CSWD

Client Side Web Development

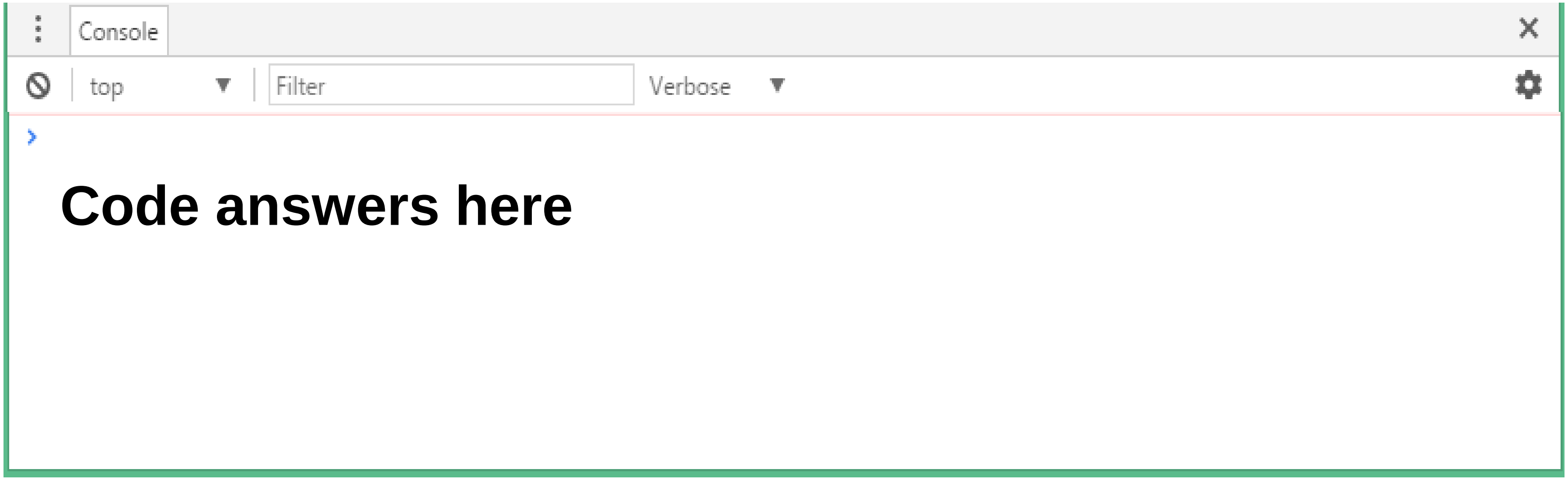
Week 6

Working with Arrays, Functions, Objects …

...

```
Code here
```

Console Output:



Code answers here

Variable and Function Scope

Variables defined inside a function with a new var are only valid inside this function, not outside it!

Let's define two functions as follows:

Local variable

```
function withVar(){
     var myVar = 3;
     console.log('Inside withVar myVar is: '
      + myVar);
}
function withoutVar(){
     myVar = 6;
     console.log('Inside withoutVar myVar is: '
      + myVar);
}
```

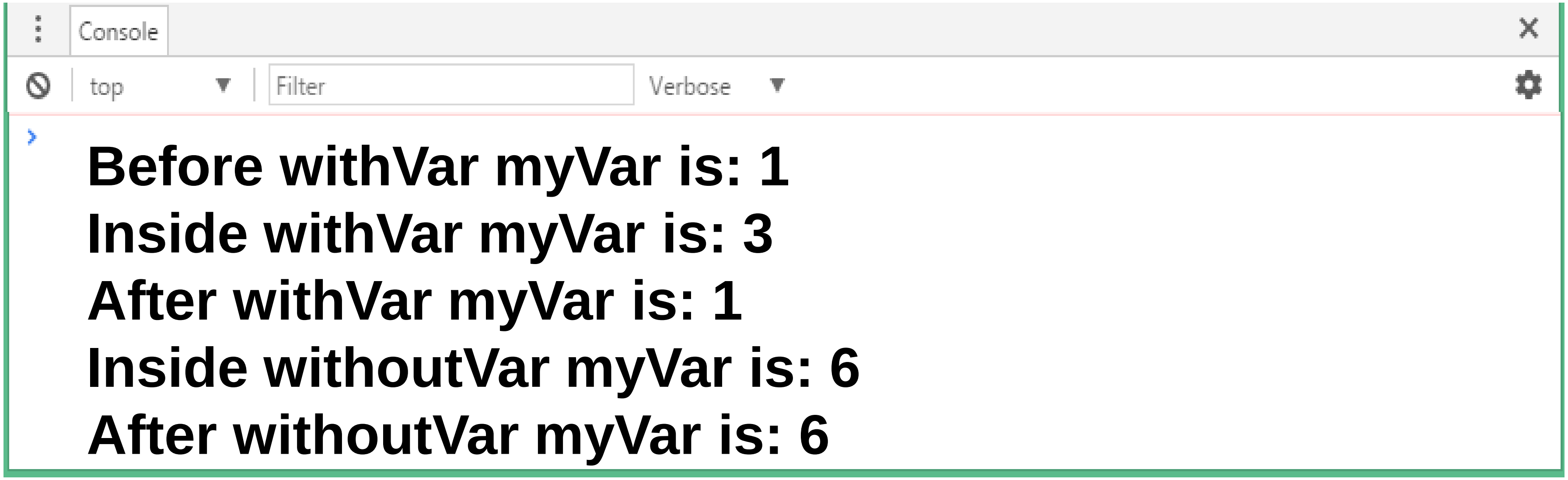
Global variable

Variable and Function Scope …

- Variables defined outside functions are called global variables and are potentially dangerous.
- We should try to keep all our variables contained inside functions.
  - This ensures that our script will “play nicely” with other scripts that may be applied to the page.
  - Many scripts use generic variable names like num or currentValue etc
    - If these are defined as global variables, the scripts will override each other’s settings ☹

```
var myVar=1 // Global variable
console.log('Before withVar myVar is:' + myVar);
withVar();
console.log('After withVar myVar is:' +myVar);
withoutVar();
console.log('After withoutVar myVar is:' +myVar);
```

Console Output:



```
Before withVar myVar is: 1
Inside withVar myVar is: 3
After withVar myVar is: 1
Inside withoutVar myVar is: 6
After withoutVar myVar is: 6
```

Functions are data

- Functions in JavaScript are actually data.
- This means that you can create a function and assign it to a variable, as follows:

```
var f = function () {
  return 1;
};
```

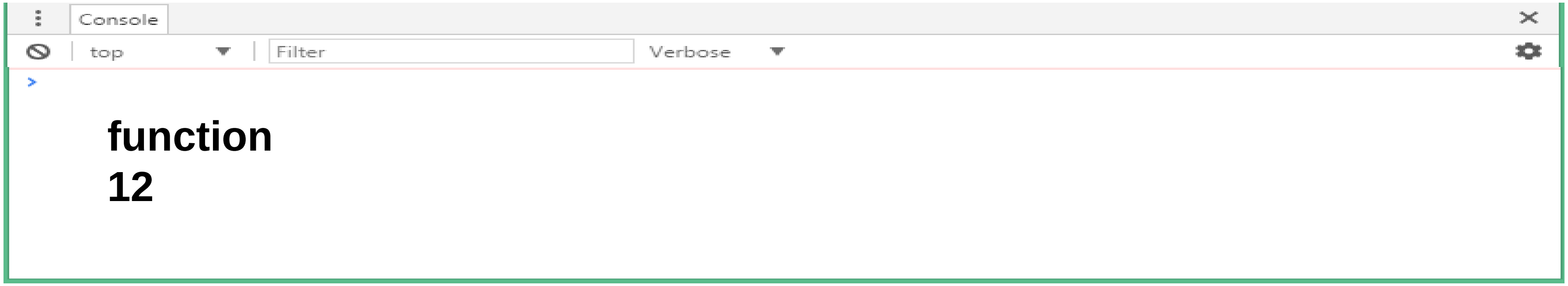
Highlighted part is a function expression

- This way of defining a function is sometimes referred to as function literal notation.
- So, JavaScript functions are data, but a special kind of data with the following two important features:
  - ➢ they contain code and
  - ➢ they are executable (i.e. they can be invoked)

Functions are data cont …

- **Functions in JavaScript are invoked by adding parentheses after its name.**
  - **this works regardless of how the function was defined e.g.**

```
var multiply = function (a, b) {
  return a * b;
};
var times = multiply;
console.log( typeof times );
times( 3, 4 );
```



```
function
12
```

Anonymous functions

- **As you now know, there exists a *function expression* syntax where you can have a function defined like the following:**

```
var f = function (a) {
  return a;
};
```

- **This is also often called an anonymous function (as it doesn't have a name), especially when such a function expression is used even without assigning it to a variable.**
- **Potential uses:**
  - ➢ **You can pass an anonymous function as a parameter to another function so the receiving function can do something useful with the function that you pass.**
  - ➢ **You can define an anonymous function and execute it right away.**

Callback functions

- Here's an example of a function that accepts two functions as parameters, executes them, and returns the sum of what each of them returns:

```
function invokeAdd(a, b) {
  return a() + b();
}
```

- Now, let's define two simple additional functions using a function declaration pattern that only returns hardcoded values:

```
function four() {
  return 4;
}
```

```
function two() {
   return 2;
}
```

Q^2. Given the following statement write down the expected output:

invokeAdd(two, four);

Callback functions cont …

- When you pass a function, A, to another function, B, and then B executes A, it's often said that A is a callback function.

```
function B( A ) {
  return A();
}
```

- If A doesn't have a name, then you can say that it's an anonymous callback function.

```
function B( function (){
 return someresult;
} )
```

Immediate functions

- **Here’s another application of an anonymous function-calling a function immediately after it's defined e.g.**

```
(
 function ( args ) {
   alert('Hello '+args);
 }
)('Class');   // outputs: Hello Class
```

a function expression inside parentheses followed by another set of parentheses.

- **The second set says execute now and is also the place to put any arguments that your anonymous function might require.**

JavaScript: Object-Based Language

- There are three object categories in JavaScript: Native Objects, Host Objects, and User-Defined Objects.
  - Native objects: defined by JavaScript.
    - String, Number, Array, Image, Date, Math, etc.
  - Host objects : supplied and always available to JavaScript by the browser environment.
    - window, document, forms, etc.
  - User-defined objects : defined by the author/programmer
- Initially, probably used Native and Host objects created by the browser and their methods and properties
  - Need to become familiar with user-defined objects

Keeping Scripts Safe

- We’ve seen that we can keep variables safe by defining them locally via the var keyword.
- The reason was to avoid other functions relying on variables with the same name and the two functions overwriting each other’s values.
- The same applies to functions.
  - As you can include several JavaScript files to the same HTML document in separate script elements your functionality might break as another included document has a function with the same name ☹

Keeping Scripts Safe – objects 1

- **We can define an object and use our functions as methods of this object e.g.**

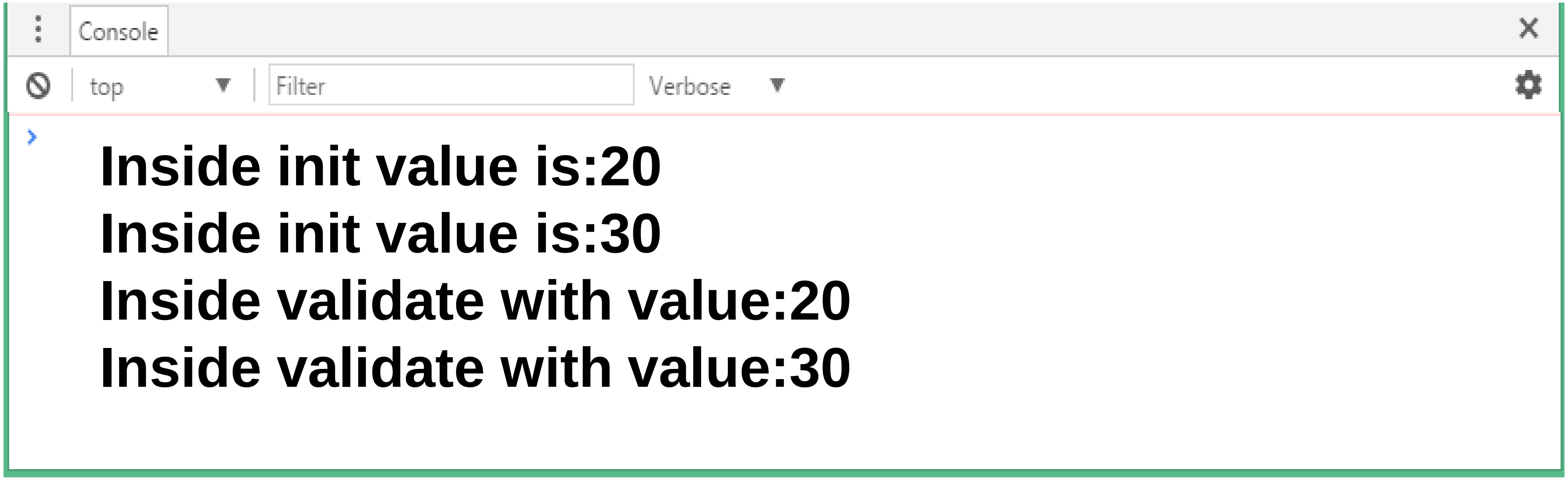
```
function myObject(value) {
  this.value = value;
  this.init = init;
  this.validate = validate;
}
function init(amount) {
  this.value += amount;
  console.log("Init value is:" + this.value);
}
function validate() {
  console.log("Validate value is:" + this.value ");
}
```

Keeping Scripts Safe - testing

- To call these functions you need to use myscript.init( ) and myscript.validate( ) e.g.

```
var x = new myObject();x.init(20);
var y = new myObject();y.init(30);
    x.validate(); y.validate();
```

Console Output:



```
Inside init value is:20
Inside init value is:30
Inside validate with value:20
Inside validate with value:30
```

Keeping Scripts Safe – objects 2

- **We can define a new object and use our functions as methods of this object e.g.**

