BOSTON
UNIVERSITY

Department of Computer Science

CS412 Intro to JavaScript

@perrydBUCS

Some quick history

- JavaScript is not Java (not even a scripted version)
- Shortly after the web was born, Marc Andreesen (founder of Netscape) hired Brendan Eich to add support for embedded Scheme in Netscape Navigator
- Around the same time, Netscape and Sun worked together to add Java support in the form of applets to the browser
- Andreesen pivoted and decided that a ‘scripty’ version of Java was called for rather than Scheme, so that syntax would be similar
- Eich wrote a prototype of the language, then called Livescript (‘Mocha’ internally), in ten day May of 1995

- Betas shipped as Livescript starting in September of 1995
- Renamed to JavaScript in December 1995 (many say to ride Java's popularity)
- Microsoft reverse-engineered the interpreter and released JScript in 1996 in IE3 and IIS
- JavaScript and JScript were roughy the same syntax-wise, but different enough that devs had to write code for both browsers
- Netscape submitted the language to ECMA (European Computer Manufacturers Association, an international standards body) in 1996
- ECMA released ECMA-262 in 1997; JScript and JavaScript were two implementations of the spec

ECMAScript spec versions

- 1997: ECMA-262 (ECMAScript) standard, JavaScript / JScript
- 1998: ECMAScript 2
- 1999: ECMAScript 3
- At this point, Microsoft continued to diverge from the standard; although work was being done on an ES4 spec, industry wrangling killed it
- 2005: AJAX
- 2009: ES5 released
- 2011: ES5.1
- 2015: ES2015
- 2016: ES2016
- 2017: ES2017

JavaScript versions

- ES and JS don't share the same numbering scheme, since the former is a spec and the latter is an implementation
- Important JS versions:
  - 1.5 (2000)
  - 1.6 (2005)
  - 1.7 (2006)
  - 1.8 (2008)
  - 1.8.5 (2010)
- There's a massive amount of code at 1.6, 1.7, and 1.8

“Modern” JavaScript

- Past v1.8.0 we typically express the version of JS we are using as related to its underlying ECMAScript specification version
- ES6 (2015)
- ES7 (2016)
- ES8 (2017)
- ES9 (2018)
- Baseline code is ES6; runtimes such as Node and V8 implement features in later versions (Node, for example, currently supports some features in ES7 and ES8…it tracks V8)

So what is V8?

- While client-side (browser) applications became the most popular way to use JS, there's always been a server-side engine as well
- Google developed and open-sourced their JS engine, called V8, in 2008 as part of the Chromium project; Lars Bak was the principal
- JavaScript is an interpreted language, however V8, written in C++, JIT-compiles directly to x86 and ARM machine code
- NodeJS, a server-side JS implementation, includes and closely tracks the V8 engine — features released in V8 appear quickly in NodeJS
- Webassembly a spec for machine-code storage of executables, allows for some pre-compilation in most browsers

NodeJS

- Even though there were server-side implementations of JS engines from the very beginning, none gained significant traction
- In late 2009 Ryan Dahl demo'd and released the first version of Node.js, a JS library that used C++ to bind event processing to a V8 instance, using libuv for low-level asynchronous I/O
- NodeJS keeps current with V8 features
- The package allows us to build performant web servers / services with JavaScript
- We'll spend quite a bit of time in Node

So…

- JavaScript is a browser-based scripting language that runs in the context of a browser — it is event-driven
- Also runs as a server with Node
- It is an interpreted language that most often is JIT compiled
- We'll see soon that JS is built for asynchronous operation
- The language we think of as JS really is just an implementation of the ECMAScript specification — there are dozens of other implementations
- Let's take a look at language features, mainly those that are different from languages you might be used to

What version?

