Murdoch
UNIVERSITY

JavaScript: Functions, Objects and NodeJS More

Lecture 2 (A)

ICT375 Advanced Web Programming
Semester 1, 2021

Lecture Objectives

- Relevance to unit objectives:
  - Learning objective 2: Writing software
- Relevance to assessments:
  - Much of your programming in this unit (Labs and Assignments) will be written in the JavaScript programming language within the Node.js environment

Lecture Outline

- Why use JavaScript for this unit?
- Advanced features of JavaScript programming language
  - Revision of JavaScript basic language features
  - JavaScript functions: anonymous, closures
  - Arrays in JavaScript
  - JavaScript object-oriented features
- How to get up to speed with JavaScript

Introduction

- This lecture will **NOT** be a general “Introduction to Programming” lecture
  - It is assumed that you are already familiar with programming from previous units like ICT159, ICT167 and ICT286
- Instead, we will review the basics, and then cover advanced features of JavaScript
  - Please read recommended textbooks for a more complete coverage of the language syntax
  - You should refer to language references (online) listed at the end of these lecture slides when you start doing your programming exercises

Advantages of JavaScript

- JavaScript is a scripting language that historically allows us to design interactive web pages
- Some of the usage are:
  - Browser detection
  - Opening pages in customized windows
  - Validating input fields before and when submitting a form
  - Changing the web page in response to user action

Disadvantages of JavaScript

- Unfortunately, JavaScript has weaknesses:
  - Though there is an agreed upon standard called **ECMAScript**, vendors apply this standard to their own implementation in their own 'unique' way (much like differences between browsers)
  - JavaScript is not as strictly ‘typed’ as other languages
    - This can introduce undesirable, sloppy programming practices
    - TypeScript was introduced to deal this problem
  - There are many different ways to do the same thing in JavaScript
    - This can lead to lack of consistency and uniformity within development teams

Why JavaScript in this Unit?

- The reasons for using JavaScript in this unit:
  - JavaScript usage is much more powerful and flexible now than it was in its traditional usage
    - A large community of programmers / developers are now taking its usage into many new areas
  - We will be using the Node.js development environment to demonstrate client / server architecture
  - Node.js is a JavaScript implementation

JavaScript in HTML

- JavaScript was originally used in HTML pages for the reasons mentioned on slide 6
- Here we provide a very brief review:
  - The primary method of inserting JavaScript into an HTML page is via the `<script>` element
  - There are six attributes for the `<script>` element (all of which are optional): async, charset, defer, language (deprecated), src, type
    - Please investigate these attributes as needed
    - You can review your material from ICT286

JavaScript in HTML

- Two main ways to use the `<script>` element:
  1. Embed JavaScript code directly into HTML pages

```
<script type="text/javascript">
   function sayHI() {
      alert("HI!");
   }
</script>
```

  2. Include JavaScript code from an external file; this requires the use of the **`src`** attribute to provide the URL of the file with the JavaScript code in it

```
<script type="text/javascript" src="example.js">
</script>
```

JavaScript in HTML

- Traditionally, all `<script>` elements were placed within the `<head>` element on a HTML page

```
<!DOCTYPE html>
<html>
  <head>
      <title>Example HTML Page</title>
      <script type="text/javascript"
              src="example1.js"></script>
      <script type="text/javascript"
              src="example2.js"></script>
  </head>
  <body>
      <!-- content here -->
  </body>
</html>
```

JavaScript in HTML

- Modern web applications allow JavaScript references in the `<body>` element (i.e. within the page content):

```
<!DOCTYPE html>
<html>
  <head><title>Example HTML Page</title>
  </head>
  <body>
      <!-- content here -->
      <script type="text/javascript"
              src="example1.js"></script>
      <script type="text/javascript"
              src="example2.js"></script>
  </body>
</html>
```

JavaScript in HTML

- From ICT286, you know that HTML has been deprecated in favour of XHTML, which has now been superseded by HTML5
    - Thus there are differences between the three, with HTML5 and XHTML being more strict syntactically than HTML
    - This may have some impact in relation to JavaScript usage
- It is therefore your responsibility to learn (or remind yourself of) the differences between the three, and investigate when and how this could affect your JavaScript code

JavaScript Language Basics

- An identifier is the name of a variable, function, property, or function parameter
- Identifiers may consist of one or more characters in the following format:
  - The first character must be a letter, an underscore (_), or a dollar sign ($)
  - All other characters may be letters, underscores, dollar signs, or numbers
  - Meaningful identifiers should be used
  - By convention, identifiers use camel case, meaning that the first letter is lowercase and each additional word is offset by a capital letter, like this: `doSomethingImportant`

JavaScript Language Basics

- Variables, function names, and operators are all case-sensitive, meaning that a variable named ‘test’ is different from a variable named ‘Test’
  - Eg: ‘typeof’ can not be the name of a function, because it is a keyword (we will look at keywords shortly)
  - However, ‘typeOf’ is a perfectly valid function name

Comments and Statements

- JavaScript uses C-style comments
  - Single-line comments use //
  - Block comments use /* multiple lines */
- It is recommended that all statements in JavaScript be terminated with a semicolon
  - Importantly, this improves parser performance and also code readability and maintainability
- Like C, multiple statements require braces (curly brackets), to indicate a block of code

