, elt, index, array)`

- Examples
  - `let sum = (a, b) => a + b;`
  - `let acc = (a, b) => a + 2 * b;`
  - \([2, 3, 7, 1].reduce(sum) \//=> 13\)
  - \([2, 3, 7, 1].reduce(sum, 0) \//=> 13\)
  - \([2, 3, "7", 1].reduce(sum) \//=> "571"\)
  - \([2, 3, 7, 1].reduce(acc) \//=> 24\)
  - \([2, 3, 7, 1].reduce(acc, 0) \//=> 26\)
Recall: Ruby’s Reduction Chain

init → block → acc → item → resulting value → a
Iteration: Reduce

- Examples with anonymous functions

```javascript
[2, 3].reduce((a, b) => a + b);
//=> ?
```

```javascript
[
  [0, 1],
  [2, 3],
  [4, 5]
].reduce((a, b) => a.concat(b));
//=> ?
```
Iteration: Reduce

- Examples with anonymous functions

\[
[2, 3].reduce((a, b) => a + b);
//=> 5
\]

\[
[0, 1],
[2, 3],
[4, 5]
].reduce((a, b) => a.concat(b));
//=> [0, 1, 2, 3, 4, 5]
\]
Your Turn

Given: roster of students (an array)

Write a JavaScript program that outputs an html list of students (name and midterm score) whose gpa is > 3.0, such that the list is sorted by midterm score

1. Xi Chen (85)
2. Mary Smith (80)
3. Alessandro Reis (74)
Example Input

```javascript
let roster =
[
    { name: "Mary Smith",
        gpa: 3.7,
        midterm: 80 },
    { name: "Xi Chen",
        gpa: 3.5,
        midterm: 85 },
    { name: "Alessandro Reis",
        gpa: 3.2,
        midterm: 74 },
    { name: "Erin Senda",
        gpa: 3.0,
        midterm: 68 }
];
```
One Solution

document.writeln("<ol>");
document.writeln(
    roster
    .filter((s) => s.gpa > 3.0)
    .sort((a, b) => b.midterm - a.midterm)
    .map((s) => "<li>" + s.name + " (" + s.midterm + ")" + "</li>"
    )
    .join("\n")
);
document.writeln("</ol>");
To Ponder

Assume:

```javascript
var d = new Dog();
d instanceof Dog; //=> true
d instanceof Pet; //=> true
```

Questions:

- What is Dog? (A class? An interface? ...)
- What is Pet?
- How are they related? Draw the hierarchy
To Ponder
JavaScript: Objects, Methods, Prototypes
What is an Object?

- **Property:** a key/value pair
  - aka name/value pair
- **Object:** a partial map of properties
  - Keys must be unique
- Creating an object, literal notation
  ```
  let myCar = {
    make: "Acura",
    year: 1996,
    plate: "NKR462"
  };
  ```
- To access/modify an object's properties:
  ```
  myCar.make = "Ford";    // cf. Ruby
  myCar["year"] = 2006;
  let str = "ate";
  myCar["pl" + str] == "NKR463"; //=> true
  ```
Object Properties

myCar

make: "Ford"
year: 2006
plate: "NKR463"
Arrays vs Associative Arrays

0  4
1  "hi"
2  true
3  3.14

0  true
1  true
2  false

0  age
1  greeting
2  doors
3  pi

0  4
1  "hi"
2  true
3  3.14

0  true
1  true
2  false
Objects can grow

```javascript
myCar.state = "OH"; // 4 properties
let myBus = {};
myBus.driver = true; // adds a prop
myBus.windows = [2, 2, 2, 2];
```

Objects can shrink

```javascript
delete myCar.plate;
// myCar is now { make: "Ford",
// year: 2006, state: "OH" }
```
Object Properties

myCar

make: "Ford"
year: 2006
plate: "NKR463"
Object Properties

```javascript
myCar.state = "OH";
```

```
class Car {
  constructor(make, year, plate, state) {
    this.make = make;
    this.year = year;
    this.plate = plate;
    this.state = state;
  }
}
```

```
const myCar = new Car("Ford", 2006, "NKR463", "OH");
```
Object Properties

```javascript
myCar

delete myCar.plate;

make: "Ford"
year: 2006
state: "OH"
```
Testing Presence of Key

- **Boolean operator**: `in`  
  
  `propertyName in object`

- Evaluates to true iff object has the indicated property key
  
  ```javascript
  "make" in myCar //=> true
  "speedometer" in myCar //=> false
  "OH" in myCar //=> false
  ```

- Property names are strings
Iterating Over Properties

- Iterate using `for...in` syntax
  ```javascript
  for (property in object) {
    object[property]...
  }
  ```

- Notice `[]` to access each property
  ```javascript
  for (let p in myCar) {
    document.write(p + " : " + myCar[p]);
  }
  ```

- Loop over iterable (eg array) with `for...of`
  ```javascript
  for (let n of roster) {
    document.write("name: " + n);
  }
  ```
Methods

The value of a property can be:

- A primitive (boolean, number, string, null...)
- A reference (object, array, function)

```javascript
let temp = function(sound) {
    play(sound);
    return 0;
}
myCar.honk = temp;
```

