

FIT2102

Programming Paradigms

Lecture 2

JavaScript Functions, Objects and Classes

Dynamic Typing

Fluent programming with anonymous functions

Faculty of Information Technology



MONASH
University

Lecture 1 Conclusion

Assembly language offers minimal abstraction over the underlying computer architecture, it offers:

- the ability to name operations and memory locations symbolically;
- the ability to define procedures (more recently);
- conveniences for dealing with arrays and macros (not examined here)

C adds more understandable syntax,
but is still close to the machine execution model.

Languages we examine in future lectures will depart further and further from the underlying (von Neumann) computer architecture.

Lecture 2 Conclusion (cont...)

JavaScript is basically an imperative language with C or Java-like syntax, except that it is:

- Interpreted
- Functions are objects which can be assigned to variables

In coming weeks we will incorporate more “functional” abstractions into our JavaScript coding, before moving onto a language which significantly departs from the imperative model.

Imperative programming involves telling the computer how to compute step-by-step

Declarative programming describes what you want to do, not how you want to do it

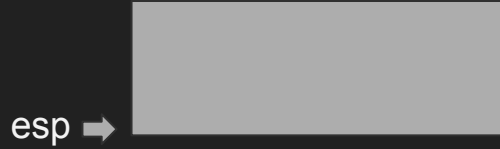
Lecture 2 Conclusion (cont... cont...)

We compared imperative and recursive loops:

- imperative loops are stateful and error prone
- recursion offers a more declarative way to code loops:
 - clear statement of **base case** and **loop invariant**
 - can be achieved with only **immutable variables**
 - STRONG CAVEAT: can result in **stack overflow**

In future we will return again to recursion in another language that supports efficient recursive loops

Nested Function Calls (inside the x86)



esp: Stack pointer register
points to next available byte on stack

Nested Function Calls



esp: “stack pointer” register
points to next available byte on stack

ebp: “base pointer” register, points to start of stack frame

Nested Function Calls



Nested Function Calls



Nested Function Calls



Stack overflow!

Learning Outcomes for Lecture 2

After lecture 2 you should be able to:

- Explain the relationship between javascript functions and objects
- Compare arrow functions and regular function syntax
- Create and apply anonymous functions to fluent style code
- Explain JavaScript's prototype scheme for creating classes from functions
- Create ES6 style classes with constructors and getters
- Compare object oriented polymorphism to dependency injection through functions

JavaScript

- A dynamically typed, interpreted language
- Developed by Brendan Eich, Netscape 1995
- Original idea: make Scheme the Netscape in-browser scripting language
- JavaScript developed as a more familiar (Java-like syntax) alternative
- Still has functions as first class citizens: functions are objects

JavaScript is Dynamically Typed

Types are associated with values rather than variables:

```
let i = 123;    // a numeric literal has type number  
i = 'a string'; // a string literal has type string, but no error here!
```

`let` - declare a reassignable (mutable) variable

`const` - declare a constant (immutable) variable

Prefer `const` where possible!

Immutable variables are easier to debug

Arrays

```
→ const tutors = ['tim', 'michael', 'yan', 'Yang', 'arthur', 'kelvin']  
tutors.length  
> 6  
tutors[1]  
> 'michael'  
tutors[1] = 'mic' ←  
tutors  
> ['tim', 'mic', 'yan', 'Yang', 'arthur', 'kelvin']
```

```
for(let i = 0; i < tutors.length; i++) {  
    console.log(tutors[i])  
}  
> tim  
Michael  
yan ...
```

The reference immutable,
but the referenced array is mutable

Avoid hand-coded loops!

```
for ([initialization]; [condition]; [final-expression])  
    statement
```

The ***initialization*** can initialise to the wrong value (e.g. n instead of $n-1$, 1 instead of 0) or initialise the wrong variable.

The ***condition*** test can use $=$ instead of $==$, $<=$ instead of $<$ or test the wrong variable, etc.

The ***final-expression*** can (again) increment the wrong variable.

The ***statement*** body might change the state of variables being tested in the termination condition since they are in scope.

JavaScript Objects

JavaScript objects are property bags:

```
const myObj = {  
  aProperty: 123,  
  anotherProperty: "tim was here"  
}
```

The following are equivalent and both involve a hashtable lookup:

```
console.log(myObject.aProperty)  
console.log(myObject['aProperty'])
```

JavaScript Functions Are Just Objects

```
function sayHello(person) {  
  console.log('hello ' + person)  
}  
sayHello('tim')  
> "hello tim"
```

We can bind a function to a variable just like any other object

```
const hi = sayHello  
hi('tim')  
> "hello tim"
```


JavaScript 101: First higher-order function

```
function square(x) {  
  return x * x;  
}
```

```
function sumTo(n, f) {  
  return n ? f(n) + sumTo(n-1, f) : 0;  
}
```

```
sumTo(10, square)
```

```
> 385
```

A higher-order function is one that:

- takes a function as argument;
- or returns a function.

