Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 4

Dr Nick Hayward

Project outline & mockup assessment

Course total = 15%

- begin outline and design of an application
  - *built from scratch - languages include*
    - JavaScript
    - Python
    - C
    - …
  - *builds upon examples, technology outlined during first part of semester*
  - *must implement algorithms & data structures*
  - *purpose, scope &c. is group's choice*
  - *NO blogs, to-do lists, note-taking…*
    - chosen topic requires approval
  - *presentation should include mockup designs and concepts*

Project mockup demo

Assessment will include the following:

- brief presentation or demonstration of current project work
  - *~5 to 10 minutes per group*
  - *analysis of work conducted so far*
  - *presentation and demonstration*
    - outline current state of app concept and design
    - show prototypes and designs
  - *due Tuesday 11th & Thursday 13th February 2020 @ 10am*

Practical usage - JavaScript Arrays

common operations on arrays - test array items

- check one or more array items to see if they match certain conditions
- help with this requirement, JS provides some useful built-in functions, `every` and `some`
  - *`every` method - pass a callback, which is called for each specified property in the array*
    - e.g. check if all properties have a specified value &c.
    - returns a boolean for the check - `true` for *all* properties matching specified value, otherwise `false`
  - *`some` method - pass a callback, which is called for each specified property in the array*
    - e.g. check at least one property matches a specified value
    - returns a boolean - `true` for at least one match, `false` for zero matches
  - *e.g.*

```
// array
const archives = [
  {name: 'waldzell', type: 'game', location: 'castalia'},
  {name: 'mariafels', type: 'benedictine'}
];

// check archives - `every` returns true for all match, `false` for a single error/omission
const everyName = archives.every(archive => 'name' in archive);
// check boolean return for `every` method in everyName
everyName === true ? console.log(`each archive has a name`) : console.log(`at least one archive is
         unnamed...`);

// check archives - `some` return true for a single match, `false` for no matches
const singleLocation = archives.some(archive => 'location' in archive);
// check boolean return value
singleLocation === true ? console.log(`at least one archive has a location`) : console.log(`no archive
         has a location...`);
```

Practical usage - JavaScript Arrays

common operations on arrays - searching arrays - part 1

- also search and find items in JS arrays
- JS provides another built-in function, `find`
- e.g.

```
// array
const archives = [
  {name: 'waldzell', type: 'game', location: 'castalia'},
  {name: 'mariafels', type: 'benedictine', location: 'czech'}
];

// find object in array
const locations = archives.find(archive => {
  // return object - not found simply returns undefined
  return archive.location === 'castalia';
});

// check search - check undefined or log archive name to console
locations !== undefined ? console.log(`archive in castalia = ${locations.name}`) :
         console.log(`location and archive not found...`);
```

- if the requested item can be found
  - *matching object will be returned*
- otherwise, the `find` method will simply return `undefined`

Practical usage - JavaScript Arrays

common operations on arrays - searching arrays - part 2

- `find` will return first matching item
  - *regardless of the number of matches*
- to search an array for all matches we can use the `filter` method instead
- e.g.

```
// filter array and return multiple matches
const filterTypes = archives.filter(archive => 'type' in archive);
```

- returns all matching items
  - *simply check length of return object,*
  - *and iterate through the results*
- e.g.

```
// check filter returns
if (filterTypes.length >= 1 ) {
  for(let archive of filterTypes) {
    console.log(`archive name = ${archive.name} and type = ${archive.type}`);
  }
} else {
  console.log(`archive types are not available...`);
}
```

Practical usage - JavaScript Arrays

common operations on arrays - searching arrays - part 3

- also possible to filter an array by index using following methods
  - `indexOf` = *find the index of a given item*
  - *e.g.*

```
const waldzellIndex = archives.indexOf('waldzell');
```

- `lastIndexOf` = find last index of multiple matched items
- e.g.

```
const waldzellIndex = archives.lastIndexOf('waldzell');
```

