Programming Language Explorations

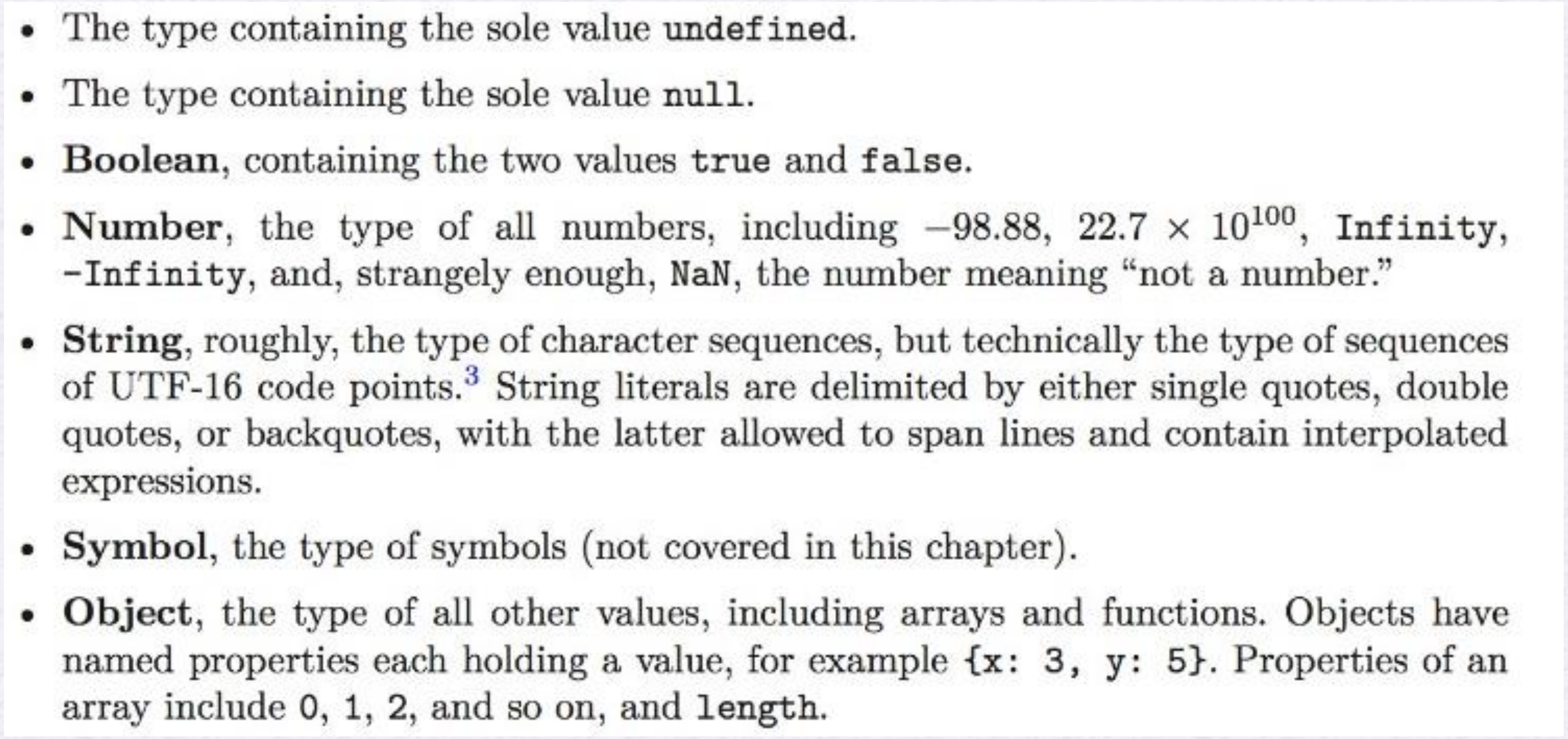
Chapter 1: JavaScript

JavaScript Data Types

Value Types
VS
Reference Types

JavaScript data types

- Values have one of exactly 7 types



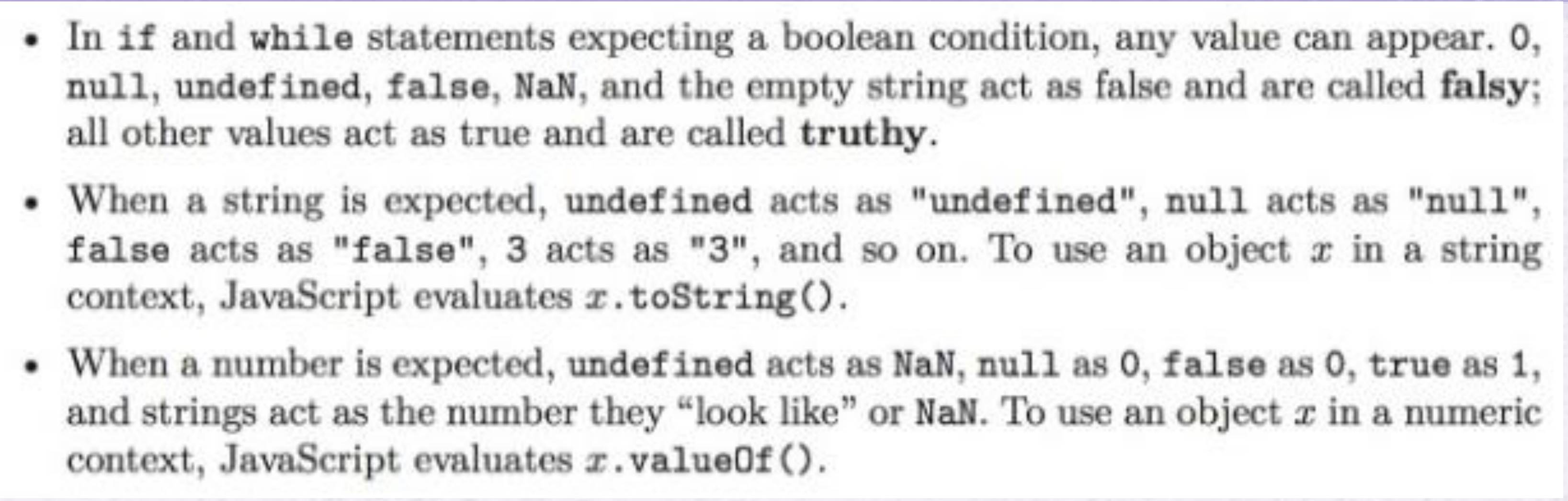
- The type containing the sole value `undefined`.
- The type containing the sole value `null`.
- **Boolean**, containing the two values `true` and `false`.
- **Number**, the type of all numbers, including -98.88, 22.7×10^{100}, `Infinity`, `-Infinity`, and, strangely enough, `NaN`, the number meaning "not a number."
- **String**, roughly, the type of character sequences, but technically the type of sequences of UTF-16 code points.[3] String literals are delimited by either single quotes, double quotes, or backquotes, with the latter allowed to span lines and contain interpolated expressions.
- **Symbol**, the type of symbols (not covered in this chapter).
- **Object**, the type of all other values, including arrays and functions. Objects have named properties each holding a value, for example `{x: 3, y: 5}`. Properties of an array include `0`, `1`, `2`, and so on, and `length`.

