JavaScript Course Introduction

(Video: 0_introduction.mp4): **Introduction**

- 00:21-01:53: Douglas Crockford introduces himself, gives a little introduction, and goes over the agenda for the course.

Part 1: Programming Style & Your Brain (00:01:54-01:15:03)

(Video: 1_A.mp4): **The Way We Think**

- 01:54-5:26: Our process of thought stems from two separate systems: System 1: Our Gut and System 2: Our Head.
- 05:27-08:10: Visual processing is something human brains do better than the fastest super-computers. However visual processing can create conflicts between our two different systems. An illusion is an example of what is seen by System 1 (our gut) but may not make sense logically to system two (our head).
- 08:11-10:51: Advertisers have always attempted to sell products to our *Gut* not to our *Head*. They try to appeal to us visually, not logically.

(Video: 1_B.mp4): **The Programming Thought Process**

- 10:52-14:38: Computer programs are the most complicated things people make. While computer have attempted to create programs (i.e. Artificial Intelligence), they aren't able to duplicate our system of thought. Humans right programs with the brains of hunters and gatherers.
- 14:39-16:39: Programming uses both System 1 (our Gut) and System 2 (our Head). Intuition is a part of programming.
- 16:40-18:01: Programming is about making tradeoffs. What any programming language (like JavaScript), there are good parts and there are bad parts. JavaScript has some of the *best good* parts and some of the worst *bad parts*.

(Video: 1_C.mp4): **JavaScript: Good Parts and Bad Parts**

- 18:02-20:00: JSLint is a code-quality tool that defines a professional subset of JavaScript. It lets you know when you are using the bad parts of JavaScript. (NOTE: The instructor video goes out from 19:23-19:32)
- 20:01-26:26: The curly brace debate: On the right or on the left? It's a simple example why we should prefer programming forms that are error resistant.
- 26:27-30:34: *Switch* statements can introduce fall-through hazards. The cascading effect can lead to more places where errors can live.

(Video: 1_D.mp4): **Understanding Good Programming Style**

- 30:35-32:00: In programming, saying “That hardly ever happens” is just another way of saying “It happens”. Good programming style can help produce better programs.
- 32:01-35:36: The evolution of literary styles (i.e. lowercase, word breaks, and punctuation) helped reduce errors when transcribing language. Same applies to programming.
- 35:37-37:50: Programs must communicate clearly to people. The programming process does not stop at the compiler. Good style makes code easier to understand.
- 37:51-41:40: There are places in JavaScript where good style also communicates intent. You see this with immediately invocable functions, semicolon use, *with* statements and more.

(Video: 1_E.mp4): **Avoiding Confusing Code (A)**

- 41:41-46:03: Confusing code should be avoided. Use programming forms that avoid adding confusion.
- 46:04-50:57: Looking at block scope versus function scope reveal common pitfalls with variable scope.

(Video: 1_F.mp4): **Avoiding Confusing Code (B)**

- 50:58-01:00:12: Additional pitfalls emerge from the use of global variables and syntax shortcuts.
- 01:00:13-01:02:35: Bad coding styles fall into four classes: Under Educated, Old School, Thrill Seekers, and Exhibitionist
- 01:02:36-01:05:38: Programming is the most complicated thing that humans do. Computer programs must be perfect. Humans are not good at perfection so designing a good programming style demands discipline.

(Video: 1_G.mp4): **Using JSLint**

- 01:05:39-01:09:20: The JSLint style was driven by the need to automatically detect defects. Not all features of a language should be used just because they are available. The key is finding a reliable subset within the language.
- 01:09:21-01:13:05: Performance-specific code can be confusing. Only optimize the code that is taking the time. Algorithm replacement is vastly more effective than code fiddling. Bugs will exist no matter what coding style you use. All programmers can do is limit the chance for bugs.
- 01:13:06-01:15:02: Part 1 wrap up and questions

Part 2: And Then There Was JavaScript (01:15:03-02:24:00)

(Video: 2_A.mp4): **The History of JavaScript**

- 01:15:03-01:28:35: JavaScript started with HyperCard. The developers of the first web browsers like Mosaic and Netscape wanted a way for people to make web pages interactive much the same way HyperCard made the Macintosh operating system interactive. First came LiveScript then came JavaScript.