  Eg: `{ … block of code … }`

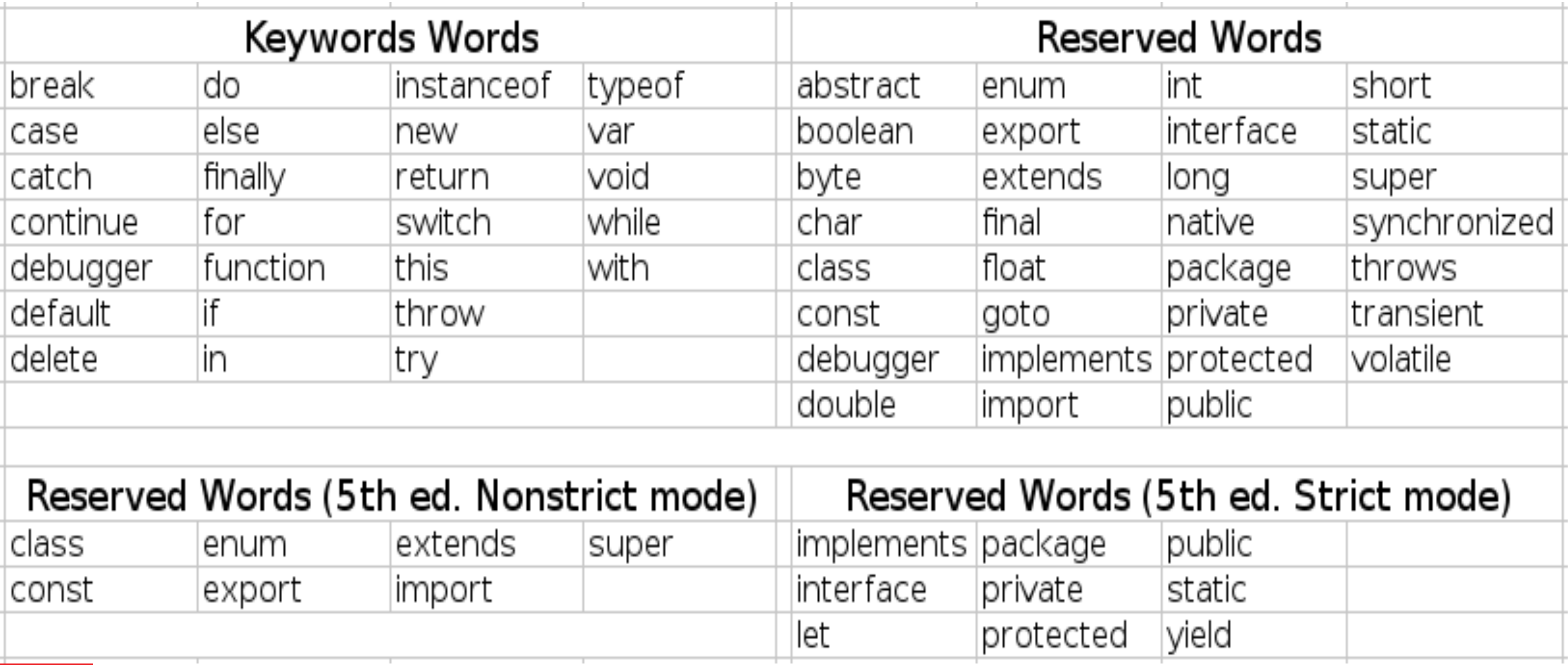
Strict Mode

- Strict mode is a parsing and execution method where some of the erratic behavior (of earlier versions) are addressed, and errors are thrown for unsafe activities
- To enable strict mode, place the following directive at the top of your JavaScript:

```
"use strict";// quotes and semicolon required
```

- Using the `strict` directive, is recommended practice

Keywords and Reserved Words



Keywords Words				Reserved Words			
break	do	instanceof	typeof	abstract	enum	int	short
case	else	new	var	boolean	export	interface	static
catch	finally	return	void	byte	extends	long	super
continue	for	switch	while	char	final	native	synchronized
debugger	function	this	with	class	float	package	throws
default	if	throw		const	goto	private	transient
delete	in	try		debugger	implements	protected	volatile
				double	import	public	
Reserved Words (5th ed. Nonstrict mode)				**Reserved Words (5th ed. Strict mode)**			
class	enum	extends	super	implements	package	public	
const	export	import		interface	private	static	
				let	protected	yield	

- The table above was presented in ICT286; you should review the keywords above

JavaScript Variables

- JavaScript variables are loosely typed, which means that a variable can hold any type of data
  - Every variable is simply a named place-holder for a value
- To define a variable, use the **var** keyword followed by the variable name
- Eg:

```
var message;          // defined or declared
message = "Hi!";   // initialized
message = 43;       // valid but not recommended
```

JavaScript Variable Scope

- It is important to note that using the **var** keyword to declare a variable makes its scope local to where it was defined
- For example, within if, if-else, switch, looping structures and functions, a **var** defined variable is local to that structure

```
function test()
{
  var message = "Hi!";   // local variable
}
test();                  // correct output
console.log(message);    // error
```

JavaScript Variable Scope

- In the previous example, the variable is defined inside the function using **var**
- This means that the local variable is destroyed as soon as the function exits
  - After calling and exiting the function, an attempt to access the variable is made, so the last line causes an error
- If you want a variable for local use only, then this is legal and appropriate
  - However, if you attempt to access a variable declared locally (from outside its scope), then an error ensues

JavaScript Variable Scope

- It is also possible to define a variable without using the **var** keyword
- Such a variable will be globally available inside and outside functions, etc.
  - **However, this is not recommended practice**, as global variables defined locally are hard to debug and can cause confusion and error

```
function test(){
  message = "Hi!";     // global variable
}
test();
console.log(message);  // prints "Hi!"
```

JavaScript Variable Scope

- A much better approach is to define a variable globally using the **var** keyword
- The variable is then still accessible wherever it is needed, but may avoid logic errors
  - However, you should exercise due care with the use of global variables

```
var message;                    // global variable
function test(){
    message = "Hi!";
}
test();
console.log(message);  // prints "Hi!"
```

JavaScript Code: Expected Standard

- For all tutorials and assignments it is expected that your JavaScript code will demonstrate the following recommended practices:
  - ‘strict’ mode should be used
  - All statements should be semicolon terminated
  - Meaningful identifiers should be used (camel-case where appropriate)
  - Variables should be declared using keyword **var** (or **let**)
    - Only use global variables when necessary
  - Correct code layout should be used
  - Application design **must** be modular

JavaScript Data Types

- There are 5 simple data types (also called primitive types) in JavaScript:
  1. Undefined - has only one value: the special value **undefined**
  2. Null - has only one value: the special value **null**
  3. Boolean - has only two possible literal values: **true** or **false**
  4. Number - uses the IEEE-754 format to represent integers and floating-point values
  5. String - represents a sequence of zero or more 16-bit Unicode characters

JavaScript Data Types

- There is also a complex data type:
  6. Object is an unordered list of `property:value` pairs
- There is no way to define your own data types in JavaScript, so all values can be represented as one of the previous six data types
- Defining an object is often considered defining your own data type
  - This will be discussed further when we look at O-O programming in JavaScript

JavaScript Data Types

- These data types were covered in detail in ICT286, so we will not go into the details here
- However, you should review that material for yourself
  - For your convenience, the appropriate slides have been included at the end of this set of lecture slides (73-82)