- There are unfortunately quite a few versions in active use
- We will primarily be working in ES6
- Node 11.8 is current and implements V8 7.0.276
- nb: I tend to bounce back and forth between ES6 and JS when talking…they're the same thing

JS: Basic language features

- Like most languages, JS is built of *expressions* and *statements*
  - An expression: a = 2; (actually a few expressions…how many?)
  - A statement: a = b + 2;
- Statements in JS end in a semicolon (;)
  - Except when they don't…

User I/O

- Most apps are UI driven, so input data comes from reading a box on a web page, from data in an event, and so on
- If you need to prompt the user, there's a 'prompt' function available, though it's a little flimsy:

```
day = prompt('What day is it?')
```

- Double quotes work here, too; convention is to use single quotes in JS and double quotes in HTML for strings
- NOTE: 'prompt( )' is not part of the base language…it's implemented at the browser; the function won't work in bare Node server-side

- ‘Printing’ is tricky in JS
- Recall that the language was intended to be run entirely in a browser
- Where would you print something?
- Like Java, there is a place to print — the console

```
console.log(‘Go Pats!’);
```

- This works on the client and the server; on the client you’ll need to open the browser console to see output

Comments

- // for inline comments
- /*   */ for block comments

Variables

- There are (at least) four ways to declare a variable

```
a = 42

var a = 42

let a = 42

const a = 42
```

- The differences have to do primarily with scope, though **const** also declares a constant value (though there's a catch, coming up)
- Note that in the case of a bare variable (ie a=42 above) JS assumes the declaration

Hoisting

- Prior to ES6, we only had **var** variables to work with
- During the first passes of the runtime engine, variable and function *declarations* are found and **hoisted** to the top of the enclosing context
- Pre-ES6 JS has two scopes: global and function
- In most cases this means that variable declarations are hoisted to the top of the file
- And so…

```
a = 42;
var a;
```

- …works just fine

- Note that only declarations, not initializations, are hoisted, so

```
console.log(b)  //b is undefined
var b = 22; //initialize b
```

- fails (b is initialized and so not hoisted)

Variable scope

- Hoisting makes some sense, but what about

```
for (var count = 0; count < 5; count++) {
    console.log(count);
}

console.log(count);
```

- What is printed?

- The problem here is that constructs that we expect to create a block-level local scope do not, such as **if**, **for**, **while**, and **switch**
- In the previous example, the entire file was in scope (global scope)
- Local scope (versus global scope) is created when using **var** only in function blocks
- It can get tricky when you have a large file and you aren't paying attention to variable scopes, especially with hoisted declarations, and ESPECIALLY since we are all used to short-cutting loops like this

```
for (counter = 0; counter < 5; counter++) {
    //counter has global scope here
}
```

Scope using let and const

- ES6 addresses the scope issue with two new variable declaration keywords, both of which honor block scope and are **not** hoisted
- In other words, they behave the way we expect scoped variables to behave
- Prior to this we just had global scope and function scope
- With both let and const, you may not use the variable prior to declaration (non-JS programmers give you a puzzled look and a hearty DUH if you say this out loud)

A word about const

- Variables declared const are fixed in value
- They must be defined when they are declared
- So