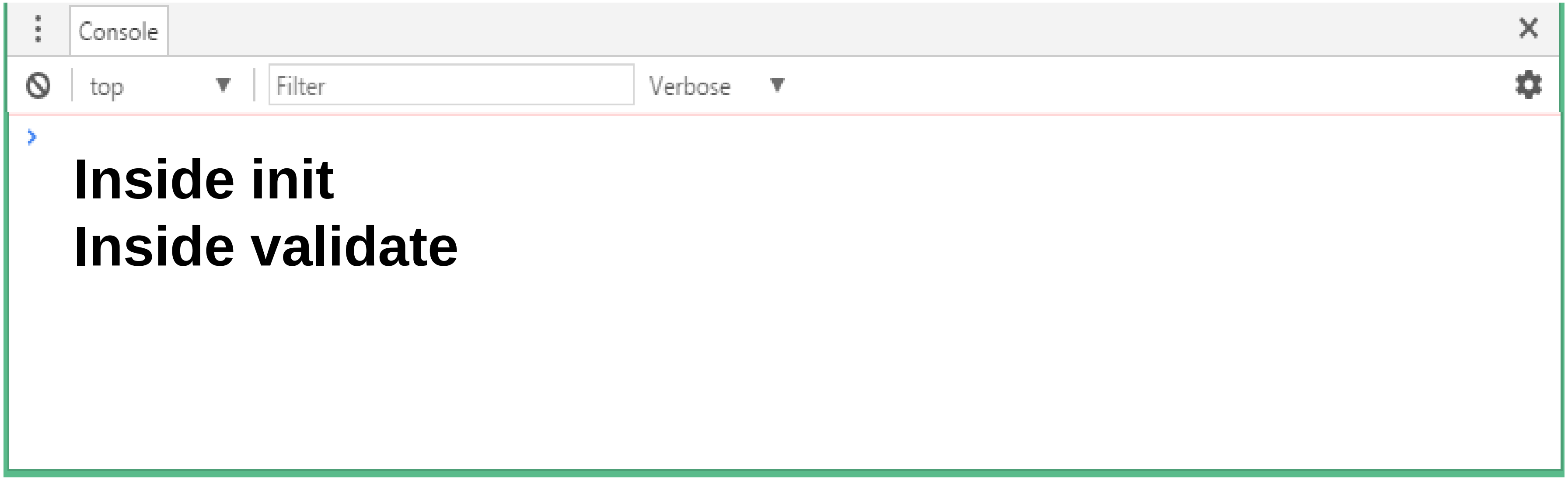
```
myScript = new Object();
myScript.init = function(){
  console.log("Inside init");
};

myScript.validate = function() {
   console.log("Inside validate");
};
```

Keeping Scripts Safe - testing

- **To call these functions you need to use myscript.init( ) and myscript.validate( ) e.g.**

```
myScript.init();
myScript.validate();
```

Console Output:



```
Inside init
Inside validate
```

Keeping Scripts Safe – object literal

- The object literal approach uses a shortcut notation to create the object and apply each of the functions as object methods instead of stand-alone functions. e.g.

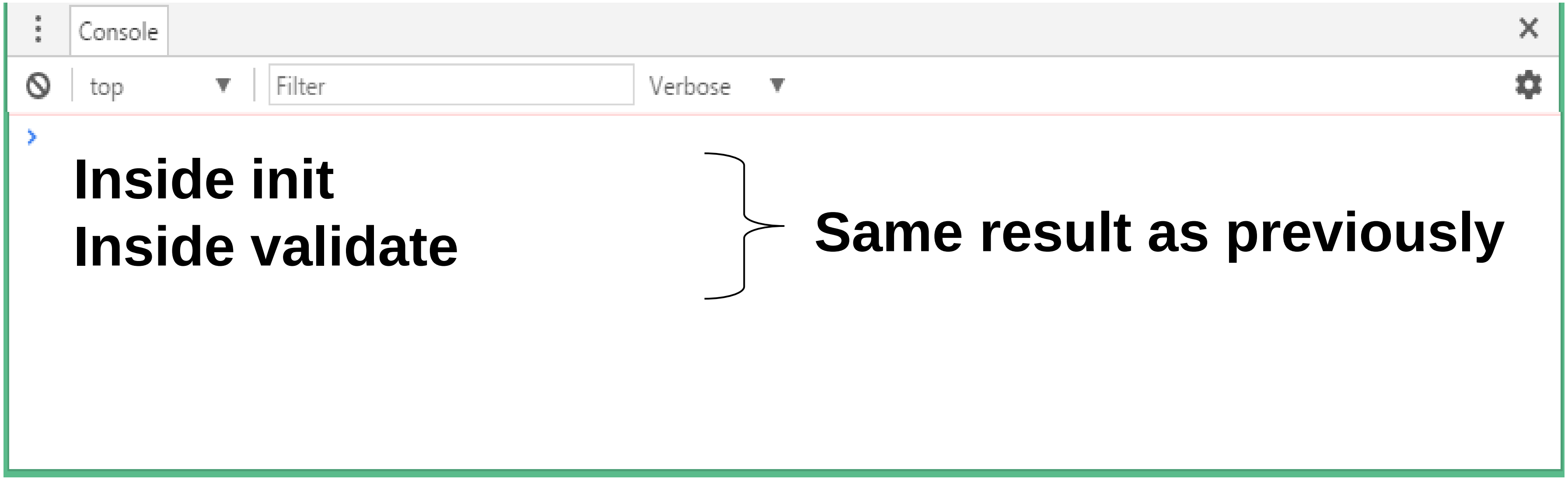
```
var myScript = {
  init: function(){
    console.log("Inside init");
  },

  validate: function() {
    console.log("Inside validate");
  }
}
```

Keeping Scripts Safe - testing

- **To call these functions you need to use myscript.init( ) and myscript.validate( ) e.g.**

```
myScript.init();
myScript.validate();
```

Console Output:



```
Inside init
Inside validate
```

Same result as previously

Keeping Scripts Safe – object literal vars

- If you want to use variables that should be accessible by all methods inside the object, you can do that with syntax that is quite similar. e.g.

```
var myScriptObject = {
  myVar: 42,
  tmpStr: "Hello World!",
  init:function(){
    console.log(this.tmpStr);
  },
  validate:function() {
    console.log( this.myVar == 42 );
  }
}
```

Q^2. Assume that the myScriptObject declaration comes after the previous use of myVar on slide 5. What value will myVar now have?

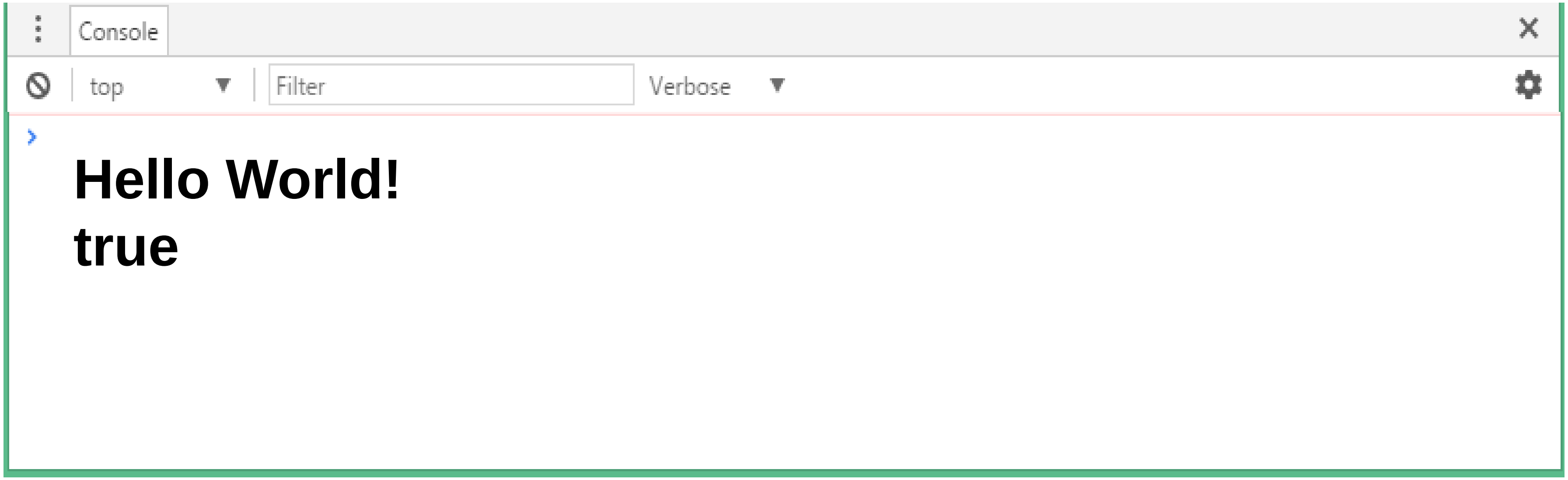
```
        console.log('myVar is: ' + myVar);
```

Keeping Scripts Safe - testing

- **To call these functions you need to use myscript.init( ) and myscript.validate( ) e.g.**

```
myScriptObject.init();
myScriptObject.validate();
```

Console Output:

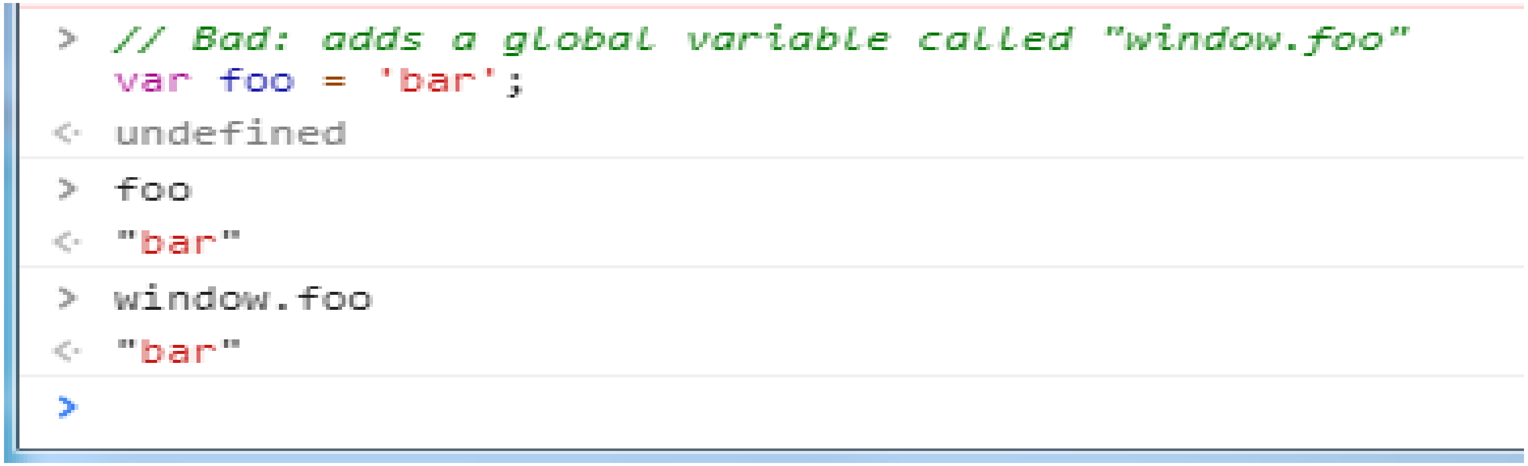


```
Hello World!
true
```

Do not leak global variables

- **Avoid adding variables to the global scope if you don't need to.**
  - **The snippet below will implicitly add a global variable**

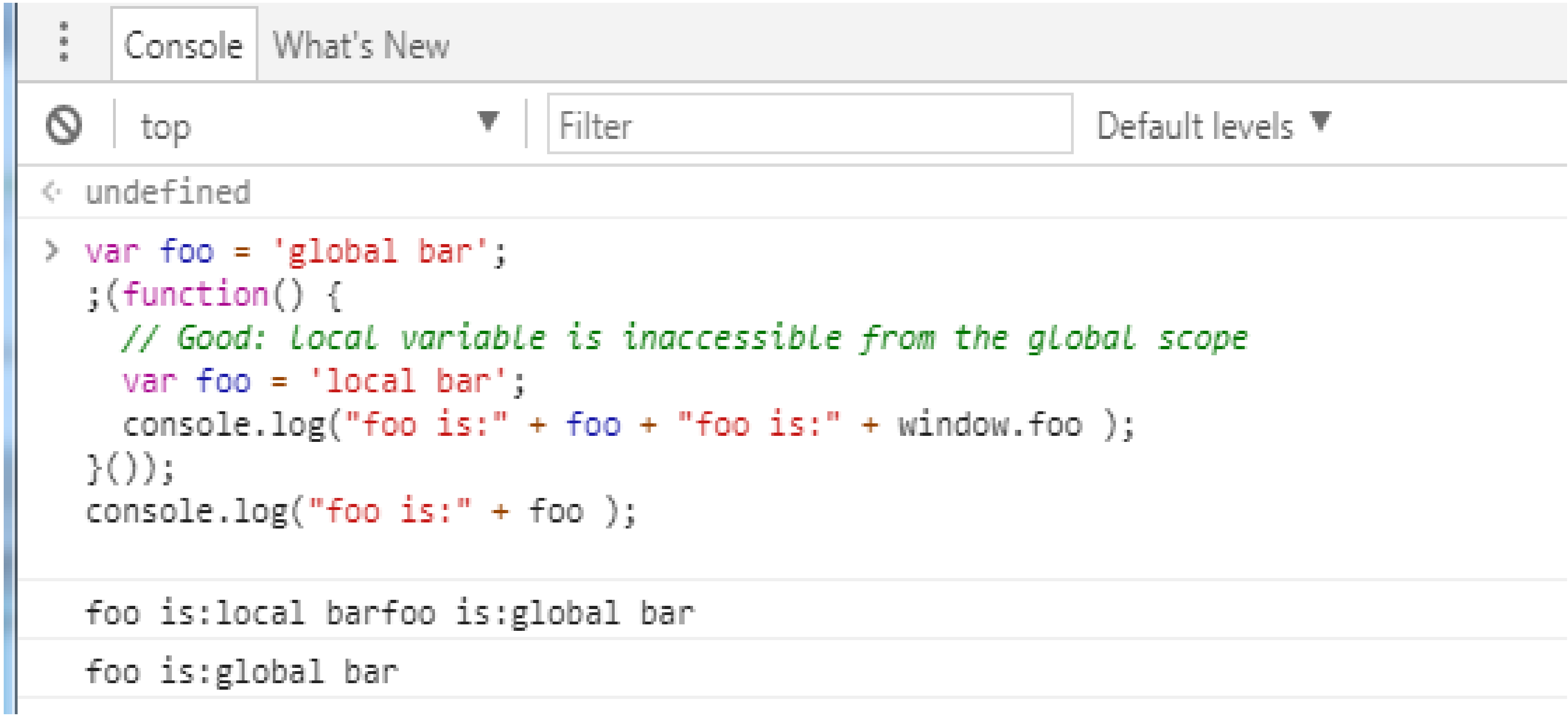
```
// Bad: adds a global variable called
"window.foo"

var foo = 'bar';
```



```
> // Bad: adds a global variable called "window.foo"
  var foo = 'bar';
< undefined
> foo
< "bar"
> window.foo
< "bar"
>
```