More succinctly:

```javascript
myCar.honk = function(sound) {
    play(sound);
    return 0;
}
```
Example: Method

```javascript
let myCar = {
    make: "Acura",
    year: 1996,
    plate: "NKR462",
    honk: function(sound) {
        play(sound);
        return 0;
    }
};
```
Example: Method (with Sugar)

```javascript
let myCar = {
    make:  "Acura",
    year:  1996,
    plate:  "NKR462",
    honk(sound)  {
        play(sound);
        return 0;
    }
};
```
Object Properties

myCar

- make: "Acura"
- year: 1996
- plate: "NKR462"
- honk:

```cpp
play(sound);
return 0;
```
Keyword “this” in Functions

- Recall distinguished formal parameter
  \( x.f(y, z); //x \text{ is the distinguished argmt.} \)

- Inside a function, keyword “this”
  
  ```javascript
  function report() {
      return this.plate + this.year;
  }
  ```

- At run-time, “this” is the distinguished argument of the invocation
  
  ```javascript
  myCar = { plate: "NKR462", year: 1996 }; 
  yourCar = { plate: 340, year: 2013 }; 
  myCar.register = report; 
  yourCar.info = report; 
  myCar.register(); //=> "NKR4621996" 
  yourCar.info(); //=> 2353
  ```
Object Properties

```
myCar
  plate: "NKR462"
  year: 1996
  register

return this.plate + this.year;

yourCar
  plate: 340
  year: 2013
  info
```
Constructors

- Any function can be a constructor
- When calling a function with “new”:
  1. Make a brand new (empty) object
  2. Call the function, with the new object as the distinguished parameter
  3. Implicitly return the new object to caller
- A “constructor” often adds properties to the new object simply by assigning them

```javascript
function Dog(name) {
    this.name = name;  // adds 1 property
    // no explicit return
}
let furBall = new Dog("Rex");
```

- Naming convention: Functions intended to be constructors are capitalized
Example

```javascript
function Circle(x, y, radius) {
    this.centerX = x;
    this.centerY = y;
    this.radius = radius;
    this.area = function() {
        return Math.PI * this.radius * this.radius;
    }
}

let c = new Circle(10, 12, 2.45);
```
Creating a Circle Object

```javascript
let c = new Circle(10, 12, 2.45);
```

```javascript
this.centerX = x;
this.centerY = y;
... Etc ...
```
Creating a Circle Object

```javascript
let c = new Circle(10, 12, 2.45);
```

```javascript
this.centerX = x;
this.centerY = y;
... Etc ...
```
Creating a Circle Object

```javascript
let c = new Circle(10, 12, 2.45);

Circle

{ centerX: 10, centerY: 12, radius: 2.45 }

... Etc ...

return Math.PI * this.radius * this.radius
```

### Variables
- `centerX`
- `centerY`
- `radius`
- `area`
Creating a Circle Object

```javascript
let c = new Circle(10, 12, 2.45);
```

```javascript
Circle
this.centerX = x;
this.centerY = y;
... Etc ...
```

```javascript
return Math.PI * this.radius * this.radius
```
Creating a Circle Object

```javascript
let c = new Circle(10, 12, 2.45);
```

```javascript
centerX = 10
centerY = 12
radius = 2.45
area = return Math.PI * this.radius * this.radius

this.centerX = x;
this.centerY = y;
... Etc ...
```

Circle

```javascript
return Math.PI * this.radius * this.radius
```
Creating Many Circle Objects

```javascript
for (let i=0; i<1000; i++) {
    new Circle(0, 0, i);
}
```

```javascript
this.centerX = x;
this.centerY = y;
... Etc ...
```

```javascript
return Math.PI * this.radius * this.radius
```

Circle

centerX
10

centerY
12

radius
2.45

area

How many of these?
Prototypes

- Every object has a *prototype*
  - A hidden, indirect property (``[[Prototype]]``)
- What is a prototype?
  - Just another object! Like any other!
- When accessing a property (*i.e. obj.p*)
  - First look for `p` in `obj`
  - If not found, look for `p` in `obj`'s prototype
  - If not found, look for `p` in that object's prototype!
  - And so on, until reaching the basic system object
Prototype Chaining

- greeting: "hi"
- doors: 3.14
- pi: 3.14
- age: 4
- tostring: hasOwnProperty
- push: pop
- etc...
Class-Based Inheritance

interfases

extends

implements

classes

static

extends

static

instantiates

objects
Prototype: Get vs Set of Proprty

- Consider two objects
  ```javascript
  let dog = { name: "Rex", age: 3 };  
  let pet = { color: "blue" };  
  ```

- Assume `pet` is `dog`'s prototype
  ```javascript
  // dog.name == "Rex"
  // dog.color == "blue" (follow chain)
  pet.color = "brown"; // set in proto
  // dog.color is "brown" (prop changed)
  dog.color = "green"; // set in object
  // pet.color is still "brown" (hiding)
Delegation to Prototype

dog

name
"Rex"

age
3

pet

color
"blue"

`dog.color == ?`  
// get follows prototype chain
Delegation to Prototype

```
pet.color = "brown";
// set changes object

dog.color == ?
// get follows prototype chain
```
Delegation to Prototype

```javascript
// set changes object!

let dog = { name: "Rex", age: 3, color: "green" };

// get follows prototype chain

dog.color = "green";

dog.color == ?

