Problem Solving for Computer Science

IS51021C

Goldsmiths Computing

January 25, 2021

Problem 2:

Now you need to organise a joint birthday party for two people

You are given a list from each person, each list is actually a list of friends and a list of enemies

Combine both lists into a single list of people to invite:

- Friends of both should be at the top of list
- A friend of one and enemy of the other should be at the bottom
- Enemy of both should not be on the list

Try writing some JavaScript code!
e.g. a function with two arrays as arguments

Problem 2:

Now you need to organise a joint birthday party for two people

You are given a list from each person, each list is actually a list of friends and a list of enemies

Combine both lists into a single list of people to invite:

- Friends of both should be at the top of list
- A friend of one and enemy of the other should be at the bottom
- Enemy of both should not be on the list

This is a hard and vague problem
Test approach to solving problems rather than ‘getting the right answer’

Start simple with examples - make simplifying assumptions

```
var list1 = [{name:"Wonder Woman",friend:true},
             {name:"Batman",friend:true},
             {name:"Lex Luthor",friend:false},
             {name:"Catwoman",friend:false}];

var list2 = [{name:"Wonder Woman",friend:true},
             {name:"Batman",friend:true},
             {name:"Lex Luthor",friend:false},
             {name:"Catwoman",friend:true}];
```

Break problem down into simpler problems - delete enemies on both lists

```
// First we can get rid of enemies on both lists
for (var i = 0; i < list1.length; i++) {
    if (!(list1[i].friend || list2[i].friend)) {
        list1.splice(i,1);
        list2.splice(i,1);
    }
}
```

```
// First we can get rid of enemies on both lists
for (var i = 0; i < list1.length; i++) {
    if (!(list1[i].friend || list2[i].friend)) {
        list1.splice(i,1);
        list2.splice(i,1);
    }
}
```

We have now simplified the problem

Move onto the next element - put friends on both lists into a new list

```
// Next we can create a new array and put good friends at the top
var finalList = [];
for (var i = 0; i < list1.length; i++) {
    if (list1[i].friend && list2[i].friend) {
        finalList.unshift(list1[i].name);
        list1.splice(i,1);
        list2.splice(i,1);
    }
}
```

```
// Next we can create a new array and put good friends at the top
var finalList = [];
for (var i = 0; i < list1.length; i++) {
    if (list1[i].friend && list2[i].friend) {
        finalList.unshift(list1[i].name);
        list1.splice(i,1);
        list2.splice(i,1);
    }
}
```

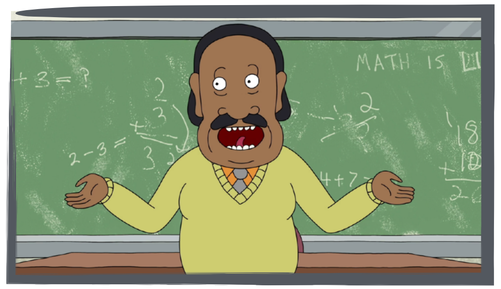
Now we are left with little to do…

```
// Finally the OK friends can go at the bottom
for (var i = 0; i < list1.length; i++) {
        finalList.push(list1[i].name);
}
```

We solved a form of the initial problem by solving several simpler problems
We can then worry about whether we can improve the method

The module so far

THEORY



EXPERIMENT

Lecture 1

What is a problem?

- Data types independent of programming language: **abstract data types**

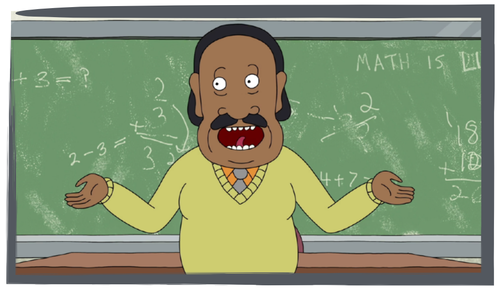
Lecture 2

Basics of JavaScript

- Primitive **data types** in JavaScript

The module so far

THEORY



EXPERIMENT

Lecture 1

What is a problem?

- Data types independent of programming language: **abstract data types**

Lecture 2

Basics of JavaScript

- Primitive **data types** in JavaScript

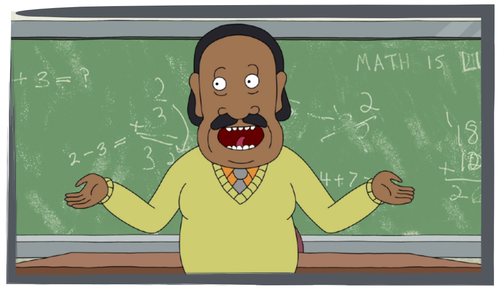
Theoretical model

Language specific

Values specified & allowed operations

The module so far

THEORY



EXPERIMENT

Lecture 1

What is a problem?

- Data types independent of programming language: **abstract data types**

Lecture 2

Basics of JavaScript

- Primitive **data types** in JavaScript

Theoretical model

Language specific

Values specified & allowed operations

true, false

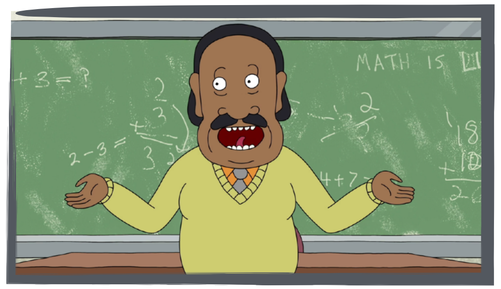
OR, AND, NOT

`true false`

`||` `&&` `!`

About the module

THEORY



Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a
…

Implementation

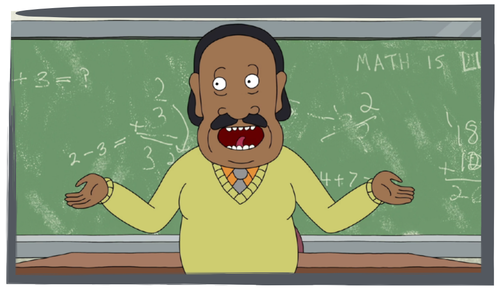
EXPERIMENT

JavaScript data types:

Number (float)	`1` `NaN`
Boolean	`true` `false`
String	`""`
Undefined	`undefined`

About the module

THEORY



EXPERIMENT

Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a
…

Implementation →

?? →

JavaScript data types:

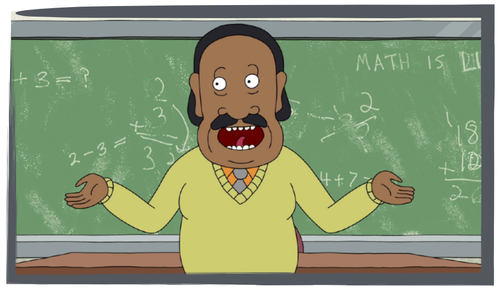
Number (float) `1` `NaN`
Boolean `true` `false`
String `""`
Undefined `undefined`

Objects: e.g. Arrays

Composed of primitive data

About the module

THEORY



EXPERIMENT

Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a
…

Implementation

??

JavaScript data types:

Number (float) `1` `NaN`
Boolean `true` `false`
String `""`
Undefined `undefined`

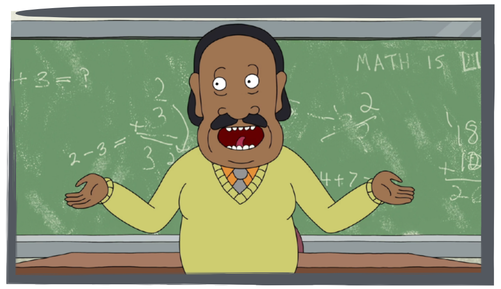
Objects: e.g. Arrays

What do objects implement?

What are the corresponding abstract data types?

About the module

THEORY



EXPERIMENT

Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a

Dynamic Arrays
Vectors
Queues
Stacks

Abstract data structures

Implementation

JavaScript data types:

Number (float) `1` `NaN`
Boolean `true` `false`
String `""`
Undefined `undefined`

Objects: e.g. Arrays

What do objects implement?
Lots of extremely useful things!

What are the corresponding abstract data types?
Lots of extremely useful things!

Today

1. JavaScript Objects and Arrays
2. Abstract Data Structures
3. Constructors

Today

1. **JavaScript Objects and Arrays**
2. Abstract Data Structures
3. Constructors

Objects

Type	Literals (Values)	
Boolean	`true` `false`	
Number (float)	`1` `3.14` `NaN`	
String	`"ps_for_cs"` `""`	← Empty string
Undefined	`undefined`	Variables without values
Null	`null`	Nothing

All other* data types are **objects**

*Technically Null an object in JavaScript (one of its “oddities”)

Objects revision

Objects can collect together multiple pieces of data: properties

Can also include methods: functions that can act on the properties of the object

Objects revision

Objects can collect together multiple pieces of data: properties

`object.bool` `object.num1` `object.num2`

Can also include methods: functions that can act on the properties of the object

```
var object = {
    bool:true,
    num1:48,
    num2:42,
}
```

Objects revision

Objects can collect together multiple pieces of data: properties

`object.bool` `object.num1` `object.num2`

Can also include methods: functions that can act on the properties of the object

```
var object = {
    bool:true,
    num1:48,
    num2:42,
    addNumbers:function(){
        return this.num1 + this.num2;
    }
}
```

```
console.log(object.addNumbers());
```

Objects revision

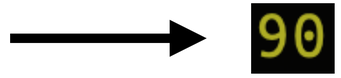
Objects can collect together multiple pieces of data: properties

`object.bool` `object.num1` `object.num2`

Can also include methods: functions that can act on the properties of the object

```
var object = {
    bool:true,
    num1:48,
    num2:42,
    addNumbers:function(){
        return this.num1 + this.num2;
    }
}
```

```
console.log(object.addNumbers());
```



Objects revision

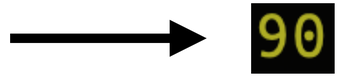
Objects can collect together multiple pieces of data: properties

`object.bool` `object.num1` `object.num2`

Can also include methods: functions that can act on the properties of the object

```
var object = {
    bool:true,
    num1:48,
    num2:42,
    addNumbers:function(){
        return this.num1 + this.num2;
    }
}
```

```
console.log(object.addNumbers());
```



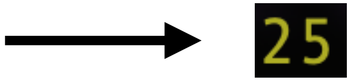
“this” refers to the object containing the method

Objects revision

We can also store objects inside other objects

```
var object = {
    bool:true,
    num1:48,
    num2:42,
    numbers:{num1:24,num2:25,num3:43}
}
```

```
console.log(object.numbers.num2)
```



Objects: an analogy

I like to think of objects as a bit like houses

A house is a single thing with properties, e.g. number of bedrooms, size of kitchen, garden…

Objects: an analogy

I like to think of objects as a bit like houses

```
{
    numBedrooms:2,
    garden:true
}
```

A house is a single thing with properties, e.g. number of bedrooms, size of kitchen, garden…

If we own the house, we can also amend it

Objects: an analogy

I like to think of objects as a bit like houses

```
{
    numBedrooms:2,
    garden:true,
    removeGarden:function() {
        if(garden) {
            this.garden = false;
            return "Garden Removed!";
        }
        return "No Garden To Remove";
    }
    addBedrooms:function(n) {
        this.numBedrooms = this.numBedrooms + n;
        return "New Bedroom Built!";
    }
}
```

A house is a single thing with properties, e.g. number of bedrooms, size of kitchen, garden…

If we own the house, we can also amend it…

We are missing some important information: the **address** of the house!