Cont …

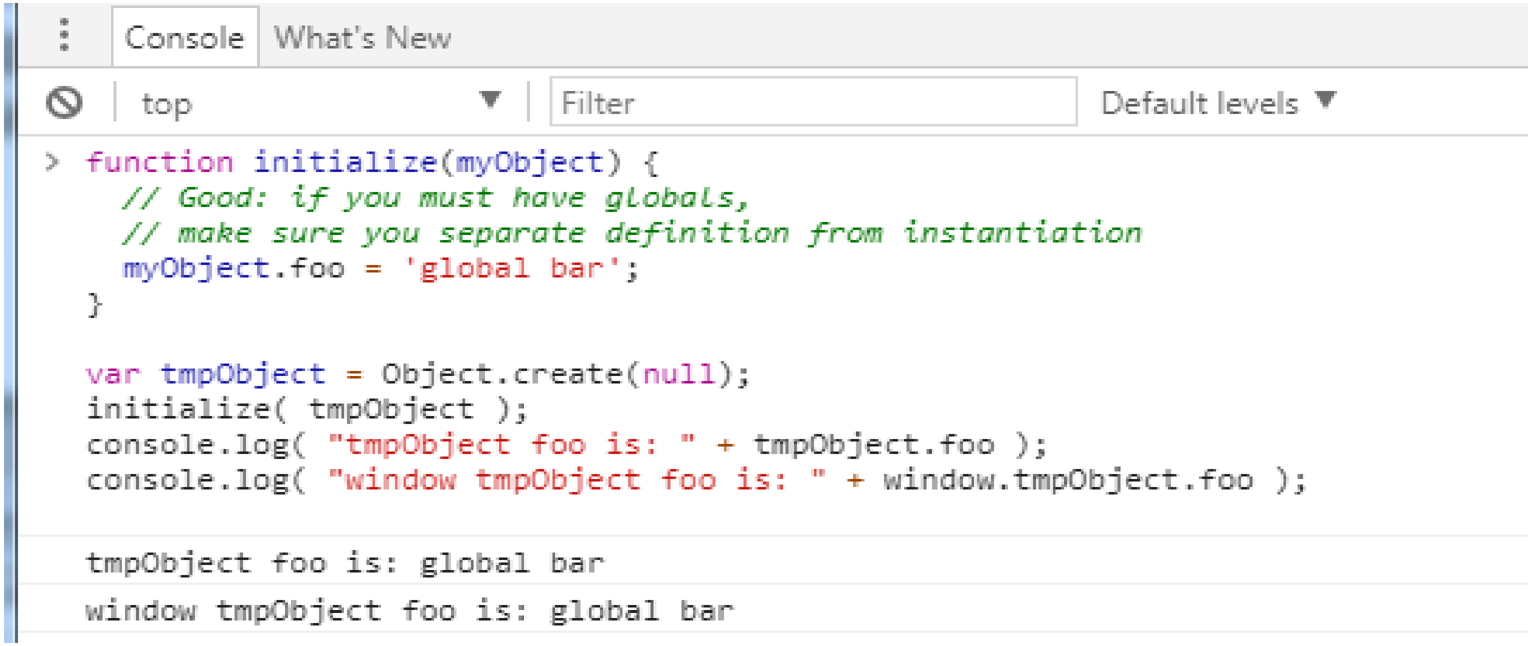
- To prevent variables from becoming global, always write your code in a closure/anonymous function - or have a build system that does this for you:



```
< undefined
> var foo = 'global bar';
  ;(function() {
    // Good: local variable is inaccessible from the global scope
    var foo = 'local bar';
    console.log("foo is:" + foo + "foo is:" + window.foo );
  }());
  console.log("foo is:" + foo );

  foo is:local barfoo is:global bar
  foo is:global bar
```

Cont …

- **If you need to register a global variable, then you should make it a big thing and only do it in one specific place in your code.**
  - **This isolates instantiation from definition, and forces you to look at your ugly state initialization instead of in multiple places**



```
> function initialize(myObject) {
    // Good: if you must have globals,
    // make sure you separate definition from instantiation
    myObject.foo = 'global bar';
  }

  var tmpObject = Object.create(null);
  initialize( tmpObject );
  console.log( "tmpObject foo is: " + tmpObject.foo );
  console.log( "window tmpObject foo is: " + window.tmpObject.foo );
```

```
tmpObject foo is: global bar
window tmpObject foo is: global bar
```

Javascript Arrays Defined

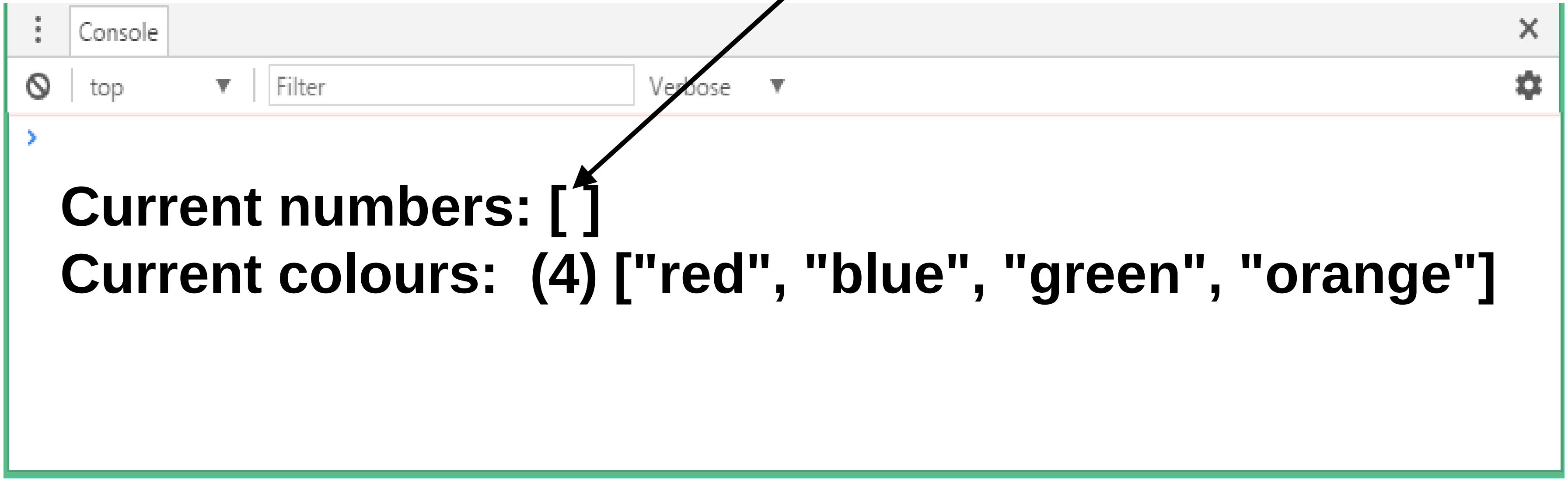
- A JavaScript array is actually a specialized type of JavaScript object, with the indices being property names that can be integers used to represent offsets.
- However, when integers are used for indices, they are converted to strings internally in order to conform to the requirements for JavaScript objects.
- Because JavaScript arrays are just objects, they are not quite as efficient as the arrays of other programming languages.
- While JavaScript arrays are, strictly speaking, JavaScript objects, they are specialized objects categorized internally as arrays.
- The Array is one of the recognized JavaScript object types, and as such, there is a set of properties and functions you can use with arrays.
- Arrays in JavaScript are very flexible.
- There are several different ways to create arrays, access array elements, and perform tasks such as searching and sorting the elements stored in an array.

Declare an array - 1

```
var numbers = [], colours;
colours = ["red", "blue", "green", "orange"];
console.log("Current numbers: ", numbers);
console.log("Current colours: ", colours);
```

Console Output:

Empty set



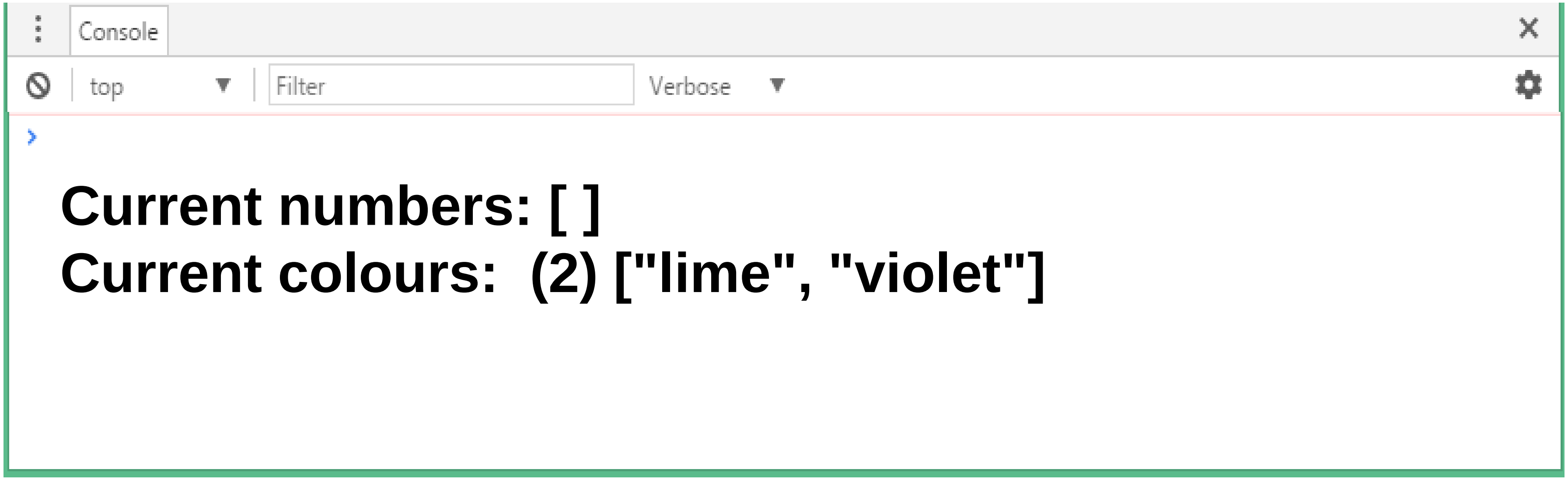
```
Current numbers: [ ]
Current colours:  (4) ["red", "blue", "green", "orange"]
```

Declare an array - 2

Empty array

```
var numbers = new Array();
var twoColours = new Array("lime", "violet");
console.log("Current numbers: ", numbers);
console.log("Current colours: ", twoColours);
```

Console Output:



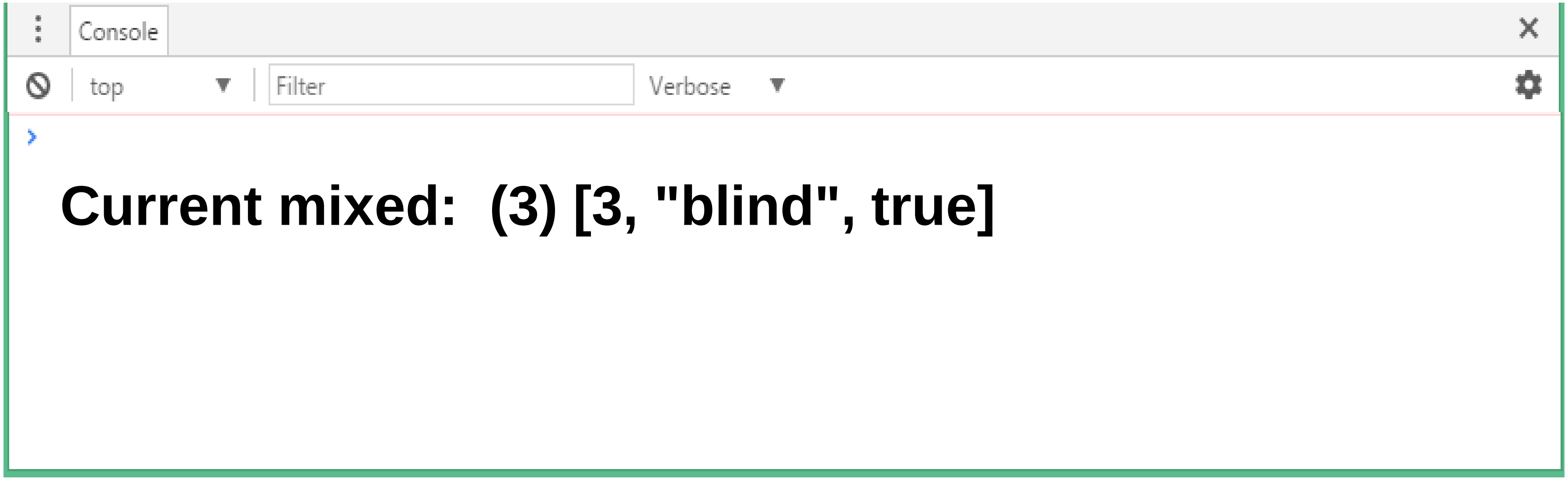
```
Current numbers: [ ]
Current colours:  (2) ["lime", "violet"]
```

Declare an array - 3

Different types!

```
var mixedRA = new Array(3, "blind", true );
console.log("Current mixed: ", mixedRA);
```