Anonymous Functions

```
function hi(person) {  
    console.log('hello ' + person)  
}
```

```
const hi = function(person) {  
    console.log('hello ' + person)  
}
```

```
['tim', 'sally', 'anne'].forEach(hi)  
> "hello tim"  
> "hello sally"  
> "hello anne"
```

- Named function
- Anonymous function assigned to a variable
- Anonymous function passed to another function

JavaScript 101: First higher-order function

```
function sumTo(n, f) {  
  return n ? f(n) + sumTo(n-1, f) : 0;  
}
```

```
sumTo(10, function(x) {  
  return x * x;  
})
```

> 385

Arrow Function Syntax

```
function(x) { return <expression> }
```

Is (almost) equivalent to:

```
x => <expression>
```

Arrow functions can have more than one parameter:

```
function(a, b) { return <expression> }
```

```
(a, b) => <expression>
```

First higher-order function

```
function sumTo(n, f) {  
  return n ? f(n) + sumTo(n-1, f) : 0;  
}
```

```
sumTo(10, x => x * x)
```

> 385

Function type annotations (preparation for TypeScript):

```
function f(x) {  
    return x*2;  
}
```

TypeScript (TS) would infer a return type for this function of number (because that is the return type of '*').

In TS, we could make the type of this function explicit:

```
function f(x: number): number {  
    return x*2;  
}
```

We can speak more *generically* about types with *type variables*:

```
function f<T>(x:T, g:(x:T)=>T): T {  
    return g(x);  
}
```

When we write this function, we don't care about the specific type of T, but everything that is a T must have the same type as everything else that is a T.

Array Cheat Sheet: Pure Methods on Array

Where `a` is an array with elements of type `U`:

(Note: these are not correct TS annotations, but a Haskelly “shorthand”)

```
a.forEach(f: U=> void): void           // apply the function f to each element of the array

// Pure functions:
a.slice(): U[]                         // copy the whole array
a.slice(start: number): U[]           // copy from the specified index to the end of the array
a.slice(start: number, end: number): U[] // copy from start index up to (but not including) end index
a.map(f: U=> V): V[]                   // apply f to elements and return result in new array of type V
a.filter(f: U=> boolean): U[]          // returns a new array of the elements for which f returns true
a.concat(b: U[]): U[]                  // return a new array with the elements of b concatenated after
                                        // the elements of a (can take additional array arguments)
a.reduce(f: (V, U)=> V, V): V          // Uses f to combine elements of
                                        // the array into a single result of type V
```

All of the above are pure in the sense that they do not mutate `a`, but return the result in a new object.

Methods of Array

```
tutors.forEach(person=> console.log('hello ' + person))
```

```
> hello tim
```

```
hello michael
```

```
hello yan
```

```
hello yang
```

```
hello arthur
```

```
hello kelvin
```

```
tutors.map(person=> "hello " + person)
```

```
> ['hello tim', 'hello michael', 'hello yan', 'hello yang', 'hello arthur', 'hello kelvin']
```

```
tutors.filter(person=> person[0] == 'y')
```

```
> ['yan', 'yang']
```

```
tutors.map(p => p.length)
```

```
    .reduce((t,s)=>t+s,0)
```

```
> 29
```


Fluent programming with chained methods

```
tutors.map(e=>e.slice(0,3))  
  .filter(e => e[0] != 't')  
  .map(e=>e.toUpperCase())  
  .reduce((t,e)=> (!t[e] ? t[e]=1 : t[e]++, t), {})
```

t: accumulator

e: array element

Initial value passed into reduce

Multiple operations can be performed inside a single expression by separating them with a comma:

(op1, op2, op3)

Value of the expression will be the value of the last term, op3

```
> {MIC: 1, YAN: 2, ART: 1, KEL: 1}
```

Classes

```
class Person {  
  constructor(name, occupation) {  
    this.name = name  
    this.occupation = occupation  
  }  
  get greeting() {  
    return `Hi, my name's ${this.name} and I ${this.occupation}!`  
  }  
  sayHello() {  
    console.log(this.greeting)  
  }  
}
```

```
const tim = new Person("Tim", "lecture Programming Paradigms")  
tim.sayHello()
```

Hi, my name's Tim and I lecture Programming Paradigms!