Location, location, location!

We are missing some important information: the **address** of the house!

Location, location, location!

Objects are stored in memory, so we need a way to *find* them

Variables store the **address** of an object: these are called *references*

Variables are containers for *simple* data

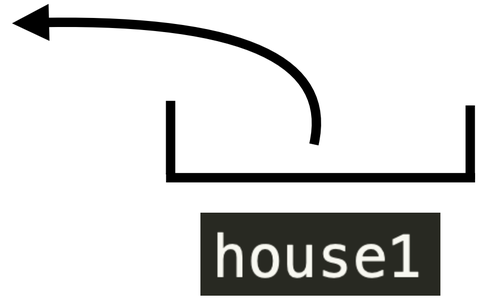
We need to “assign” objects to variables:

```
var house1 = {
    numBedrooms:2,
    garden:true
};
```

Variable stores a *reference* to the object: tells us where to look to find the object

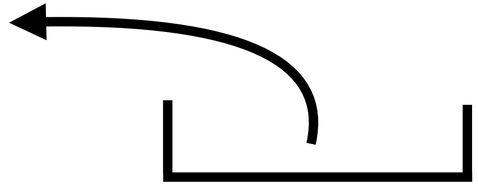
```
{
    numBedrooms:2,
    garden:true
}
```

Arrow is reference



```
{
    numBedrooms:2,
    garden:true
}
```

Arrow is reference



`house1`

1 Main Road
The Town

We can store the address on the back of an envelope

1 Main Road
The Town

1 Main Road
The Town

1 Main Road
The Town

Address can be written in many places
We can copy the address
All refer to the same house

1 Main Road
The Town

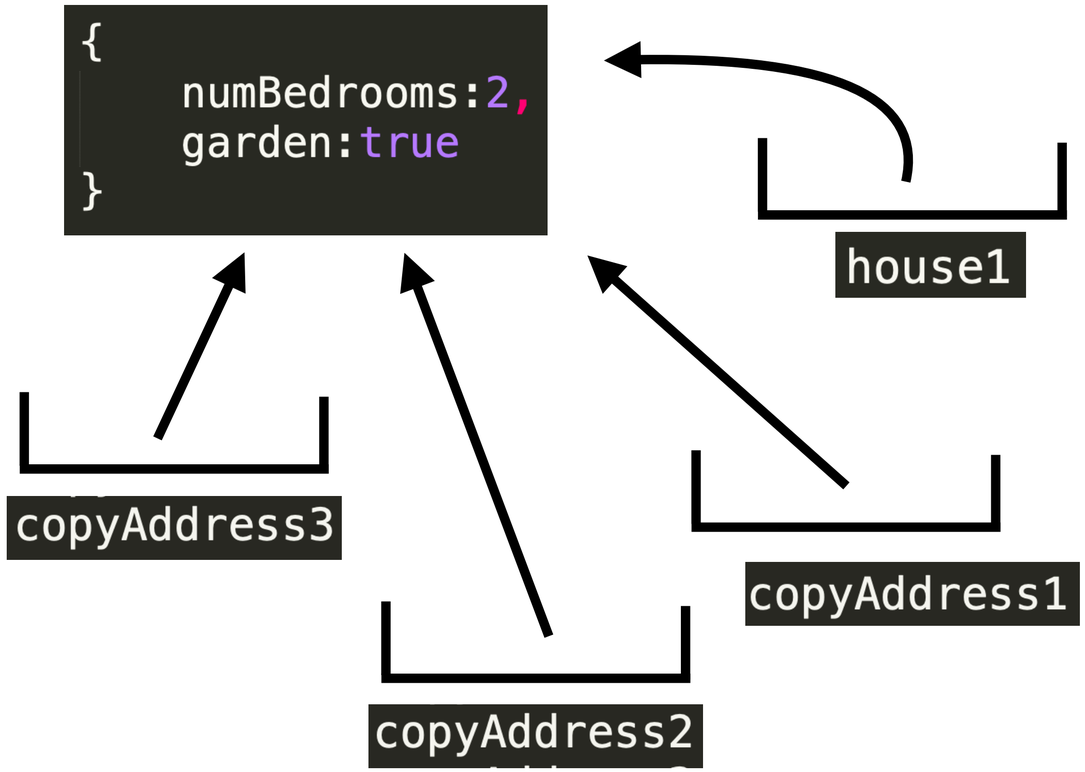
1 Main Road
The Town

1 Main Road
The Town

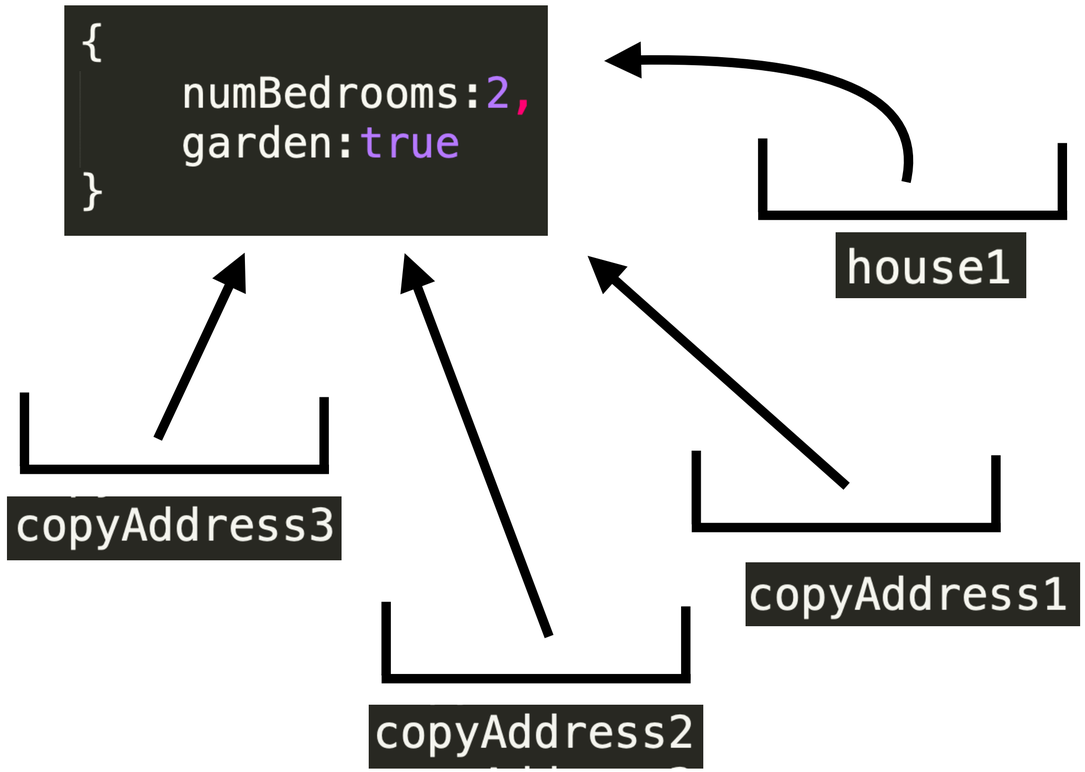
Address can be written in many places
We can copy the address
All refer to the same house

These references to object can be assigned
many times to new variables
All refer to the same object

```
var house1 = {
    numBedrooms:2,
    garden:true
};

var copyAddress1 = house1;
var copyAddress2 = house1;
var copyAddress3 = copyAddress1;
```



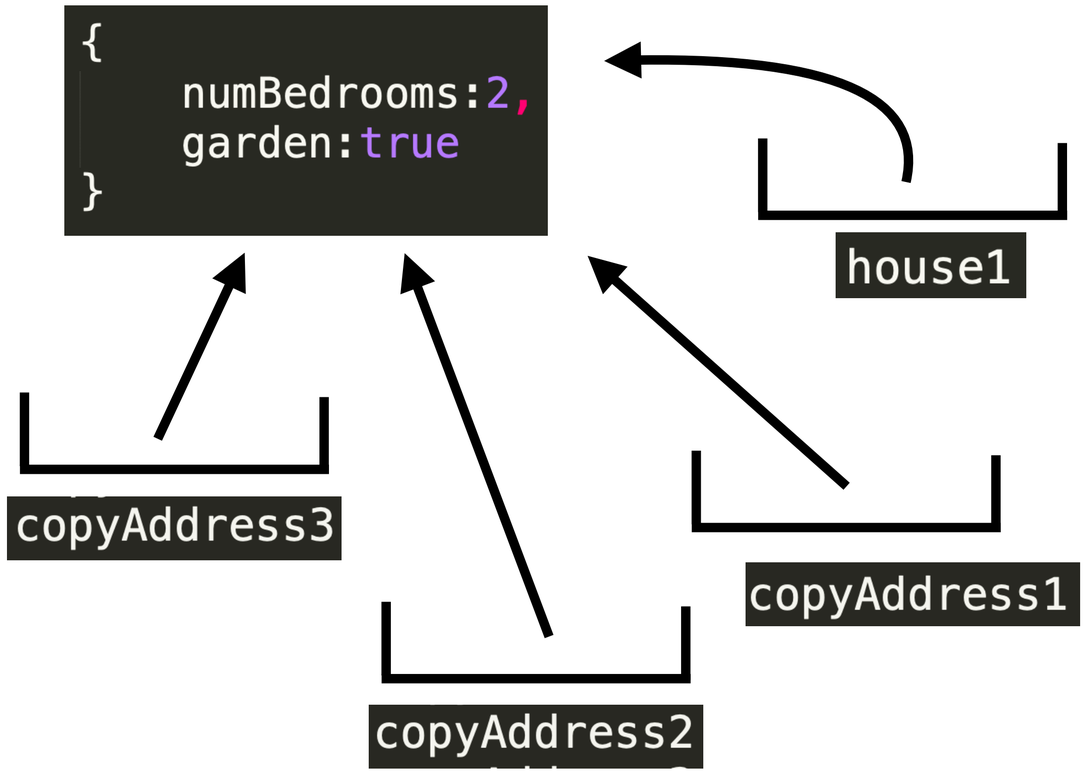
```
var house1 = {
    numBedrooms:2,
    garden:true
};

var copyAddress1 = house1;
var copyAddress2 = house1;
var copyAddress3 = copyAddress1;
```




```
console.log(copyAddress1 === house1);
```