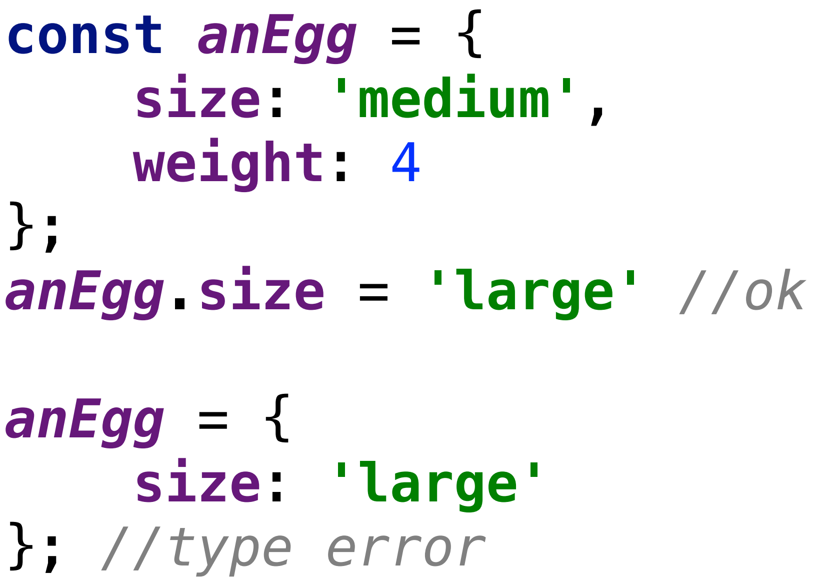
```
const foo;
foo = 42;
```

- is an error. The correct way is

```
const foo = 42;
```

Another word about const

- We haven't discussed objects yet, but a const reference to an object just locks the variable to the object; the internal object variables are not automatically const

- 

```
const anEgg = {
    size: 'medium',
    weight: 4
};
anEgg.size = 'large' //ok

anEgg = {
    size: 'large'
}; //type error
```

Variable advice

- Declare variables at the spot you need them (not at the top of a file)
- Use meaningful variable names
- Assume that var is deprecated — **only use const and let**
- Default to **const**; if something isn’t constant, then it’s **let**

JS variable types

- JavaScript is a loosely typed language
- In many languages, different types of variables are stored in differently sized memory allocation, so an integer might be two bytes while a long is four bytes wide
- Arrays, especially…most languages require static typing of array variables
- JS instead infers the type from context
- We say that the language implements **Duck Typing**
- This can lead to some interesting issues, especially around testing for equality

What does this print?

```
let a = "42"
let b = 42

console.log(a == b)
```

- JavaScript is the Yellow Lab of languages — it just wants you to be happy
- In this example

```
let a = "42"
let b = 42

console.log(a == b)
```

- JS sees the string on the left, the number on the right, and leaps to the conclusion that you really meant to treat them both as numbers
- JS converts the string to a number, then does the compare
- This is called type coercion, and JS does it a lot

Equality operators

- This often isn’t what we want!
- JS provides type-specific equality operators to explicitly state your intent
- Type-specific
  - ===
  - !===
- Duck typed
  - ==
  - !=
- Note that there isn’t a type-specific comparison set (e.g. no <==)

Other operators

- Operators are, for the most part, the same as other languages
  - Math: + - * / **
  - Assignment: =
  - Logic: && II

Arrays

- Since I mentioned arrays…
- Arrays use [ ] notation
- They can hold mixed types (unlike many languages)
- And of course they start indexing at 0 (zero)

Loops, conditionals, blocks, etc

- For the most part the rest of the JavaScripts's primitives are similar to other languages
- Loops are while and for (and do…while)
- Conditionals are the traditional if statements (if, else, else if)
- Blocks are defined with curly braces
  - As long as you are using let and const, blocks also define scope

- We'll look at functions in the next lecture

CS412 JS Functions and Objects

@perrydBUCS

Built-in types

- There are seven built-in types in JS
    - string
    - number
    - boolean
    - null
    - undefined
    - object
    - symbol
- Most of these behave as expected, except for null (we'll see that in a moment)

Symbol type

- Symbols are new in ES6
- They let us generate a unique identifier
- The identifier can be used as keys in structures such as maps, or as a way to uniquely identify a label, such as in an enumeration
- Note that JS doesn’t have an ‘enum’ operator as do other languages — we write enums either as strings or in objects
- Symbols have quite a few properties and methods, but they don’t seem to be in heavy use yet

undefined versus null

- Both are JS primitives
- An uninitialized variable will be ‘undefined’ until a value is set
- A few other operations will result in an ‘undefined’ value

- ‘null’ represents the absence of a value

- null == undefined (is true)   but   null === undefined (is false)

- Best practice: Use null to explicitly set an empty variable, and let JS handle undefined, even though they behave roughly the same

also…

- typeof(undefined) is “undefined”
- typeof(null) is “Object”

- (typeof( ) returns a string)