Console Output:



```
Current mixed:  (3) [3, "blind", true]
```

Accessing and Writing Array Elements

- Data is assigned to array elements using the [ ] operator in an assignment statement.
  - For example, the following loop assigns the values 1 through 100 to an array:

```
var nums = []; var i;
for (i=0; i<100; ++i) { nums[i] = i+1; }
```

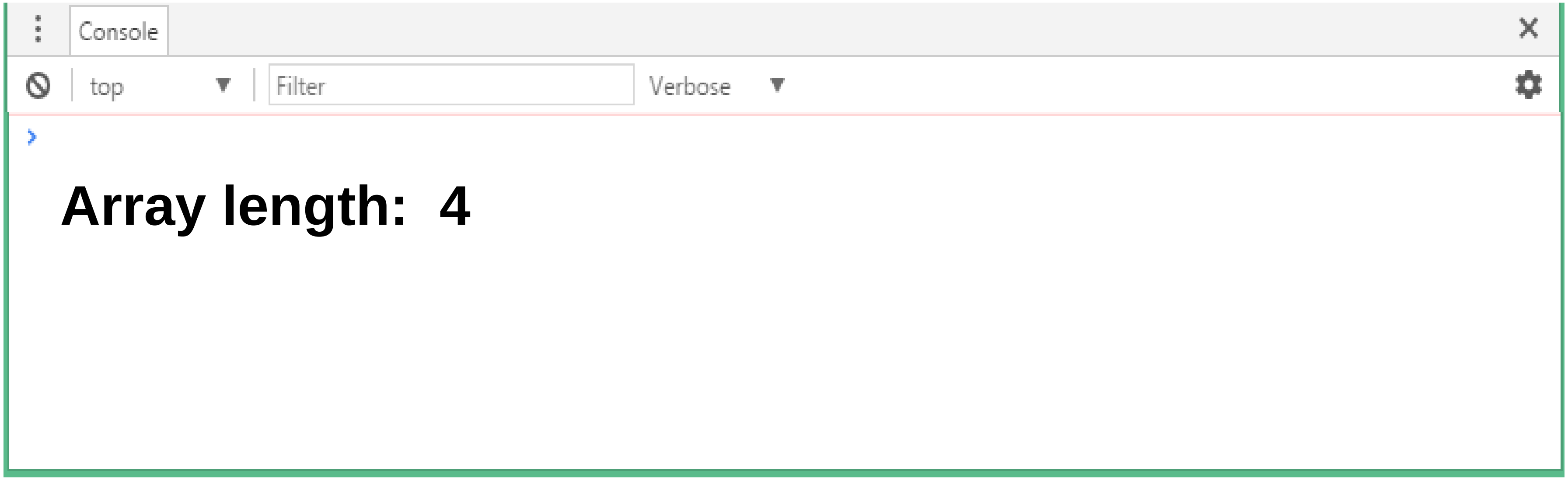
- Array elements are also accessed using the [ ] operator e.g.

```
var numbers = [1,4,10];
var sum = numbers[0] + numbers[1] + numbers[2];
console.log(sum); // displays 15
```

PROPERTIES

```
// Get a property of an object by name:
console.log("Array length: ", colours.length);
```

Console Output:

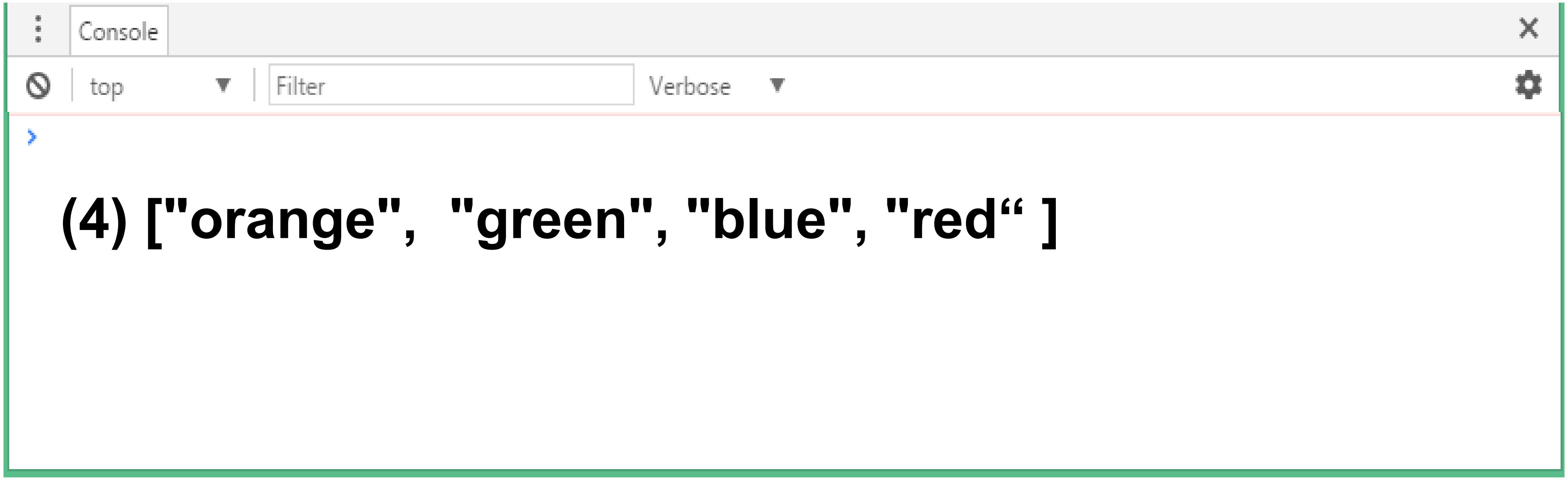


```
Array length:  4
```

Methods – reverse()

```
// Reverse the array:
colours.reverse();
```

Console Output:

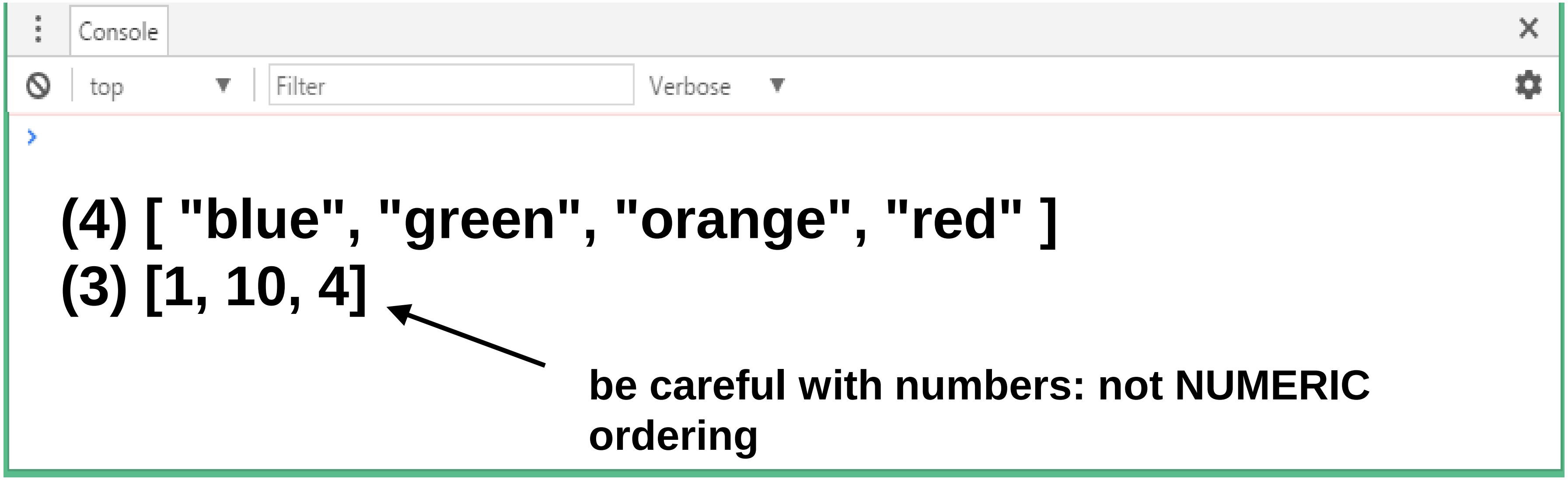


```
(4) ["orange",  "green", "blue", "red“ ]
```

Methods – sort()

```
// sort the array:
colours.sort();
numbers.sort();
```

Console Output:



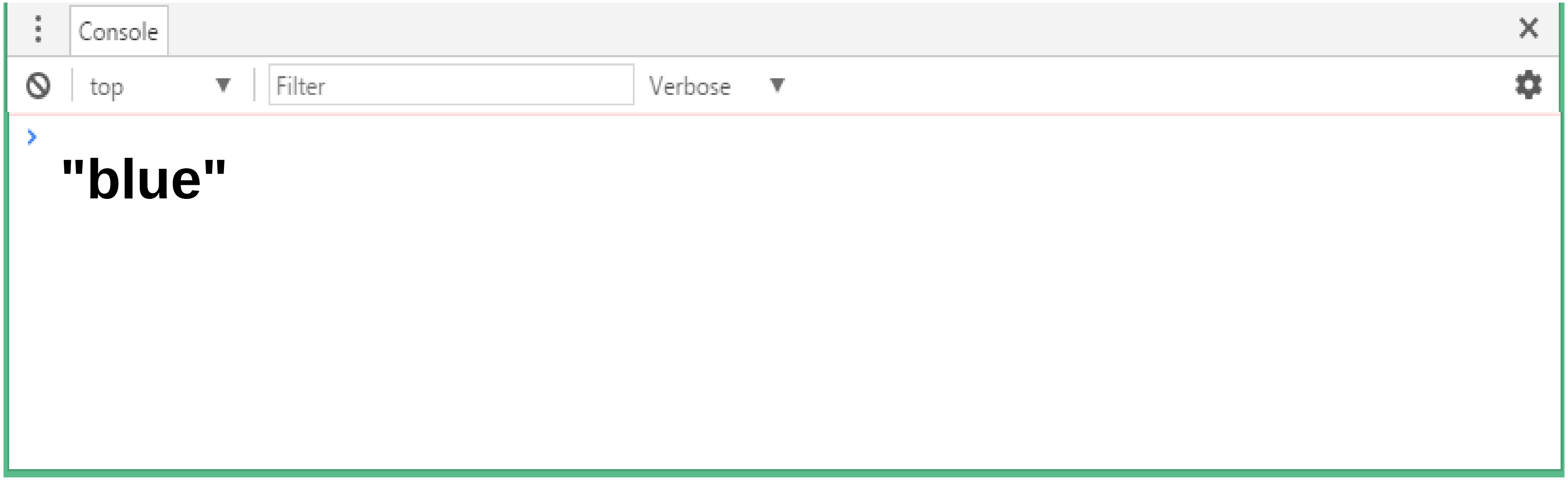
```
(4) [ "blue", "green", "orange", "red" ]
(3) [1, 10, 4]
```

be careful with numbers: not NUMERIC ordering

Methods … shift()

```
// Remove the first value of the array:
colours.shift();
```

Console Output:

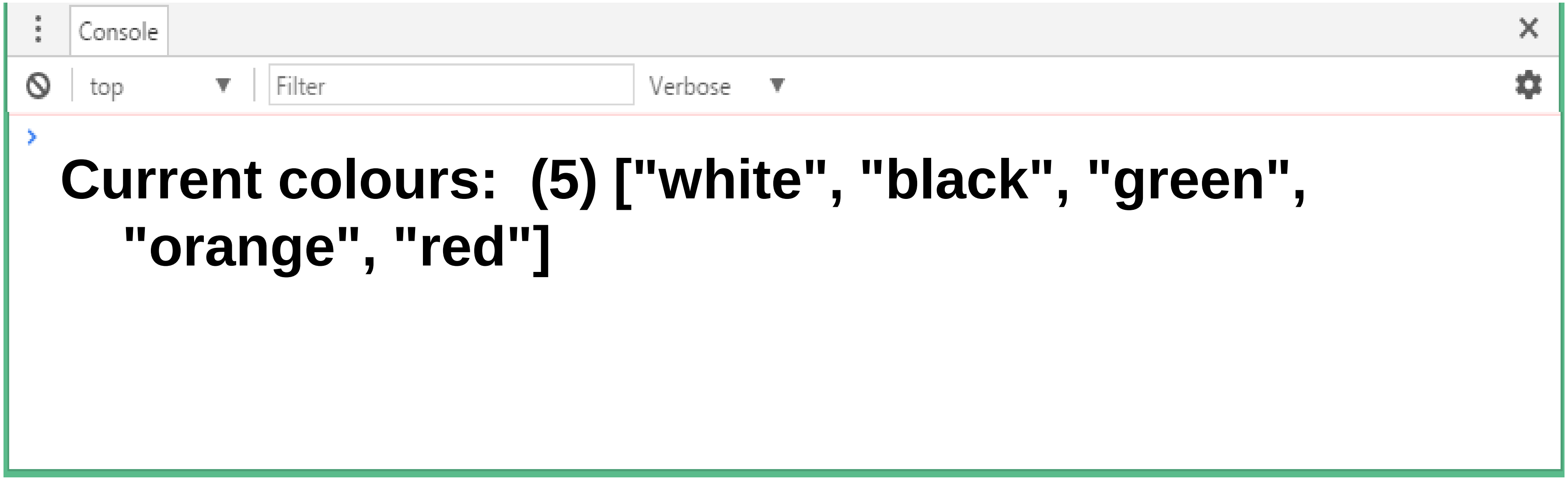


```
> 
  "blue"
```

Methods ...unshift()

```
// Add comma-separated list of values to array
colours.unshift("white", "black");
// now display current list of colours
console.log("Current colours: ", colours);
```

Console Output:

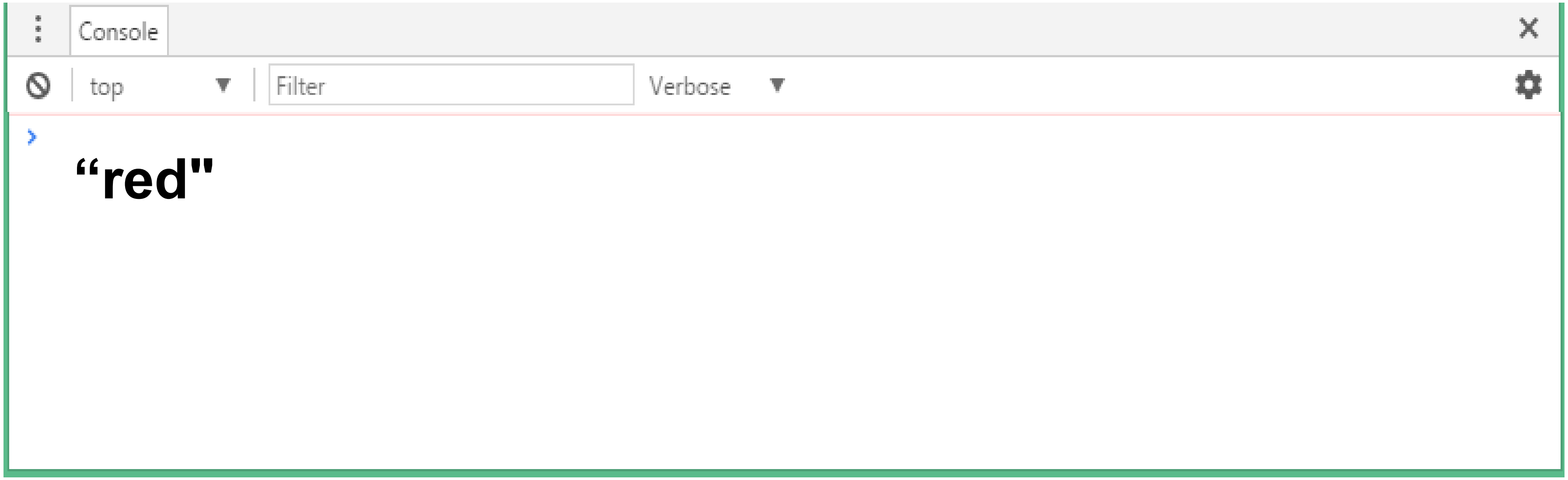


```
Current colours:  (5) ["white", "black", "green",
    "orange", "red"]
```

Methods … pop()

```
// Remove the last value of the array:
colours.pop();
```

Console Output:

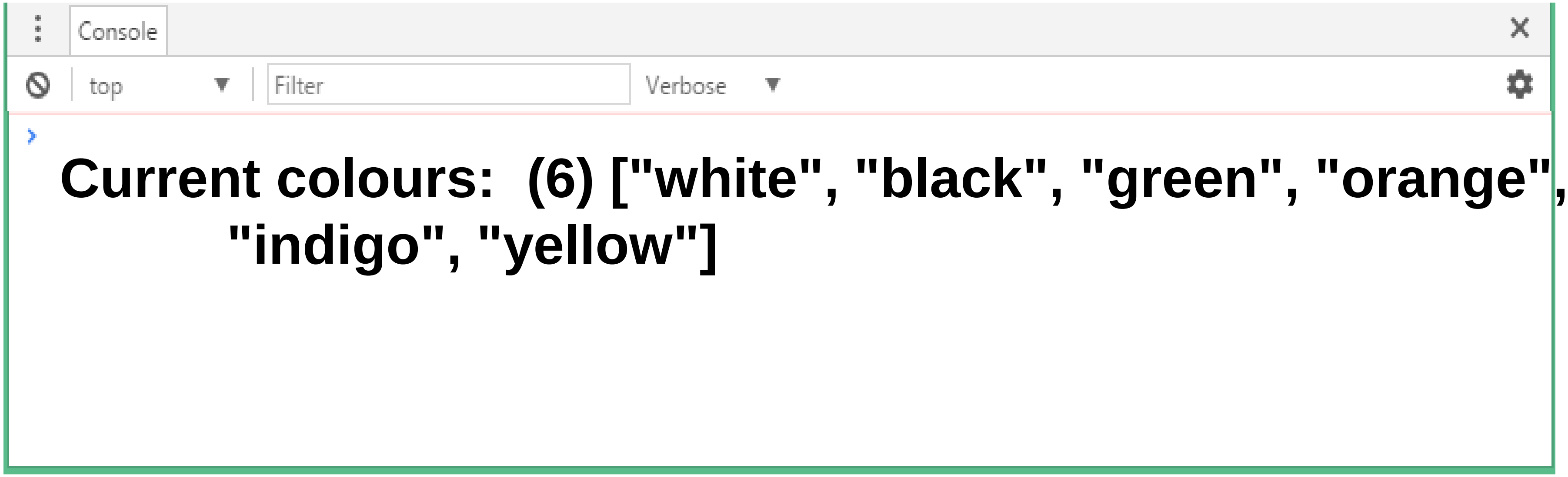


“red"

Methods ...push()

```
// Add comma-separated list to end of array
colours.push("indigo", "yellow");
// now display current list of colours
console.log("Current colours: ", colours);
```

Console Output:

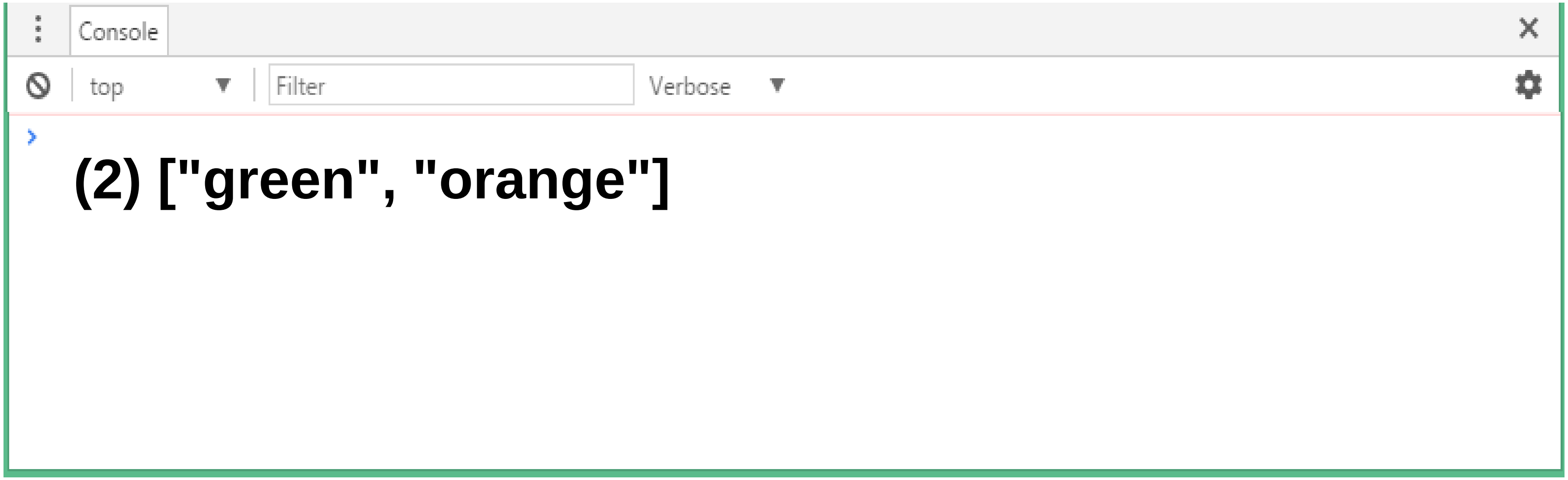


```
Current colours:  (6) ["white", "black", "green", "orange",
        "indigo", "yellow"]
```

Methods ... splice()

```
// Find the specified position (pos) and
// remove n number of items from the array.
// Args: colours.splice(pos,n)
// e.g. start at 2nd item and remove 2 items
colours.splice(2, 2);
```

Console Output:

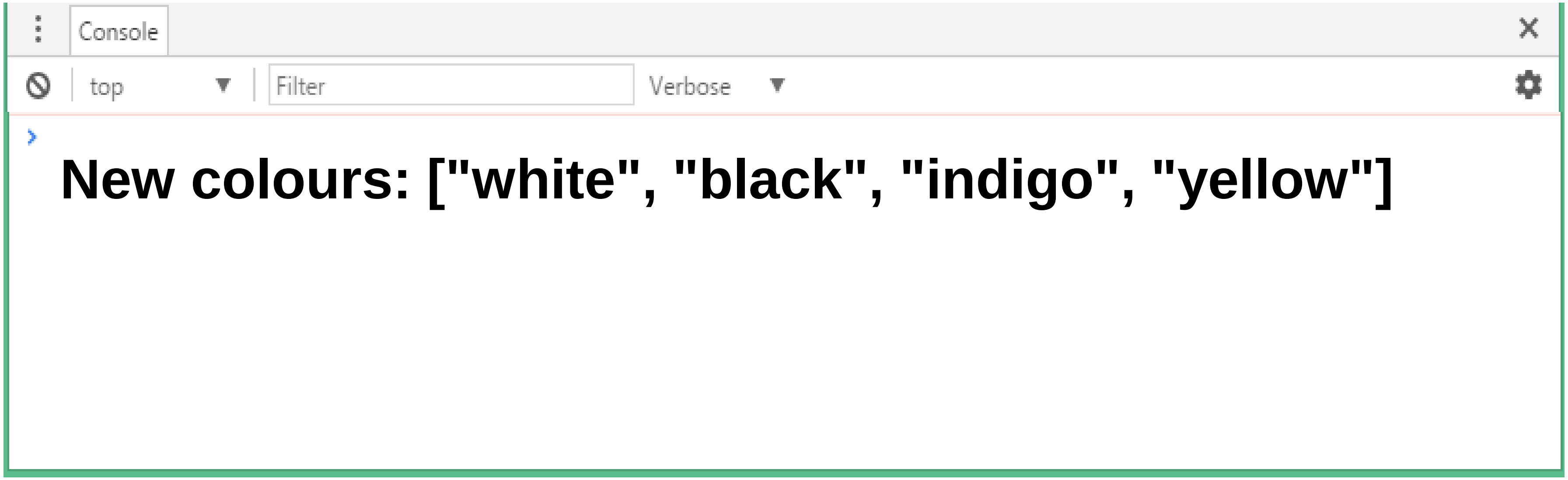


```
(2) ["green", "orange"]
```

Methods ...

```
// Create a copy of an array.
// Typically assigned to a new variable
var newColours = colours.slice();
console.log("New colours: ", newColours);
```

Console Output:



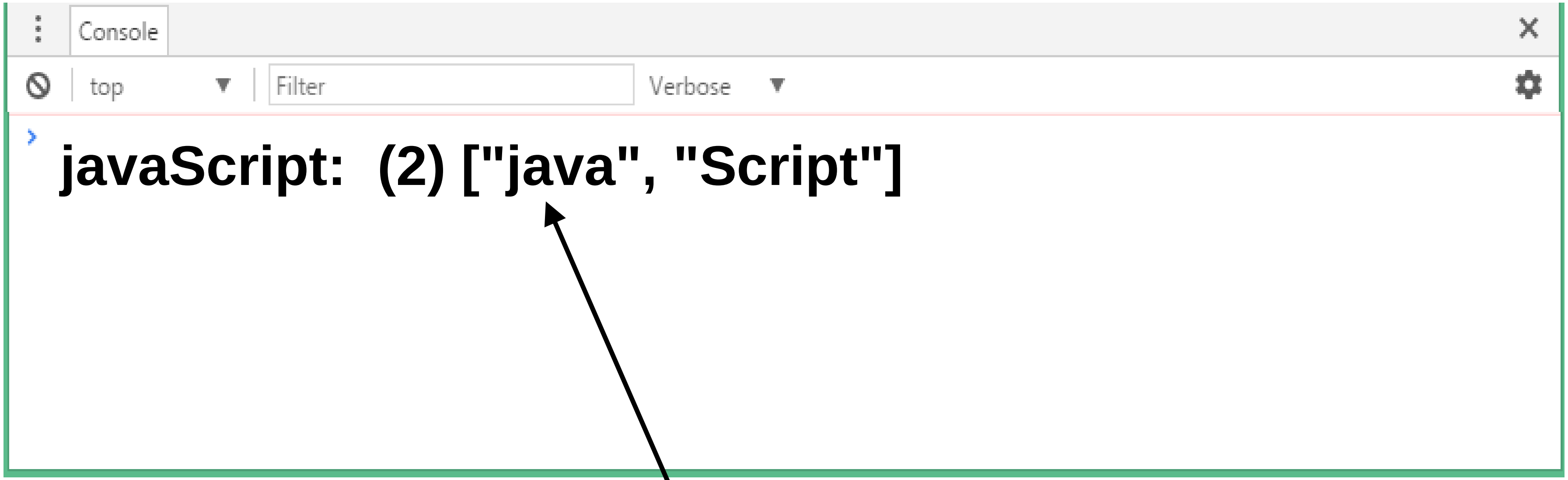
```
New colours: ["white", "black", "indigo", "yellow"]
```

CSWD

Methods ...concat()

```
// Alternative to create a copy of an array.
// use concat()
var java = ["java"]; script = ["Script"];
var javaScript = java.concat(script);
console.log("javaScript: ", javaScript);
```

Console Output:



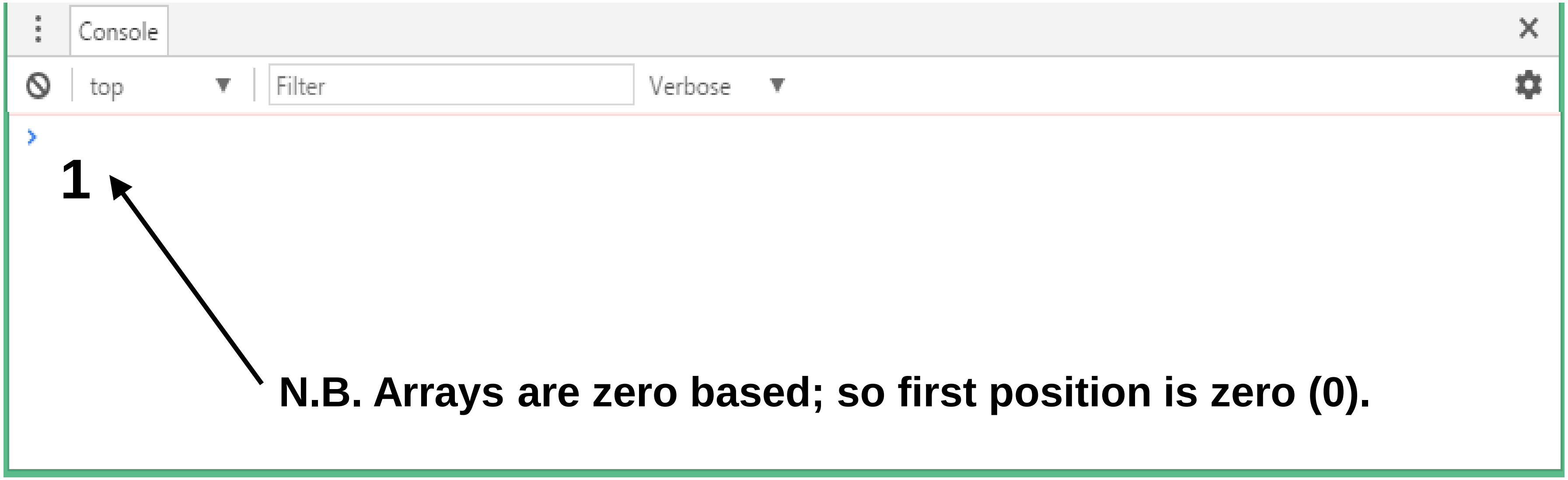
```
javaScript:  (2) ["java", "Script"]
```

N.B. New Array with 2 elements.

