JavaScript Programming

How do you program in JavaScript?

From Wikipedia:

> …
>
> ... supporting **object-oriented**, **imperative**, and **functional programming**
>
> ...

- Originally programming conventions (i.e. patterns) rather than language features
  - ECMAScript adding language features (e.g. `class`, `=>` , etc.)

Object-oriented programming: methods

- With first class functions a property of an object can be a function

```
let obj = {count: 0};
obj.increment = function (amount) {
    this.count += amount;
    return this.count;
}
```

- Method invocation: calls function and binds `this` to be object

```
obj.increment(1);  // returns 1
obj.increment(3);  // returns 4
```

this

- In methods `this` will be bound to the object

```
let o = {oldProp: 'this is an old property'};
o.aMethod = function() {
  this.newProp = "this is a new property";
  return Object.keys(this);  // will contain 'newProp'
}
o.aMethod(); // will return ['oldProp','aMethod','newProp']
```

- In non-method functions:
  - `this` will be the global object
  - Or if `"use strict"; this` will be `undefined`

functions are objects - can have properties

```
function plus1(value) {
    if (plus1.invocations == undefined) {
        plus1.invocations = 0;
    }
    plus1.invocations++;
    return value + 1;
}
```

- `plus1.invocations` will be the number times function is called
- Acts like static/class properties in object-oriented languages

function are objects: Have methods

```
function func(arg) { console.log(this,arg); }
```

- `toString()` method - return function as source string
  - `func.toString()` returns `'function func(arg) { console.log(this,arg); }'`
- `call()` method - call function specifying `this` and arguments
  - `func.call({t: 1}, 2)` prints `'{ t: 1 } 2'`
  - apply() like call() except arguments are passed as an array - `func.apply({t: 2},[2])`
  - `this` is like an extra hidden argument to a function call and is used that way sometimes
- `bind()` method - creates a new function with this and arguments bound
  - `let newFunc = func.bind({z: 2}, 3);`
  - `newFunc()` prints `'{ z: 2 } 3'`

Object-oriented programming: classes

Functions are classes in JavaScript: Name the function after the class

```
function Rectangle(width, height) {
    this.width = width;
    this.height = height;
    this.area = function() { return this.width*this.height; }
}
let r = new Rectangle(26, 14);   // {width: 26, height: 14}
```

Not correct way of adding methods

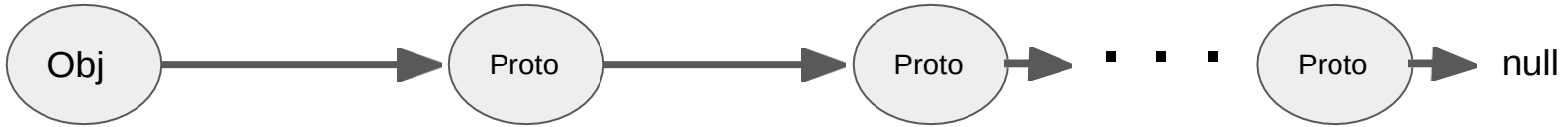
Functions used in this way are called **object constructors**:

```
r.constructor.name == 'Rectangle'
```

console.log(r): `Rectangle { width: 26, height: 14, area: [Function] }`

Object-oriented programming: inheritance

- Javascript has the notion of a **prototype** object for each object instance
  - Prototype objects can have prototype objects forming a **prototype chain**



- On an object property read access JavaScript will search the up the prototype chain until the property is found
  - Effectively the properties of an object are its **own** property in addition to all the properties up the prototype chain. This is called prototype-based inheritance.
- Property updates are different: always create property in object if not found

Using prototypes

```
function Rectangle(width, height) {
    this.width  =  width;
    this.height = height;
}
Rectangle.prototype.area = function() {
    return this.width*this.height;
}
let r = new Rectangle(26, 14);   // {width: 26, height: 14}
let v = r.area();       // v == 26*14
Object.keys(r) == [ 'width', 'height' ] // own properties
```

Note: Dynamic - changing prototype will cause all instances to change

Prototype versus object instances

```
let r = new Rectangle(26, 14);
```

Understand the difference between:

```
r.newMethod = function() { console.log('New Method called'); }
```

And:

```
Rectangle.prototype.newMethod =
        function() { console.log('New Method called'); }
```

Prototype versus object instances

```
function Parent(gender){
  this.gender = gender;
  this.yellAtChild = function(){console.log('Somebody gonna get a hurt real bad!');}
}
// Let's create dad and mom and start yelling at kids.
var dad = new Parent('male');
var mom = new Parent('female');
dad.yellAtChild(); // Somebody gonna get a hurt real bad!
mom.yellAtChild(); // Somebody gonna get a hurt real bad!
// ERROR: Not possible to do this way.
Parent.yellAtChild = function() { .... }
// You need to override the `yellAtChild` method for each object instance.
dad.yellAtChild = function(){ console.log('Shut up!');};
mom.yellAtChild = function(){ console.log('Go to bed!');}
dad.yellAtChild(); // Shut up!
mom.yellAtChild(); // Go to bed!
```