- The first six types are **primitive types**; Object is a **reference type**.

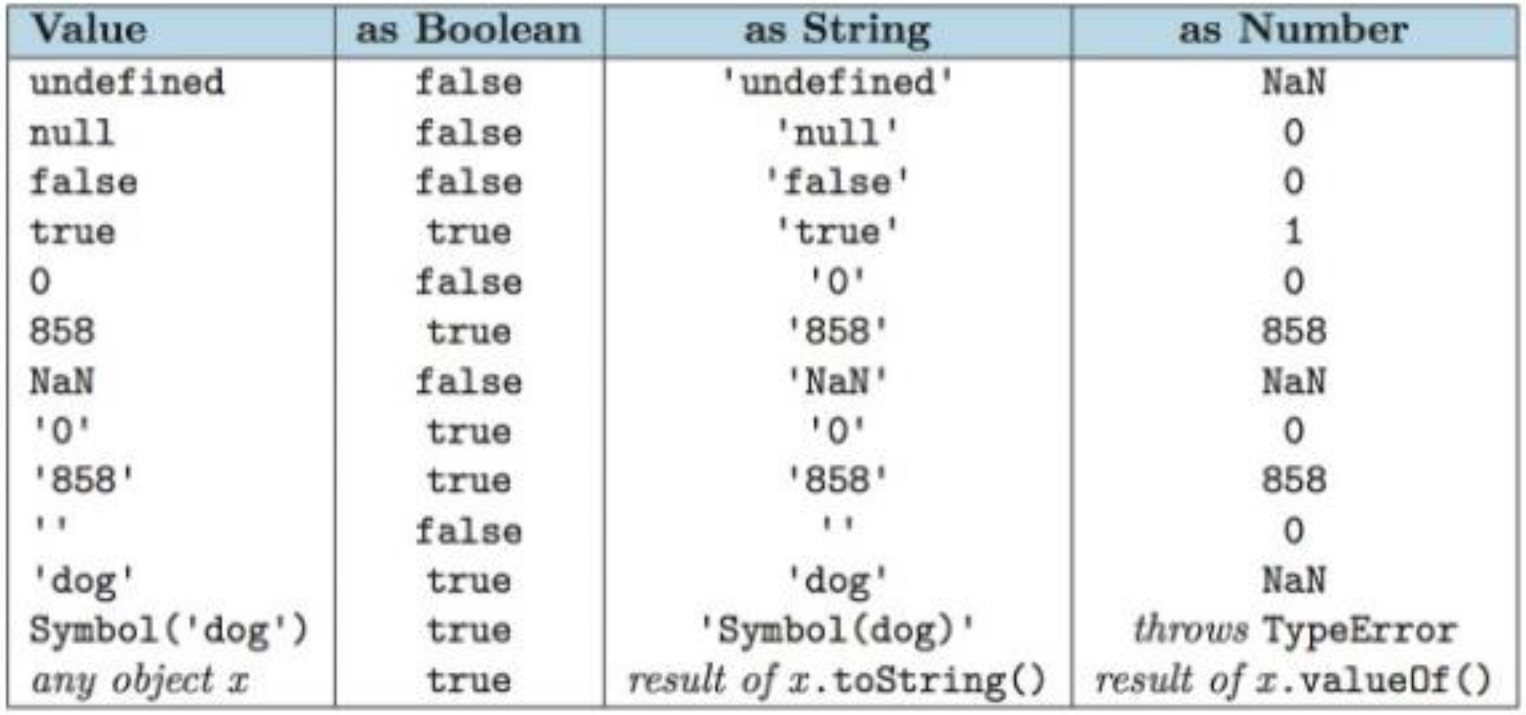
JavaScript is **dynamic** & **weakly-typed**

- Strongly typed vs Weakly typed
  - Strongly typed languages don't permit <u>many</u> implicit type conversions, whereas weakly typed languages do.
- Static typing vs Dynamic typing
  - Static: Types checked before run-time.
  - Dynamic: Types checked on the fly, during execution.

JavaScript implicit type conversions (coercions)



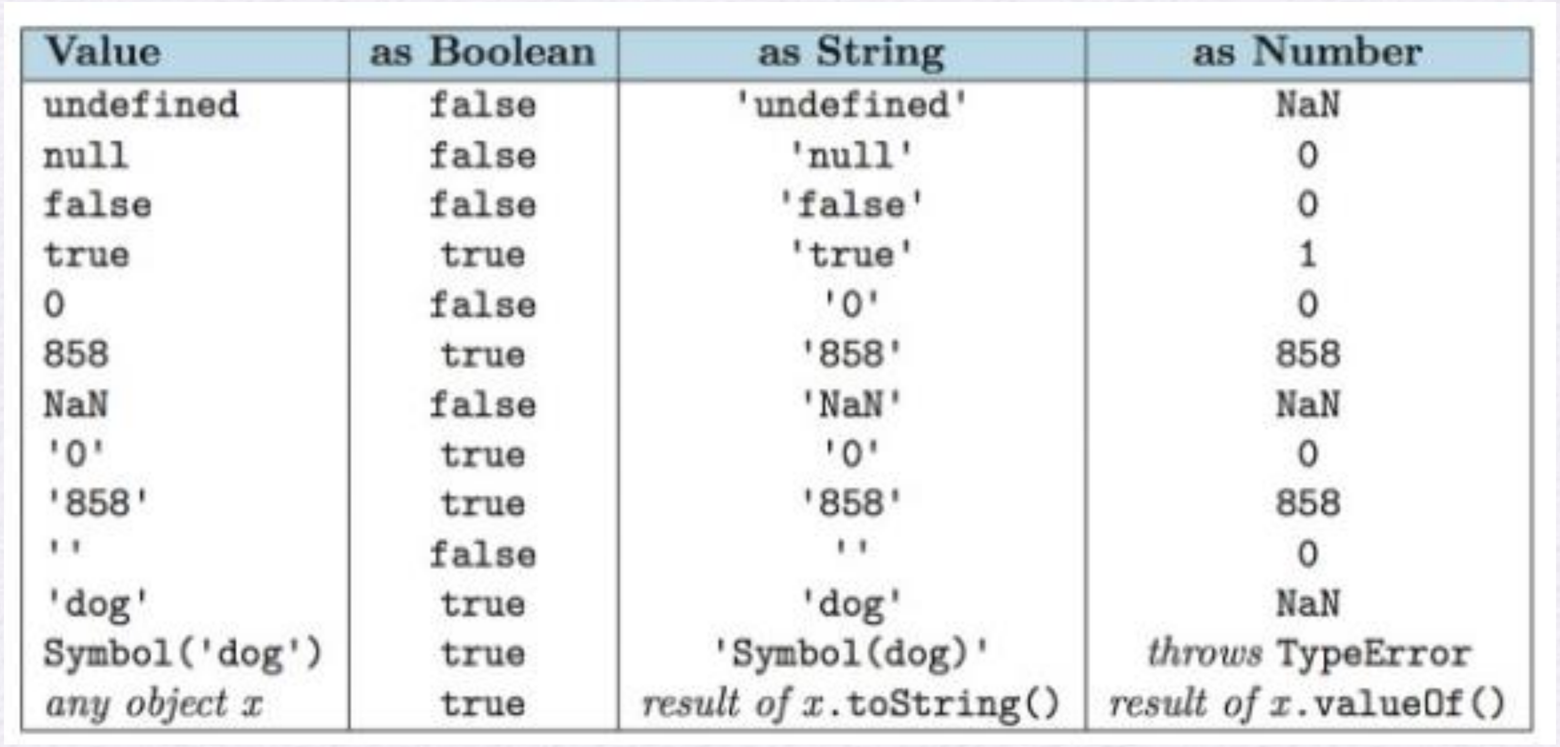
- In `if` and `while` statements expecting a boolean condition, any value can appear. 0, `null`, `undefined`, `false`, `NaN`, and the empty string act as false and are called **falsy**; all other values act as true and are called **truthy**.
- When a string is expected, `undefined` acts as `"undefined"`, `null` acts as `"null"`, `false` acts as `"false"`, 3 acts as `"3"`, and so on. To use an object *x* in a string context, JavaScript evaluates *x*`.toString()`.
- When a number is expected, `undefined` acts as `NaN`, `null` as 0, `false` as 0, `true` as 1, and strings act as the number they "look like" or `NaN`. To use an object *x* in a numeric context, JavaScript evaluates *x*`.valueOf()`.