- That null has a type of “Object” was a bug in an early specification that was incorporated into ECMAScript, and there’s so much code that relies on it, fixing the bug is worse than letting it go

JS functions

- Functions are similar to other languages
- We can declare a named function

```
function adder(left, right) {

    return left + right;
}
console.log(`${adder(30,12)}`);
```

- Or declare a variable, then point it to a function

```
let adder2
adder2 = function (left, right) {
    return left+right;
}

console.log(`${adder2(30,12)}`);
```

- Note that const doesn’t work for adder2 since const requires a definition

using const

- I tend to define functions as const like this…
-

```
const adder2 = function (left, right) {
    return left + right;
}

console.log(`${adder2(30,12)}`);
```

=>

- ES6 introduced new function definition syntax based on CoffeeScript
- It’s a little more succinct but functionally equivalent (haha)

```
const adder3 = (left, right) => left + right;
console.log(`${adder3(30,12)}`);
```

- If a function has a single arg, no ( ) is required

```
const adder4 = left => left + 12;
console.log(`${adder4(30)}`);
```

- No args? use ( )

```
const adder5 = () => 30 + 12;
console.log(`${adder5(30)}`);
```

multi-line and =>

- Functions with multiple lines use { and } to enclose the function body

```
const adder6 = () => {
    const thirty = 30;
    const twelve = 12;
    return thirty + twelve;
}
console.log(`${adder6()}`);
```

- Note that in the previous one-line examples, the return is implicit

Functions as arguments

- Functions are first-class objects in JS, so they can be treated like any variable
- This means that we can pass a function

```
const doMath = (value, operation) => operation(value);

let result = doMath(
    30,
    val => val + 12
)
console.log(result);
```

- …or return a function

```
const getOperation = operator => {

    switch (operator) {
        case '+':
            return (left, right) => left + right;
            break;
    }
}

let mathFunction = getOperation('+');
console.log(mathFunction(30,12))
```

Passing lambdas

- Passing unnamed (lambda) functions is extremely common
- We typically use them to handle asynchronous events
- These are called **callbacks**
- A somewhat contrived example:

```
const adder7 = (left, right, cb) => cb(left + right);

let result = adder7(15, 6,
        sum => sum*2
        )
console.log(result)
```

IIFEs

- Immediately Invoked Function Expressions
- Remember the global scope issues mentioned earlier?
- It gets even worse when we start including other JS files in our code… what if the included file has a global variable with the same name?
- Java fixes this with namespaces
- We fix it by constructing a file-level function that runs immediately
- This creates a function-level scope for the entire file
- Here’s an example…

```
(function() {
//Everything else goes here
})()
```

- This works because the ( ) at the very end executes the function
- Just like add(2,3)…the (2,3) executes add with the two params
- The opening ‘(‘ and the matching ‘)’ on the last line are there to prevent JS from thinking this is just a function definition

JS Objects

- ‘object’ is a little bit of a stretch name…ES6 out of the box doesn’t provide classic object-oriented features such as data hiding
- We can still write OO in JS as long as we are aware of the limitations
- Typescript and newer ES specs do provide a fairly full OO implementation
- Still, objects in JS are pretty useful

Object notation

- Objects are enclosed in curly braces { }
- They can contain both attributes and behaviors (variables and functions)
- Constructors are used as in classic OO languages, however for one-off objects they aren't required
- When using a constructor, the **new** keyword instantiates an object
- The **this** keyword points to the in-context object

ADOs, POJOs, etc

- In the absence of any constructor or functions, we very often treat an object as an abstract data object or plain old JavaScript object:

```
const colorCodes = {
    blue:   1,
    red:    2,
}

console.log(`Blue is code ${colorCodes.blue}`);
```

About const here…

- In this example, const refers to the variable colorCodes, not the elements in the object
- That means that we can change them

```
const colorCodes = {
    blue:   1,
    red:    2,
}
colorCodes.blue = 42;
console.log(`Blue is code ${colorCodes.blue}`);

//Prints 42
```