Prototype versus object instances

```
function Parent(gender){
  this.gender = gender;
}
// Attach the common function to prototype.
Parent.prototype.yellAtChild = function(){
  console.log('Somebody gonna get a hurt real bad!');
};
// Let's create dad and mom and start yelling at kids.
var dad = new Parent('male');
var mom = new Parent('female');
dad.yellAtChild(); // Somebody gonna get a hurt real bad!
mom.yellAtChild(); // Somebody gonna get a hurt real bad!
Parent.prototype.yellAtChild = function(){
  console.log('You are grounded.');
};
dad.yellAtChild(); // You are grounded
mom.yellAtChild(); // You are grounded
```

Inheritance

```
Rectangle.prototype = new Shape(...);
```

- If desired property not in `Rectangle.prototype` then JavaScript will look in `Shape.prototype` and so on.
  - Can view prototype objects as forming a **chain**. Lookups go up the prototype chain.
- Prototype-based inheritance
  - Single inheritance support
  - Can be dynamically created and modified

ECMAScript version 6 extensions

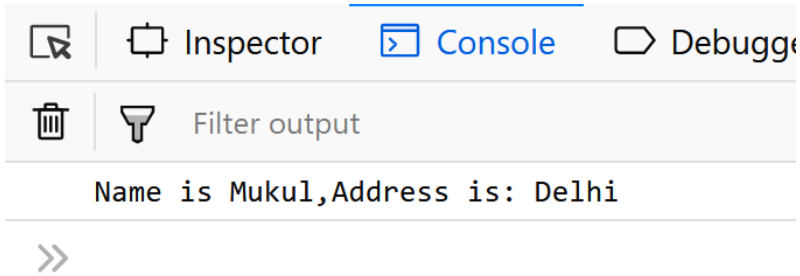
```
class Rectangle extends Shape {  // Definition and Inheritance
  constructor(height, width) {
     super(height, width);
     this.height = height;
     this.width = width;
  }
  area() {                        // Method definition
    return this.width * this.height;
  }
  static countRects() {          // Static method
     ...
  }
}

let r = new Rectangle(10,20);
```

React.js example class

```
class HelloWorld extends React.Component {
   constructor(props) {
    super(props);
    ...
  }
  render() {
    return (
      <div>Hello World</div>
    );
  }
}
```

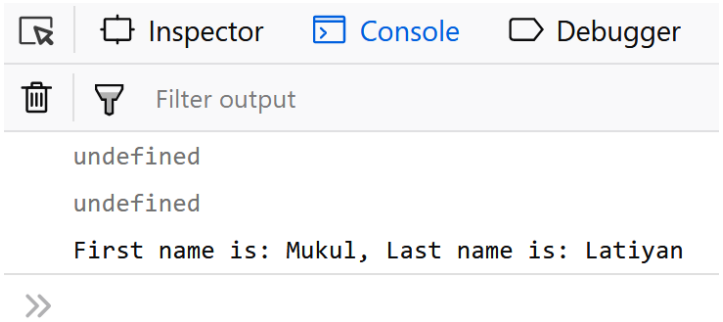
Encapsulation

```
// Encapsulation example
class person {
    constructor(name, id) {
        this.name = name;
        this.id = id;
    }
    add_Address(add) {this.add = add;}
    getDetails() {
        console.log(`Name is ${this.name},
        Address is: ${this.add}`);
    }
}
let person1 = new person('Mukul', 21);
person1.add_Address('Delhi');
person1.getDetails();
```



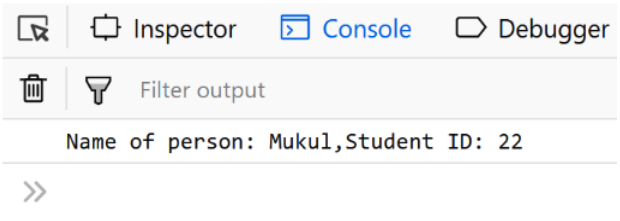
Abstraction

```
// Abstraction example
function person(fname, lname) {
    let firstname = fname;
    let lastname = lname;
    let getDetails_noaccess = function () {
        return (`First name is: ${firstname} Last name is: ${lastname}`);
    }
    this.getDetails_access = function () {
        return (`First name is: ${firstname}, Last name is: ${lastname}`);
    }
}
let person1 = new person('Mukul', 'Latiyan');
console.log(person1.firstname);
console.log(person1.getDetails_noaccess);
console.log(person1.getDetails_access());
```



Inheritance

```
// Inheritance example
class person {
    constructor(name) { this.name = name;}
    toString() { return (`Name of person: ${this.name}`); }
}
class student extends person {
    constructor(name, id) {
        // super keyword for calling the above class constructor
        super(name);
        this.id = id;
    }
    toString() {
        return (`${super.toString()}, Student ID: ${this.id}`);
    }
}
let student1 = new student('Mukul', 22);
console.log(student1.toString());
```