`typeof` Operator

- As JavaScript is loosely typed, we often need to determine the data type of a value stored in a given variable
- The **`typeof`** operator returns one of the following string values:
  - "**undefined**" if the value is undefined
  - "**boolean**" if the value is a Boolean
  - "**string**" if the value is a string
  - "**number**" if the value is a number
  - "**object**" if the value is an object (other than a function) or **null**
  - "**function**" if the value is a function

Operators and Control Structures

- JavaScript provides the following operators:
  - Increment/Decrement (pre and post ++, --)
  - Mathematical Operators (+, -, *, /, % (modulus on integer division))
  - Assignment Operators (=, +=, -=, *=, /=)
  - Relational Operators (==, ===, >, >= , <, <=)
    - The == operator will compare for equality after doing any necessary (implicit) type conversion
    - The === operator performs identically to == except it does not perform any type conversion

Operators and Control Structures

- Logical Operators (&&, ||, !)
- The Conditional Operator

```
variable = boolean_expression ? true_value : false_value;
```

- Bitwise and Shift operators
  - Look up for yourself

Operators and Control Structures

- JavaScript provides the following control structures:
  - if, if-else, and nested if-else statements
  - switch-case statements
  - for loop (and variations)
  - while loop
  - do-while loop
- Operators and control structures work the same way as they do in C, C++, and Java
- You should investigate for yourself in the case of any slight variances

Functions in JavaScript

- Functions in JavaScript are declared using the **function** keyword, followed by an optional set of parameters (in parentheses) and then the body of the function
- The basic syntax is as follows:

```
function functionName([param0,param1,...,paramN]) {
  // statements
}

function sayHi(name, message) { // note no data types
  console.log("Hello " + name + ", " + message);
}
```

Functions in JavaScript

- The previous function can be called as follows:

```
// passing string literals as arguments
sayHi("Nicholas", "how are you today?");
OR
// passing pre-defined/initialized variables as arguments
var arg1 = " … "; var arg2 = " … ";
sayHi(arg1, arg2);
```

- Any function can return a value at any time by using the `return` statement followed by an optional value

```
function sum(num1, num2) {
  return (num1 + num2);
}
```

Functions in JavaScript

- A function stops executing immediately after the last executed statement OR upon encountering the return statement (possibly returning a specified value)
- The return statement can be used without specifying a return value
  - Commonly used with branching statements
  - When used in this way, the function stops executing immediately and returns the value **undefined**

Understanding Arguments

- JavaScript functions do not care:
  - How many parameters are listed
  - The order in which they are listed
  - The data types of the parameters
- Just because you define a function to accept two parameters does not necessarily mean you have to pass in two arguments when calling the function
  - You could pass in one or three or none, and the interpreter will not complain

Understanding Arguments

- JavaScript will not complain about type mis-matches between the parameter and argument lists
- You can pass parameters or arguments in any order
- **Caution:** in order to avoid errors in logic and functionality, a disciplined approach should be adopted to keep track of such issues

Understanding Arguments

- The situation just mentioned is permitted because arguments in a function call are internally represented as elements in an array (specifically an Array object – more on this later)
  - The Array object is always passed into a function, but the function does not care what (if anything) is in the Array object

Understanding Arguments

- The name of the Array object is `arguments`
- It is passed into all functions, and can be accessed from within a function to retrieve values of any argument passed in
  - You can access the `arguments` array using the square bracket notation
  - The first argument is `arguments[0]`, the second is `arguments[1]`, and so on …

Understanding Arguments

- In our example on slide 31, the `sayHi()` function's first parameter is called ‘name’
- The corresponding argument in the function call can be accessed by referencing `arguments[0]`
- Therefore, the function can be re-written without naming the parameters explicitly:

```
function sayHi() { // note: no parameters
  console.log("Hello "+arguments[0]+", "+arguments[1]);
}
// call function with arguments
sayHi("Nicholas", "how are you today?");
```

Understanding Arguments

- Note this function is defined with no parameter list
- The `name` and `message` parameters have been removed, yet the function can still access the appropriate argument values (passed in the function call)
- The length property of the `arguments` object can be used to obtain the number of arguments available to the function:

```
function howManyArgs() {
   console.log(arguments.length);
}
```

Understanding Arguments

- Any named parameter (in a function definition) is automatically assigned the value `undefined` when no value is passed as an argument in the function call
- This means that unlike other languages, JavaScript functions cannot be *overloaded* in the traditional sense
    - Overloading requires an exact signature match
- If two functions are defined to have the same signature (overloaded), it is the last function that becomes the owner of that name

Anonymous Functions

- A function without a name is called an **anonymous** function
- You can assign such a function to a variable
  - The idea is that, if you are going to use a function as a variable (and not refer to it by its function name), then you do not need to name the function when defining it
- Thus, the following methods are equivalent

Anonymous Functions

```
// function name given, but is just wasted characters
var foo1 = function namedFunction() {
  console.log('foo1');
}
foo1(); // call the function via the variable foo1

// no function name provided, i.e. anonymous function
var foo2 = function () {
  console.log('foo2');
}
foo2(); // call the function via the variable foo2
```

Higher Order Functions

- Since JavaScript allows us to assign functions to variables, we can pass functions to other functions
- Such functions are called higher-order functions

```
function say(word) {    // function with one parameter
   console.log(word);   // prints value of parameter
}

// function with 2 parameters; a function and a value
function execute(someFunction, value) {
   someFunction(value); // calls the ‘say’ function
}

//function call; pass function ‘say’ and a string
execute(say, “Hello”);
```

Higher Order Functions

- Example of a normal and an anonymous function passed as a parameter:

```
// using normal function definition
function foo() {
   console.log('2000 milliseconds have passed');
}
setTimeout(foo, 2000); // calls function foo


// declaring anonymous function in argument list
setTimeout( function () {
   console.log('2000 milliseconds have passed');
}, 2000);   // delay 2000 milliseconds or 2 seconds
```

Closures

- A **closure** is a combination of a function and the lexical environment within which that function is declared
  - **Lexical environment** can be defined as the association of identifiers to specific variables or functions based on the nesting structure
- So, whenever we have a function defined inside another function, the **inner function** has access to the variables declared in the **outer function**