Value	as Boolean	as String	as Number
`undefined`	`false`	`'undefined'`	`NaN`
`null`	`false`	`'null'`	`0`
`false`	`false`	`'false'`	`0`
`true`	`true`	`'true'`	`1`
`0`	`false`	`'0'`	`0`
`858`	`true`	`'858'`	`858`
`NaN`	`false`	`'NaN'`	`NaN`
`'0'`	`true`	`'0'`	`0`
`'858'`	`true`	`'858'`	`858`
`''`	`false`	`''`	`0`
`'dog'`	`true`	`'dog'`	`NaN`
`Symbol('dog')`	`true`	`'Symbol(dog)'`	*throws* `TypeError`
any object x	`true`	*result of x*`.toString()`	*result of x*`.valueOf()`

JavaScript implicit type conversions (coercions)

A value that is false or would be converted to false is called **falsy.** All other values are **truthy**.



Value	as Boolean	as String	as Number
undefined	false	'undefined'	NaN
null	false	'null'	0
false	false	'false'	0
true	true	'true'	1
0	false	'0'	0
858	true	'858'	858
NaN	false	'NaN'	NaN
'0'	true	'0'	0
'858'	true	'858'	858
''	false	''	0
'dog'	true	'dog'	NaN
Symbol('dog')	true	'Symbol(dog)'	*throws* TypeError
any object x	true	*result of x*.toString()	*result of x*.valueOf()

***Exercise**: For each of the following values, state whether they are truthy or falsy:* 9.3*,* 0*,* [0]*,* false*,* true*,* ""*,* "${"}"*,* \`${"}\`*,* []*,* [[]]*,* {}*.*

JavaScript objects

- Javascript objects are variables that can contain many values.

- This code assigns many values (Fiat, 500, white) to a variable named car:

```
var car = {type:"Fiat", model:"500", color:"white"};
```

  - The values are written as **property:value** pairs in a comma-delimited list inside curly braces

```
console.log(car.model);

console.log(car["model"]);
```

2 different ways to access values in an object.

Adding *property, value* pairs to an existing object

- To create an empty object use:

```
car = {};
```

- To add property, value pairs to an existing object use:

```
car["type"] = "Hyundai";
car["model"] = "Sonata";
console.log(car);
```

↳

```
{ type:'Hyundai', model:'Sonata' }
```

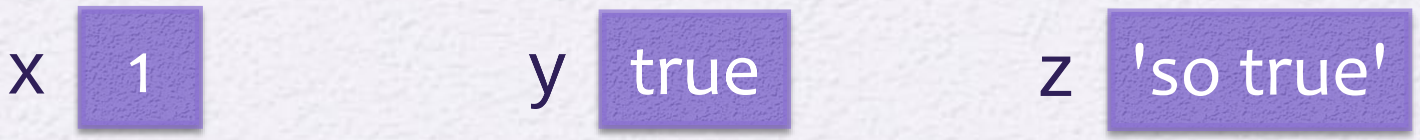
```
car["color"] = "blue";
console.log(car);
```

↳

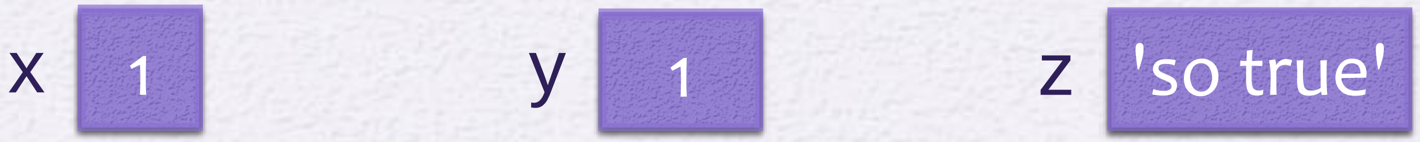
```
{ type:'Hyundai', model:'Sonata', color:'blue' }
```

JavaScript primitive types
(memory content is **a value**)

- let x = 1;   let y = true;   let z = 'so ' + y



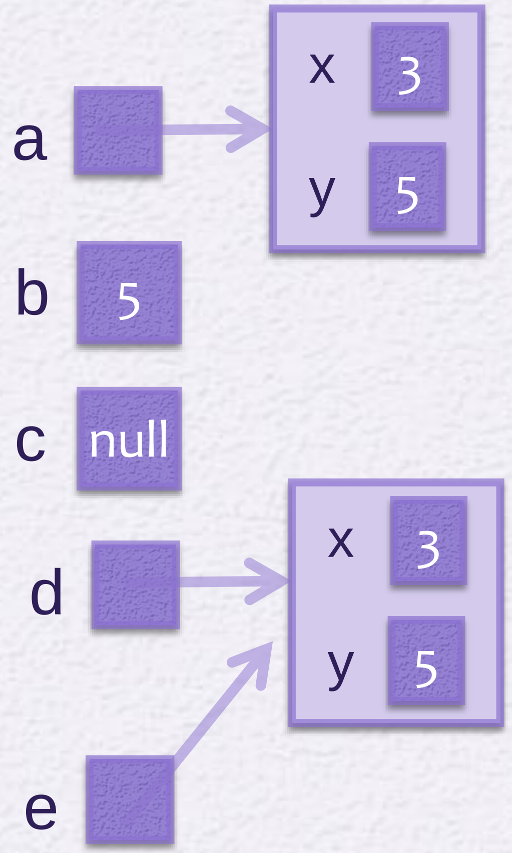
- y = x



- Values of a **primitive type** are written directly inside the variable boxes.