- `findIndex` = effectively works the same as `find` but returns an index value
- e.g.

```
const waldzellIndex = archives.findIndex(archive => archive === 'waldzell');
```

Algorithms and Data Structures

recursion and patterns

- as seen with custom linked list data structure
  - *access and iteration is a key consideration*
  - *e.g. general use, effective re-use…*
- e.g. no existing index for each item in the linked list
  - *choose to use a pattern such as recursion to check and access the list*
- recursion is a common technique used in design of many algorithms
  - *and app development in general*
- key benefit of recursion is option to define a *base* case and *recursive* case
  - *to help solve a given problem*
- recursion commonly provides an elegant way to solve complex problems
- its usage may also be seen as somewhat divisive
  - *i.e. controversial depending upon context*

Video - Algorithms and Data Structures

Recursion

Recursion - UP TO 2:27

Source - Recursion - YouTube

Algorithms and Data Structures

recursion for fun - part 1

- consider a jar of *10,000* sweets with various colours
  - *only a single winning gold sweet*
- we might design an algorithm with recursion
- initially two procedures we may define
  - *pick a sweet*
  - *check the sweet's colour*
- second procedure may also be used to check sweet's colour
  - *i.e. does it match prize gold sweet*
- defines whether we need a recursive call to first procedure
  - *or process has finished*

Algorithms and Data Structures

recursion for fun - part 2

- outline required steps to achieve overall process of finding gold sweet
- procedures will include various tasks to help resolve overall process
- e.g.

```
1. open the jar of sweets to begin the search
2. choose a sweet from the jar and check its colour
3. if the sweet is *not* gold, add it to a second jar
4. if the sweet is *gold*, the prize has been found and the process ends
5. repeat...
```

Algorithms and Data Structures

recursion for fun - part 3

- define a general series of steps as follows

```
1. check each sweet in the jar
2. if the sweet is *not* gold...repeat step 1
3. if the sweet is *gold*, you win the prize...
```

- we can see difference between these initial approaches to solving same problem
- first example might use a simple `while` loop, e.g.
  - *`while the jar of sweets is not empty, choose a sweet and check its colour...`*
- second example uses *recursion*
  - *i.e. keep calling first step, or function, until a break is achieved*

Algorithms and Data Structures

recursion for fun - part 4

- consider this problem using two sample implementations
  - *reflect sample outlines*
- first example uses a `while` loop
  - *outlined as follows using pseudocode*

```
search_sweets(main_jar)
  while main_jar is not empty
    sweet = main_jar.pick_a_sweet()
    if sweet.is_not_gold()
      second_jar.add_sweet(sweet)
    else
      print "gold sweet found, you win!"
      exit
```

- while main sweet jar still contains sweets
  - *`pick_a_sweet()` function will choose a sweet*
  - *function will need to return chosen sweet*
  - *check its colour and remove it from the main jar*
- then check current sweet's colour
  - *add it to the second jar if it's not gold*
- if sweet is gold, you win and the loop will exit

Algorithms and Data Structures

recursion for fun - part 5

- example in JavaScript is as follows

```
// FN: search passed jar of sweets
function searchSweets(main_jar) {
    // declare
    const second_jar = [];
    while (main_jar.length > 0) {
        // pop last item in main_jar array - or use shift() for first item...
        const sweet = main_jar.pop();
        // check if sweet is gold
        if (sweet !== 'gold') {
          console.log(`${sweet} sweet is not gold...`);
          // if not gold, add to second jar
          second_jar.push(sweet);
        } else {
          // you win...gold sweet found in main jar
          console.log(`you win, ${sweet} sweet found!`);
          // exit loop...
          return;
        }
    }
}

// define main jar with variety of sweets
const main_jar = ['blue', 'green', 'red', 'orange', 'gold', 'yellow', 'pink'];
// check main jar for a gold sweet...
searchSweets(main_jar);
```