Methods … indexOf()

```
// Find position of first element that matches
// the search parameter after index position.
// Args: colours.indexOf(search, index)
// Returns -1 if no match found
colours.indexOf("black",0);
```

Console Output:



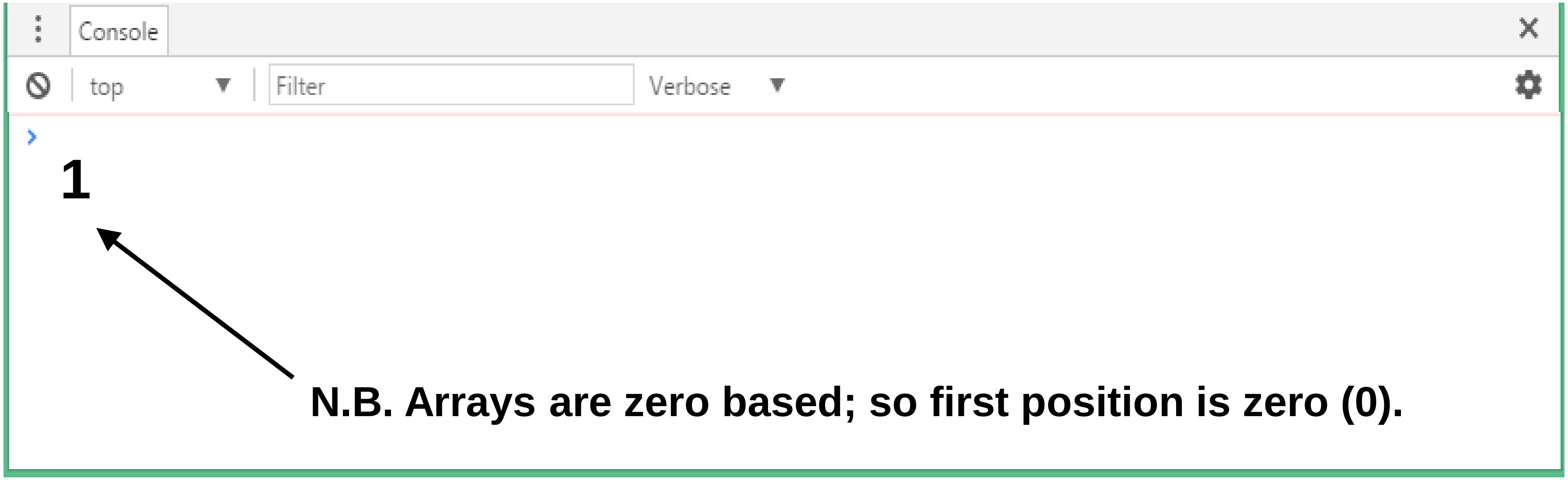
```
1
```

N.B. Arrays are zero based; so first position is zero (0).

Methods … lastIndexOf()

```
// Find position of last element that matches
// the search parameter after index position.
// Args: colours.lastIndexOf(search, index)
// Returns -1 if no match found
colours.lastIndexOf("black",0);
```

Console Output:



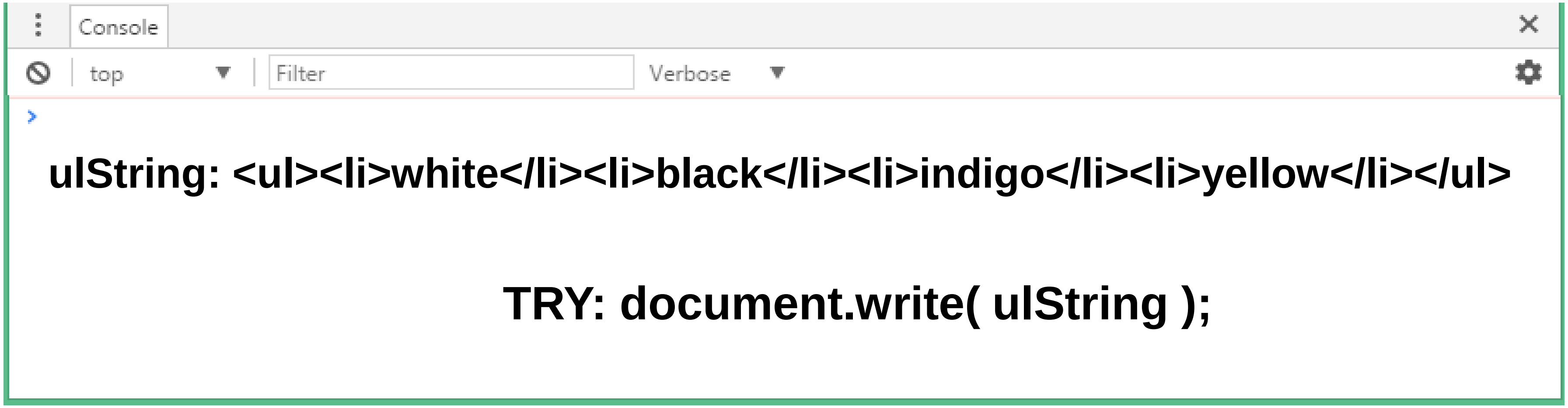
```
1
```

N.B. Arrays are zero based; so first position is zero (0).

Methods … join()

```
// Return elements of array as char separated
// string. Separator argument can be used to
// change comma.
// Args: colours.join(separator)
var ulString = '<ul><li>' +
    colours.join("</li><li>") + '</li></ul>';
console.log("ulString: ", ulString);
```

Console Output:



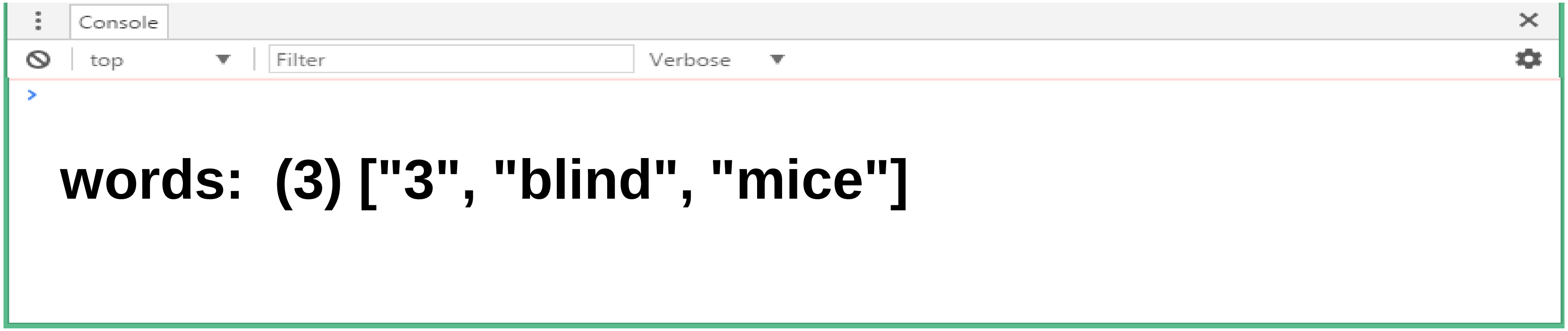
```
>
ulString: <ul><li>white</li><li>black</li><li>indigo</li><li>yellow</li></ul>
```

TRY: document.write(ulString);

Methods ... split()

```
// creates an array from a string.
// Splits using a separator argument
// Args: string.split(separator)
var arrayString = "3 blind mice";
var words = arrayString.split(" ");
console.log("words: ", words);
```

Console Output:



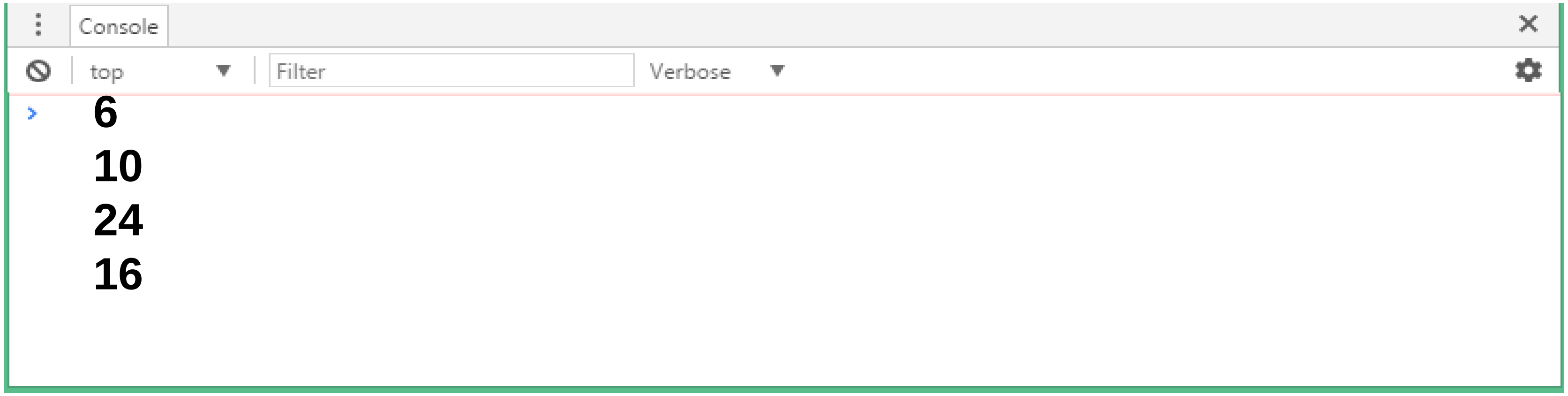
```
>
words:  (3) ["3", "blind", "mice"]
```

Q^2.
```
var jumbledWords = arrayString.split("i");
console.log("jumbledWords : ", jumbledWords );
```

Iterator Functions – forEach()

- forEach() takes a function as an argument and applies the called function to each element of an array.

```
eg function double( num ) {
         console.log( num * 2 );
   }
var numbers = [3, 5, 12, 8];
numbers.forEach( double );
```

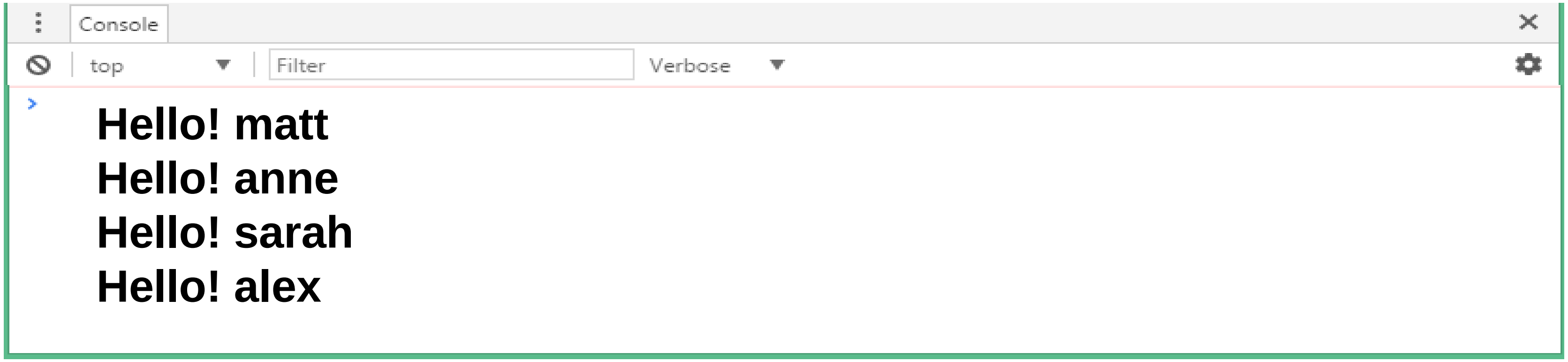


```
6
10
24
16
```

Iterator Functions – forEach()

- forEach() also works with strings.

```
eg function sayHello( name ) {
         if ( name != "" ) console.log( "Hello! " + name);
    }
var names = ["matt", "anne", "sarah", "", "alex"];
names.forEach(sayHello);
```



```
Hello! matt
Hello! anne
Hello! sarah
Hello! alex
```