What gets printed in the console?

```
var house1 = {
    numBedrooms:2,
    garden:true
};

var copyAddress1 = house1;
var copyAddress2 = house1;
var copyAddress3 = copyAddress1;
```




```
console.log(copyAddress1 === house1);
```

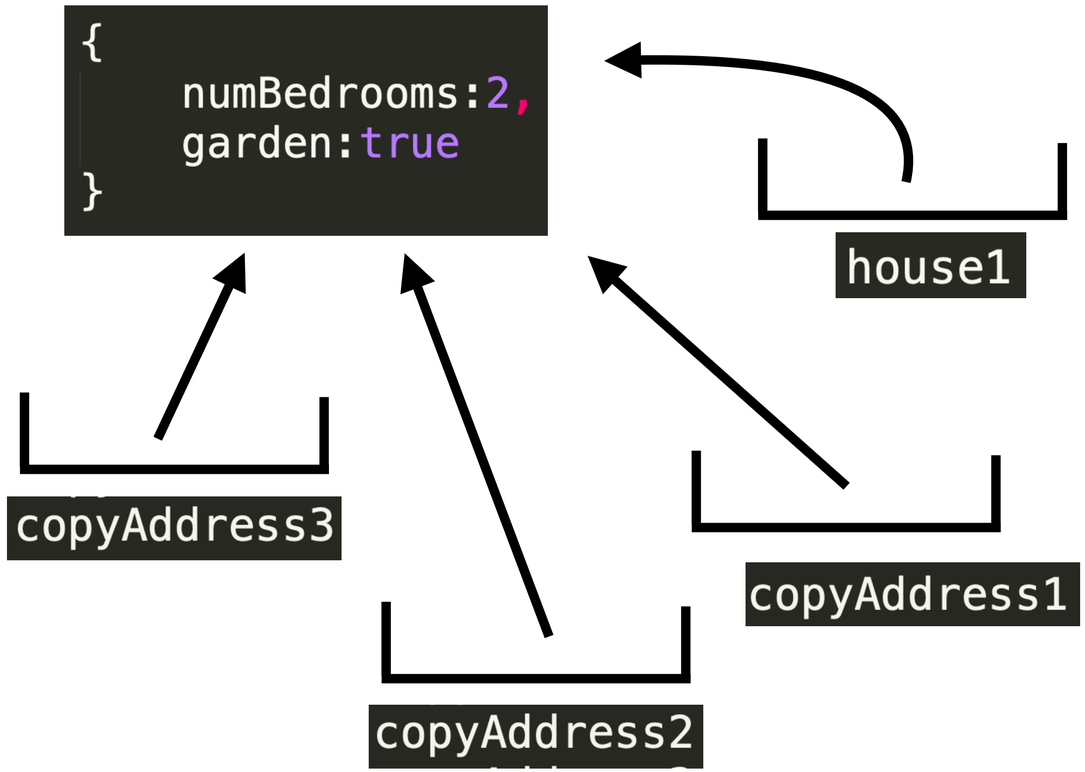
What gets printed in the console?

```
true
```

The addresses
(references) are the same!

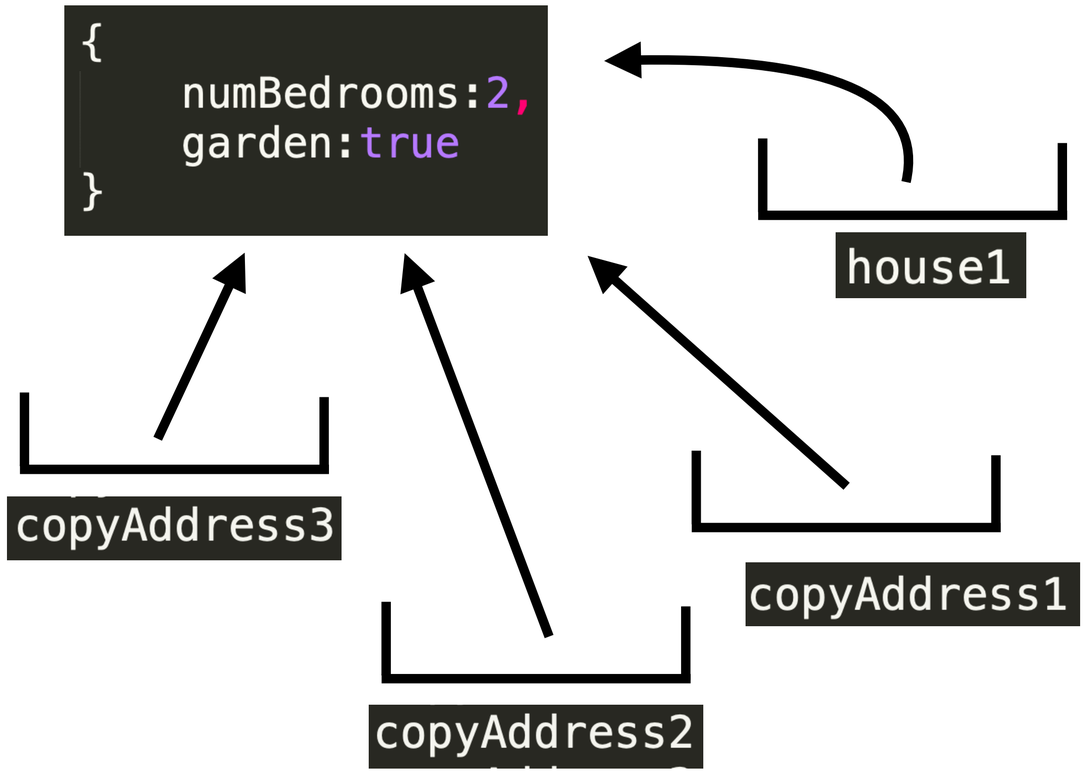
```
var house1 = {
    numBedrooms:2,
    garden:true
};

var copyAddress1 = house1;
var copyAddress2 = house1;
var copyAddress3 = copyAddress1;
```

We can change a property of the object using any of the variables



```
var house1 = {
    numBedrooms:2,
    garden:true
};

var copyAddress1 = house1;
var copyAddress2 = house1;
var copyAddress3 = copyAddress1;

copyAddress2.numBedrooms = 3;
```

We can change a property of the object using any of the variables

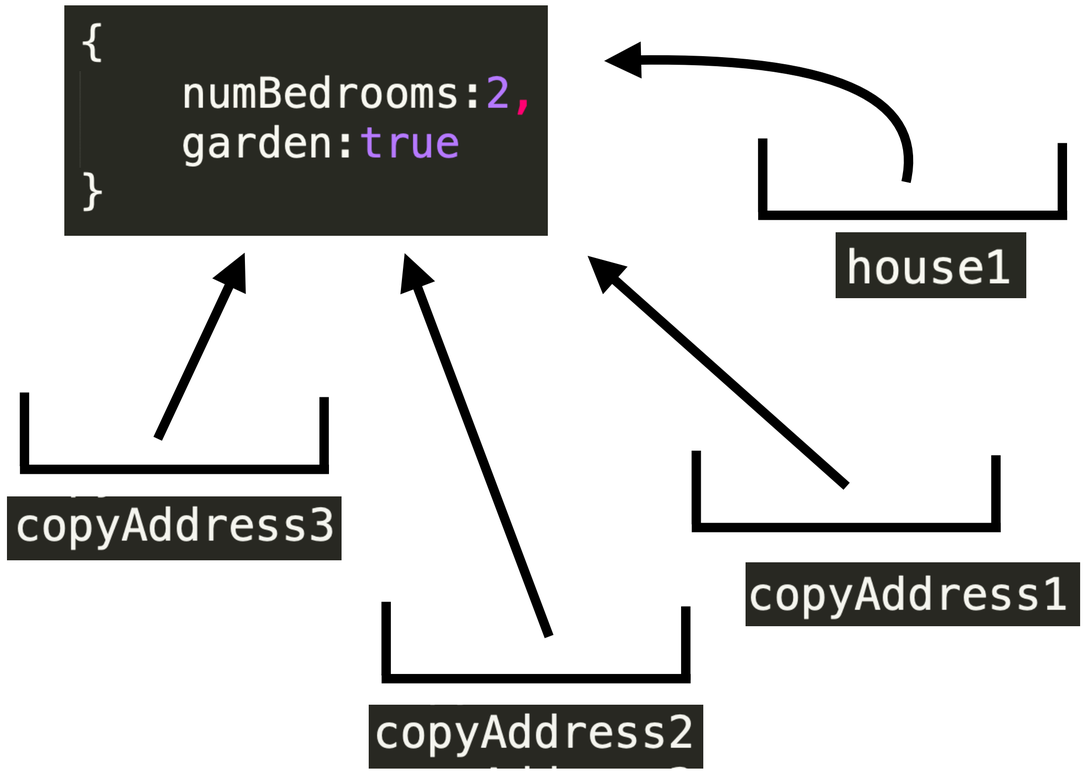



```
console.log(house1.numBedrooms);
```

What gets printed in the console?

```
var house1 = {
    numBedrooms:2,
    garden:true
};

var copyAddress1 = house1;
var copyAddress2 = house1;
var copyAddress3 = copyAddress1;

copyAddress2.numBedrooms = 3;
```

We can change a property of the object using any of the variables




```
console.log(house1.numBedrooms);
```

What gets printed in the console?

3

The same would be true of the other two variables

They all refer to the same object

Understanding the distinction between data and data location is ***essential*** for computer science

Imagine we have a row of houses that have all the same properties

1 Main Road
The Town

2 Main Road
The Town

3 Main Road
The Town

What differentiates the houses is the address

Imagine we have a row of houses that have all the same properties

1 Main Road
The Town

2 Main Road
The Town

3 Main Road
The Town

```
var house1 = {
    numBedrooms:2,
    garden:true
};

var house2 = {
    numBedrooms:2,
    garden:true
};

var house3 = {
    numBedrooms:2,
    garden:true
};
```

```
console.log(house1 === house2);
```

What gets printed in the console?