- able to loop through passed jar of sweets
  - *check each one until we find winning gold sweet*
- we make a number of assumptions
  - *i.e. passed jar as an array, value of sweet's colour as a string....*
- also a slow search
  - *only have information for required colour of winning sweet*
  - *need to iterate through whole jar*

Algorithms and Data Structures

recursion for fun - part 6

- second implementation
  - *consider a solution using recursion*
- initially consider algorithm using pseudocode

```
search_sweets(main_jar)
  if main_jar is not empty
    sweet = main_jar.pick_a_sweet()
    if sweet.is_not_gold()
      second_jar.add_sweet(sweet)
      search_sweets(main_jar)
    else gold sweet found
  else jar is empty
```

- recursive example follows same underlying pattern as `while` option
- instead of loop we may now call `search_sweets()` method
  - *for all sweets in the jar*
  - *or until we find the gold sweet*

Algorithms and Data Structures

recursion for fun - part 7

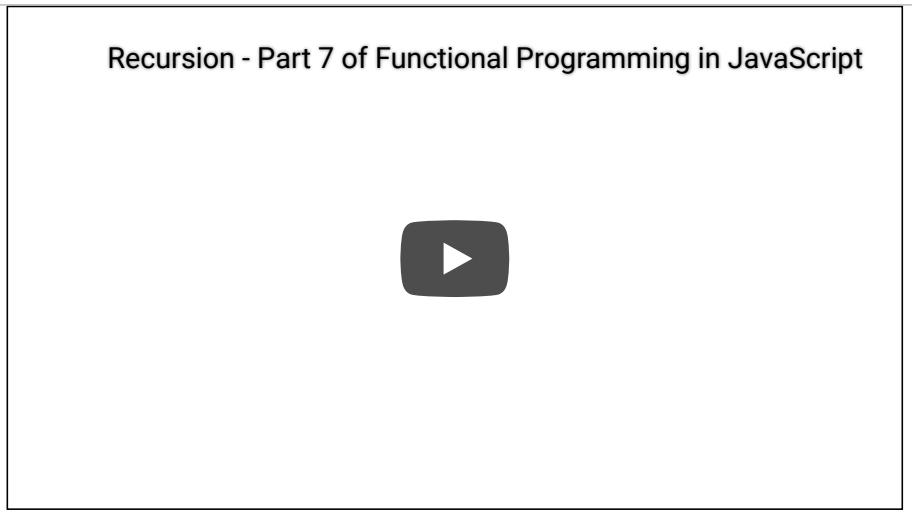
- implement this algorithm using recursion with JavaScript

```
// FN: search passed jar of sweets
function searchSweets(main_jar) {
    // declare second_jar for removed sweets...
    const second_jar = [];
    // check main_jar has sweets left...
    if (main_jar.length > 0) {
        // get a sweet and remove from main_jar
        const sweet = main_jar.pop();
        // check sweet colour - gold wins prize...
        if (sweet !== 'gold') {
          console.log(`${sweet} sweet is not gold...`);
          // if not gold, add to second jar
         second_jar.push(sweet);
         // recursive call - pass remainder of main_jar
         searchSweets(main_jar);
       } else {
         // you win...gold sweet found in main jar
        console.log(`you win, ${sweet} sweet found!`);
      }
    } else {
        // main_jar is empty - no gold sweet found...
        console.log(`jar is now empty...you lose, try again!`);
    }l,

}

// define main jar with variety of sweets
const main_jar = ['blue', 'green', 'red', 'orange', 'golden', 'yellow', 'pink'];
// check main jar for a gold sweet...
searchSweets(main_jar);
```

- need to check availability of sweets in jar
  - *allows us to check for winning gold sweet*
- conditional statements follow same pattern as previous JavaScript example
- may now recursively call `searchSweets()` function