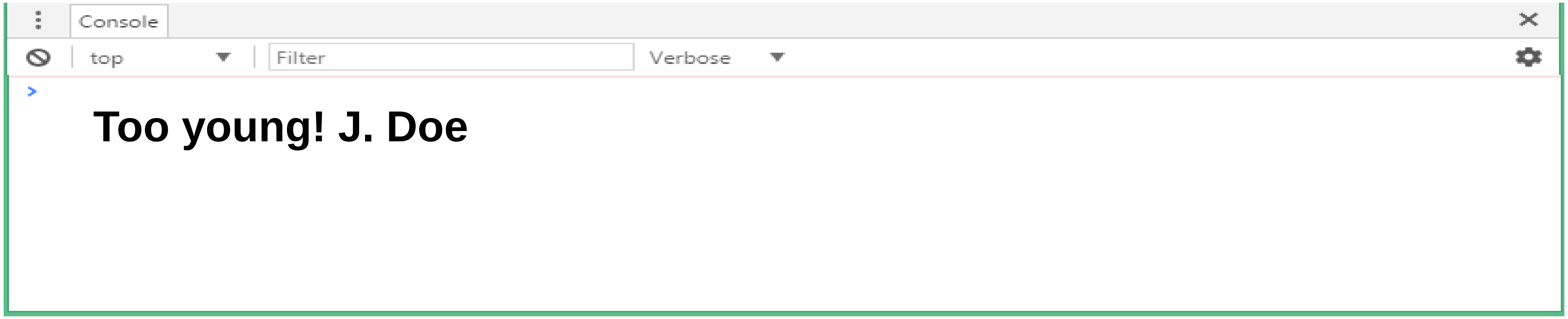
Q^2. Build an unordered list of the names using a forEach() iterator

Iterator Functions – forEach()

- forEach() also works with objects.

eg

```
function allow( person ) {
  if ( person.age < 18 )
      console.log( "Too young! " + person.name);
  }
var john = {}; john.age = 16; john.name = "J. Doe";
var anne = {}; anne.age = 18; anne.name = "A. Mate";
var people = [john, anne];
people.forEach(allow);
```

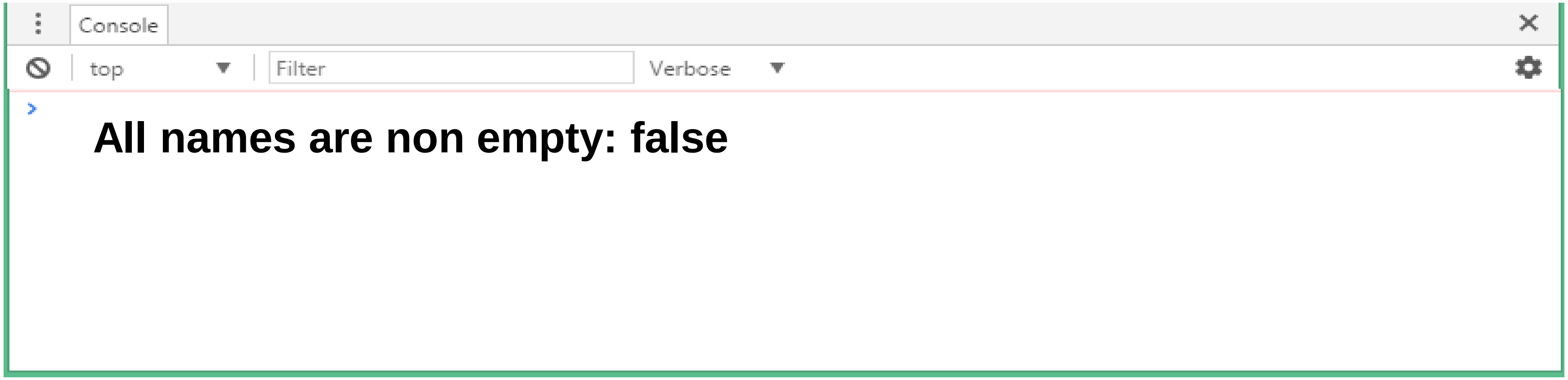


```
Too young! J. Doe
```

Iterator Functions – every()

- every() applies a Boolean function to an array and returns true if the function can return true for every element in the array.
- eg: `function nonEmpty( name ) {`

```
            return ( name != "" );
        }
var names = ["matt", "anne", "sarah", "", "alex"];
var validNames = names.every(nonEmpty);
console.log( "All names are non empty:", validNames );
```



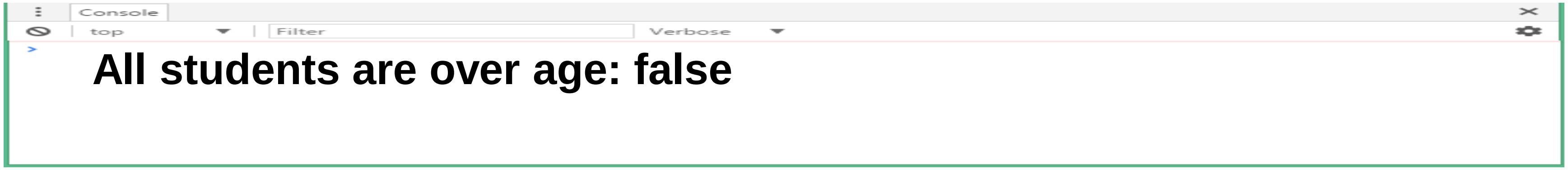
```
All names are non empty: false
```

Iterator Functions – some()

- some() applies a Boolean function to an array and returns true if the function can return true for every element in the array.

- eg:

```
function isUnder18( person ) {
                return ( person.age < 18 );
           }
var john = {}; john.age = 16; john.name = "J. Doe";
var anne = {}; anne.age = 18; anne.name = "A. Friend";
var students = [john, anne];
var overAge = ! students.some(isUnder18);
console.log( "All students are over age:", overAge );
```



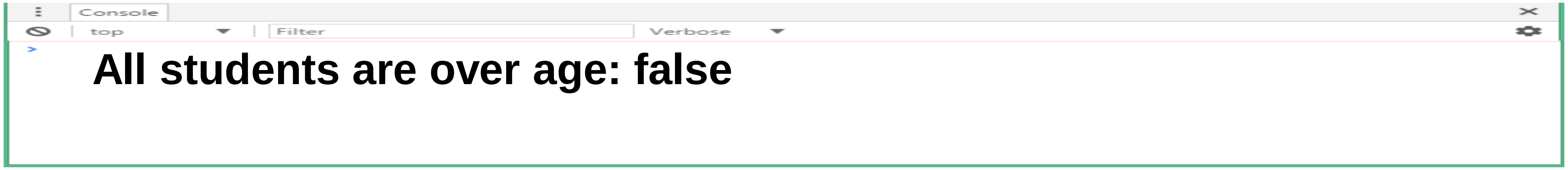
```
All students are over age: false
```

Iterator Functions – some()

- some() applies a Boolean function to an array and returns true if the function can return true for every element in the array.

- eg:

```
function isUnder18( person ) {
                return ( person.age < 18 );
            }
var myStudents = [];
myStudents.push({"name":"Peter", "age":17});
myStudents.push({"age":18, "name":"Mary"});
var overAge = ! myStudents.some(isUnder18);
console.log( "All students are over age:", overAge );
```

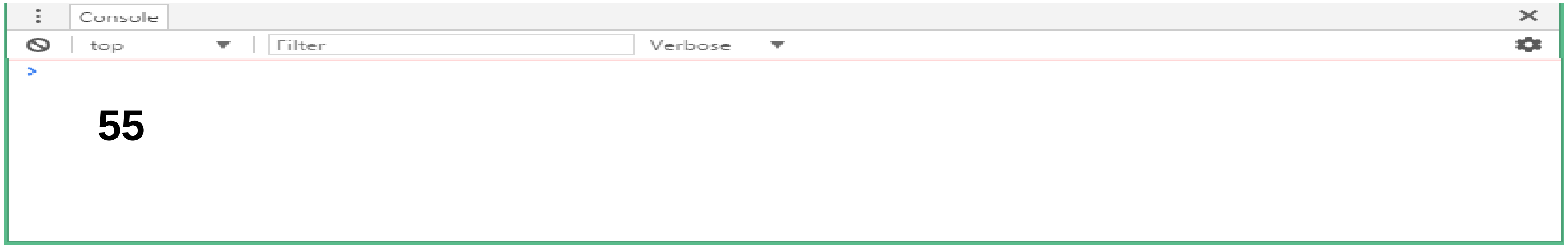


```
All students are over age: false
```

Iterator Functions – reduce()

- The reduce() function applies a function to an accumulator and the successive elements of an array until the end of the array is reached, yielding a single value.

```
function add(runningTotal, currentValue) {
  return runningTotal + currentValue;
}
var nums = [1,2,3,4,5,6,7,8,9,10];
var sum = nums.reduce(add);
console.log(sum);
```

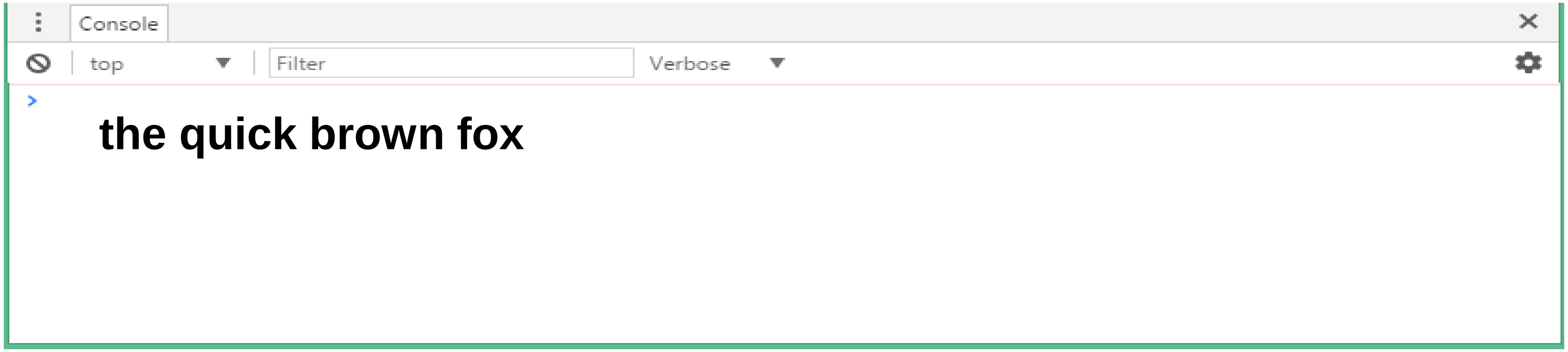


```
55
```

Iterator Functions cont …

- 2nd example using strings to perform concatenation:

```
function concat(accumulatedString, item) {
  return accumulatedString + item;
}
var words = ["the ", "quick ","brown ", "fox "];
var sentence = words.reduce(concat);
console.log( sentence );
```

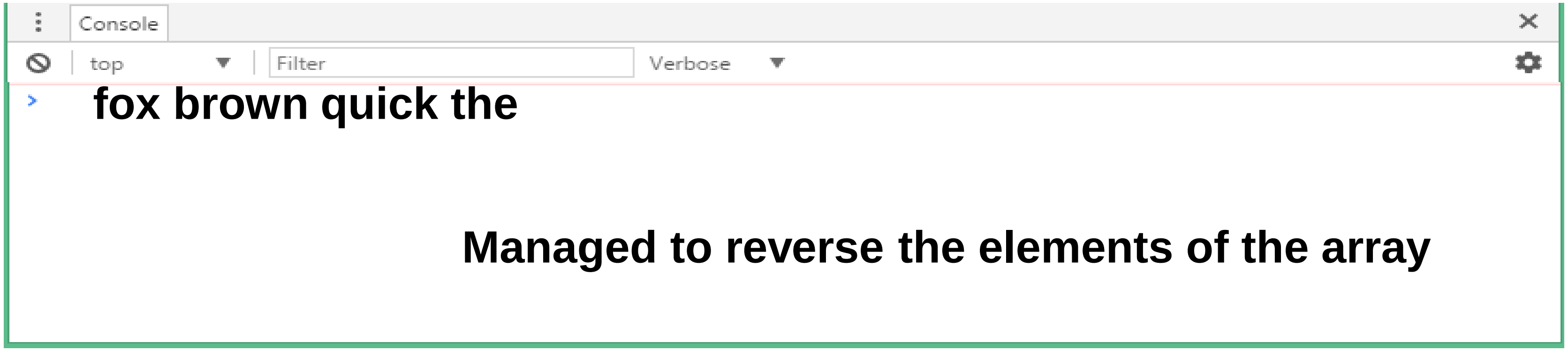


```
the quick brown fox
```

Iterator Functions – reduceRight()

- **JavaScript also provides a reduceRight() function, which works similarly to reduce(), only working from the righthand side of the array to the left,**

```
function concat(accumulatedString, item) {
  return accumulatedString + item;
}
var words = ["the ", "quick ","brown ", "fox "];
var sentence = words.reduceRight(concat);
console.log( sentence );
```



```
> fox brown quick the
```

Managed to reverse the elements of the array

Iterator Functions – map()

- There are two iterator functions that return new arrays: map() and filter().
- The map() function works like the forEach() function, applying a function to each element of an array.
  - The difference between the two functions is that map() returns a new array with the results of the function applied to each element of the array.

```
function upgrade( grade ) {
      return grade += 5;
}
var marks = [77, 65, 81, 42, 83];
var newMarks = marks.map(upgrade);
console.log(newMarks); // displays (5) [82,70,86,47,88]
```

Iterator Functions cont …

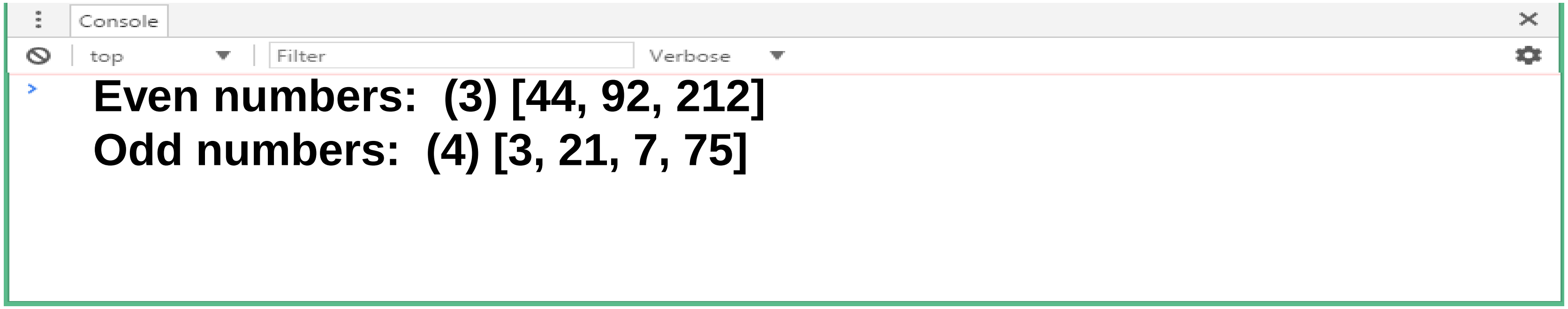
- 2nd example using strings:

```
function firstChar(word) {
  return word[0];
}

var words = ["for", "your", "information"];
var acronym = words.map(firstChar);
console.log(acronym.join("")); // "fyi"
```