Live coding exercise:

To be announced...

Creating Objects from a Prototype

```
function Person(name, occupation) {  
  this.name = name  
  this.occupation = occupation  
}  
  
Person.prototype.sayHello = function() {  
  console.log(`Hi, my name's ${this.name} and I'm ${this.occupation}!`)  
}  
  
const tim = new Person("Tim", "fun at parties")  
tim.sayHello()  
  
> Hi, my name's Tim and I'm fun at parties!
```

Note: with arrow syntax `this` refers to the enclosing scope, rather than the function object, i.e. can't use `=>` to declare methods that operate on the object

Creating Objects from a Prototype

```
function Person(name, occupation) {  
  this.name = name  
  this.occupation = occupation  
}
```

```
Person.prototype.sayHello = function() {  
  console.log(`Hi, my name's ${this.name} and I ${this.occupation}!`)  
}
```

```
const tim = new Person("Tim", "lecture Programming Paradigms")  
tim.sayHello()  
> Hi, my name's Tim and I lecture Programming Paradigms!
```

Note: with arrow syntax `this` refers to the enclosing scope, rather than the function object, i.e. can't use `=>` to declare methods that operate on the object

Creating Objects from Classes

```
class Person {  
  constructor(name, occupation) {  
    this.name = name  
    this.occupation = occupation  
  }  
  sayHello() {  
    console.log(`Hi, my name's ${this.name} and I ${this.occupation}!`)  
  }  
}
```

```
const tim = new Person("Tim", "lecture Programming Paradigms")  
tim.sayHello()  
> Hi, my name's Tim and I lecture Programming Paradigms!
```

Creating Objects from Classes

```
class Person {  
  constructor(name, occupation) {  
    this.name = name  
    this.occupation = occupation  
  }  
  sayHello() {  
    console.log(`Hi, my name's ${this.name} and I ${this.occupation}!`)  
  }  
}
```

```
const tim = new Person("Tim", "lecture Programming Paradigms")  
tim.sayHello()  
> Hi, my name's Tim and I lecture Programming Paradigms!
```

Getter property methods

```
class Person {  
  constructor(name, occupation) {  
    this.name = name  
    this.occupation = occupation  
  }  
  get greeting() {  
    return `Hi, my name's ${this.name} and I ${this.occupation}!`  
  }  
  sayHello() {  
    console.log(this.greeting)  
  }  
}
```

```
const tim = new Person("Tim", "lecture Programming Paradigms")  
tim.sayHello()
```

Hi, my name's Tim and I lecture Programming Paradigms!

Object-Oriented Polymorphism

```
class LoudPerson extends Person {  
    sayHello() {  
        console.log(this.greeting.toUpperCase())  
    }  
}  
  
const tims = [  
    new Person("Tim", "lecture Programming Paradigms"),  
    new LoudPerson("Tim", "shout about Programming Paradigms")  
]  
  
tims.forEach(t => t.sayHello())
```

Hi, my name's Tim and I lecture Programming Paradigms!

HI, MY NAME'S TIM AND I SHOUT ABOUT PROGRAMMING PARADIGMS!

Dependency Injection with Functions


```
class Person {  
  constructor(name, occupation, voiceTransform = g => g) {  
    this.name = name  
    this.occupation = occupation  
    this.voiceTransform = voiceTransform  
  }  
  sayHello() {  
    console.log(this.voiceTransform(this.greeting))  
  }  
}
```

Default to “identity” function



```
const tims = [  
  new Person("Tim", "lecture Programming Paradigms"),  
  new Person("Tim", "shout about Programming Paradigms", g => g.toUpperCase())  
]  
tims.forEach(t => t.sayHello())
```

We “inject” a
dependency on
toUpperCase from
outside the Person class



Hi, my name's Tim and I lecture Programming Paradigms!

HI, MY NAME'S TIM AND I SHOUT ABOUT PROGRAMMING PARADIGMS!

Closures and higher-order functions

A function and the set of variables it accesses from its enclosing scope is called a *closure*

```
function add(x) {  
  return y => y+x;  
}  
  
let addNine = add(9)  
addNine(10)  
> 19  
  
addNine(1)  
> 10
```

Functions that take other functions as parameters or that return functions are called *higher-order functions*.

Conclusion

In the tute, and more seriously from next week, we'll start to really explore the power of higher-order functions.

JavaScript has no compile time type checking. This gives:

- an economy of syntax
- no help when combining higher-order functions

Next week we'll start to use TypeScript, which extends JavaScript syntax with type annotations to help with the latter problem