Imagine we have a row of houses that have all the same properties

1 Main Road
The Town

2 Main Road
The Town

3 Main Road
The Town

```
var house1 = {
    numBedrooms:2,
    garden:true
};

var house2 = {
    numBedrooms:2,
    garden:true
};

var house3 = {
    numBedrooms:2,
    garden:true
};
```

```
console.log(house1 === house2);
```

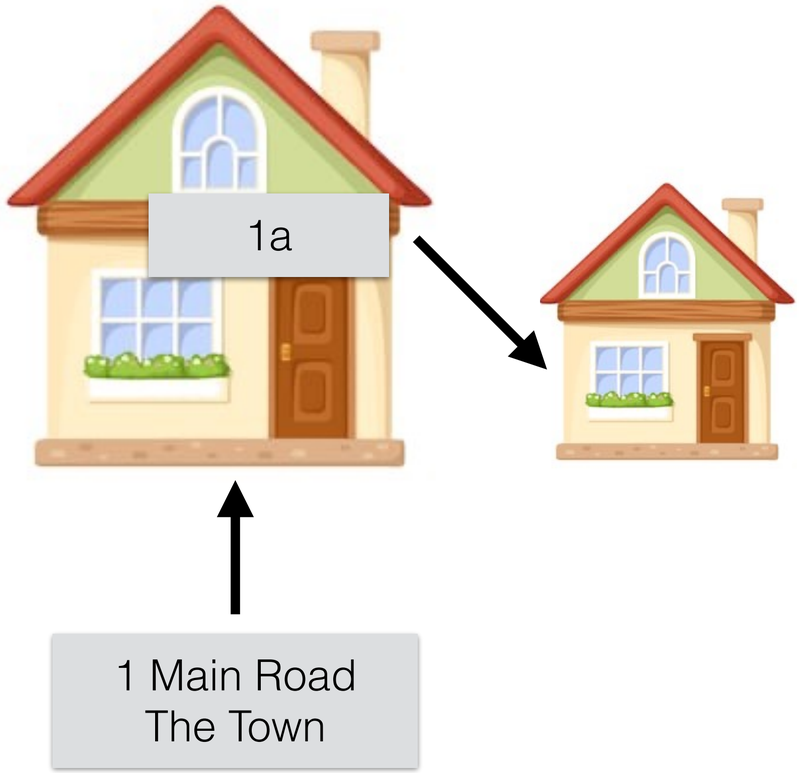
What gets printed in the console?

```
false
```

Each new assignment creates a **new object**

Like building a new house

Houses can have separate extensions




```
var house = {
    numBedrooms:2,
    garden:true,
    extension:{numBedrooms:1,garden:false}
};
```

Object inside an object

The property extension gives us a reference to the extension

We'll come back to this next week

An array is a *special* kind of object

When we create new arrays we create objects

Variables store references to arrays

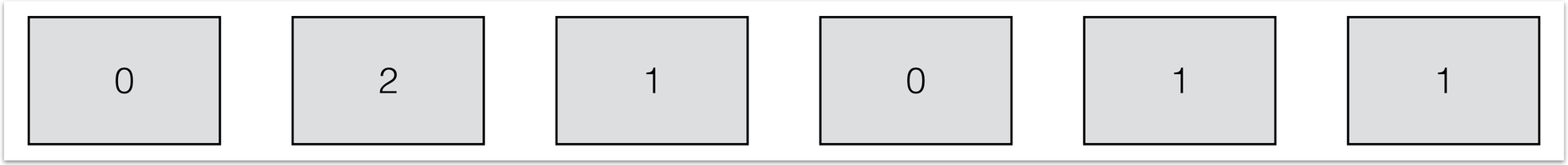
An array is a *special* kind of object

When we create new arrays we create objects

Variables store references to arrays

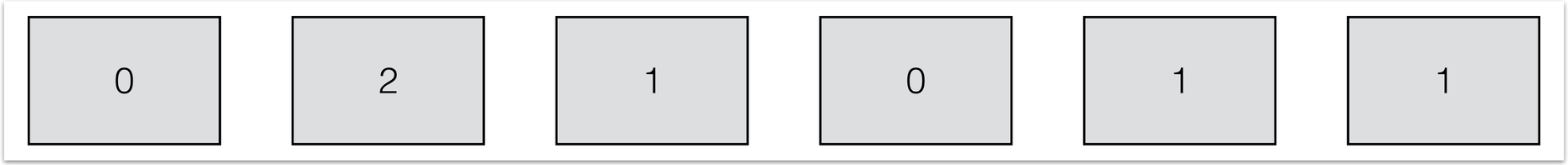
```
var arr1 = [1, 2, 3];
var arr2 = arr1;

arr1[0] = 0;

console.log(arr2);
```

What are the properties of an array?




```
var arr = [0, 2, 1, 0, 1, 1];
```

What are the properties of an array?




```
var arr = [0, 2, 1, 0, 1, 1];
```

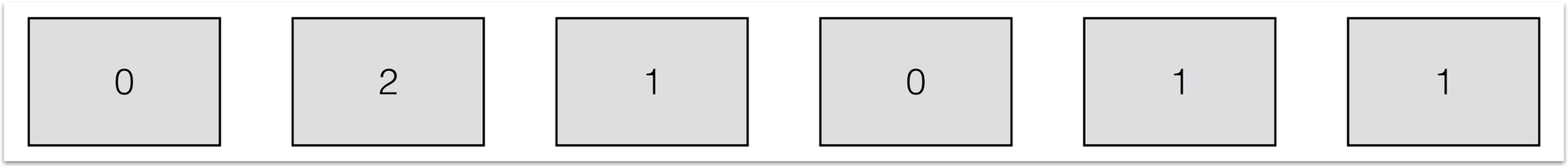
The elements

```
arr[2]
```

Its length

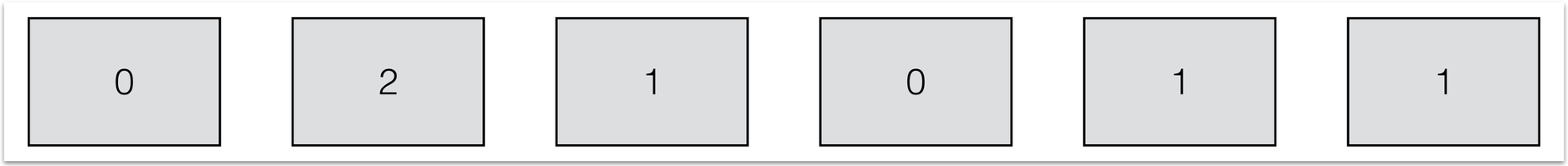
```
arr.length
```

What are the methods for an array?




```
var arr = [0, 2, 1, 0, 1, 1];
```

What are the methods for an array?




```
var arr = [0, 2, 1, 0, 1, 1];
```

Adding and removing elements from front:

```
arr.unshift(0);
```

```
arr.shift();
```

Adding and removing elements from back:

```
arr.push(3)
```

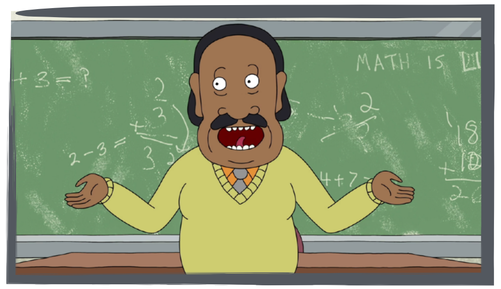
```
arr.pop();
```

Adding and removing elements from anywhere else:

```
arr.splice(0,1);
```

```
arr.slice();
```
etc

THEORY



EXPERIMENT

Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a
…

Implementation

JavaScript data types:

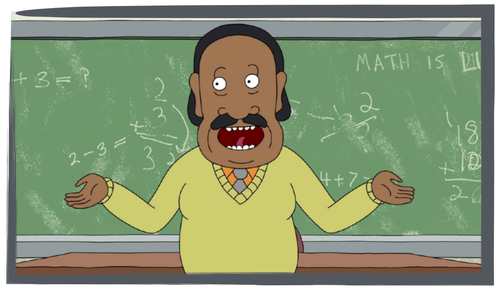
Number (float) `1` `NaN`
Boolean `true` `false`
String `""`
Undefined `undefined`

? → **Array**

What is the *Abstract Data Structure* implemented by a JavaScript Array?

About the module

THEORY



EXPERIMENT

Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a

Dynamic Array

Implementation

JavaScript data types:

Number (float) `1` `NaN`
Boolean `true` `false`
String `""`
Undefined `undefined`

Array

What is the *Abstract Data Structure* implemented by a JavaScript Array?

The Dynamic Array

The Dynamic Array is an **abstract data structure**

It is independent of any programming language

JavaScript implements it

Admin

- Worksheet 2 available today from 11am

- Quiz 2 available today from 4pm - 2.5% of your final grade
  - Deadline for first quiz: **1st February 4pm**
  - Deadline for second quiz: **8th February 4pm**
  - **I forgot to say**: if you attempt quiz, final grade is:

3 + 7<your best attempt>/10

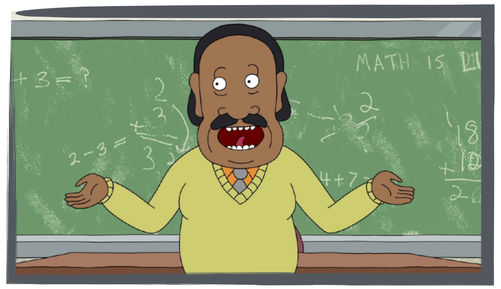
  - No attempt gets *0/10*

Today

1. JavaScript Objects and Arrays
2. **Abstract Data Structures**
3. Constructors

Abstract Data Structures

THEORY



EXPERIMENT

Lecture 1

What is a problem?