Video - Algorithms and Data Structures

Recursion for Fun



Recursion and Fun - JavaScript - UP TO 7:32

Source - Recursion and Fun - JavaScript - YouTube

recursion - linked list

- consider earlier linked list
  - *may define an algorithm for implementing procedures on a list using recursion*
- e.g. following algorithm may be outlined to find last element of defined list

```
last(list) {
  if ( isEmpty(list) )
    error('error - list empty...')
  else if ( isEmpty(rest(list)) )
    return first(list)
  else
    return last(rest(list))
}
```

- if we consider *complexity* of this algorithm
  - *the procedure has **linear time complexity***
  - *i.e. if length of list is increased, execution time will likewise increase by same factor*
- performance does not mean that lists are always inferior to arrays
- lists are not an ideal data structure for certain uses
  - *regardless of applied common algorithms*
- i.e. when an application needs to access last element of a longer list

Algorithms and Data Structures

stacks and the call stack

- a brief segue into *stacks*
  - *in particular a consideration of call stack used with program execution*
- key to understanding execution of many algorithms in code
  - *e.g. a better understanding of recursion for development*
- if we don't understand how order of execution is tallied and reconciled by applications
  - *we'll struggle to clearly understand nature of algorithms*
  - *and their general usage*

Algorithms and Data Structures

stacks - intro - part 1

- *stack* data structure commonly represented as
  - *a modified, restricted array or list*
  - *used in various programming and scripting languages*
- *stack* is an efficient data structure
  - *used for many development purposes*
- Data may only be added and removed from top of structure
  - *affords ease of implementation and speed*
- commonly refer to a stack as *last in, first out*
  - *or LIFO*
- stack of plates in a restaurant kitchen
  - *a good analogy of this structure's usage*
  - *dirty plates are added to top of stack*
  - *a dishwasher will wash these plates from the top down*
  - *so last plate on the stack will be washed first*

Video - Algorithms and Data Structures

Python Stacks

Python Stacks

Source - Stack - YouTube

Algorithms and Data Structures

stacks - intro - part 2

- *push* method may add a value to the end of an array
- *pop* method may be used to remove last value in array
- stack is a data structure
  - *allows values to be pushed into it*
  - *and popped from it as needed*
- last item added is now the first removed
- Stacks may be used for many different purposes in development * e.g. execution requests to function calls
- a stack is a common structure
  - *e.g. storing lists of items for ordered usage*

stacks - intro - part 3

- similar in nature to a linked list
- restricted use of a *Stack* normally defines alternative names for primitive operators
- conceptually - commonly define construction and access as follows
  - *i.e. for a custom implementation of a Stack data structure*
- constructor to enable instantiation of Stack data structure
  - *e.g.* `EmptyStack` *or simply* `Stack`
- basic selectors for required default functionality
  - `top(stack)` *- return top element from stack*
  - `pop(stack)` *- returns stack without top element*

stacks - intro - part 4

- specific implementation of such selectors may vary from language to language
- fundamental concept of the data structure remains consistent
- also see similar `true` and expected relationships for a stack
- similar to those seen for a linked list
- e.g.
  - *`isEmpty(EmptyStack)`*
  - *`not isEmpty(push(x, s))` (for any `x` and `s`)*
  - *`top(push(x, s)) = x`*
  - *`pop(push(x, s)) = s`*

Algorithms and Data Structures

stacks - intro - part 5

- conceptually define following as useful comparison
  - *a* `list` *and* `stack`

structure	constructors	selectors	condition
list	EmptyList, MakeList	first, rest	isEmpty
stack	EmptyStack, push	top, pop	isEmpty

Video - Algorithms and Data Structures

Stacks and the Call Stack - part 1

Call Stack - UP TO 4:50

Source - Call Stack - YouTube

Algorithms and Data Structures

stacks and the call stack - part 1

- as a computer executes code, commands, and various logic...
- uses an internal *stack*
  - *the call stack*
  - *records and checks order of execution*
- e.g. consider the following Python code

```
def greetings(name):
    print "hello, " + name + "!"
    more_greetings(name)
    print "ready to leave..."
    goodbye()
```

