Comp 460 - Algorithms & Complexity

Spring Semester 2020 - Week 6

Dr Nick Hayward

Video - Algorithms and Data Structures

Stacks and the Call Stack - part 2

Call Stack

Source - Call Stack - YouTube

sequential execution vs event management

- also compare execution of *call stack* with event management
- common example of this is use of single thread for plain JavaScript
- compare with Node.js
    - *use of events*
    - *events management*
    - *deferred patterns*
    - *…*

Video - Algorithms and Data Structures

Event Driven Architecture - part 1

Event Driven Architecture - UP TO 2:40

Source - Event Driven Architecture - YouTube

Server-side considerations - Node.js

what is Node.js?

- Node.js is, in essence, a JavaScript runtime environment
  - *designed to be run outside of the browser*
- designed as a general purpose utility
- can be used for many different tasks including
  - *asset compilation*
  - *monitoring*
  - *scripting*
  - *web servers*
- with Node.js, role of JS is changing
  - *moving from client-side to a support role in back-end development*

Server-side considerations - Node.js

speed of Node.js

- a key advantage touted for Node.js is its speed
- many companies have noted the performance benefits of implementing Node.js
  - *including PayPal, Walmart, LinkedIn…*
- a primary reason for this speed boost is the underlying architecture of Node.js
- Node.js uses an **event-based** architecture
- instead of a threading model popular in compiled languages
- Node.js uses a single event thread by default
- all I/O is asynchronous

Video - Node.js

How Node.js works - part 1

How Node.js works - UP TO 1:32

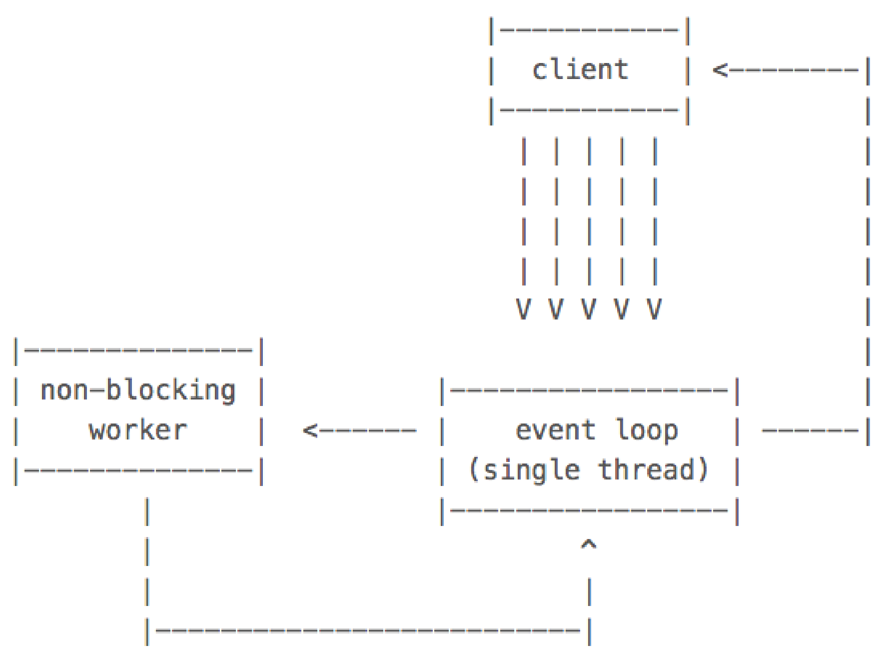
Source - How Node.js works - YouTube

Server-side considerations - Node.js

conceptual model for processing in Node.js

- how does Node.js, and its underlying processing model, actually work?
- client sends a hypertext transfer protocol, HTTP, request
  - *request or requests sent to Node.js server*
- event loop is then informed by the host OS
  - *passes applicable request and response objects as JavaScript closures*
  - *passed to associated worker functions with callbacks*
- long running jobs continue to run on various assigned worker threads
- responses are sent from the non-blocking workers back to the main event loop
  - *returned via a callback*
- event loop returns any results back to the client
  - *effectively when they're ready*