JavaScript objects
(memory content is **a reference**)

- const a = {x:3, y:5};
- const b = a.y;
- const c = null;
- const d = {x:3, y:5};
- const e = d;



- Objects are **reference type**
- Values of a **reference type** are actually references to entities holding the object properties.

JavaScript arrays

- Creating an array:
```
        var cars = ["Saab", "Volvo", "BMW"];
                    or
        var cars = new Array("Saab", "Volvo", "BMW");
```

- Access the Elements of an Array
```
var name = cars[0];
```

- Changing an Array Element
```
cars[0] = "Opel";
console.log(cars[0]);     //will print “Opel”
```

- Access the Full Array
```
        var cars = ["Saab", "Volvo", "BMW"];
        console.log(cars);      //will print the whole array
                                //['Saab', 'Volvo', 'BMW']
```

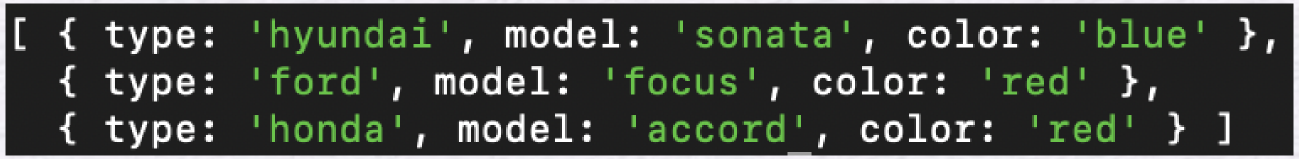
Adding values to an existing array

- Create empty array:

  ```
  var cars = [];
  ```

- Add items (objects) to the array, one at a time:

```
cars[0] = {"type": "hyundai", "model": "sonata", "color": "blue"};
cars[1] = {"type": "ford", "model": "focus", "color": "red"};
cars[2] = {"type": "honda", "model": "accord", "color": "red"};
console.log(cars);
```



```
[ { type: 'hyundai', model: 'sonata', color: 'blue' },
  { type: 'ford', model: 'focus', color: 'red' },
  { type: 'honda', model: 'accord', color: 'red' } ]
```

- QUESTION: How do I access the color of the first car in the array?

  ```
  cars[0].color
  cars[0]["color"]
  ```

Arrays VS Objects

- Arrays use numbers to access its "elements". In this example, *person[0]* returns John:

```
var person = ["John", "Doe", 46];
console.log(person[0]);          //will print “John”
```

  - *Notice how you can have variables of different types in the same Array*

- Objects use **names** to access its "members". In this example, *person.firstName* returns John:

```
var person = {firstName:"John", lastName:"Doe", age:46};
console.log(person["firstName"]);  //will print “John”
console.log(person.firstName);  //will print “John”
```

2 different ways to access firstname.

Arrays & Objects

- Array Elements Can Be Objects

```
var people = [ {firstName:"Jay-Z", lastName:"Carter", age:48},        → First object
          {firstName:"Eminem", lastName:"Mathers", age:45},            → Second object
          {firstName:"Drake", lastName:"Graham", age:35} ]             → Third object
```

```
console.log(people[0]);
        { firstName: 'Jay-Z', lastName: 'Carter', age: 48 }
console.log(people[0].firstName);
        'Jay-Z'
console.log(people[0]["firstName"]);
        'Jay-Z'
```

Arrays & Objects

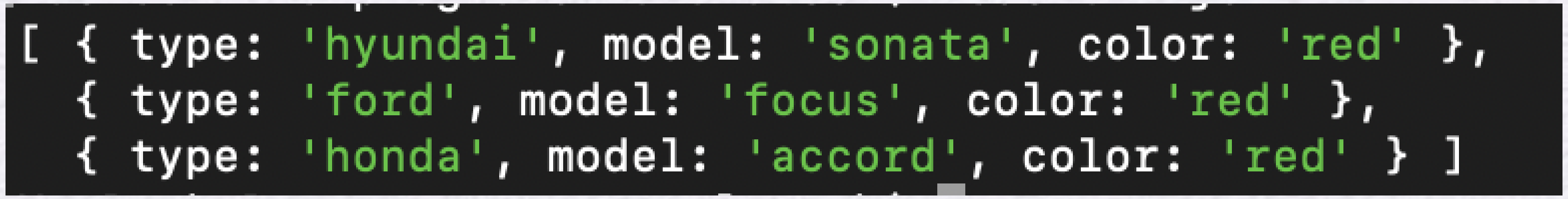
```
car1 = {"type": "hyundai", "model": "sonata", "color": "red"};

car2 = {"type": "ford", "model": "focus", "color": "red"};

car3 = {"type": "honda", "model": "accord", "color": "red"};


var cars = [car1, car2, car3];

console.log(cars);
```

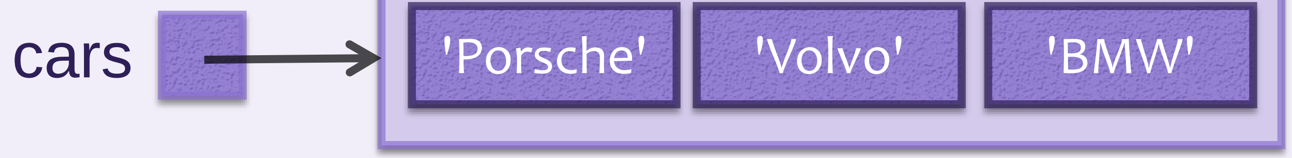


```
[ { type: 'hyundai', model: 'sonata', color: 'red' },
  { type: 'ford', model: 'focus', color: 'red' },
  { type: 'honda', model: 'accord', color: 'red' } ]
```

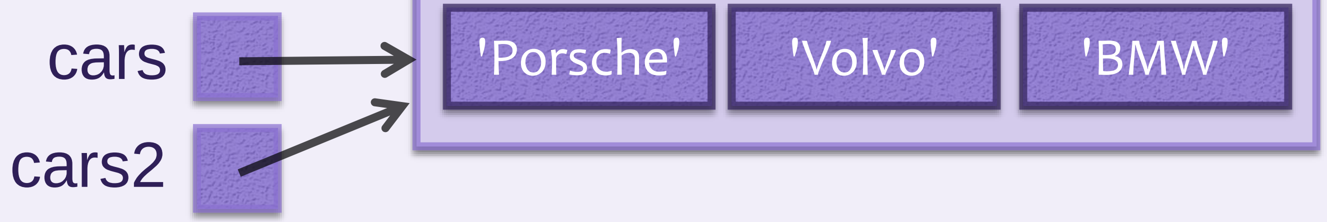
Arrays are also **reference type**

- Arrays are considered objects and therefore are **reference type**
- Copying an array to a variable simply <u>copies the reference</u>.