Functions in objects

- A function is just another element in an object
- If the object will be instantiated with **new**, we have to use the **this** keyword to reference any internal object properties
- Constructors are identified by naming a function:

```
function Egg() {
    this.weight = 0
    //we'll only worry about weight
    this.setWeight = function (min, max) {
            this.weight =  Math.random() * (max - min) + min;
    }
    this.getWeight = function () {
        return this.weight;
    }
}
const egg = new Egg(); //Instantiation
egg.setWeight(2,8) //set the weight to between min, max ounces
```

Object destructuring

- ES6 adds a handy way to pass multiple parameters into a function using an object
- Consider:

```
const divider = ({top, bottom}) => top / bottom;

console.log(divider({top:8, bottom: 2}))  //4
console.log(divider({bottom:2, top: 8}))  //4
```

- As long as the names in the parameter object match, values will be assigned to the appropriate variable in the called function
- This gets rid of having to remember in what order a function wants its params to be

Also for return values

- Destructuring works in both directions

```
const squareAndCube = x => [x*x, x*x*x, x*x*x*x];
const [s,c,d] = squareAndCube(3);
console.log(`Square: ${s}\nCube: ${c}\nQuad: ${d}`);
```

Classes and inheritance

- ES6 provides a much cleaner way to write class definitions than in ES5
-
```
class food  {
    constructor(size) {
        this.size = size
    }
     getSize () { return this.size}
}
class egg extends food {
    constructor(color, size) {
        super(size)
        this.color = color
    }
    getColor() { return this.color}
}

let myFood = new food(4.0)
let myEgg = new egg("blue", 5)
console.log("Size: ", myFood.getSize())
console.log("Color: ", myEgg.getColor())
```

BOSTON UNIVERSITY

Department of Computer Science

CS412 Generators, Iterators, Default Params

@perrydBUCS

Closures and global variables

- Consider this snippet

```
let funcs = [];

for (var num=0; num<4; num++) {
    funcs.push(
        () => console.log(`Num: ${num}`)
    )
}
funcs[2];
```

- What is printed on the console?

- What’s happening here is that the use of **var** in the loop definition causes num to have global visibility
- Each iteration of the loop binds a reference to **num** in a function
- Since num is global, when we run the functions from the array, they all point to the same variable instance

- To do this right we need the closure created by the function definition to refer to its own instance of num

```
let funcs = [];

for (let num=0; num<4; num++) {
    funcs.push(
        () => console.log(`Num: ${num}`)
    )
}
funcs[2]();
```

- let binds to block scope, and when the function is defined, its closure gets a reference to the current value of **num**

Spread / Rest operator

- The . . . operator (three dots typed together) is both the **spread** and the **rest** operator in ES6
- We usually see … used in the context of an array, but technically it should work with any iterable
- The idea is that we want to ‘explode’ an array into individual items (spread), or take a bunch of items and jam them into an array (rest)

spread

- For example…

```
let spr = (a, b, c) => console.log(a,b,c);
let anArray = [1,2,3];
spr(...anArray);
```

- This prints 1 2 3

rest

- And

  ```
  let rst = (a, b, ...c) => console.log(a,b,c);
  rst(1,2,3,4,5,6);
  ```

- This prints 1 2 [3, 4, 5, 6]

rest and for-of

- A common use case for rest is when you have a function that expects an arbitrary number of parameters

```
let func = (...args) => {
    console.log(args);
    for (const arg of args) {
        console.log(arg);
    }
}
func(1,2,3,4,5,6);
```

- This prints   [1, 2, 3, 4, 5, 6] and then 1 2 3 4 5 6, each on a separate line

for-in

- Mentioning since the syntax looks similar…
- If you need to get the properties of an object, use a for-in loop rather than for-of

```
let foo = {
    color: 'red',
    size: 'large'
};
for (const val in foo) {
    console.log(val);
}
```

- This prints color size (each on a separate line)