- Data types independent of programming language: **abstract data types**

Lecture 2

Basics of JavaScript

- Primitive **data types** in JavaScript

Theoretical model

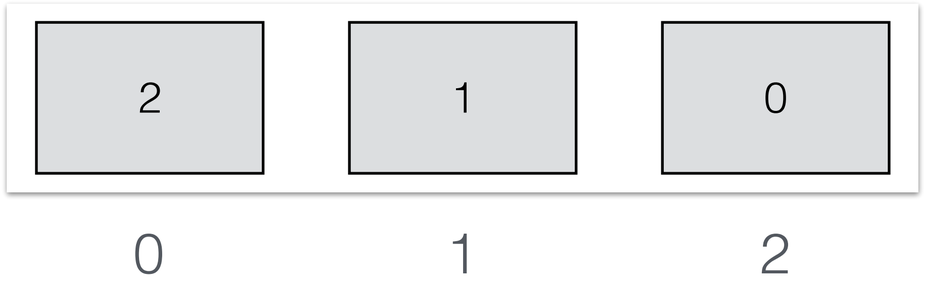
Language specific

Values specified & allowed operations

Dynamic Array

Values specified & allowed operations

Elements of data ordered on a line
Indexed



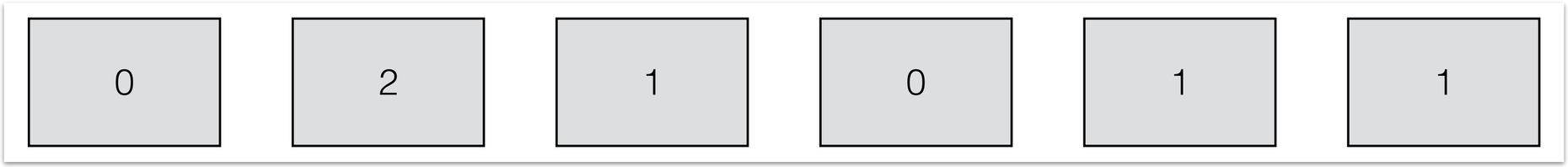
Dynamic Array

Values specified & allowed operations

Elements of data ordered on a line
Indexed

Operation	Description
length	How many elements in Dynamic Array
select[k]	Returns the value stored at element with index k
store![o,k]	Store the value o stored at element with index k
removeAt![k]	Remove element with index k Shift everything from k+1 onwards one to the left
insertAt![o,k]	Insert element with value o at index k Shift everything from k onwards one to the right

Example:



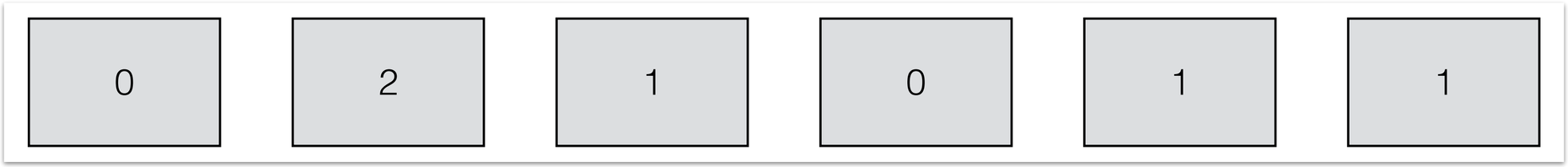
length select[2]

store![4, 3]

removeAt![2]

insertAt![10,4]

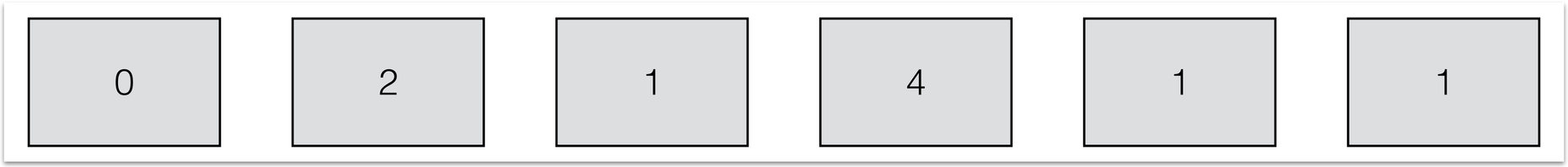
Example:



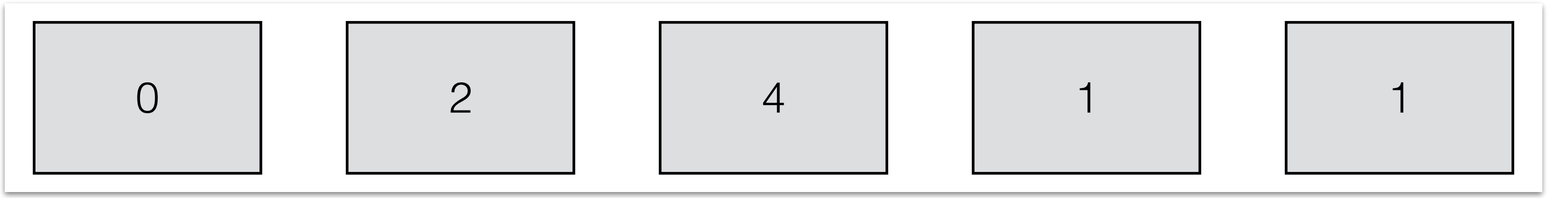
0 1 2 3 4 5

length 6 select[2] 1

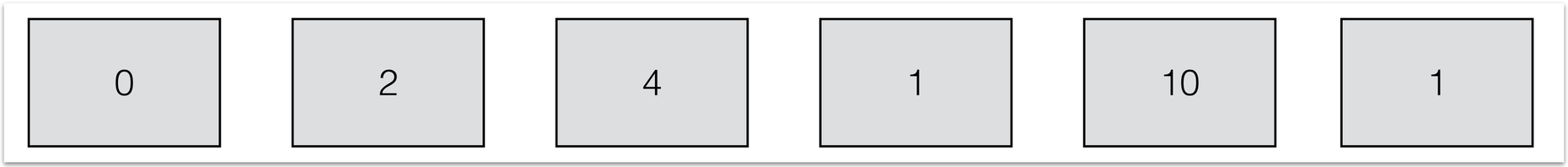
store![4, 3]



removeAt![2]



insertAt![10,4]



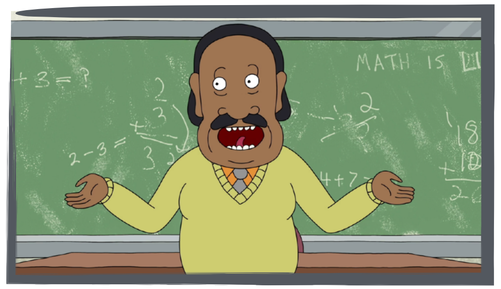
Dynamic Array

Operation	Description	JS implementation
length	How many elements in Dynamic Array	`arr.length`
select[k]	Returns the value stored at element with index k	`arr[2]`
store![o,k]	Store the value o stored at element with index k	`arr[1] = 2;`
removeAt![k]	Remove element with index k Shift everything from k+1 onwards one to the left	`arr.splice(0,1);`
insertAt![o,k]	Insert element with value o at index k Shift everything from k onwards one to the right	`arr.splice(2, 0, 2);`

Or, possibly `arr.push(3)` `arr.pop();`

Vectors

THEORY



Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a
Dynamic Arrays ✓
Vectors
Queues
Stacks

Abstract data structures

Implementation

EXPERIMENT

JavaScript data types:

Number (float) `1` `NaN`
Boolean `true` `false`
String `""`
Undefined `undefined`

Objects: e.g. Arrays

The Vector is an **abstract data structure**

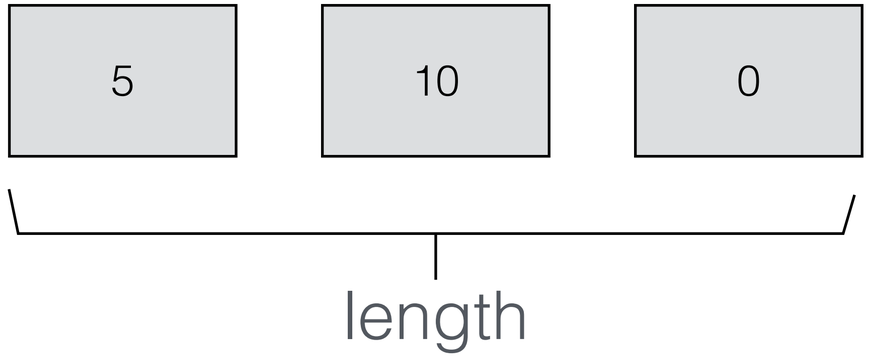
It is independent of any programming language

Arrays in Java and C++ implement it

It is like a Dynamic Array where the length can't change

This name is particular to this module - the word vector is used differently in different contexts, e.g. in C++

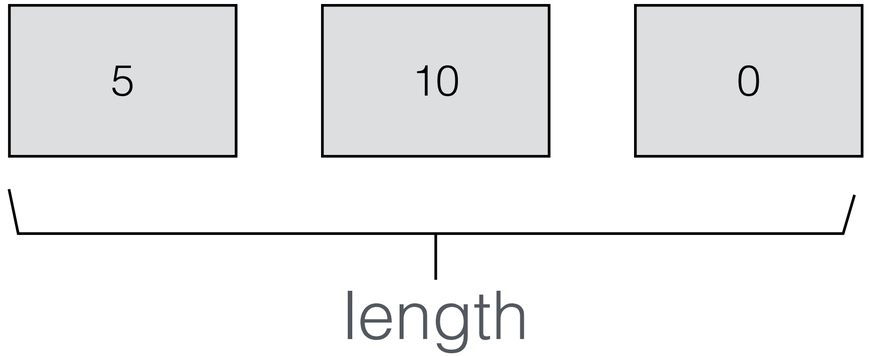
A vector:



Empty vector:

Number of elements **cannot be changed**

A vector:



Allowed operations:

length	Reveals number of elements
select[k]	Reads a specific element k
store![o,k]	Writes value o to element k