Image - Client-side and server-side computing



Node.js - conceptual model for processing

Video - Node.js

How Node.js works - part 2

How Node.js works - UP TO 3:37

Source - How Node.js works - YouTube

Server-side considerations - Node.js

threaded architecture

- concurrency allows multiple things to happen at the same time
- common practice on servers due to the nature of multiple user queries
- Java, for example, will create a new thread on each connection
  - *threading is inherently resource expensive*
- size of a thread is normally a few MB of memory
- naturally limits the number of threads that can run at the same time
- also inherently more complicated to develop platforms that are thread-safe
  - *thereby allowing for such functionality*
- due to this complexity
  - *many languages, eg: Ruby, Python, and PHP, do not have threads that allow for real concurrency*
  - *without custom binaries*
- JavaScript is similarly single-threaded
  - *able to run multiple code paths in parallel due to **events***

Server-side considerations - Node.js

event-driven architecture

- JavaScript originally designed to work within the confines of the web browser
- had to handle restrictive nature of a single thread and single process for the whole page
- synchronous blocking in code would lock up a web page from all actions
  - *JavaScript was built with this in mind*
- due to this style of I/O handling
  - *Node.js is able to handle millions of concurrent requests on a single process*
- added, using libraries, to many other existing languages
  - *Akka for Java*
  - *EventMachine for Ruby*
  - *Twisted for Python*
  - *...*
- JavaScript syntax already assumes events through its use of callbacks
- NB: if a query etc is CPU intensive instead of I/O intensive
  - *thread will be tied up*
  - *everything will be blocked as it waits for it to finish*

Video - Algorithms and Data Structures

Event Driven Architecture - part 2

Event Driven Architecture - UP TO 5:14

Source - Event Driven Architecture - YouTube

Server-side considerations - Node.js

callbacks

- in most languages
  - *send an I/O query & wait until result is returned*
  - *wait before you can continue your code procedure*
- for example, submit a query to a database for a user ID
  - *server will pause that thread/process until database returns result for ID query*
- in JS, this concept is rarely implemented as standard
- in JS, more common to pass the I/O call a **callback**
- in JS, this **callback** will need to run when task is completed
  - *eg: find a user ID and then do something, such as output to a HTML element*
- biggest difference in these approaches
  - *whilst the database is fetching the user ID query*
  - *thread is free to do whatever else might be useful*
  - *eg: accept another web request, listen to a different event…*
- this is one of the reasons that Node.js returns good benchmarks and is easily scaled
- NB: makes Node.js well suited for I/O heavy and intensive scenarios

recursion and the call stack - part 1

- stack may be used to represent execution logic for recursive function
- consider following Python code for calculating factorial
  - *e.g. for 3 ! - factorial(3)*

```
def factor(x):
    if x == 1:
        return 1
    else:
        return x * factor(x-1)

print(factor(3))
```

- check logic for pattern to recursive calls
  - *and call stack they use...*

recursion and the call stack - part 2

- e.g. call function with passed value of `3`
  - *outline call stack and recursive execution*

app execution

- inital passed value of `3`
  - *`x = 3`*

```
------------
|  factor  |
|----------|
| x  |  3  |
------------
```

- then we check `x` against a value of `1`
  - *not `1`*
  - *continue to `else`*
  - *return `x` multiplied by `factor(x-1)` - first recursive call*
    - `factor(2)` is added to the call stack and executed

```
------------
|  factor  |
|----------|
| x  |  2  |
------------
|  factor  |
|----------|
| x  |  3  |
------------
```

recursion and the call stack - part 3

```
------------
|  factor  |
|----------|
| x  |  2  |  ----
------------     |
|  factor  |     |-- n.b. both calls have a variable `x` with different values
|----------|     |
| x  |  3  |  ----
------------
```