- 01:28:36-01:30:06: ECMAScript evolved to its third edition in 1999. The fourth edition was abandoned. The fifth edition was released in 2009. It turns out ten years between editions ended up being a good thing for JavaScript.
- 01:30:07-01:31:43: The bad parts of JavaScript stem from Legacy, Good Intentions, and Haste. For the most part, these can be avoided.

(Video: 2_B.mp4): **JavaScript Fundamentals**

- 01:31:44-01:34:16: JavaScript is an object-oriented language. An object is a dynamic collection of properties. Almost everything in JavaScript is an object. Object literals are not only a good part of the language, but influenced the development of JSON.
- 01:34:17-01:41:04: While JavaScript does not have classes, it does have *prototypes*. Prototypes are much easier to work with than classes. Prototypes can utilize delegation or differential inheritance. However, in JavaScript, inheritance can sometimes work against you. Many other data types in JavaScript inherit from Object. These include Number, Boolean, String, Array Date, RegExp, and Function.

(Video: 2_C.mp4): **Numbers**

- 01:41:05-01:52:59: Exploring the Number type and using the Math object. *NaN* can be confusing, but ultimately it's the result of undefined or erroneous operation.

(Video: 2_D.mp4): **Strings and Arrays**

- 01:53:00-02:00:41: A look at the way Strings behave in JavaScript and common String operations.
- 02:00:42-02:06:14: In many languages, an Array is linear sequence of memory divided into equal sized blocks. In JavaScript, Arrays behave very differently. Sometimes they can be more efficient, most of the time they are less efficient.

(Video: 2_E.mp4): **Objects**

- 02:06:15-02:08:26: Arrays and Object are often interchangeable. It's best to use Objects when the names are arbitrary strings. Arrays are often more efficient when the names are sequential numbers.
- 02:08:27-02:13:31: All values are objects except for *null* and *undefined*. The *typeof* prefix operator will return a string identifying the type of a value. Use *Array.isArray* for Array validation.

(Video: 2_F.mp4): **Common JavaScript Statements**

- 02:13:32-02:20:13: JavaScript is syntactically a C family language and has very familiar identifiers, comments and operators.
- 02:20:14-02:22:29: Many of the statements in JavaScript like *if*, *switch*, *for* and *try/throw* will mirror those found it other C family languages.
- 02:22:30-02:24:00: Part 2 wrap-up and questions.

Part 3: Function the Ultimate (02:24:01-03:09:01)

(Video: 3_A.mp4): **Background on Functions**

- 02:24:01-02:28:41: Many object-oriented languages have methods, classes, constructors, and modules. JavaScript just has the *Function*. Best-practice suggests you declare all variables at the top of the function and declare all functions before you call them.
- 02:28:42-02:30:53: Within functions, you can either return the result of an expression or return nothing (*undefined*). Each function also receives two pseudo parameters: *arguments* and *this*.
- 02:30:54-02:34:39: The invocation operator can surround zero or more arguments. If a function is called with too many arguments, the extra arguments are ignored. Too few arguments will force *undefined* to be passed to the missing arguments. The four ways to call a function are Function form, Method form, Constructor form and Apply form.

(Video: 3_B.mp4): **Functions as Subroutines**

- 02:34:40-02:38:55: The idea of a subroutine is the origin of functions. Subroutines allow for code reuse, modularity and recursion.
- 02:38:56-02:49:19: With closures, the context of an inner function includes the scope of the outer function. An inner function continues to have access to that context even after the parent functions have returned. Function scope works like block scope.

(Video: 3_C.mp4): **Prototypal Inheritance (A)**

- 02:49:20-02:53:47: JavaScript can begin to look like a classical object-oriented language. Inheritance can be achieved through the prototype chain. However, pseudo-classical inheritance can look foreign to developers and lead to some issues.
- 02:53:48-02:55:05: The Prototypal Inheritance approach adds a constructor function which ensures proper inheritance. While it may not be better, it cleans up the implementation.

(Video: 3_D.mp4): **Prototypal Inheritance (B)**

- 02:55:06-03:07:28: Using functions as modules may be the best way to create inheritance. This module pattern can be easily transformed into a powerful constructor pattern using functional inheritance.
- 03:07:29-03:09:01: Part 3 Wrap-up and questions.