Closures

```
// demonstrating normal functionality
function outerFunction(arg) {
  var variableInOuterFunction = arg;
  // inner function to output variable value
  function bar() {
     // Access the variable from the outer scope
    console.log(variableInOuterFunction);
  }
  // Call the local (inner) function to
  // demonstrate that it has access to arg
  bar();
}
// prints hello fucntion!
outerFunction('hello function!');
```

Closures

- However, with closures, the inner function can still access the variables from the outer scope even after the outer function has returned
  - Variables are still bound in the inner function and are not dependent on the outer function
  - Advanced JavaScript usage often makes use of this functionality
    - We will use this functionality when developing Web clients and servers
  - See more examples in

    https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

Closures

```
// demonstrating closure functionality
function outerFunction(arg) {
  var variableInOuterFunction = arg;
  // returning an anonymous inner function
  return function () {
    console.log(variableInOuterFunction);
  }
  // we do not call the inner function here!
}
var innerFunction=outerFunction('hello closure!');

// outerFunction has already returned at this point
innerFunction(); // prints hello closure!
```

Arrays

- If you need to keep track of many related items, individual variables may not be convenient
- JavaScript, like other languages, provides arrays for this purpose
  - Remember from other units, that an array is considered a complex data type
  - Typically, an array can only store data of the same data type (i.e., they are non-heterogeneous), and they are not dynamic (if the array needs to be expanded, a programmer must allocate more memory using appropriate runtime methods)

Creating an Array in JavaScript

- In JavaScript, you declare the array name (just as you would a variable) and then supply a list of comma separated values
  - What you name the array is up to you, but you should follow the same naming conventions as for single variables
  - Each comma separated value in the list represents one element in the array
- To indicate an array, you can put a list of elements between opening and closing square brackets

Creating an Array in JavaScript

```
var days =
   ['Mon','Tues','Wed','Thurs','Fri','Sat','Sun'];
```

- You can create an empty array without any elements and add items to the array as needed whilst the program is running

```
var playList = [];
```

- You can store any mix of values (data types) in an array i.e. numbers, strings, Boolean, etc.

```
var prefs = [1.0, 223, 'www.oreilly.com', false];
```

Creating an Array in JavaScript

- So, two very important points about arrays in JavaScript which distinguish them from their implementation in many other languages:
  - **They will dynamically increase in size as new elements are added**; the programmer is thus relieved of manually handling memory
  - They are heterogeneous; i.e., **you can store different data types in the same array**
    - The advisability of storing heterogeneous data types in the same array is questionable, but is convenient when dealing with data from Web pages
    - This requires a disciplined approach to programming

Accessing Elements in Arrays

- Access to array elements (for insertion, modification, and retrieval) is the same as with other languages
  - i.e., via an index number, which starts at 0

```
var days =
        ['Mon','Tues','Wed','Thurs','Fri','Sat','Sun'];
alert(days[0]);     // retrieves, prints element 0
days[3] = 'Fred';   // modifies element at index 3
```

- You can access the array length:

```
alert(days.length); // prints 7
```

The Array Object

- An JavaScript array is really an object and it can also be created with Array constructor.
- A JavaScript array uses indexes to access its elements

```
// a new empty Array object using a constructor
var arr = new Array();
// creates a new Array object with 4 elements
var arr4 = new Array(1, "Hi", { a: 2 },
                  function(){console.log('boo');});
// print "Hi"
console.log(arr4[1]);
```

Accessing Elements in Array with Methods

- As an object, a JavaScript array has some special property (such as length) and methods (inherited from the Array.prototype global object)
- JavaScript Array objects have methods:
  - push() – 1 or more added to end of array
  - unshift() – 1 or more added to beginning
  - pop() – 1 only removed from end of array
  - shift() – 1 only removed from beginning
  - splice() – 1 or more added or removed from designated position in array

Associative Arrays

- Many programming languages support arrays with **named** indices or keys
    - Arrays with named indices are called **associative arrays** (or hashes or maps)
- JavaScript does **not** support associative arrays (i.e. named indices).
- In JavaScript, arrays always use numbered indices

Associative Arrays: WARNING !!

- You can add named indices to an array, Eg:

```
var person = [1, "two"];
person["firstName"] = "John";
person["age"] = 46;
```

- However, the standard array property and methods would not apply to the elements with the named indices. E.g.,

```
console.log(person.length); // print 2, not 4

console.log(person[3]); // print undefined
```

Associative Arrays

- To re-iterate, in JavaScript **arrays** use *numbered indices* and **objects** use *named indices*
- Therefore:
  - You should use **arrays** when you want the element index to be *numbers*
  - You should use **objects** when you want the element index to be *strings* (i.e. text)

JavaScript Objects

- Prior to ES6, JavaScript, JavaScript does not support classes.
- It uses `Object` type instead of classes to create objects.
- New objects can be created by using the new operator followed by a constructor of type `Object`

Object Type

- A constructor is simply a function whose purpose is to create a new object

  Eg: `var Person = new Object();`

- The above example creates a new instance of the `Object` reference type and stores it in the variable called `Person`

Object Type

- The constructor being used is `Object()`, which creates a simple reference with only the default properties and functionality
- To this point, most of the reference value examples have used the `Object` type
- Although instances of the `Object` type do not have much functionality, they are ideally suited to storing and transmitting data around an application

Object Type

- There are two ways to explicitly create an instance of type `Object`
- One way (as we have seen) is with the **new** operator and the `Object` constructor:

```
var Person = new Object();

// then can add properties and/or
// functionality using the ‘dot’ notation
Person.name = "Nicholas";
Person.age = 29;
```

Murdoch UNIVERSITY

Object Type

- We can also add methods to the object using anonymous function. Eg:

```
Person.print = function (){
    console.log("name: " + this.name);
    console.log("age: " + this.age);
}
```

- Note that in the above example, we need to use the reserved word `this` to access the properties of the object.