Getting values of object properties

- Sometimes you just need the names of the properties, or to see if an object has a specific property (though there is a function on Object to do that explicitly)
- Most of the time you need the values…

```
let foo = {
    color: 'red',
    size: 'large'
};
for (const val in foo) {
    console.log(foo[val]);
}
```

- This prints   red   large   (each on a separate line)

Default function params

- It’s often handy to provide a set of defaults for a function
- The pre-ES6 way of handling this is astoundingly ugly

```
//From YDKJ
function foo(x,y) {
    x = x || 11;
    y = y || 31;

    console.log( x + y );
}

foo();           // 42
foo( 5, 6 );     // 11
foo( 5 );        // 36
foo( null, 6 );     // 17
```

- This has a really awful side effect due to the way that JS defines ‘truthy’ and ‘falsy’ values
- In JS, the value 0 is false
- What happens when we do this…

```
//From YDKJ
function foo(x,y) {
    x = x || 11;
    y = y || 31;

    console.log( x + y );
}

foo(0,31);
```

- We can fix this problem with 0 by checking a little more closely…

```
function foo(x,y) {
    x = (x !== undefined) ? x : 11;
    y = (y !== undefined) ? y : 31;

    console.log( x + y );
}

foo( 0, 42 );      // 42
foo( undefined, 6 );  // 17
```

- Why does the second call print 17?
- What happens if you actually want to pass in **undefined** for some reason?

- The fix for handling undefined as an actual value looks like

```
function foo(x,y) {
    x = (0 in arguments) ? x : 11;
    y = (1 in arguments) ? y : 31;

    console.log( x + y );
}

foo( 5 );                 // 36
foo( 5, undefined );  // NaN
```

- What if you want to pass the second value but not the first?

```
foo( , 5 ); // NaN
```

Order of default params

- Not surprisingly, you can only omit values at the end of a param list
- Can’t omit ones in the middle, either
- This is true in C++ (and most languages), so it makes sense that JS, written in C++, has the same behavior

ES6 default params

- In ES6, default params are set explicitly

```
let bar = (a, b=22) => a + b;

console.log(
    bar(20)
)
```

- The assignment is similar to the

```
x !== undefined ? x : 11
```

operation from a previous slide, with similar side effects

Default expressions

- The default values can also be expressions…

```
let bar = (a = baz(a), b=22) => a + b;

let baz = a => a*2;

console.log(
    bar(20)
)
```

- This prints 42
- Why?

Lazy execution

- The expression in the default param list is only executed if it is needed, that is if the param is either undefined or omitted
- This should work…

```
let bar = (a = 22, b = baz(a)) => a + b;

let baz = a => a*2;

console.log(
    bar(20), bar()
)
```

Iterators

- An iterator is a function that returns the values of an iterable item one at a time
- For example, if we have the array [1,2,3,4,5], an iterator on the array would first return 1, then 2, the 3, and so on
- Even though iteration is a basic language concept, JS didn’t have formal iterators until ES6
- Most built-in objects implement the Iterable interface, and user-defined objects also can provide an iterator across their internal data members
- The interface also provides a flag (done) that is set to true when you’ve released the last item

- Here’s an array, which by default implements Iterable

```
const anArray = [1,2,3];

const arrayIterator = anArray[Symbol.iterator]();

let val = arrayIterator.next();
console.log(`Val: ${val.value}, Flag: ${val.done}`);

val = arrayIterator.next();
console.log(`Val: ${val.value}, Flag: ${val.done}`);

val = arrayIterator.next();
console.log(`Val: ${val.value}, Flag: ${val.done}`);

val = arrayIterator.next();
console.log(`Val: ${val.value}, Flag: ${val.done}`);
```

Generators

- Sometimes we want an iterable that isn’t a set list of data…it should create a new value using some pattern each time it is called
- ES6 gives us generators for this purpose, along with some new syntax and keywords
- When a generator has exhausted its values, it returns a **done** flag set to **true**
- These are basically pause-able functions…they don’t run to completion
- They also are restartable
- Best illustrated with an example…

- Generators are functions that are marked with the * symbol
- Each time the generator is called, it returns the next item in its **yield** list
- The yield might also be an expression (we'll see this shortly)

```
function* listGen () {
    yield 1;
    yield 2;
    yield 3;
}

const x = listGen();

console.log(`${x}`)
```

- What does this print?