- as we execute `greetings()` function
  - *also execute other defined custom functions*
  - *i.e. `more_greetings()` and `goodbye()`*
- also execute `print` function - internal to Python
- initially consider custom functions relative to call stack

```
def more_greetings(name):
    print "how are you, " + name + "?"

def goodbye():
    print "take care, goodbye!"
```

Algorithms and Data Structures

stacks and the call stack - part 2

- as we execute function `greetings()`
  - *system will allow memory*
  - *i.e. a container or box specific to that function call*
- memory will contain function, variable `name` with passed value
- e.g.

```
----------------------
|      greetings     |
|--------------------|
| name:    |  daisy  |
----------------------
```

- every time we make a function call
  - *system will save values for all of the variables for that call*
- our code printed the inital greeting

```
"hello, Daisy!"
```

stacks and the call stack - part 3

- then execute another function call `more_greetings()`
- system will again allocate memory for this specific function call

```
---------------------
|   more_greetings   |
|--------------------|
| name:    |  daisy  |
---------------------
|      greetings     |
|--------------------|
| name:    |  daisy  |
---------------------
```

stacks and the call stack - part 4

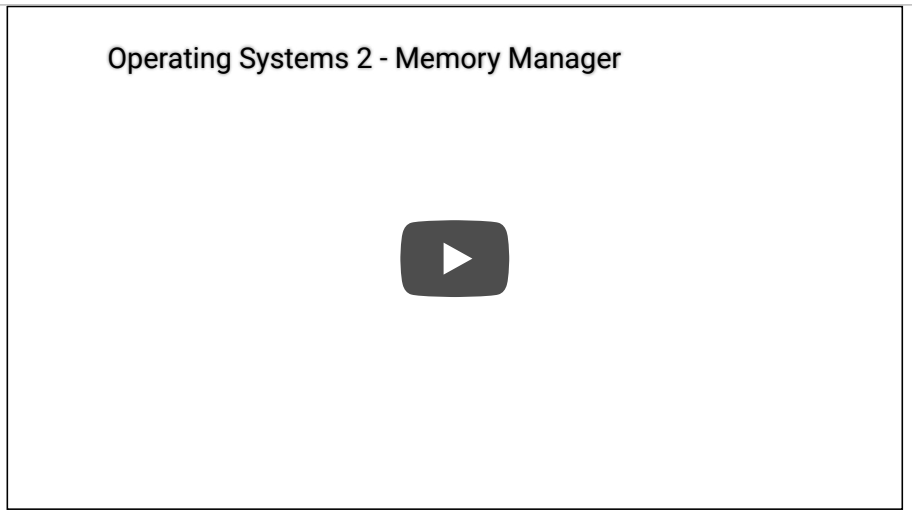
- system is using a *stack* for this memory storage
- i.e. its stacking boxes of memory for current function calls
- then print the second greeting

```
"how are you, Daisy?"
```

- return to function call
- top of call stack
  - *i.e. box of memory for* `more_greetings`
  - *now popped off stack*

Video - Algorithms and Data Structures

Stacks and the Call Stack



Memory Manager - UP TO 7:11

Source - Memory Manager - YouTube

stacks and the call stack - part 5

- stack returns to the following

```
----------------------
|      greetings     |
|--------------------|
| name:    |  daisy  |
----------------------
```

- as we called `more_greetings()` function
  - *initial function for `greetings()` only partially completed*
- i.e. call a function from another function
  - *current function calling execution is paused*
  - *paused in partially completed state*
- values of defined variables for function still stored in memory

Algorithms and Data Structures

stacks and the call stack - part 6

- now back to `greetings()` function
- may continue to execute that function
- e.g. continue by printing

```
"ready to leave..."
```

- then call `goodbye()` functon
- function will be added to top of stack
  - *then executed*
  - *then exit from function back to `greetings()` function*
- at end of initial function call for `greetings()`
  - *exit that function and stack is now clear*
- call stack may store required variables for multiple functions
  - *required order of execution for defined code*