// set changes object!
```

Diagram:

- dog
  - name: "Rex"
  - age: 3
  - color: "green"
- pet
  - color: "brown"
Prototypes Are Dynamic Too

- Prototypes can add/remove properties
- Changes are felt by all children

```javascript
// dog is { name: "Rex", age: 3 }
// dog.mood & pet.mood are undefined
pet.mood = "happy"; // add to pet
// dog.mood is now "happy" too
pet.bark = function() {
  return this.name + " is " + this.mood;
}
dog.bark(); //=> "Rex is happy"
pet.bark(); //=> "undefined is happy"
```
Delegation to Prototype

dog
- name: "Rex"
- age: 3

pet
- color: "brown"
- mood: "happy"

```
dog.bark();
pet.bark();
```
Connecting Objects & Prototypes

How does an object get a prototype?

```javascript
let c = new Circle();
```

Answer

1. Every function has a prototype property
   - Do not confuse with hidden `[[Prototype]]`!
2. Object's prototype `link`—`[[Prototype]]`—is set to the function's prototype property

When a function `Foo` is used as a constructor, i.e. `new Foo()`, the value of `Foo`'s prototype property is the prototype object of the created object
Prototypes And Constructors

Circle

```javascript
this.centerX = x;
this.centerY = y;
... Etc ...
```
Prototypes And Constructors

c = new Circle()

prototype
	his.centerX = x;
this.centerY = y;
... Etc ...

Circle

constructor

area

Computer Science and Engineering  The Ohio State University
Prototypes And Constructors

\[
c = \text{new Circle()}
\]

Prototype:

```
this.centerX = x;
this.centerY = y;
... Etc ...
```
Prototypes And Constructors

\[ c = \text{new Circle()} \]

```
this.centerX = x;
this.centerY = y;
... Etc ...
```
Idiom: Put Methods in Prototype

```javascript
function Dog(n, a) {
  this.name = n;
  this.age = a;

  this.bark = function(sound) {
    return this.name + " says " + sound;
  }
}

// bad: method is added to object itself
```
Method is in Object

```
r = new Dog()
Dog.prototype
return this.name + "says" + sound;
```

```
this.name = x;
this.age = a;
this.bark = ...
```
function Dog(n, a) {
    this.name = n;
    this.age = a;
}

Dog.prototype.bark = function(sound) {
    return this.name + " says " + sound;
};

// good: add method to prototype
Idiom: Methods in Prototype

```javascript
class Dog {
    constructor(n, a) {
        this.name = n;
        this.age = a;
    }
    bark(sound) {
        return this.name + "says" + sound;
    }
}

// best: ES6 classes (syntactic sugar)
```
Methods in Prototype

```
r = new Dog();

this.name = x;
this.age = a;

return this.name + " says " + sound;
```
class Dog {
    name: "Fur"; // property in prototype!
    age: 0;

    constructor(n, a) {
        this.name = n; // hides prototype property
        this.age = a;
    }

    bark(sound) {
        return this.name + " says " + sound;
    }
}
Methods in Prototype

```javascript
r = new Dog()

name: "Rex"
age: 6

Dog.prototype

constructor

name: "Fur"
age: 0
bark

this.name = x;
this.age = a;

return this.name + "says" + sound;
```
Meaning of `r instanceof Dog`

\[ r = \text{new Dog()} \]

```javascript
Dog.prototype
constructor
this.name = x;
this.age = a;
return this.name + "says" + sound;
```

\[ r.__proto__.constructor == \text{Dog} \]
function Animal() { ... };  
function Dog() { ... };  

Dog.prototype = new Animal();  
// create prototype for future dogs

Dog.prototype.constructor = Dog;  
// set prototype's constructor  
// properly (ie should point to Dog())
Setting up Prototype Chains

```javascript
r = new Dog()

new Animal() // Dog.prototype

name: "Rex"

Dog.prototype

new Animal() // Animal.prototype

constructor

Dog

Animal

prototype

constructor

prototype
```
Prototype Chains

- `instanceOf` is checked transitively up the prototype chain
  
  ```javascript
  r instanceof Dog //=> true
  r instanceof Animal //=> true
  r instanceof Object //=> true
  ```

- Q: Identify in the previous diagram
  ```javascript
  r.__proto__.__proto__.constructor
  Dog.prototype.__proto__
  .constructor.prototype
  ```
To Ponder
Summary

- Objects as associative arrays
  - Partial maps from keys to values
  - Can dynamically add/remove properties
  - Can iterate over properties

- Method = function-valued property
  - Keyword this for distinguished parameter

- Any function can be a constructor

- Prototypes are "parent" objects
  - Delegation up the chain of prototypes
  - Prototype is determined by constructor
  - Prototypes can be modified