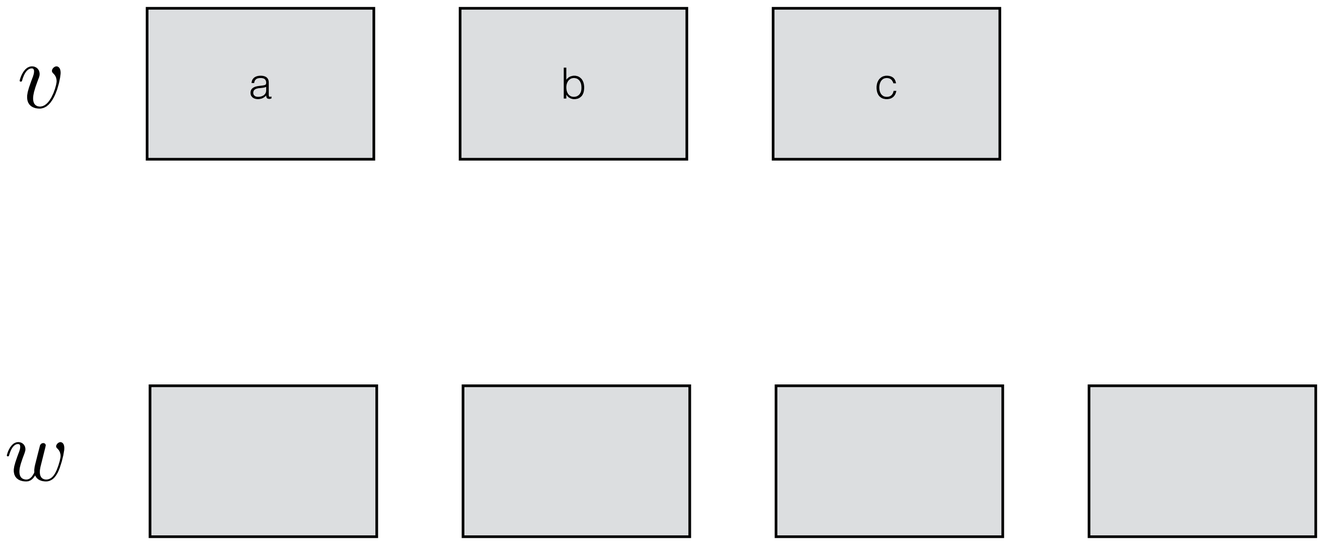
A vector: 

length	Reveals number of elements
select[k]	Reads a specific element k
store![o,k]	Writes value o to element k

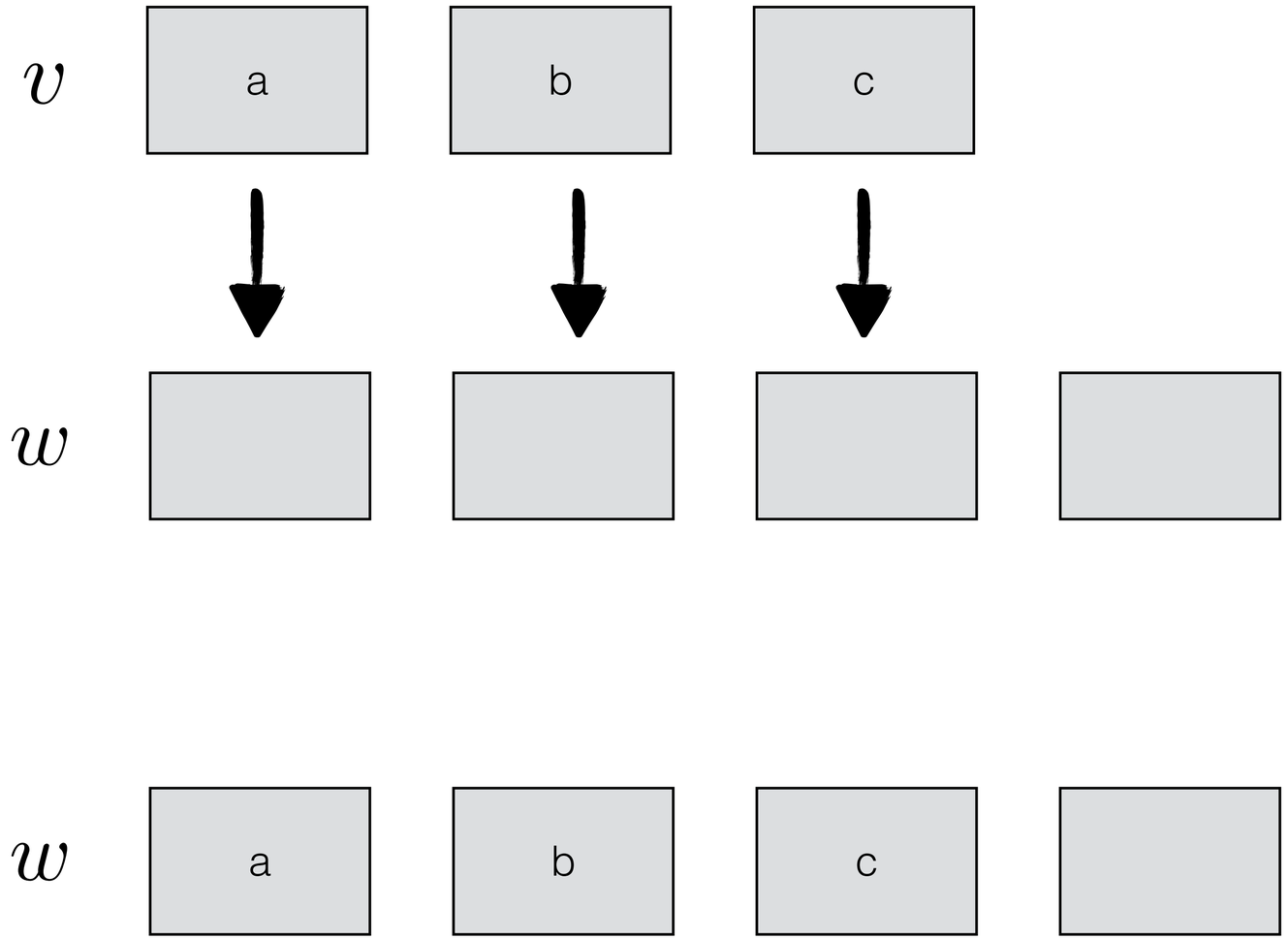
We can always create new vectors of fixed size

Memory allocation

We can always create new, empty vectors of fixed size

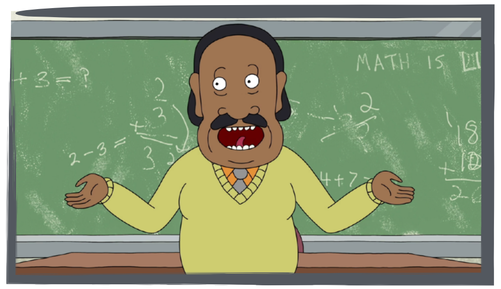


We can always create new, empty vectors of fixed size



Queues

THEORY



Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a
Dynamic Arrays ✓
Vectors ✓
Queues
Stacks

Abstract data structures

Implementation

EXPERIMENT

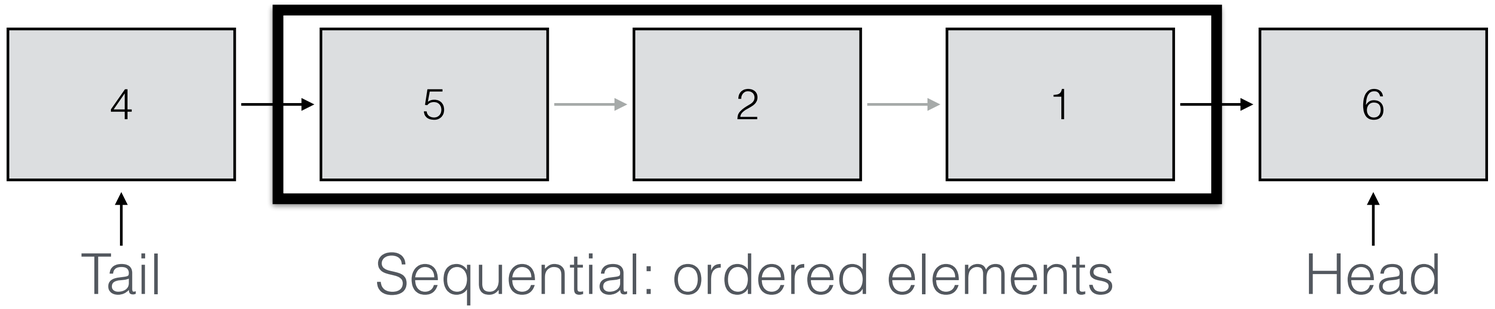
JavaScript data types:

Number (float) `1` `NaN`
Boolean `true` `false`
String `""`
Undefined `undefined`

Objects: e.g. Arrays

Queues*

A queue:



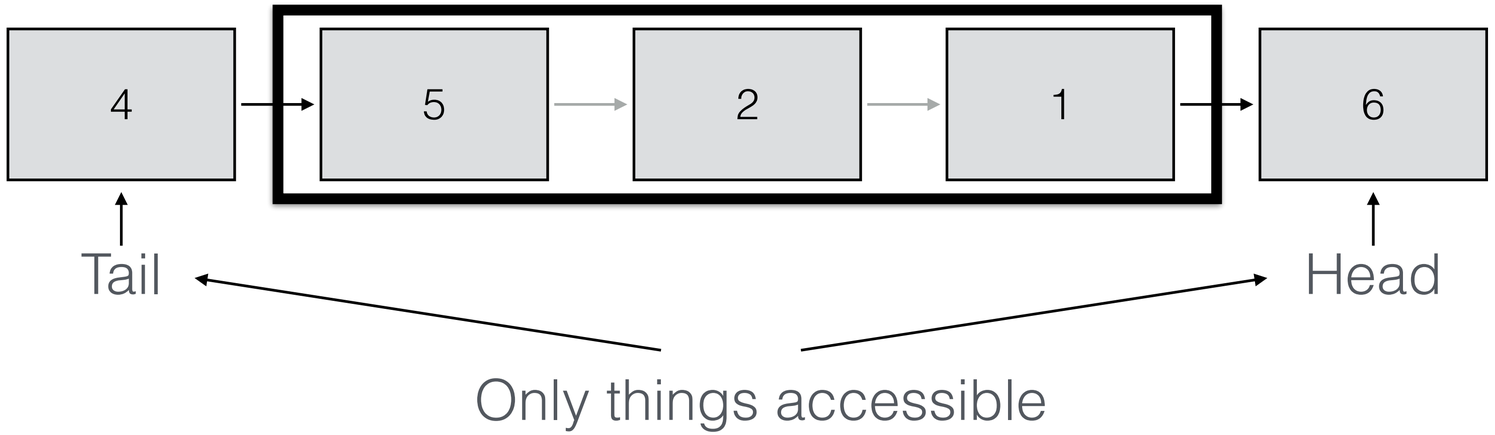
Empty queue:

FIRST-IN-FIRST-OUT (FIFO)

*The most British abstract data structure

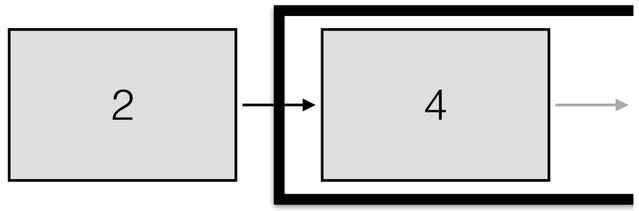
Queues

A queue:

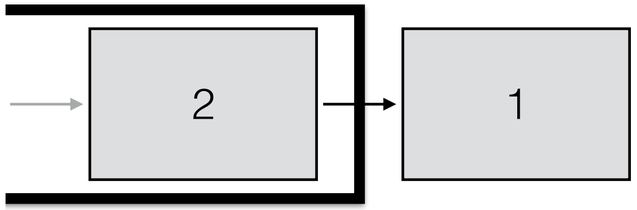


Number of elements can change:

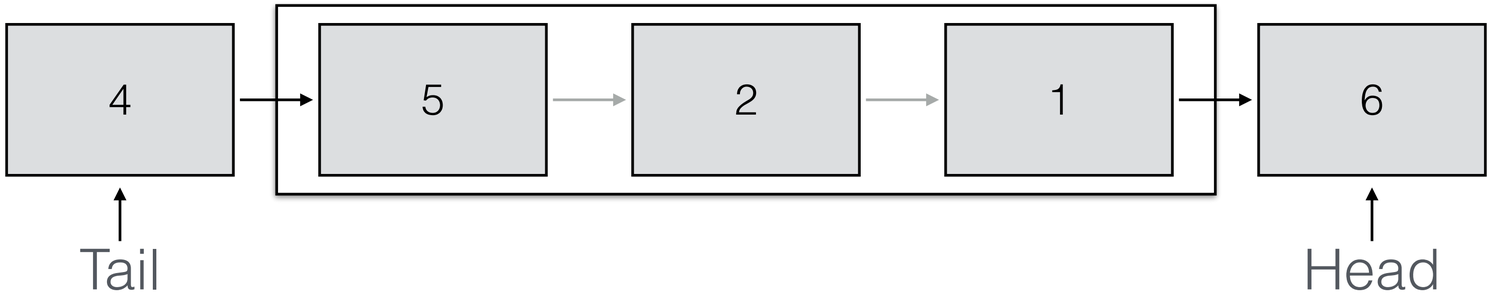
Add element to tail



Remove element from head

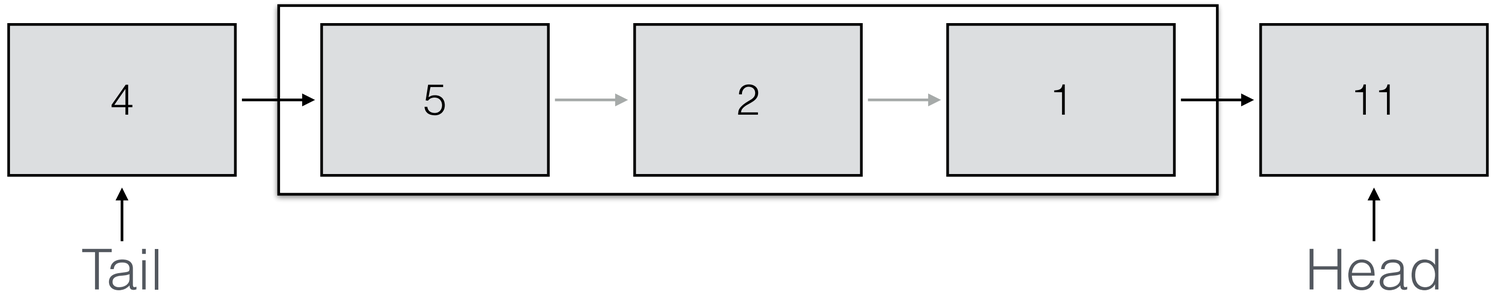


Queues



Allowed operations:

head	Reads out the value stored in the head
dequeue!	Removes the head element
enqueue![o]	Adds element to tail with value o
empty?	Checks if queue is empty

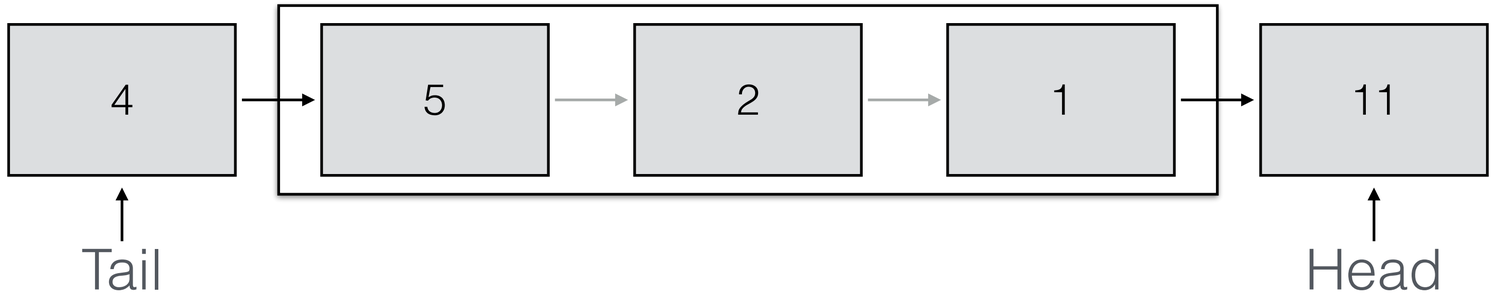


head

dequeue!

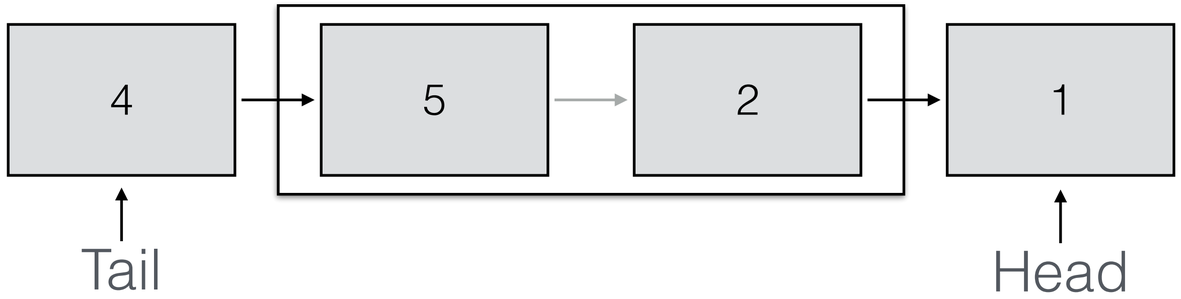
enqueue![2]

empty?

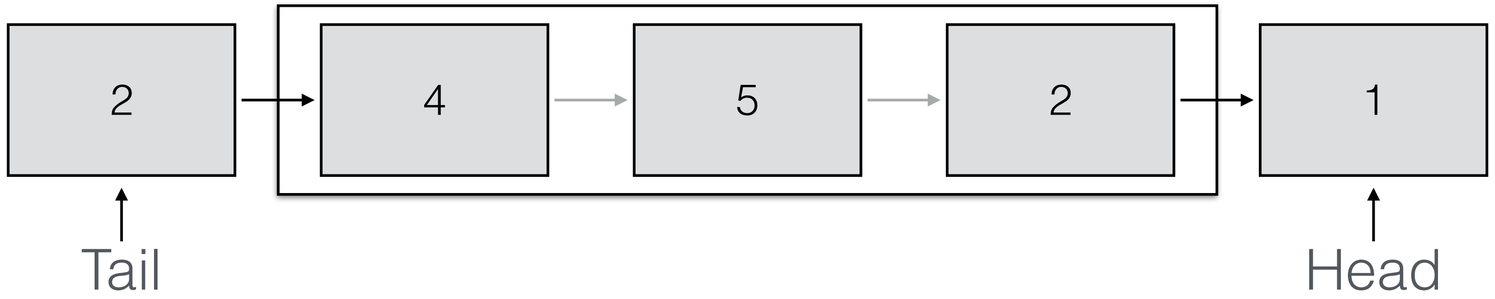


head 11

dequeue!



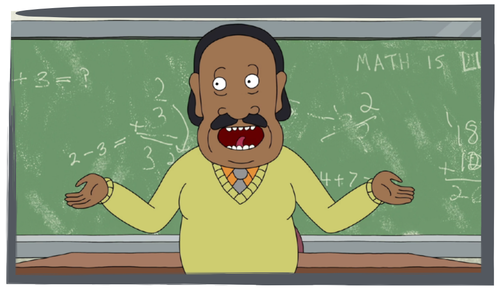
enqueue![2]



empty? False

Stacks

THEORY



EXPERIMENT

Abstract data types:

Integer: *-1, 0, 1, 2*
Boolean: *true, false*
Floating point: *11 x 10^{-2}*
Characters: $, a
Dynamic Arrays ✓
Vectors ✓
Queues ✓
Stacks

Abstract data structures

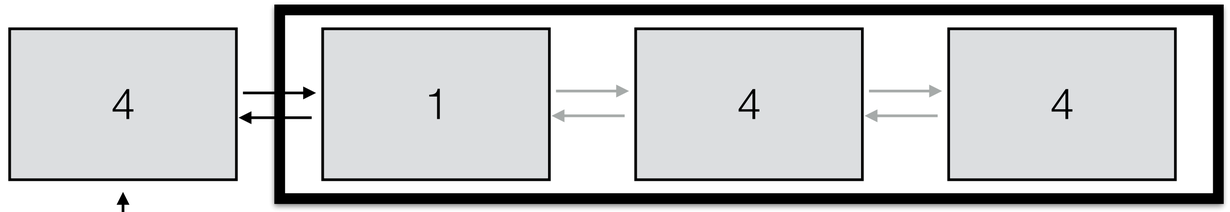
Implementation

JavaScript data types:

Number (float) `1` `NaN`
Boolean `true` `false`
String `""`
Undefined `undefined`

Objects: e.g. Arrays

Stacks



Sequential: ordered elements

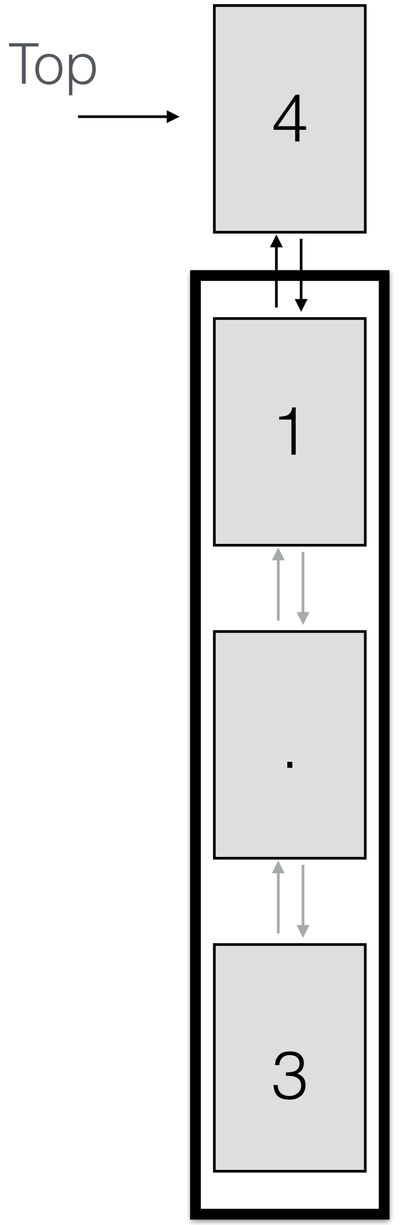
Empty stack:

LAST-IN-FIRST-OUT (LIFO)

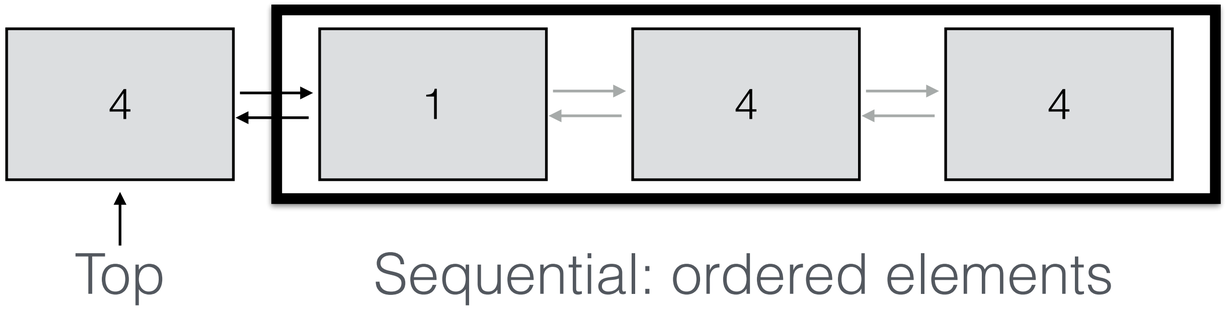
Extremely useful in organising function calls

Stacks

LAST-IN-FIRST-OUT (LIFO)



Stacks

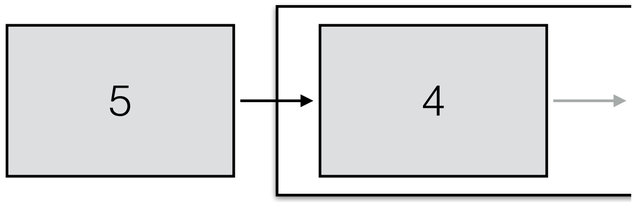


Sequential: ordered elements

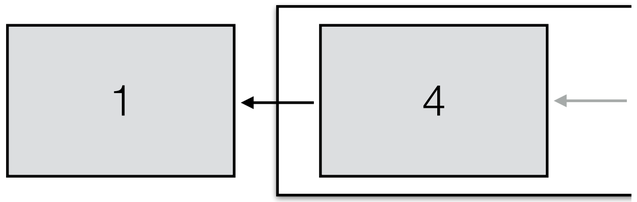
LAST-IN-FIRST-OUT (LIFO)

Number of elements can change:

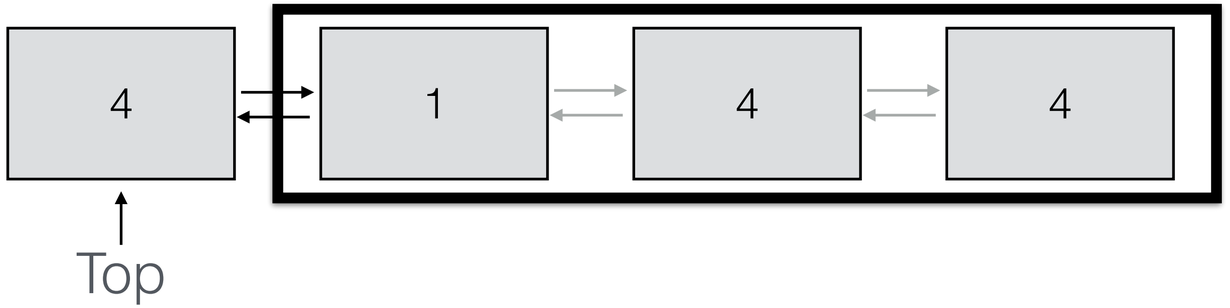
Add element to top



Remove element from top

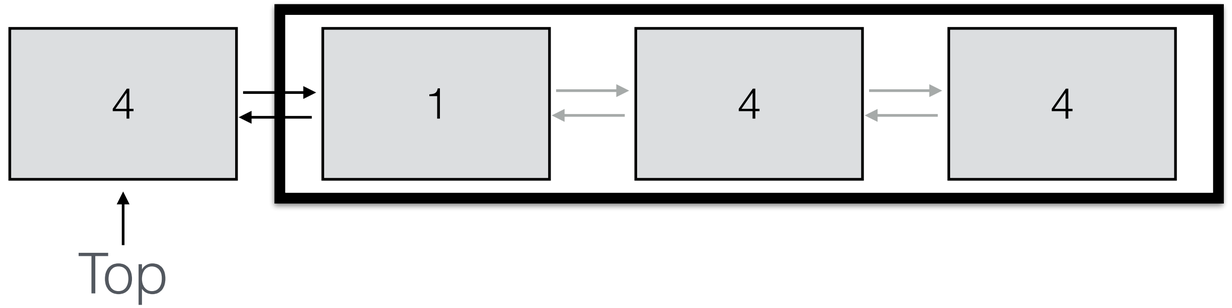


Stacks



Allowed operations:

push![o]	Adds a new element to the top with value o
peek	Reads out the value of the top element
pop!	Removes top element and returns its value
empty?	Checks if stack is empty

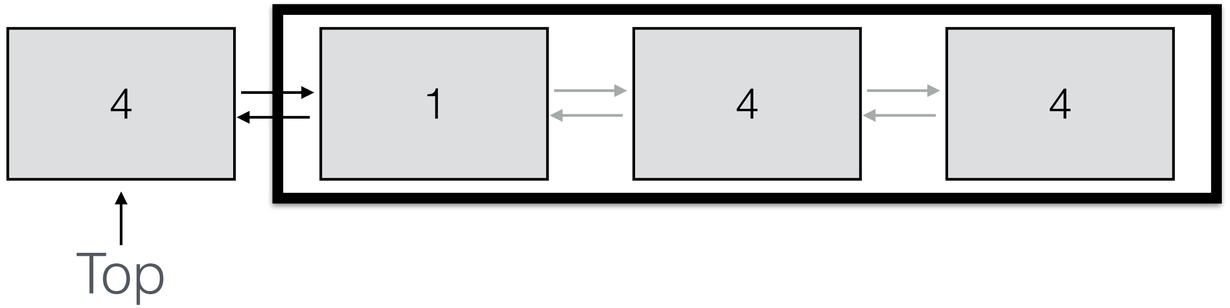


peek

push![45]

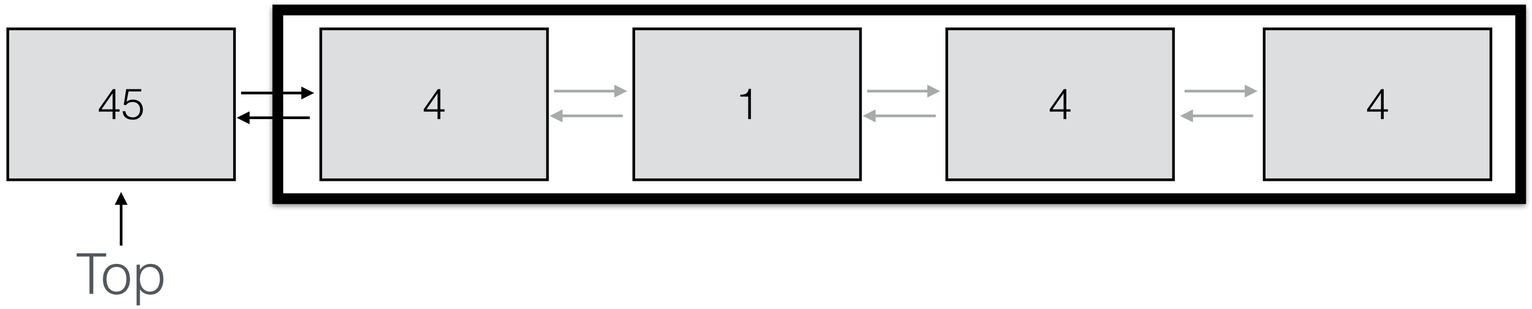
pop!

empty?

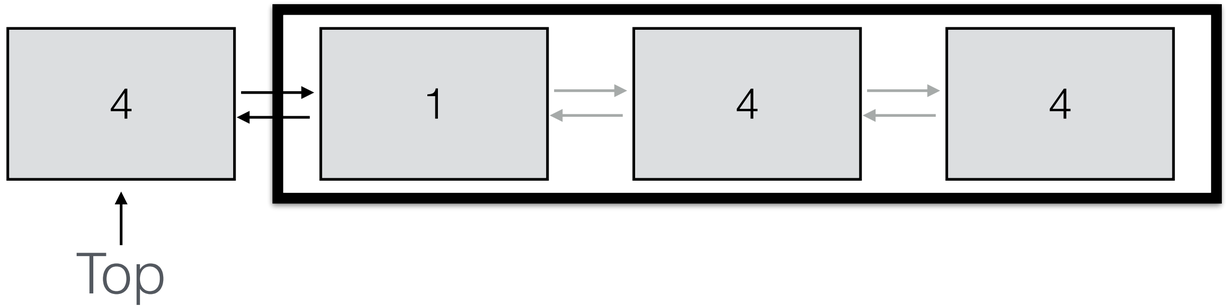


peek 4

push![45]



pop!



empty? False

What does this have to do with “full stack development”?

Not a lot: comes from software, or solution stack

Problem 3:

Given 48 toys and 42 sweets. Most number of guests invited such that all toys and sweets are distributed equally?

6

Now the guests have siblings:

Guest 1: 0 siblings
Guest 2: 2 siblings
Guest 3: 1 sibling
Guest 4: 0 siblings
Guest 5: 1 sibling
Guest 6: 1 sibling

Maximum number of guests where **6 people in total** (both guests and siblings) get toys and sweets?

Try writing some JavaScript code!