Algorithms and Data Structures

stacks and the call stack - part 7

- order of execution for functions and application code in JavaScript
  - *defined by call stack*
- *call stack* provides context for ordered execution of code
- e.g. a conceptual stack of ordered execution

```
not in function
    in greetings function
        in console.log
    in greetings function
        in moreGreetings function
    in greetings function
        in console.log
    in greetings function
        in goodbye function
not in function
    in console.log
not in function
```

Algorithms and Data Structures

stacks and the call stack - part 8

- this stack represents the following sample JavaScript code

```
function greetings(name) {
  console.log("hello " + name + "!");
  moreGreetings(name);
  console.log("ready to leave...");
  goodbye();
}

function moreGreetings(name) {
  console.log("how are you, " + name + "?");
}

function goodbye() {
  console.log("take care, goodbye!")
}

greetings("Daisy");
console.log("now finished...");
```

- context for this code's execution stored in *call stack*

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*

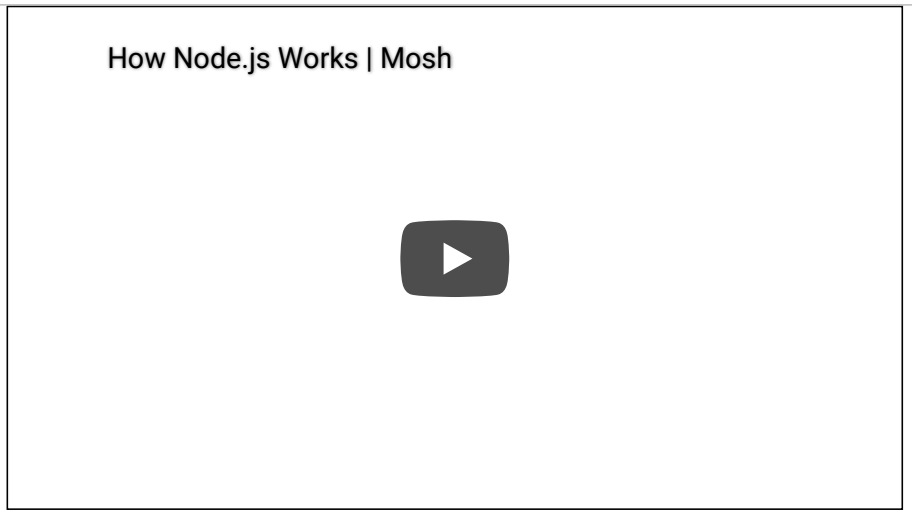
Server-side considerations - Node.js

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

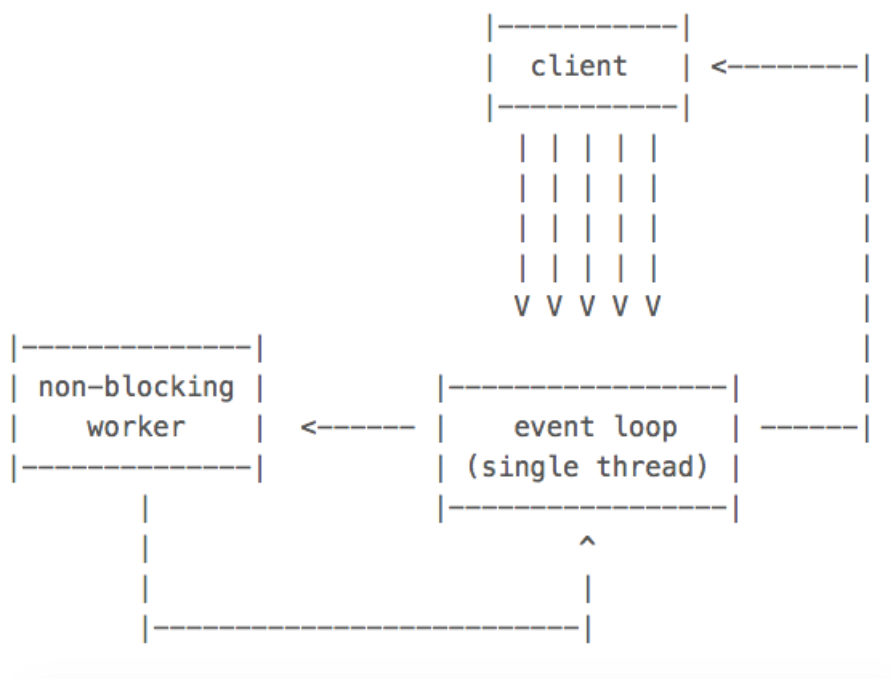
Source - How Node.js works - YouTube

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*

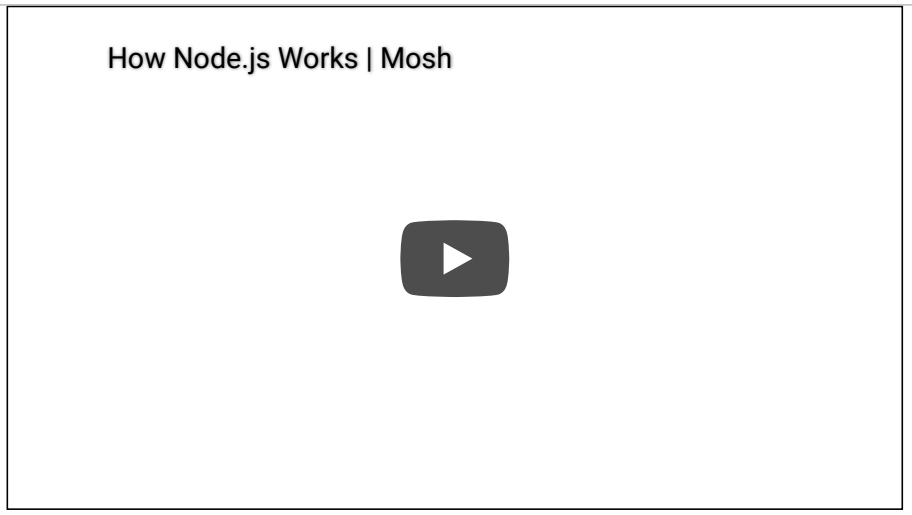
Image - Client-side and server-side computing



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

Algorithms and Data Structures

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...*

Algorithms and Data Structures

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`*

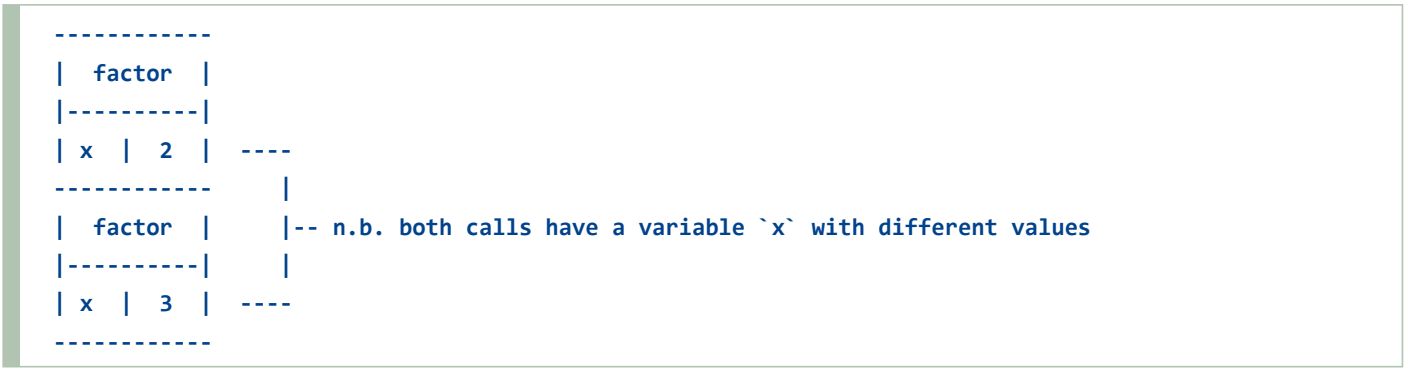
```
------------
|  factor  |
|----------|
| x  |  3  |
------------
```