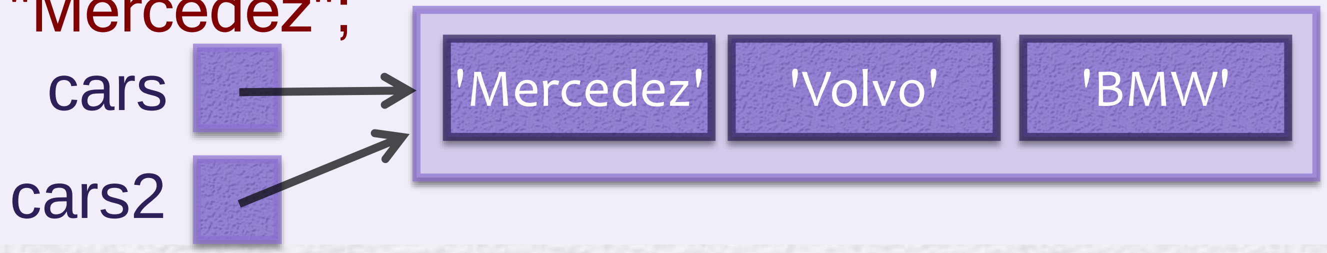
```
var cars = ["Porsche", "Volvo", "BMW"];
```



```
var cars2 = cars;
```



```
cars2[0] = "Mercedez";
```

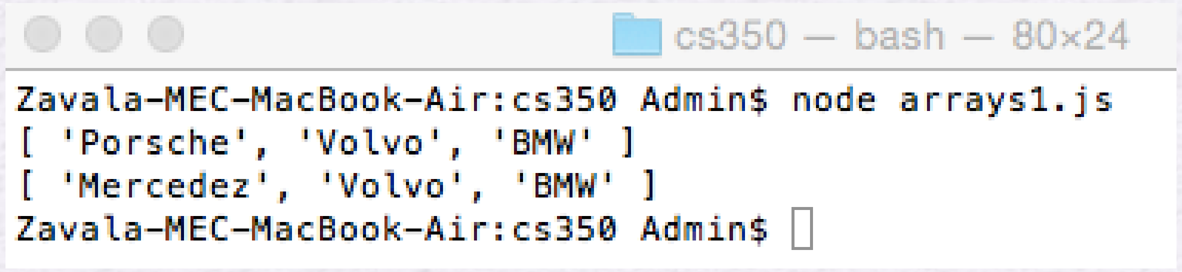


Arrays are also **reference type**

- Arrays are considered objects and therefore are **reference type**
- Copying an array to a variable simply <u>copies the reference</u>.

```
var cars = ["Porsche", "Volvo",
console.log(cars);

var cars2 = cars;
cars2[0] = "Mercedez";
console.log(cars);
```

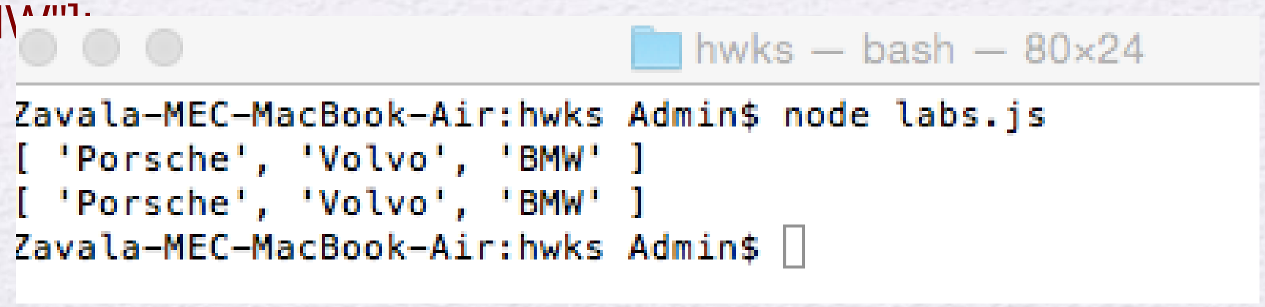


```
Zavala-MEC-MacBook-Air:cs350 Admin$ node arrays1.js
[ 'Porsche', 'Volvo', 'BMW' ]
[ 'Mercedez', 'Volvo', 'BMW' ]
Zavala-MEC-MacBook-Air:cs350 Admin$ 
```

- **slice** can be used to make a copy of the <u>values</u> in the array:

```
var cars = ["Porsche", "Volvo", "BMW"];
console.log(cars);

var cars2 = cars.slice();
cars2[0] = "Mercedez";
console.log(cars);
```

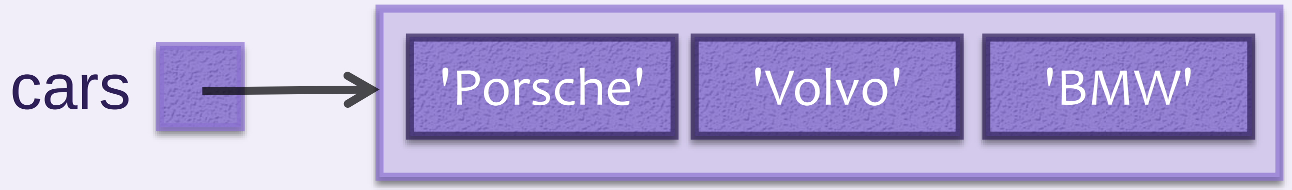


```
Zavala-MEC-MacBook-Air:hwks Admin$ node labs.js
[ 'Porsche', 'Volvo', 'BMW' ]
[ 'Porsche', 'Volvo', 'BMW' ]
Zavala-MEC-MacBook-Air:hwks Admin$ 
```

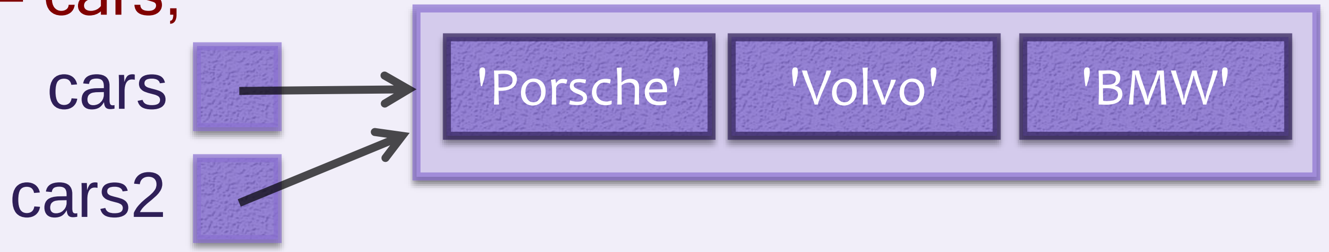
Slice makes a copy of the values in the array

- Arrays are considered objects and therefore are **reference type**
- Copying an array to a variable simply copies the reference.