- we’re now executing top of call stack - `factor(2)`
  - *`x = 2`*
  - *check x against a value of 1*
  - *continue to `else`*
  - *return x multiplied by `factor(x-1)` - second recursive call*
    - `factor(1)` is added to the call stack and executed

recursion and the call stack - part 4

- we now have three calls in the stack

```
------------
|  factor  |
|----------|
| x  |  1  |  ----
------------     |
|  factor  |     |-- n.b. value cannot be accessed outside function context
|----------|     |
| x  |  2  |  ----
------------     |
|  factor  |     |-- n.b. both calls have a variable `x` with different values
|----------|     |
| x  |  3  |  ----
------------
```

- we're now executing the top of the stack - `factor(1)`
  - *`x = 1`*
  - *we can now return 1*
  - *pop `factor(1)` from call stack*
  - *this is the first call we may `return` from...*
- now return to second recursive call
  - *`return 2 * 1`*
  - *pop `factor(2)` from call stack*
- now return to first recursive call
  - *`return 3 * 2`*
  - *pop `factor(3)` from call stack*
- print 6

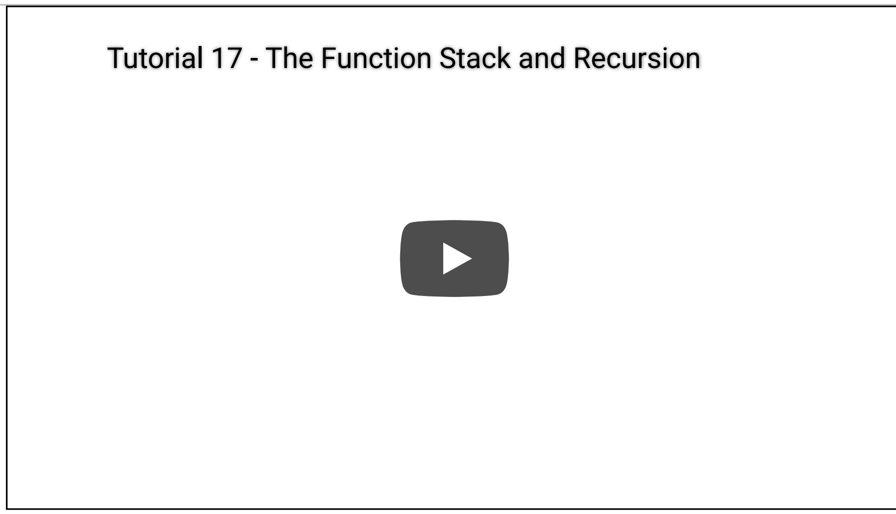
recursion and the call stack - part 5

- pseudocode outline for pattern of execution for this function
  - *e.g. 3! - factorial(3)*

```
factor(3)
    x = 3
    return x * factor(3-1) // recurse 1
    factor(2)
        x = 2
        return x * factor(2-1) // recurse 2
        factor(1)
            x = 1
            return 1 // pop factor(1) from call stack
        return 2 * 1 // 1 is returned from recurse 2
        return 2 // pop factor(2) from call stack
    return 3 * 2 // 2 is returned from recurse 1
    return 6 // pop factor(3) from call stack

print 6 // stack now clear, execution ends...
```

Video - Algorithms and Data Structures

Recursion and the Call Stack - Java



Recursion and the Call Stack - Java

Source - Recursion and the Call Stack - Java - YouTube

recursion and the call stack - part 6

- we may see how useful a stack is to the execution of recursion
  - *used as a record of execution*
  - *a clear order of remaining execution*
- call stack acts as record of half-completed function call
  - *each call with its own record of incomplete execution waiting to finish*
- key benefit to this stack usage
  - *no need to keep a manual record*
  - *e.g. of executed and incomplete function calls*
  - *call stack keeps record...*
- using call stack is convenient
  - *but it does come with a cost*
  - *e.g. for recursive calls*
- adding information to call stack requires memory usage
  - *this can fill quickly*
  - *e.g. when we use recursive function calls*
- if memory usage is causing an application's execution to freeze or crash
  - *consider modifying recursion to iteration*
  - *use a cache for certain functions and function calls*
    - i.e. *memoisation*
    - identify duplicate calculations, calls...
  - *use an option such as tail recursion*

