BOSTON UNIVERSITY

Department of Computer Science

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*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())
```