Iterator Functions – filter()

- **The `filter()` function works similarly to `every()`, but instead of returning true if all the elements of an array satisfy a Boolean function, the function returns a new array consisting of those elements that satisfy the Boolean function.**

```
eg function isEven(num) { return num % 2 == 0; }
   function isOdd(num) { return num % 2 != 0; }
   var numbers = [3, 21, 44, 7, 92, 212, 75];
   var evenNumbers = numbers.filter(isEven);
   console.log("Even numbers: ", evenNumbers );
   var oddNumbers = numbers.filter(isOdd);
   console.log("Odd numbers: ", oddNumbers );
```

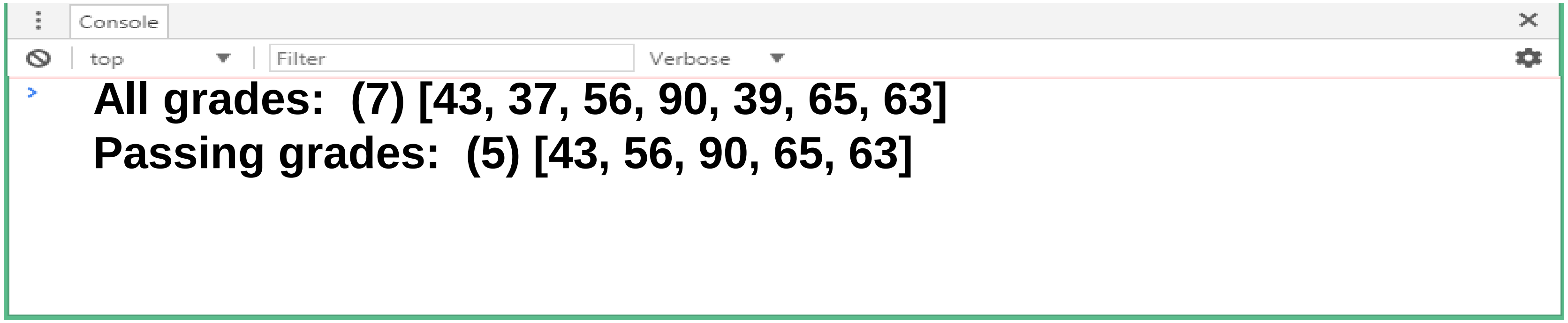


```
Even numbers:  (3) [44, 92, 212]
Odd numbers:  (4) [3, 21, 7, 75]
```

Iterator Functions cont …

- Here’s an interesting use of filter():

```
function passing(num) {
  return num >= 40;
}
var marks = [43, 37, 56, 90, 39, 65, 63];
var passMarks = marks.filter(passing);
console.log("All grades: ", marks);
console.log("Passing grades: ", passMarks);
```

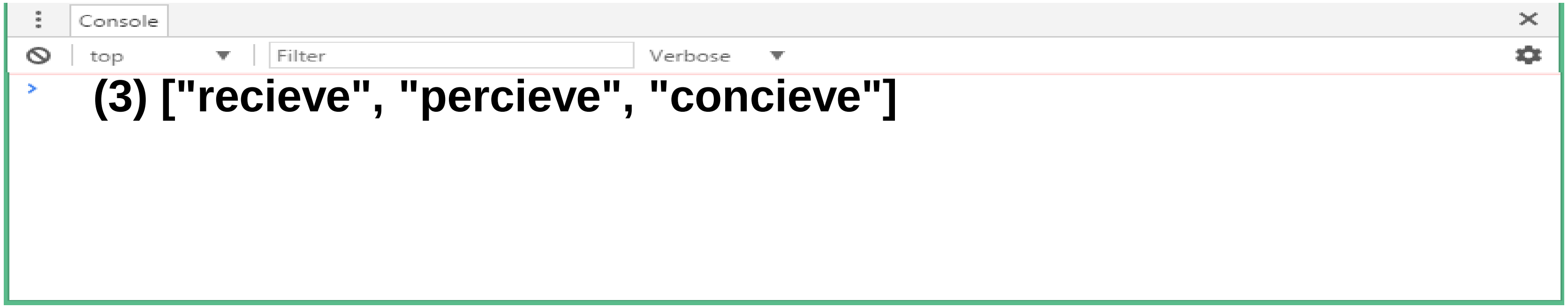


```
All grades:  (7) [43, 37, 56, 90, 39, 65, 63]
Passing grades:  (5) [43, 56, 90, 65, 63]
```

Iterator Functions cont …

- Of course, we can also use filter() with strings.
- This example finds misspelled words failing the spelling rule "i before e except after c":

```
function afterC(str) {
  return (str.indexOf("cie") > -1);
}
var words =
  ["recieve","deceive","percieve","deceit","concieve"];
var misspelled = words.filter(afterC);
console.log(misspelled);
```



```
(3) ["recieve", "percieve", "concieve"]
```

Arrays of Objects

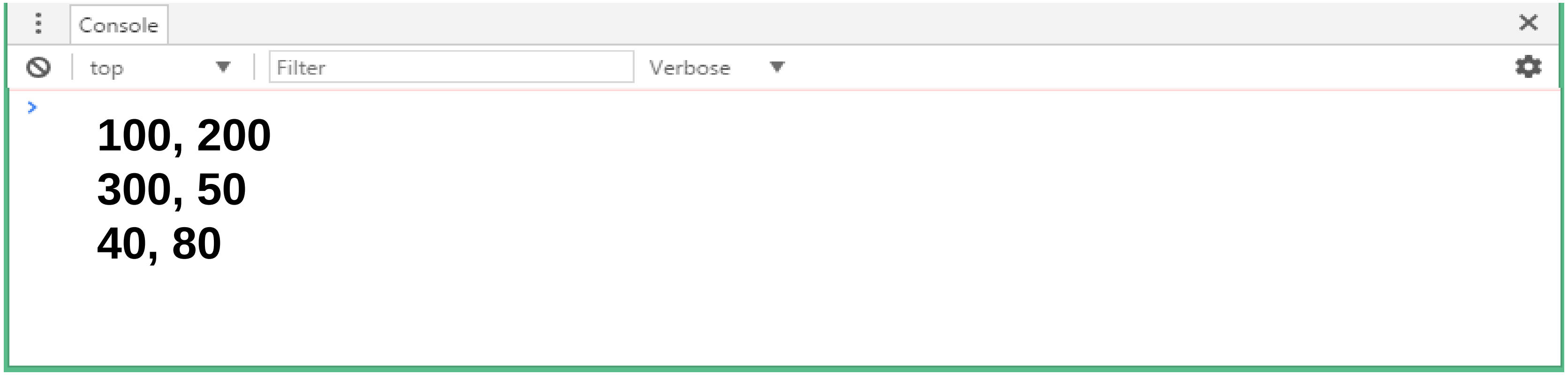
- **Arrays can also consist of objects, and all the functions and properties of arrays work with objects.**

```
function Point(x,y) {
  this.x = x;
  this.y = y;
}
function displayPoints(ra) {
  var i = 0;
  for (i= 0; i < ra.length; ++i) {
    console.log(ra[i].x + ", " + ra[i].y);
  }
}
```

Q^2. Rewrite displayPoints using ra.forEach()

Arrays of Objects cont …

```
var p1 = new Point(100, 200);
var p2 = new Point(300, 50);
var p3 = new Point(40, 80);
var points = [p1,p2,p3];  // array of objects
displayPoints( points );
```

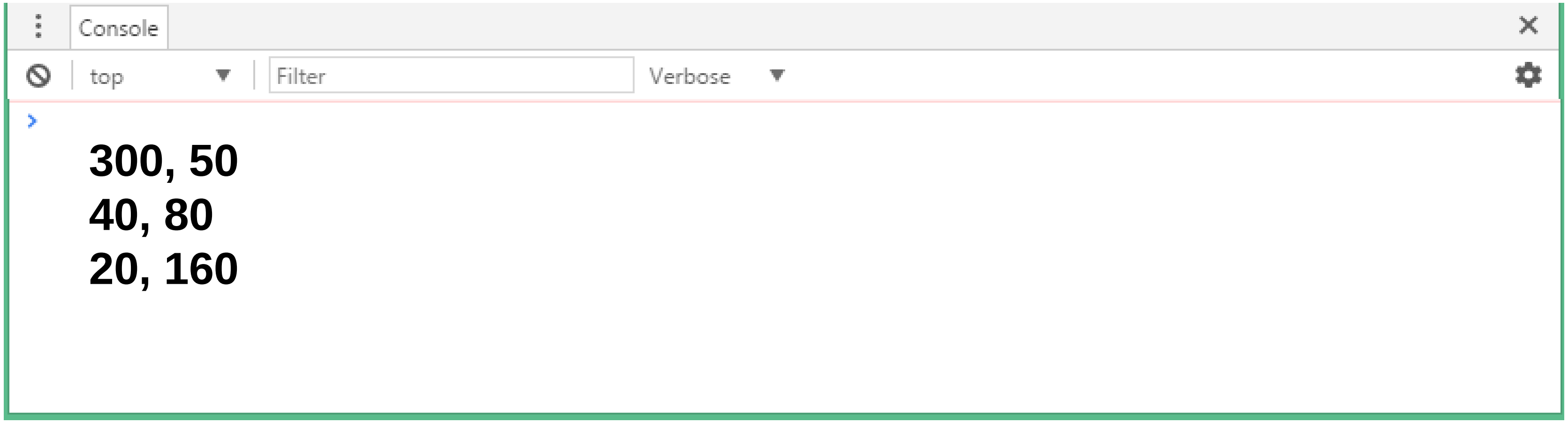


```
100, 200
300, 50
40, 80
```

Arrays of Objects cont …

- Add/remove more data points to array

```
var p4 = new Point(20, 160);
points.push(p4);    // add new point to end of array
points.shift();     // remove first point from array
displayPoints( points );
```



```
300, 50
40, 80
20, 160
```

Arrays in Objects

- **Objects can also have arrays!**

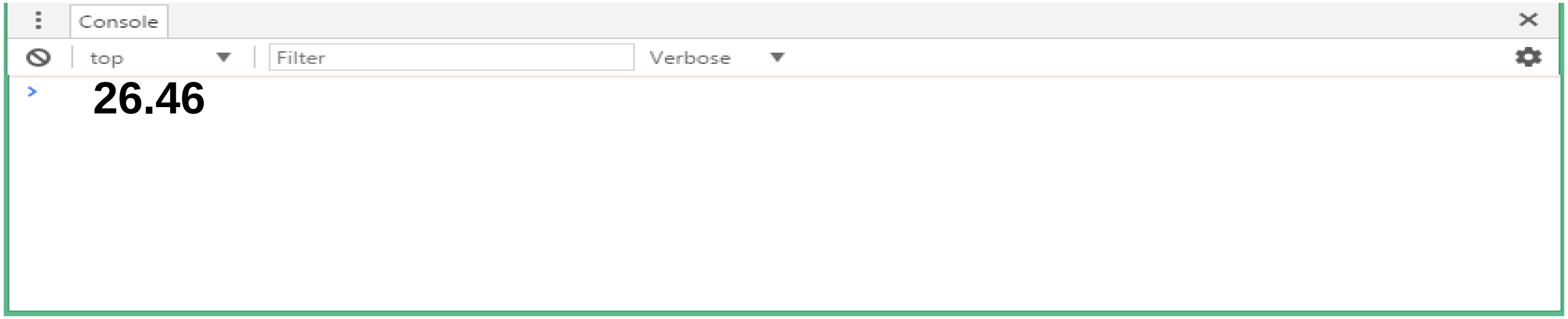
```
function BMIValues() {
  this.dataStore = [];        // here's our empty array
  this.add = add;             // define two methods
  this.average = average;
}
function add(bmiValue) {
  this.dataStore.push(bmiValue);   // add value to end of array
}
function average() {
  var total = 0; var i = 0;
  for (i = 0; i < this.dataStore.length; i++) {
      total += this.dataStore[i];
  }
  return total / this.dataStore.length;
}
```

Rewrite using reduce()

Q^2. Where's the potential error in this code?

Arrays in Objects cont …

```
var bmis = new BMIValues();
bmis.add(25.6);
bmis.add(31.2);
bmis.add(21.5);
bmis.add(25.5);
bmis.add(28.5);
console.log(bmis.average());
```



```
> 26.46
```

Summary - Function usage examples:

- The term “first-class” means that something is just a value.
  - A first-class function is one that can go anywhere that any other value can go—there are few to no restrictions.
  - A number in JavaScript is surely a first-class thing, and therefore a first-class function has a similar nature:

1) eg A number can be stored in a variable and so can a function:

```
var fortytwo = function() { return 42 };
```

2) eg A number can be stored in an array slot and so can a function:

```
var fortytwos = [ 42, function() { return 42 } ];
```

3) eg A number can be stored in an object field and so can a function:

```
var fortytwos = {number:42, fun:function(){return 42}};
```

4) eg A number can be created as needed and so can a function:

```
42 + (function() { return 42 })(); // => 84
```

5) eg A number can be passed to a function and so can a function:

```
function weirdAdd(n, f) { return n + f() }
weirdAdd(42, function() { return 42 }); // => 84
```

6) eg A number can be returned from a function and so can a function:

```
return 42;
return function() { return 42 };
```

Summary

- **Covered overview of JavaScript:**
  - **Arrays**
  - **Functions**
  - **Objects**
- **Next week:**
  - **Possibly take a look at JavaScript Patterns**