Object Literals

- The other way uses the **object literal notation**

```
var Person = {}; // equivalent to previous example
```

  - This is a short-hand form of object definition designed to simplify creating an object with numerous properties:

```
var Person = {
  name : "Nicholas", // note colon and comma
  age : 29           // no comma after last property
};                   // note semicolon to close
```

  - The assignment operator indicates a value is expected next; in this case, an object literal

Object Literals

- As well as properties, we can also add functionality to object literals via functions:

```
var Person = {
   name  : "Nicholas",
   age : 29,
   job : "Software Engineer",

   sayName : function(){
      console.log(this.name); // note use of this
   }                    // no comma after last statement
};                      // note semicolon to close
```

- The function `sayName()` just prints the value of the `name` property of the `Person` object

JavaScript Classes

- JavaScript class is introduced in ES6 (ECMAScript 2015). We can now use JavaScript a class to create objects.

```
// declare a class
class Person {
    constructor(name, age, job){
       this.name = name;
       this.age = age;
       this.job = job;
    }
    sayName(){   // a method
      console.log(this.name);
    }
 }
 // create a new object
let person = new Person("Greg", 27, "Doctor");
console.log(person.sayName()); // prints "Greg"
```

JavaScript Classes

- Notice in the previous slide:
  - By convention, a class name always begins with an uppercase letter
  - You must always declare a constructor inside a class, whose name is exactly constructor.
  - The properties are assigned directly into the object using the keyword **this**
  - There is no return statement, because constructors in any language do not have a return statement
  - A method is declared with the following syntax:

```
method_name ( … ) {   … }
```

Leverage Your Existing Programming Skills

- To get the best out of JavaScript for this unit:
  - Be sure you understand the power of functions (arguments, anonymous, closures) and O-O features
  - Be familiar with the JavaScript documentation
  - Be prepared to research independently as needed
  - If you are having trouble with something, keep researching and working until you solve it
  - Do not forget what you have learned in other programming units
    - Follow the best practices shown to you

Further Reading

- This lecture does NOT cover the JavaScript language comprehensively
- You should utilize any of the materials suggested in the next two slides
- Visit the JavaScript homepage for useful materials, and visit one of the online tutorials suggested
- JavaScript does not provide much in the way of syntax error output, so visit the **javascriptlint** site and learn to use it correctly

JavaScript References

- Professional JavaScript for Web Developers Zakas, N.C.
- JavaScript homepage:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript
- Online JavaScript tutorials
  - http://www.w3schools.com/
  - https://www.codeschool.com/courses/javascript-road-trip-part-1
- For correct usage of JavaScript:
  - http://javascriptlint.com/

JavaScript References

- JavaScript: a beginner's guide, John Pollock. 2nd ed., 2004.
- JavaScript step by step, Steve Suehring.
- Beginning JavaScript, Wilton, Paul; McPeak, Jeremy, 2010.
- JavaScript: the definitive guide, David Flanagan, 2010.
- JavaScript and JSON essentials, Sai Srinivas Sriparasa, 2013.
- Principles of Object-Oriented JavaScript, N. C. Zakas.
- Object Oriented JavaScript, Stoyan Stefanov.

Data Type `Undefined`

1. When a variable is declared using **var**, but not initialized, it is automatically assigned the value of `undefined`

```
var message;
alert(message == undefined);  // true
```

   - Generally, you should **not** explicitly set a variable to be **undefined**
   - A variable containing the value of **undefined** is different from a variable that has not been defined at all (i.e. **var** has not been used)

Data Type `Null`

2. Logically, a **null** value is an empty object pointer
   - This is why the `typeof` operator returns "object" when it is passed a `null` value

```
var car = null;
alert(typeof car); // output is "object"
```

   - It **is** advisable to initialize an object pointer variable to `null`; you can then explicitly check if the value is `null` or an object reference

```
if (car == null){
    //do something with car
}
```

Data Type `Boolean`

3. Boolean values are distinct from numeric values, so `true` is not equal to 1, and `false` is not equal to 0
   - All other types of values have Boolean equivalents in JavaScript
   - The `Boolean()` casting function can be called on any type of data to convert it to its Boolean equivalent

```
var message = "Hello world!";
var messageAsBoolean = Boolean(message);
```

   - The `Boolean()` casting function will always return a Boolean value

Data Type `Boolean`

- The rules for what is assigned when a value is converted to `true` or `false` depend more on its data type than the actual value

DATA TYPE	VALUES CONVERTED TO TRUE	VALUES CONVERTED TO FALSE
Boolean	true	false
String	Any non-empty string	"" (empty string)
Number	Any non-zero number (including infinity)	0, NaN (See the "NaN" section below.)
Object	Any object	null
Undefined	n/a (i.e. cannot be true)	undefined

  - So, it is important to understand what variable you are using (and what value you are storing in it) in a flow-control statement

Data Type Number

4. There are several different number literal formats
   - The most basic is a decimal integer

```
var intNum = 55;  // integer
```

   - The floating-point value must include a decimal point and at least one number after (to the right of) the decimal point

```
var intNum = 0.1; // or .1 not recommended
```

   - A special numeric value is NaN, which is used to indicate a failed mathematical operation (as opposed to a syntax error)
   - Number has various functions and operators such as `parseInt()`,`parseFloat()`,`Number.MIN_VALUE`, `Number.MAX_VALUE`

Data Type String

5. Strings can be delineated by either double quotes or single quotes
   - A string beginning with a double quote must end with a double quote, and a string beginning with a single quote must end with a single quote
   - There are the following character literals:

LITERAL	MEANING	LITERAL	MEANING
\n	New line	\r	Carriage return
\t	Tab	\f	Form feed
\b	Backspace	\\	Backslash (\)
\’	Single quote (') - used when a string is delineated by single quotes Example: 'He said, \'hey.\"		
\"	Double quote (") - used when a string is delineated by double quotes Example: "He said, \"hey.\""		

Data Type String

- The length property returns the string length

  `alert(text.length);`

- Like Boolean, other data types can be converted to string using the `String()` casting method or the `toString()` method
- Like Java, strings in JavaScript are immutable
  - i.e., once a string has been created, its value cannot change
  - To modify the string held by a variable, the original string must be destroyed and the variable filled with another string containing a new value

Data Type Object

6. Objects in JavaScript start out as non-specific groups of data and functionality
   - Objects are created by using the **new** operator followed by the name of the object type to create
   - If there are no arguments, the parentheses can be omitted (though **this is not recommended practice**)

```
var obj = new Object();
```

   - You can create your own objects by creating instances of the Object type and adding properties and/or functionality to it

Data Type Object

- The Object type is the base from which all other objects are derived
  - All properties and methods of the Object type are also available to other objects
  - Each Object instance has the following properties and methods:
    - Constructor
    - hasOwnProperty(propertyName)
    - isPrototypeOf(object)
    - propertyIsEnumerable(propertyName)
    - toLocaleString()
    - toString()
    - valueOf()