Video - Algorithms and Data Structures

Recursion, the Call Stack, and Overflow…part I

!!Con 2019- Tail Call Optimization: The Musical!! - UP TO 3:55

Source - !!Con 2019- Tail Call Optimization: The Musical!! - YouTube

recursion and the call stack - part 7

- an example of tail recursion for `3!` - factorial(3)

```
def factor(x, tail):
    print("factor x =",x)
    if x == 1:
        print("return from (x == 1) = 1")
        return tail
    else:
        print("x =",x)
        return factor(x - 1, x * tail)

# set initial tail to 1
print(factor(3, 1))
```

Algorithms and Data Structures

recursion and the call stack - part 8

- pseudocode outline for pattern of execution for tail recursion
- e.g. `3!` - factorial(3)

```
factor(3, 1)
    x = 3, tail = 1
    return factor(3 - 1, 3 * 1) // recurse 1
    factor(2, 3)
        x = 2, tail = 3
        return factor(2 - 1, 2 * 3) // recurse 2
        factor(1, 6)
            x = 1, tail = 6
            return 6 // pop factor(1, 6) from call stack
        return 6 // pop factor(2, 3) from call stack
    return 6 // pop factor(3, 1) from call stack

print 6 // stack now clear, execution ends
```

Video - Algorithms and Data Structures

Recursion, the Call Stack, and Overflow…part II

!!Con 2019- Tail Call Optimization: The Musical!! - UP TO 7:58

Source - !!Con 2019- Tail Call Optimization: The Musical!! - YouTube

recursion and the call stack - part 9

- stack may be used to represent execution logic
  - *e.g. for a recursive function in JavaScript*
- code example uses a *call stack* to ensure expected execution

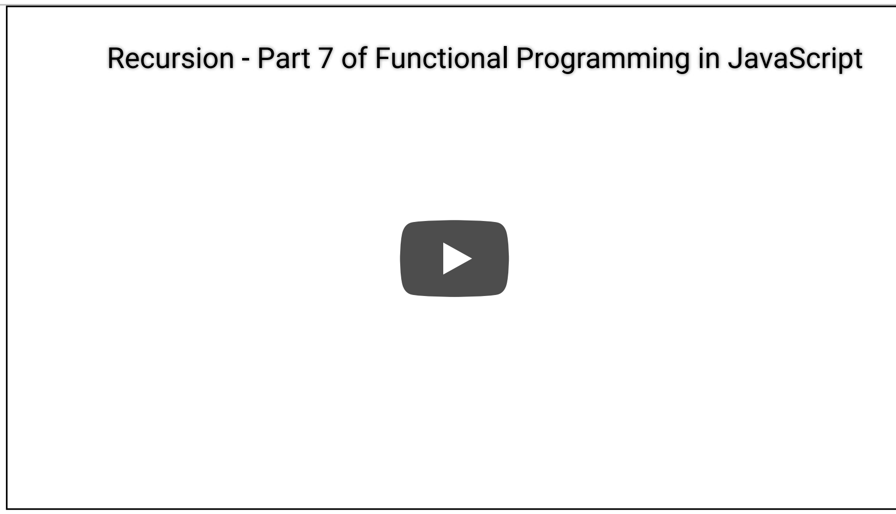
```
function findSolution(target) {
    function find(current, history) {
        if (current == target) {
            return history;
        } else if (current > target) {
            return null;
        } else {
            return find(current + 5, `(${history} + 5)`) || find(current * 3, `(${history} * 3)`);
        }
    }
    return find(1, "1");
}

console.log(findSolution(24));
```