- 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  |
------------
```

Algorithms and Data Structures

recursion and the call stack - part 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

Algorithms and Data Structures

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

Algorithms and Data Structures

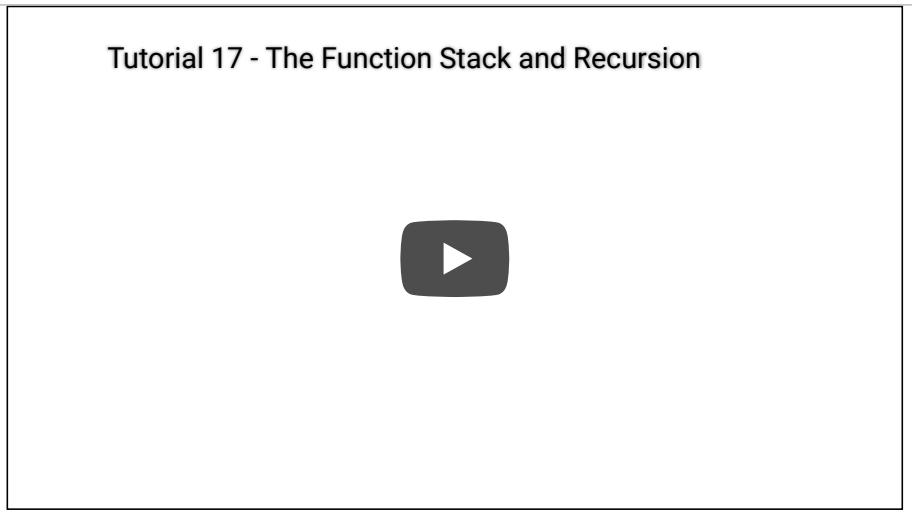
recursion and the call stack - part 5

- 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

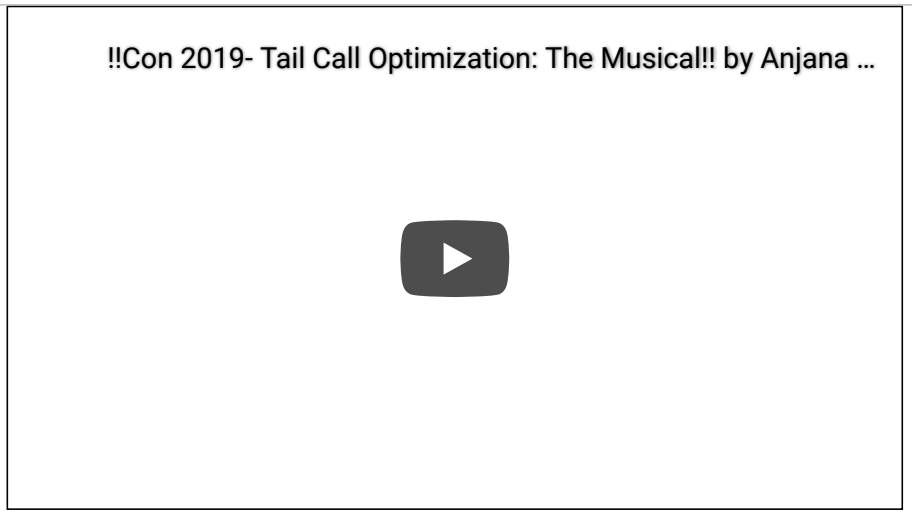
Algorithms and Data Structures

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

Resources

JavaScript

- MDN - Array
- MDN - Prototype
- MDN - Proxy

Python

- Stacks - YouTube

Various

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