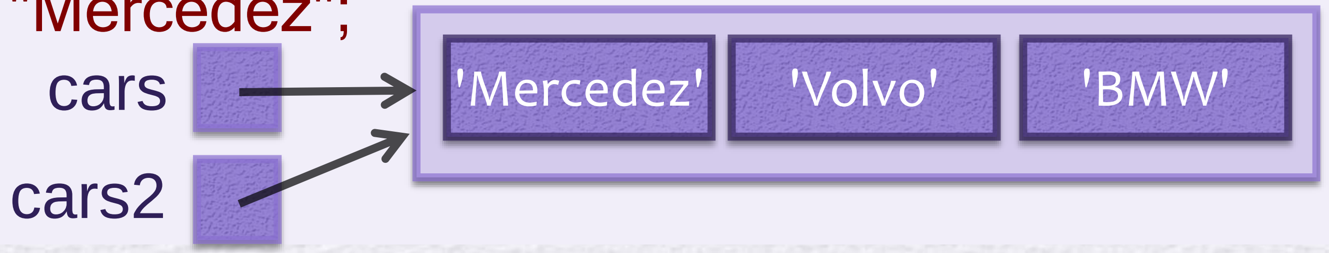
```
var cars = ["Porsche", "Volvo", "BMW"];
```



```
var cars2 = cars;
```

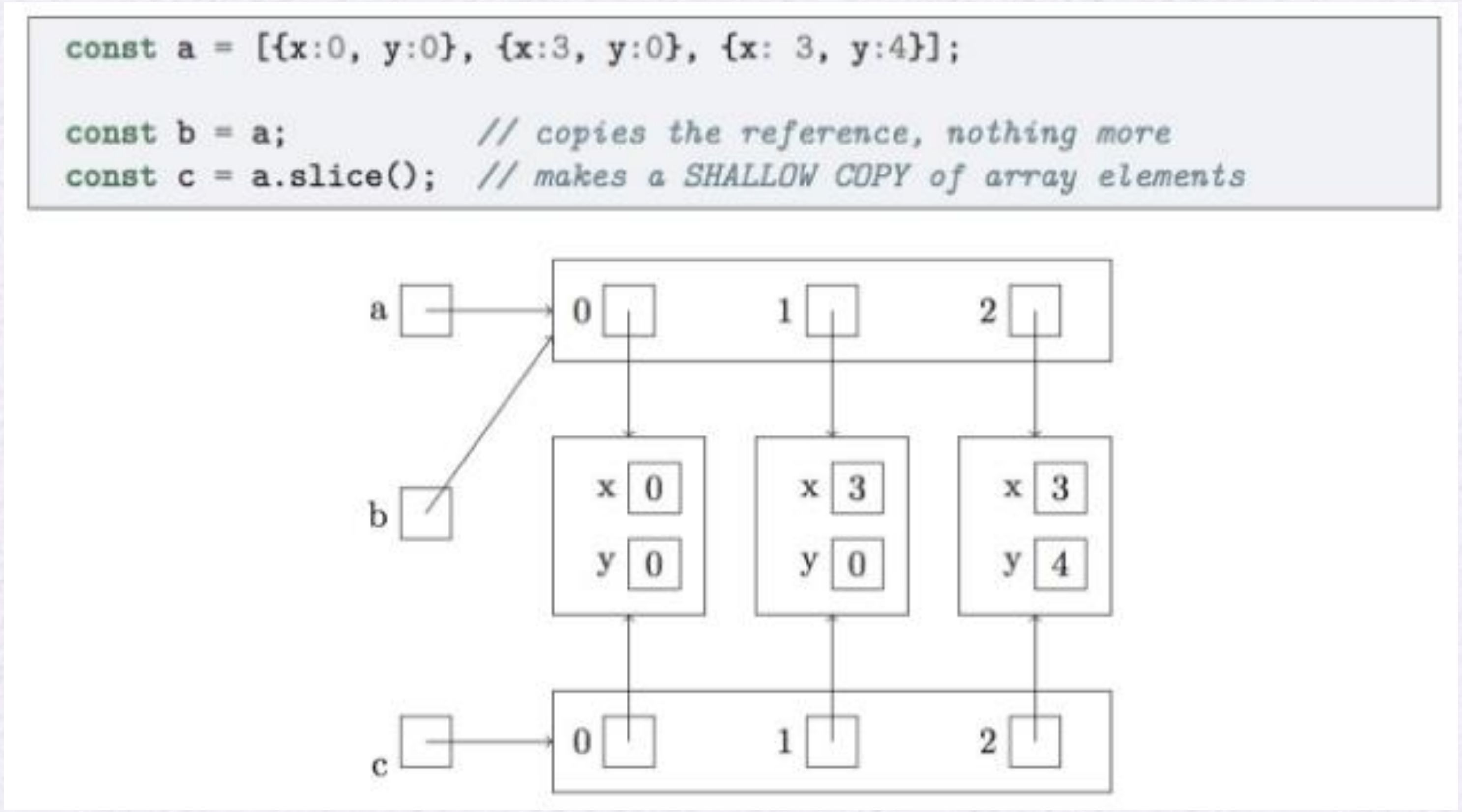


```
cars2[0] = "Mercedez";
```



Shallow copy VS Deep copy

- **slice** can be used to make a **shallow copy**.



```
const a = [{x:0, y:0}, {x:3, y:0}, {x: 3, y:4}];

const b = a;           // copies the reference, nothing more
const c = a.slice();   // makes a SHALLOW COPY of array elements
```

- A **deep copy** would be a completely independent copy
  - To make a deep copy, you would need to iterate through the components of an object, copying primitives and recursively creating deep copies of objects.