Murdoch
UNIVERSITY

Node.js: Fundamentals

Lecture 2 (B)

ICT375 Advanced Web Programming
Semester 1, 2021

Lecture Objectives

- Relevance to unit objectives:
  - Learning objective 1: Understand the technical details of key Web technologies
  - Learning objective 2: Writing software
- Relevance to assessments:
  - Some of your programming in this unit (Labs and Assignments) will require the use of the Node.js environment to demonstrate client / server architecture

Lecture Outline

- Introduction to Node.js as an implementation of JavaScript
  - Node.js concepts, usage, and performance
  - Node.js core modules
  - Node.js modules: importing / exporting
- How to get up to speed with Node.js

Introduction to Node.js

- Node.js is an open-source, cross-platform runtime environment for developing *server-side* Web applications
  - Its applications are written in JavaScript and can be run within the Node.js runtime environment on a wide variety of platforms (including macOS, Windows, and Linux/Unix servers)

Introduction to Node.js

- Node.js provides an event-driven architecture designed to optimize an application's *throughput* and *scalability* for real-time Web applications
  - It provides a *non-blocking* I/O API so that Web application do not just hang during I/O
  - It uses Google’s V8 JavaScript engine to execute code
  - A large percentage of the basic modules are written in JavaScript and are designed to reduce the complexity of writing server applications

Introduction to Node.js

- Like PHP, Node.js is primarily used to build network programs (eg: Web servers)
- The main difference between PHP and Node.js is that:
  - Most functions in PHP block until completion
  - Functions in Node.js are designed to post long lasting tasks to a thread pool, and then return to the caller in a non-blocking fashion
    - This allows queueing parallel tasks without explicit threading (i.e., you do not have to program threads, Node.js handles any threading)

Node.js Thread Pool

- As just mentioned, execution of parallel tasks in Node.js is handled by a thread pool
  - The main thread-call functions post tasks to the shared task queue
  - Inherently *non-blocking* system functions (like networking) translate to kernel-side non-blocking sockets
  - Inherently *blocking* system functions (like file I/O) run in a blocking way on their own thread
  - When a thread in the thread pool completes a task, it informs the main thread
  - The main thread in turn wakes up and executes a registered callback

Event-Driven Asynchronous Callbacks

- The Node.js execution model has only a single process, but generates new threads as required to handle requests
  - This is different from Apache’s pre-forking, which uses new processes to handle new requests
  - If there is a slow task somewhere in the process, this affects the whole process
  - Everything comes to a halt until the slow task has finished – this is **synchronous** processing
  - For a server, this could possibly mean many clients having to wait for requests to be responded to

Event-Driven Asynchronous Callbacks

- To understand the problem, consider a trivial example of synchronous processing:

```
var result = database.query(“SELECT * FROM hugetable”);
console.log(“Hello World!”);
```

- The interpreter has to read all rows from the database before executing the log function
  - As the database is *huge*, this may take some time
  - Any other processing pending will be put on hold
- Node.js introduces the event loop and uses callbacks to overcome this problem

Event-Driven Asynchronous Callbacks: Event Loop

- Upon starting a server, variables are initiated, functions are declared, and the **event loop** process simply waits for an event to occur
  - The event loop runs in a continuous cycle when there is nothing do, and waits for events
  - If a request is received, a thread is generated and processing of the request is handed to that thread
  - When a request has completed, execution returns to the event loop, which waits for another request
  - If multiple threads are executing requests, the event loop enables each one to finish in its own time and not interfere with each other

Event-Driven Asynchronous Callbacks

- Callback functions are triggered when an **asynchronous** function returns its result:
  - That is, when a request thread completes its task, it returns its result
  - This triggers an event, which then calls an anonymous callback function

Event-Driven Asynchronous Callbacks

- So, we can re-write the code from our previous example to pass in an anonymous function to our query:

```
database.query("SELECT * FROM hugetable", function (rows) {
   var result = rows;
});
console.log("Hello World!");
```

- This allows Node.js to handle the query asynchronously
  - Assumes that the `database.query()` method is part of an asynchronous library

Event-Driven Asynchronous Callbacks

- The query is sent to the database
  - Instead of waiting for the entire database read to finish, an event listener is registered to trigger when the database server has finished reading
  - At this point the result of the query is returned and the anonymous function is executed
- Meanwhile, execution of the log function occurs immediately after the event listener is registered (i.e., when the query is sent)
  - Execution then enters the event loop to process any incoming requests or completed instructions

Node.js Libraries

- Node.js contains a built-in standard library (providing core functionality) to allow an application to act as a stand-alone Web server
- Node.js is typically used where light-weight, real-time response is needed
  - Like Web-based gaming and communication applications
- It can also be used to build large, scalable network applications

Node.js Libraries

- Node.js has access to a rich library of various JavaScript modules, which simplifies (to a great extent) development of web applications
- Thousands of open-source libraries have been built for Node.js, most of which are hosted on the Node Package Manager (npm) website

Node.js = Runtime Environment + JavaScript Library

Node.js and JavaScript Globals

- Node.js and browser JavaScript differ when it comes to globals:
  1. Node.js does not directly deal with a browser window, whereas browser JavaScript has a `window` object (which is globally available)
  2. Browser JavaScript, by default, puts everything into its global scope (i.e. `window` object)
  3. Node.js, by default, was designed to put everything into local scope
     - In case we need to access globals, there is a global object; and when we need to export something, we should do so explicitly

Node.js Core Modules

- Node.js does not come with a *heavy* standard library
- The core modules of Node.js are a bare minimum, and other external modules can be obtained from the **npm** registry
- A JavaScript module is just a JavaScript file
- A JavaScript module forms its own local scope
- The main core modules (and their classes, methods, and events) include the following:

Node.js Core Modules

1. **http** is the main module responsible for the Node.js HTTP server (http://nodejs.org/api/http.html#http_http); its main methods are as follows:
   - http.createServer(): returns a new web server object
   - http.listen(): begins accepting connections on the specified port and hostname
   - http.createClient(): is a client and makes requests to other servers; this is now deprecated, so developers should instead use **http.request()**