recursion and the call stack - part 10

- initial `findSolution()` function is called
  - *passed parameter of* `24` *is the value to check*
- function returns an executed `find()` function
  - *initial test values for current and history*
- part of this function's execution
  - *checks initial values until it reaches* `else` *part of conditional statement*
- returns `find()` function
  - *called recursively*
  - *initially checking against addition of 5*
  - *continues to check possible values with `+ 5`*
  - *either succeeds or moves onto right side of logical OR,* `||`*,* logical OR checks with `* 3`*
- it will either succeed or fail with these recursive checks
- structure that permits this recursion to execute
  - *structure is the call stack*
- *call stack* provides a defined pattern to execution
  - *pattern allows the code to run as expected*

Video - Algorithms and Data Structures

Recursion for Fun - part 2



Recursion and Fun - JavaScript - UP TO 14:46

Source - Recursion and Fun - JavaScript - YouTube

stack operations - part 1

- we may define a stack as a simple list of elements
  - *may be accessed from only one end*
  - *known as the top of the stack*
- i.e. refer to data structure as *last in, first out*
- known limitation of this structure
  - *lack of access to elements not at top of stack*
- a simple difference between structures
  - *i.e. basic list or array and specific stack*
- to access the bottom element
  - *all elements above must first be popped*

stack operations - part 2

- stack operations are simple, e.g.
  - *add elements to the top*
  - *pop elements from the top*
- also means specific restrictions must be in place
  - *ensures only these operations are allowed*
  - *i.e to define usage for the data structure*
  - *if not, it ceases to be a stack*

stack operations - part 3

- complementary operations are commonly available
  - *may vary relative to language implementation*
- e.g. a stack may permit the following
  - *view the top element in the stack*
  - *this is not pop - element is not removed from stack*
  - *operation known as peeking*
  - *clear operation will remove all elements from stack*
  - `length` *property returns number of elements in stack*
  - `empty` *returns whether stack has any values or not*

Video - Algorithms and Data Structures

Recursion, the Call Stack, and Overflow…part III

!!Con 2019- Tail Call Optimization: The Musical!! - UP TO END

Source - !!Con 2019- Tail Call Optimization: The Musical!! - YouTube

example implementations

- choose various existing data structures to define custom stack
- e.g. might use an array or list
- choice will often depend on support in chosen programming language
- e.g. in JS - common option is an `array` object
- define constructor for `stack` object
  - *then extend prototype for custom properties and methods*

stack constructor

- initial constructor is as follows

```
// CONSTRUCTOR = Stack object
function Stack() {
    /* define instance properties for stack
    * - empty array for instantiated stack
    * - options might include max length, restricted data type &c.
    */
    this.store = [];
}
```

- instantiate a basic Stack object
  - *simply defining an empty array*
  - *use array as store for Stack data structure*
- constructor may be updated to include
  - *type checks and restrictions*
  - *initial values for Stack*
  - *required access context*
  - *...*

extend the prototype

- initially extend Prototype for this object
- add required functionality for a basic stack
- e.g. define functions for the following
  - *add data*
  - *delete data*
  - *get size of Stack*

prototype - add data

- *add data* function
  - *add passed data to top of stack*

```
// PROTOTYPE - add method for value pushed to top of stack
Stack.prototype.add = function (value) {
    this.store.push(value);
    console.log(`value added = ${value}`);
}
```

- underlying `store` object is an array for Stack
  - *use default* `push()` *method to add required data*

prototype - delete data

- *delete data* function defined as follows

```
Stack.prototype.delete = function () {
    const deletedValue = this.store.pop();
    console.log(`last value deleted = ${deletedValue}`);
}
```