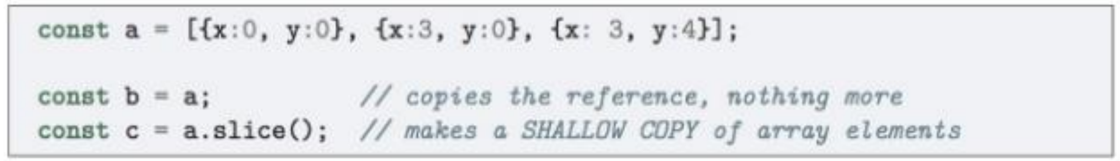
Shallow copies; exercise 1

- Show the memory content (variable boxes) after the following code is executed:

```
var people = [ {firstName:"Jay-Z", lastName:"Carter", age:48},
               {firstName:"Eminem", lastName:"Mathers", age:45},
               {firstName:"Drake", lastName:"Graham", age:35} ]
var rappers = people;
rappers[0] = {firstName:"Kendrick", lastName:"Lamar", age:31};
```

EXAMPLE

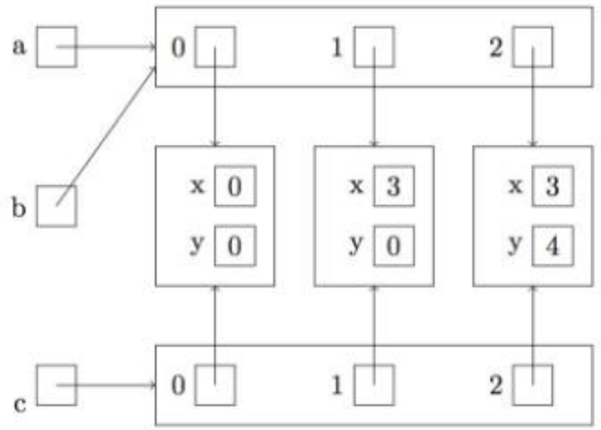
Code:



```
const a = [{x:0, y:0}, {x:3, y:0}, {x: 3, y:4}];

const b = a;          // copies the reference, nothing more
const c = a.slice();  // makes a SHALLOW COPY of array elements
```

Memory content after the code is executed:



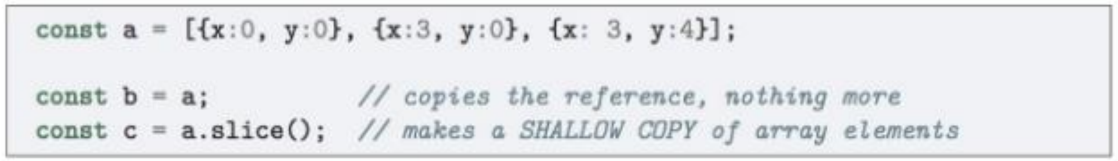
Shallow copies; exercise 2

- Show the memory content (variable boxes) after the following code is executed:

```
var people = [ {firstName:"Jay-Z", lastName:"Carter", age:48},
               {firstName:"Eminem", lastName:"Mathers", age:45},
               {firstName:"Drake", lastName:"Graham", age:35} ]
var rappers = people.slice();
rappers[0] = {firstName:"Kendrick", lastName:"Lamar", age:31};
```

EXAMPLE

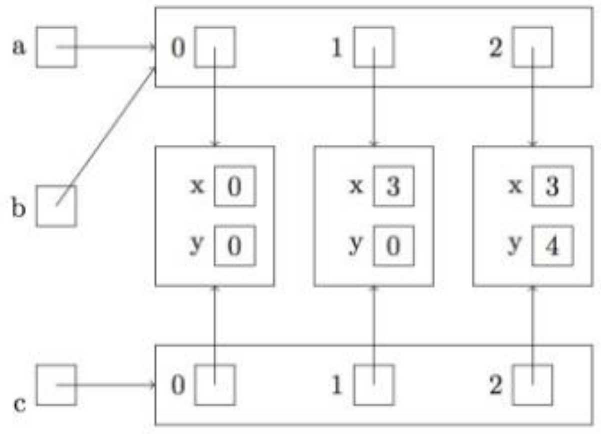
Code:



```
const a = [{x:0, y:0}, {x:3, y:0}, {x: 3, y:4}];

const b = a;           // copies the reference, nothing more
const c = a.slice();   // makes a SHALLOW COPY of array elements
```

Memory content after the code is executed:



Shallow copies; exercise 3

- Show the memory content (variable boxes) after the following code is executed:

```
var people = [ {firstName:"Jay-Z", lastName:"Carter", age:48},
               {firstName:"Eminem", lastName:"Mathers", age:45},
               {firstName:"Drake", lastName:"Graham", age:35} ]
var rappers = people.slice();
rappers[0].firstName="Kendrick";
rappers[0].lastName="Lamar";
rappers[0].age=31;
```

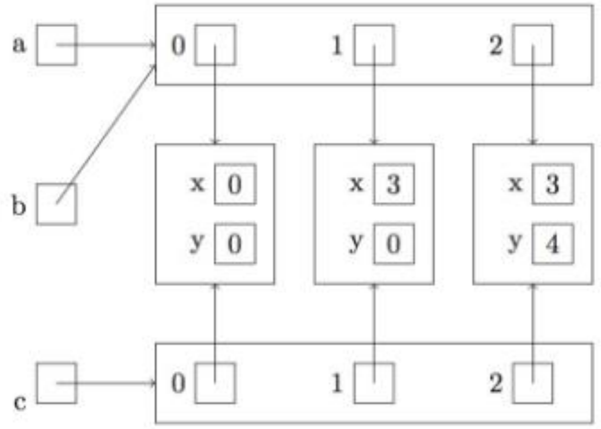
EXAMPLE

Code:

```
const a = [{x:0, y:0}, {x:3, y:0}, {x: 3, y:4}];

const b = a;           // copies the reference, nothing more
const c = a.slice();   // makes a SHALLOW COPY of array elements
```

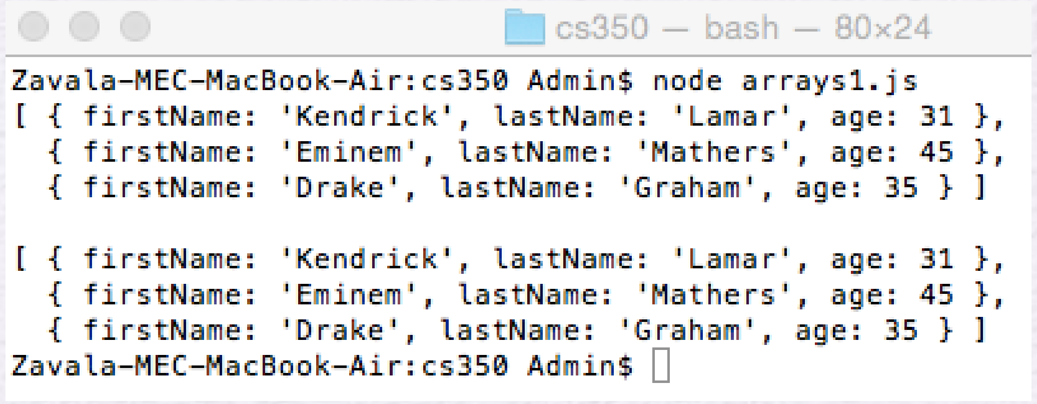
Memory content after the code is executed:



Shallow copies; program 1

- What would be printed by the following program?

```
var people = [ {firstName:"Jay-Z", lastName:"Carter", age:48},
               {firstName:"Eminem", lastName:"Mathers", age:45},
               {firstName:"Drake", lastName:"Graham", age:35} ]
var rappers = people;
rappers[0] = {firstName:"Kendrick", lastName:"Lamar", age:31};
console.log(people);
console.log(rappers);
```



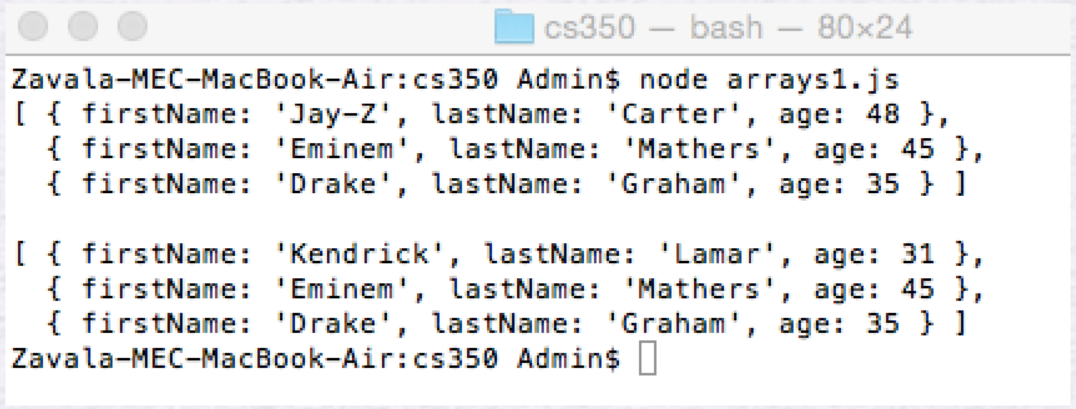
```
Zavala-MEC-MacBook-Air:cs350 Admin$ node arrays1.js
[ { firstName: 'Kendrick', lastName: 'Lamar', age: 31 },
  { firstName: 'Eminem', lastName: 'Mathers', age: 45 },
  { firstName: 'Drake', lastName: 'Graham', age: 35 } ]

[ { firstName: 'Kendrick', lastName: 'Lamar', age: 31 },
  { firstName: 'Eminem', lastName: 'Mathers', age: 45 },
  { firstName: 'Drake', lastName: 'Graham', age: 35 } ]
Zavala-MEC-MacBook-Air:cs350 Admin$ 
```

Shallow copies; program 2

- What would be printed by the following program?

```
var people = [ {firstName:"Jay-Z", lastName:"Carter", age:48},
               {firstName:"Eminem", lastName:"Mathers", age:45},
               {firstName:"Drake", lastName:"Graham", age:35} ]
var rappers = people.slice();
rappers[0] = {firstName:"Kendrick", lastName:"Lamar", age:31};
console.log(people);
console.log(rappers);
```



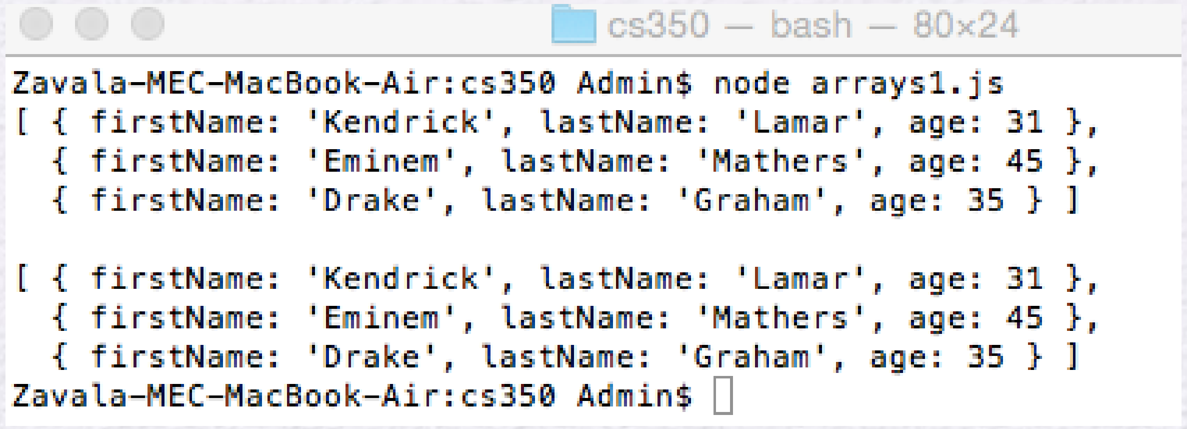
```
Zavala-MEC-MacBook-Air:cs350 Admin$ node arrays1.js
[ { firstName: 'Jay-Z', lastName: 'Carter', age: 48 },
  { firstName: 'Eminem', lastName: 'Mathers', age: 45 },
  { firstName: 'Drake', lastName: 'Graham', age: 35 } ]

[ { firstName: 'Kendrick', lastName: 'Lamar', age: 31 },
  { firstName: 'Eminem', lastName: 'Mathers', age: 45 },
  { firstName: 'Drake', lastName: 'Graham', age: 35 } ]
Zavala-MEC-MacBook-Air:cs350 Admin$ 
```

Shallow copies; program 3

- What would be printed by the following program?

```
var people = [ {firstName:"Jay-Z", lastName:"Carter", age:48},
               {firstName:"Eminem", lastName:"Mathers", age:45},
               {firstName:"Drake", lastName:"Graham", age:35} ]
var rappers = people.slice();
rappers[0].firstName="Kendrick";
rappers[0].lastName="Lamar";
rappers[0].age=31;
console.log(people);
console.log(rappers);
```



```
Zavala-MEC-MacBook-Air:cs350 Admin$ node arrays1.js
[ { firstName: 'Kendrick', lastName: 'Lamar', age: 31 },
  { firstName: 'Eminem', lastName: 'Mathers', age: 45 },
  { firstName: 'Drake', lastName: 'Graham', age: 35 } ]

[ { firstName: 'Kendrick', lastName: 'Lamar', age: 31 },
  { firstName: 'Eminem', lastName: 'Mathers', age: 45 },
  { firstName: 'Drake', lastName: 'Graham', age: 35 } ]
Zavala-MEC-MacBook-Air:cs350 Admin$
```