Node.js Core Modules

- http.ServerRequest(): passes incoming requests to request handlers
  - data: emitted when part of message is received
  - end: emitted exactly once for each request
  - request.method(): the request method as a string
  - request.url(): request URL string
- http.ServerResponse(): creates this object internally by an HTTP server - not by the user - and is used as an output of request handlers
  - response.writeHead(): sends a response header to the request
  - response.write(): sends a response body to the request
  - response.end(): sends and ends a response body

Node.js Core Modules

2. **querystring** provides utilities for dealing with query strings (i.e. data after the '?' in the url) (http://nodejs.org/api/querystring.html)
   - querystring.stringify(): serializes an object to a query string
   - querystring.parse(): de-serializes a query string to an object

Node.js Core Modules

3. **util** provides utilities for debugging (http://nodejs.org/api/util.html)
   - util.inspect(): returns a string representation of an object, which is useful for debugging
4. **url** has utilities for URL resolution and parsing (http://nodejs.org/api/url.html)
   - url.parse(): takes a URL string and returns an object

Node.js Core Modules

5. **fs** handles file system operations such as reading from, and writing to, files (http://nodejs.org/api/fs.html)
   - There are synchronous and asynchronous methods in this library:
     - fs.readFile(): reads files asynchronously
     - fs.writeFile(): writes data to files asynchronously
     - fs.readFileSync(): reads files synchronously
     - fs.writeFileSync(): writes data to files synchronously

Node.js Core Modules

- Other core modules are **net, dgram, https**
- You should investigate further the main core modules covered, and the other core modules, to become familiar enough with them to work with them correctly
- The official Node.js website provides more details of all core modules, available at:

  `https://nodejs.org/api/modules`

Node.js Core Modules

- There is **no need** to install or download any of the **core** modules, they are automatically installed with the Node.js environment
- To include them in your application, all you need is to use the `require` method:

```
var httpvar = require('http');
```

Node.js Core Modules

- Note the use of the keyword `require`
- Also, the core module being imported must be in single or double quotes – in this case, '**http**'
- The statement assigns an **http** object to the instance variable *httpvar*
- *httpvar* provides access to the public methods that are supplied by the **http** module (mentioned earlier)

Importing / Exporting Modules

- Importantly, the variable *httpvar* can be given any name, but it is common practice to name it after the module; so in the previous example we could name it just *http*
- In all of our future examples we will use this convention
- You can also export your own modules, and then import them into other scripts

Importing / Exporting Modules

- In browser JavaScript (mentioned in slide 16) there is no way to include modules
  - Scripts are supposed to be linked together using a different language (eg: HTML), but dependency management is lacking
- With Node.js, CommonJS and RequireJS help solve this problem
  - Node.js borrowed many things from the CommonJS concept
  - http://www.commonjs.org/
  - http://requirejs.org/

Importing / Exporting Modules

- The CommonJS defines an API to handle many common application needs, ultimately providing a standard library as rich as those of Python, Ruby and Java
- An application developer can write an application using the CommonJS API and then run that application across different JavaScript interpreters and host environments

Importing / Exporting Modules

- With CommonJS-compliant systems, you can use JavaScript to write:
  - Server-side JavaScript applications
  - Command line tools
  - Desktop GUI-based applications
  - Hybrid applications

Importing / Exporting Modules

- RequireJS is a JavaScript file and module loader
- It is optimized for in-browser use, but it can be used in other JavaScript environments like Node.js
- Using a modular script loader like RequireJS improves the speed and quality of your code

Importing / Exporting Modules

- As an example, let us make a module to start a server
  - We put the code in a script called `server.js`
- We need to export the necessary parts of our script
  - Other scripts that may wish to utilize the server module only need to run the script to start the server
  - So, to make a module to start a server, we can put the server into a function named **startServer** and export the function:

Importing / Exporting Modules: Exporting A Server

```
var http = require('http');
function startServer() {
  function onRequest(request, response) {
     response.write('hello client!');
     response.end();
  }
  http.createServer(onRequest).listen(8888);
  console.log('Server running');
}
exports.startServer = startServer;
```

- Don’t worry if you do not understand the code details at this point, we will discuss this server script next week in more detail
- The main point is that we have exported the server

Murdoch UNIVERSITY

Importing / Exporting Modules: Using The Server

- The server can now be imported into other scripts that may wish to use it
- For example, in a main application script called `index.js`, we can import the module and start a server

```
var server = require('./server');
// some code
server.startServer();
```

- The application `index.js` now has access to the exported functions of `server.js`

Importing / Exporting Modules

- So, to export an object in Node.js, use:

```
exports.name = function_name;
```

- Another example of exporting an object:

```
var messages = {
   find : function(req, res, next) { ... },
   add : function(req, res, next) { ... },
   format : 'title | date | author'
}
exports.messages = messages;
```

Importing / Exporting Modules

- An example of importing this code would be:

```
var msgs = require('./messages’);
```

- This assumes that `messages.js` is located in the current working directory and contains the previous code to export the object

Library Modules

- We have seen that to use core modules, we just use the `require` directive

```
require('http');
```

- We have also seen how we can export our own modules for use by other scripts
  - These too use the `require` directive, providing the correct path to the script that exports the module is supplied
- What about external library packages?
  - To import modules from external libraries requires another mechanism

Library Modules With NPM

- Node.js platform provides a package management system called the Node Package Manager (**npm**), which allows for seamless Node.js package management

  (https://npmjs.org/doc/files/npm-folders.html)

  - Installation of packages works similarly to Git in that it traverses the working tree to find a current project

Library Modules With NPM

- Install Node.js packages as follows:

```
npm install <package_name>
```

- An example:

```
npm install node-formidable
```

- To then use this package in a program, write:

```
var formidable = require('formidable');
```

Library Modules With NPM

- There are two ways to install packages with **npm**:
  1. Globally: you would typically do this as the admin or superuser, for packages to be available for all users; you cannot do this in the labs, but can on your own computer
  2. Locally: each user installs their own packages
- You choose which kind of installation to use based on:
  - How you want to use the package in a project
  - Other system-wide considerations

Node.js Modules: Global

1. If you are installing a package that you want all users to be able to use on the command line, install it **globally**
   - To install a package globally you supply the **-g** flag to the `npm install` command

     ```
     npm install <package_name> -g
     ```

   - The package binaries end up in your PATH environment variable
   - Manual pages are also installed
   - Again, only the superuser can install globally