- same as *add data*
  - *use default `pop()` method for `store` array*
  - *added to custom Stack data structure*

prototype - size of Stack

- also define `size()` function for Stack
  - *use built-in Array property for `length`*

```
Stack.prototype.size = function () {
    const size = this.store.length;
    console.log(`store size = ${size}`);
}
```

prototype - peek Stack

- useful option is *peeking* at the top of Stack
- e.g.

```
Stack.prototype.peek = function () {
    const peekValue = this.store[(this.store.length-1)]
    console.log(`top value = ${peekValue}`);
}
```

- function will return copy of top value
    - *will not delete item from Stack and underlying `store` array*

prototype - clear stack

- common operation for a Stack is to clear all entries,
  - *yet preserve the Stack itself*
- i.e. resetting store array for instantiated Stack object
- e.g.

```
Stack.prototype.clear = function () {
    // resets Stack's array store - clears all items
    this.store = [];
}
```

prototype - check empty stack - part 1

- check an instantiated Stack object for entries
  - *i.e. determine if stack is empty or not*

```
Stack.prototype.empty = function () {
    if (this.store.length === 0) {
        return true;
    } else {
        return false;
    }
}
```

prototype - check empty stack - part 2

- conditional logic has been placed in this function
  - *i.e. not passed down chain of logic to requesting application call*
  - *means function is self-contained*
- function returns valid response regardless of execution context
- as we develop Stack's Prototype methods
  - *add further restrictions and controls*
  - *clearly defines how to use this data structure*
- also define what and how may be returned
  - *custom data structure customised to context, usage...*

Video - Algorithms and Data Structures

Prototype in JavaScript - part 1

Prototype in JavaScript - UP TO 1:00

Source - Prototypes in JavaScript - YouTube

Video - Algorithms and Data Structures

Prototype in JavaScript - part 2

Prototype in JavaScript - UP TO 6:41

Source - Prototypes in JavaScript - YouTube

control access to the stack - part 1

- Stack object and methods
  - *now working as expected*
- Stack is still open to mis-use
  - *due to array object in the Stack*
- restrict and control access to this Stack data structure
  - *e.g. using a Proxy*

control access to the stack - part 2

- to use a Proxy with our Stack constructor
  - *define a custom `construct` trap*
- may also use Reflect API to define defaults for handlers
- `construct` trap intercepts calls
  - *i.e. to defined `new` operator for a given constructor*

control access to the stack - part 3

- define initial Proxy wrapper for passed constructor

```
/*
* PROXY
*/
function proxyConstruct(constructor) {

    const handler = {
        construct(constructor, args) {
            console.log('proxy constructor...');
            // const stack = Reflect.construct(constructor, args);
            return new constructor(...args);
        }
    };
    return new Proxy(constructor, handler);
}
```

control access to the stack - part 4

- then pass basic `Stack` constructor to the proxy

```
// proxy wrapper for Stack constructor
const proxiedStack = new proxyConstruct(Stack);
// instantiate proxied Stack & check store...
console.log(new proxiedStack().store);
```

control access to the stack - part 5

- instantiation of a proxied Stack object
  - *allows us to wrap constructor for Stack*
- may still use prototype methods for instantiated Stack object
- benefit of using a proxy for the constructor
  - *control of initial object instantiation*
- i.e. if object cannot be instantiated
  - *access to Prototype methods becomes irrelevant*

Video - Algorithms and Data Structures

Proxy in JavaScript

Using a JavaScript Proxy - UP TO 3:28

Source - Proxy in JavaScript - YouTube

Resources

JavaScript

- MDN - Array
- MDN - Prototype
- MDN - Proxy
- Prototypes in JavaScript - YouTube
- Proxy in JavaScript - YouTube

Python

- Stacks - YouTube

Various

- !!Con 2019- Tail Call Optimization: The Musical!! - YouTube
- Call Stack - YouTube
- Event Driven Architecture - YouTube
- Memory Manager - YouTube
- Recursion and Fun - JavaScript - YouTube
- Recursion and the Call Stack - Java - YouTube