- Generators return an iterator, which we then must access in order to walk through the list of generated items
- Essentially you are creating a custom iterator

```
function* listGen () {
    yield 1;
    yield 2;
    yield 3;
}

const x = listGen();

const y = x.next();

console.log(`${y}`)
```

Generators that yield via expression

- The generator’s state doesn’t need to be hard-coded; it can be any valid expression. Here, variables hold state:

```
function* fibs () {
    let [val1, val2, result] = [0, 1, 0]
    while (true) {
        result = val1+val2
        val1 = val2
        val2 = result
        yield result
    }
}


//Get a few fibs
myFibs = fibs()
let count = 5;
while (count --> 0) {
    console.log(myFibs.next().value)
}
```

Passing values to generators

- We can also seed a generator with an in put value or values

```
function* fibs (x = 0) {
    let [val1, val2, result] = [x,x-1,0]
//    let [val1, val2, result] = [0, 1, 0]
    while (true) {
        result = val1+val2
        val1 = val2
        val2 = result
        yield result
    }
}
//Get a few fibs
myFibs = fibs(4) //not really fib(4), just shows passing param
let count = 5;
while (count --> 0) {
    console.log(myFibs.next().value)
}
```

When is a generator done?

- For while(true) sort of loops, never
- If there is a finite sequence, the generator will emit each value in turn with the **done** flag set to **false**
  - until one more call, which emits `{value: undefined, done: true}`
  - At that point the generator's internal GeneratorState is set to completed, and the generator is done
- Generators don't have a constructor, so you can't 're-instantiate' them
- You *could* pass a generator into a new scope, which would give you a fresh copy

Getting all the values of a generator

- Since a generator function returns an iterable, we can use a **for…of** loop to iterate over its results (not for…in)

```
function* fibs () {
    let [val1, val2, result] = [0, 1, 0]
    while (result < 100) {
        result = val1+val2
        val1 = val2
        val2 = result
        yield result
    }
}
//Get a few fibs
for (fib of fibs()) {
    console.log(fib)
}
```

- The spread operator (…) works, too, since it expands an iterable

- This is essentially a function that maintains state (something we have been trained to avoid)
- Nevertheless generators can be extremely useful as a way to build a self-contained state machine that is pausable

Passing values into a generator

- We can pass initial params into a generator in the normal way
- What does this print?

```
function* test(x) {
    console.log(`In gen: ${x}`)
    yield x;

}
let xx = test(3);
console.log(xx)
```

Why isn’t anything printed?

- `let xx = test(3)` only gives us a reference to an iterator
- It doesn’t actually run the generator
- It’s the first **.next()** that runs the generator up until the first yield, then pauses
- When passing an argument, like test(3), the first time, the generator discards the argument
- An argument passed on subsequent calls is capture by the yield keyword

```
function* test() {
    console.log(`0 Starting`);
    console.log(`1 ${yield}`);
    console.log(`2 ${yield}`);

}
let xx = test();
console.log(xx.next())      //start the generator
console.log(xx.next(22))    //pass 22 to first yield
console.log(xx.next(4444))  //pass 4444 to second yield
console.log(xx.next(99))    //generator is done, no output
```

Pointing yield to an interable

- If the generator is going to return a series of known values, you can use a one-line yield statement
- yield can point to any iterable, such as an array

```
function* getArrayElements () {
    yield* [5,4,3,2,1] //note the *
}

const gae = getArrayElements();
console.log(gae.next());
console.log(gae.next());
console.log(gae.next());
```

- Yes, you can point it to another generator if you need to