Part 4: Problems (03:09:02-03:48:54)

(Video: 4_A.mp4): **Problems 1-5**

- 03:09:02-03:12:10: A sequence of problems will be presented followed by a solution. Each problem builds on the last so if you get a problem wrong, use the solution to begin the next problem. First, a quick quiz: What is x?
- 03:12:11 -03:19:32
    - Problem 1: Write a function that takes an argument and returns that argument.

    - Problem 2: Write two binary functions, *add* and *mul*, that take two numbers and return their sum and product.
    - Problem 3: Write a function that takes an argument and returns a function that returns that argument.
    - Problem 4: Write a function that adds from two invocations.
    - Problem 5: Write a function that takes a binary function, and makes it callable with two invocations.

(Video: 4_B.mp4): **Problems 6-9**

- 03:19:33-03:32:25
    - Problem 6: Write a function that takes a function and an argument, and returns a function that can supply a second argument.
    - Problem 7: Without writing any new functions, show three ways to create the *inc* function.
    - Problem 8: Write *methodize*, a function that converts a binary function to a method.
    - Problem 9: Write *demethodize*, a function that converts a method to a binary function.

(Video: 4_C.mp4): **Problems 10-12**

- 03:32:26-03:37:26
    - Problem 10: Write a function *twice* that takes a binary function and returns a unary function that passes its argument to the binary function *twice*.
    - Problem 11: Write a function *compseu* that takes two unary functions and returns a unary function that calls both of them.
    - Problem 12: Write a function *compseb* that takes two binary functions and returns a function that calls both of them.

(Video: 4_D.mp4): **Problems 13-15**

- 03:37:27-03:48:54
    - Problem 13: Write a function that allows another function to only be called once.
    - Problem 14: Write a factory function that returns two functions that implement an up/down counter.
    - Problem 15: Make a revocable function that takes a *nice* function, and returns a revoke function that denies access to the *nice* function, and an invoke function that can invoke the nice function until it's revoked.
-

Part 5: Monads & Gonads (03:48:55-04:50:36)

(Video: 5_A.mp4): **Introduction to Monads**

- 03:48:55-03:55:40: Monads are a powerful functional pattern. There are two ways to use functional programming. One ways is by simply programming with function. The other way is much more mathematical.
- 03:55:41-03:58:14: In the real world, everything changes. Immutability makes it hard to interact with the world. Monads are a loophole in the function contract.

(Video: 5_B.mp4): **The Identity Monad**

- 03:58:15-04:02:12: While some experts believe in order to understand monads you need to first learn *Haskell and Category Theory*, all you really need is a solid understanding of JavaScript. A monad is an object.
- 04:02:13-04:07:29: A monad example: The identity monad.

(Video: 5_C.mp4): **The Ajax Monad**

- 04:07:30-04:13:09: The Ajax monad is another example of a monad design pattern. We can add the ability to bind additional methods to our monad to create Ajax functionality.

(Video: 5_D.mp4): **The Maybe Monad**

- 04:13:10-04:16:42: Null pointer exceptions can create many bugs within JavaScript code. Null pointer checkers are typically used to prevent against null values. The *maybe* monad behaves similar to NaN, but can be applied to null pointers.
- 04:16:43-04:19:58: While computers are getting better at concurrency, programming languages are not. Threads are an example of concurrency and can be difficult to handle. The alternative is turn based processing but there can be issues with this technique too.

(Video: 5_E.mp4): **The Promise Monad (A)**

- 04:19:59-04:22:16: Promises are an excellent mechanism for managing asynchronicity. A promise is an object that represents a possible future value.
- 04:22:17-04:25:22: An example of using promises can be found with the Filesystem API.

(Video: 5_F.mp4): **The Promise Monad (B)**

- 04:25:23-04:33:03: Using the composition pattern, a promise can be created by nesting *when* statements. The result will look very familiar: A promise is a monad.
- 04:33:04-04:34:29: There are a number of additional resources and videos regarding monads and monad-related concepts.

(Video: 5_G.mp4): **Audience Questions**

- 04:34:30-04:41:17:
    - What is your preferred text editor?
    - What is your favorite notation?
- 04:41:18-04:50:36: Additional audience questions
    - Why is the prototyping approach in object-oriented programming more powerful than a classes-based approach?