Node.js Modules: Local

2. If you are installing a package that you only want to use in your own project, using `require('package_name')`, install it **locally**
   - This is npm’s default behaviour
   - When installing locally on command line, you must change directory to where the scripts in your project or application are located
   - Then issue the command to install the desired package (see next slide)

Installing Locally with npm

- A package can be downloaded and installed locally with the command:

```
npm install <package_name>
```

- This will create the `node_modules` directory in your current working directory (i.e., where you are located in the file system), if one does not already exist
- The package will be downloaded and installed under that directory

Installing Locally with npm

- To confirm that npm installation worked correctly, check to see that a `node_modules` directory exists and that it contains a directory for the package(s) you installed
  - You can do this on Unix or Windows systems

```
Eg:
npm install mysql
ls node_modules      (Linux)    OR
dir node_modules     (Windows)
mysql
```

Package Usage

- Once the package is installed under the `node_modules` directory, you can use it in your script

```
// Eg: in a script dbase.js
var mysql = require('mysql');
var connection = mysql.createConnection({…});
connection.connect();
connection.query(…);
```

- Run the script on command line using:

```
node dbase.js
```

Package Usage

- If you had not properly installed the `mysql` package, you would receive this error:

```
module.js:340
    throw err;
          ^
Error: Cannot find module 'mysql';
```

  - This could probably mean you have not been located in the correct directory when you installed the package
  - To fix it, run `npm install mysql` in the same directory as your script `dbase.js`

Which Package Version?

- If there is no `package.json` file for the package just installed, the latest version of the package is installed
- If there is a `package.json` file, the latest version of the package – satisfying the *semantic versioning rule* declared in the file `package.json` – is installed

What Is `package.json`?

- The `package.json` file is a good way to manage locally installed `npm` packages
- A `package.json` file offers the following:

1. It serves as documentation for the packages your project depends on
2. It allows you to specify the version of a package that your project can use by using *semantic versioning rules*
3. It makes your build re-produceable, which means that its easier to share with other developers

`package.json`: Minimum Requirements

- As a bare minimum, a `package.json` file must have the following properties:
  - `"name"` - all lowercase, 1 word, dashes and underscores allowed, no spaces allowed
  - `"version"`

```
Eg:
{
    "name": "my_package",
    "version": "1.0.0"
}
```

  - Notice the object literal notation

Creating a `package.json`

- To manually create a `package.json` file, type on command line:

  ```
  npm init
  ```

- This will initiate a command line questionnaire that will conclude with the creation of a `package.json`, in the directory where you initiated the command

Creating a `package.json`

- For correct placement of the `package.json` file, you should be located in the directory where the package is installed, before issuing the command
- However, the extended command line interface questionnaire experience may not be for everyone

Creating a `package.json`

- You can expedite the process with the default `package.json` by typing:

  `npm init` with the `--yes` or `-y` flag
- This will ask you only one question, author

  ```
  npm init --yes
  ```
- The `package.json` file will be written under the package directory (which is under `node_modules`)
  - Make sure you change to the package directory before issuing the command

Creating a `package.json`

Eg: /home/macca/node_modules/my_package/package.json

```
{
  "name": "my_package",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "macca",
  "license": "ISC",
  "repository": {
    "type": "git",
    "url": "https://github.com/macca/my_package.git"
  },
  "bugs": {
    "url": "https://github.com/macca/my_package/issues"
  },
  "homepage": "https://github.com/macca/my_package"
}
```

Explanation

`name:` defaults to author name unless in a git directory, in which case it will be the package name in the repository

`version:` always 1.0.0

`main:` always index.js

`scripts:` by default creates an empty test script

`keywords:` empty

`author:` whatever you provided the CLI

`license:` ISC

`repository:` will pull information from the current directory, if present

`bugs:` will pull information from the current directory, if present

`homepage:` will pull information from the current directory, if present

Specifying Packages

- You can also set several configuration options with the **init** command – Eg:

```
npm set init.author.name "macca"
```

- To specify the packages your project depends on, you need to list the packages you'd like to use in your `package.json`
  - "dependencies": are packages required by your application in production
  - "devDependencies": are packages only needed for development and testing

Manually Editing `package.json`

- You can manually edit dependencies in your `package.json` file
- You need to create the attribute in the package object called "dependencies" that points to an object
- This object will hold attributes named after the packages you would like to use
  - These point to a semantic versioning expression that specifies what versions of that package are compatible with your project

Manually Editing `package.json`

- If you have dependencies you only need to use during local development, you will follow the same instructions as above but in an attribute called "devDependencies"

Example Dependencies

```
{
   "name": "my_package",
   "version": "1.0.0",
   "dependencies": {
      "my_dep": "^1.0.0"
   },
   "devDependencies": {
      "my_test_framework": "^3.1.0"
   }
}
```

Specifying a `package.json`

- The easier way to add dependencies to your `package.json` is from the command line, by flagging the `npm install` command with either `--save` or `--save-dev`
- To add an entry to your `package.json` dependencies:

```
npm install <package_name> --save
```

- To add an entry to your `package.json` devDependencies:

```
npm install <package_name> --save-dev
```

Managing Dependency Versions

- **npm** uses Semantic Versioning (or SemVer or semver), to manage versions and ranges of versions of packages
- If you have a `package.json` file in your package directory and you run `npm install`, then **npm** will look at the dependencies that are listed in that file and download the latest versions satisfying semver rules for all of those dependencies

Further Reading

- This lecture provides a brief introduction to Node.js
- Next week we will cover in more depth the client and server aspects of Node.js
- You should utilize any of the materials suggested in the next two slides
- Visit the Node.js homepage for useful materials
- You can visit the online tutorials suggested

Node.js References

- Node.js homepage:
  - https://nodejs.org/en/
- Online Node.js tutorial
  - http://www.tutorialspoint.com/nodejs
  - https://docs.nodejitzu.com
- Beginning Node.js, Basarat, A.S., 2014.
- Practical Node.js, Mardan, A., 2014.
- Node.js In Practice; Young, A., and Harter, M., Jeremy, 2014.
- Node Up And Running, Hughes-Croucher, T., and Wilson M., 2012.

NPM References

- Node.js homepage:
  - https://nodejs.org/en/blog/npm/npm-1-0-global-vs-local-installation/
  - https://docs.npmjs.com/getting-started/installing-npm-packages-locally
- Online Package.json
  - https://docs.npmjs.com/files/package.json