Functional Programming

- Imperative:

```
for (let i = 0; i < anArr.length; i++) {
  newArr[i] = anArr[i]*i;
}
```

- Functional:

```
newArr = anArr.map(function (val, ind) {
   return val*ind;
});
```

- Can write entire program as functions with no side-effects

```
anArr.filter(filterFunc).map(mapFunc).reduce(reduceFunc);
```

Functional Programming - ECMAScript 6

- Imperative:

```
for (let i = 0; i < anArr.length; i++) {
  newArr[i] = anArr[i]*i;
}
```

- Functional:

```
newArr = anArr.map((val, ind) => val*ind);  // Arrow function
```

- Can write entire program as functions with no side-effects

```
anArr.filter(filterFunc).map(mapFunc).reduce(reduceFunc);
```

Arrow functions don't redefine `this`

Functional Programming – Pure vs Impure functions

- Pure functions take some input and give a fixed output. Also, they cause no side effects in the outside world.

```
const add = (a, b) => a + b;
```

  - Here, `add` is a pure function. This is because, for a fixed value of `a` and `b`, the output will always be the same.

```
const SECRET = 42;
const getId = (a) => SECRET * a;
```

  - `getId` is not a pure function. The reason being that it uses the global variable `SECRET` for computing the output. If `SECRET` were to change, the `getId` function will return a different value for the same input. Thus, it is not a pure function.

Pure functions in JavaScript

- Filter

```
array.filter(condition);
const filterEven = x => x%2 === 0;
[1, 2, 3].filter(filterEven);
// [2]
```

- Reduce

```
array.reduce(reducer);
const sum = (accumulatedSum, arrayItem) => accumulatedSum + arrayItem
[1, 2, 3].reduce(sum);
// 6
```

- Map

```
array.map(mapper)
const double = x => 2 * x;
[1, 2, 3].map(double);
// [2, 4, 6]
```

- Concat (same as Spread operator)

```
[1, 2].concat([3, 4])
// [1, 2, 3, 4]
```

You can create your own pure function!

JavaScript Closures

- Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.
  You could use a global variable, and a function to increase the counter:

```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  counter += 1;
}

// Call add() 3 times
add();
add();
add();

// The counter should now be 3
```

There is a problem with the solution above: Any code on the page can change the counter, without calling add().

JavaScript Closures

- The counter should be local to the add() function, to prevent other code from changing it:

```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  let counter = 0;
  counter += 1;
}

// Call add() 3 times
add();
add();
add();

//The counter should now be 3. But it is 0
```

It did not work because we display the global counter instead of the local counter.

JavaScript Closures

- All functions have access to the global scope. In fact, in JavaScript, all functions have access to the scope "above" them.
  JavaScript supports nested functions. Nested functions have access to the scope "above" them.
  In this example, the inner function plus() has access to the counter variable in the parent function:

```
function add() {
  let counter = 0;
  function plus() {counter += 1;}
  plus();
  return counter;
}
```

This could have solved the counter dilemma, if we could reach the plus() function from the outside.
We also need to find a way to execute counter = 0 only once.
We need a closure.

JavaScript Closures

```
const add = (function () {
      let counter = 0;
      return function () {counter += 1; return counter}
    })();

    add();
    add();
    add();

    // the counter is now 3
```

- The variable add is assigned to the return value of a self-invoking function.
- The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.
- This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope (even when it is closed).
- This is called a JavaScript closure. It makes it possible for a function to have "private" variables.
- The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

JavaScript: The Bad Parts

Declaring variables on use - Workaround: Force declarations

```
let myVar = 2*typeoVar + 1;
```

Automatic semicolon insertion - Workaround: Enforce semicolons with checkers

```
return
  "This is a long string so I put it on its own line";
```

Type coercing equals: `==` - Workaround: Always use `===`, `!==` instead

`("" == "0")` is `false` but `(0 == "")` is `true`, so is `(0 == '0')`

`(false == '0')` is `true` as is `(null == undefined)`

`with`, `eval` - Workaround: Don't use

Some JavaScript idioms

- Assign a default value

```
hostname = hostname || "localhost";
port = port || 80;
```

- Access a possibly undefined object property

```
let prop = obj && obj.propname;
```

- Handling multiple `this`:

```
fs.readFile(this.fileName + fileNo, function (err, data) {
   console.log(this.fileName, fileNo);  // Wrong!
});
```

Some JavaScript idioms

- Assign a default value

```
hostname = hostname || "localhost";
port = port || 80;
```

- Access a possible undefined object property

```
let prop = obj && obj.propname;
```

- Handling multiple `this`: `self`

```
let self = this;
fs.readFile(self.fileName + fileNo, function (err, data) {
   console.log(self.fileName,fileNo);
});
```

Some JavaScript idioms

- Assign a default value

```
hostname = hostname || "localhost";
port = port || 80;
```

- Access a possible undefined object property

```
let prop = obj && obj.propname;
```

- Handling multiple `this`:

```
fs.readFile(this.fileName + fileNo, (err, data) =>
   console.log(this.fileName,fileNo)
);
```