Arrow Functions

A short hand notation for `function()`, but it does not bind `this` in the same way.

```
var odds = evens.map(v => v + 1);  // no parentes and no brackets
var nums = evens.map((v, i) => v + i);
var pairs = evens.map(v => ({even: v, odd: v + 1}));

// Statement bodies
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});
```

How does `this` work?

```
var object = {
    name: "Name",
    arrowGetName: () => this.name,
    regularGetName: function() { return this.name },
    arrowGetThis: () => this,
    regularGetThis: function() { return this }
}

console.log(this.name)
console.log(object.arrowGetName());
console.log(object.arrowGetThis());
console.log(this)
console.log(object.regularGetName());
console.log(object.regularGetThis());
```

They work well with classes

```
class someClass {
    constructor() {
        this.name = "Name"
    }

    testRegular() {
        return function() { return this }

    }

    testArrow() {
        return () => this.name;
    }
}

var obj = new someClass();

console.log(obj.name)
console.log(obj.testRegular()());
console.log(obj.testArrow()());
```

Classes

As we know them from "real" languages. Syntactic sugar on top of prototype-inheritence.

```
class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
    super(geometry, materials);

    this.idMatrix = SkinnedMesh.defaultMatrix();
    this.bones = [];
    this.boneMatrices = [];
    //...
  }
  update(camera) {
    //...
    super.update();
  }
  get boneCount() {
    return this.bones.length;
  }
  set matrixType(matrixType) {
    this.idMatrix = SkinnedMesh[matrixType]();
  }
  static defaultMatrix() {
    return new THREE.Matrix4();
  }
}
```

Enhanced Object Literals

```
var theProtoObj = {
  toString: function() {
    return "The ProtoOBject To string"
  }
}

var handler = () => "handler"


var obj = {
    // __proto__
    __proto__: theProtoObj,

    // Shorthand for ‘handler: handler’
    handler,

    // Methods
    toString() {

     // Super calls
     return "d " + super.toString();
    },

    // Computed (dynamic) property names
    [ "prop_" + (() => 42)() ]: 42
};
```

```
console.log(obj.handler)
console.log(obj.handler())
console.log(obj.toString())
console.log(obj.prop_42)
```

String interpolation

Nice syntax for string interpolation

```
var name = "Bob", time = "today";

var multiLine = `This

Line

Spans Multiple

Lines`


console.log(`Hello ${name},how are you ${time}?`)
console.log(multiLine)
```

Destructuring

```
// list "matching"
var [a, , b] = [1,2,3];
console.log(a)
console.log(b)
```

Objects can be destructured as well.

```
nodes = () => { return {op: "a", lhs: "b", rhs: "c"}}
var { op: a, lhs: b , rhs: c } = nodes()
console.log(a)
console.log(b)
console.log(c)
```

Using Shorthand notation.

```
nodes = () => { return {lhs: "a", op: "b", rhs: "c"}}

// binds `op`, `lhs` and `rhs` in scope
var {op, lhs, rhs} = nodes()

console.log(op)
console.log(lhs)
console.log(rhs)
```

Can be used in parameter position

```
function g({name: x}) {
  return x
}

function m({name}) {
  return name
}

console.log(g({name: 5}))
console.log(m({name: 5}))
```

Fail-soft destructuring

```
var [a] = []
var [b = 1] = []
var c = [];
console.log(a)
console.log(b);
console.log(c);
```

Default

```
function f(x, y=12) {
  return x + y;
}

console.log(f(3))
```

Spread

In functions:

```
function f(x, y, z) {
  return x + y + z;
}
// Pass each elem of array as argument
console.log(f(...[1,2,3]))
```

In arrays:

```
var parts = ["shoulders", "knees"];
var lyrics = ["head", ...parts, "and", "toes"];

console.log(lyrics)
```

Spread + Object Literals

We can do cool stuff with this in object creations.

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
console.log(x); // 1
console.log(y); // 2
console.log(z); // { a: 3, b: 4 }

// Spread properties
let n = { x, y, ...z };
console.log(n); // { x: 1, y: 2, a: 3, b: 4 }
console.log(obj)
```

Sadly it is not supported yet:

```
npm install --save-dev babel-plugin-transform-object-rest-spread
```

Rest

We can allow unlimited params to function by using the rest operator.

```
function demo(part1, ...part2) {
    return {part1, part2}
}

console.log(demo(1,2,3,4,5,6))
```

Let

`Let` is the new `var`. As it has "sane" bindings.

```
{
   var globalVar = "from demo1"
}

{
   let globalLet = "from demo2";
}

console.log(globalVar)
console.log(globalLet)
```

However, it does not assign anything to `window`:

```
let me = "go";  // globally scoped
var i = "able"; // globally scoped

console.log(window.me);
console.log(window.i);
```

It is not possible to redeclare a variable using `let`:

```
let me = "foo";
let me = "bar";
console.log(me);
```

```
var me = "foo";
var me = "bar";
console.log(me)
```

Const

`Const` is for read-only variables.

```
const a = "b"
a = "a"
```

It should be noted that `const` objects can still be mutated.

```
const a = { a: "a" }
a.a = "b"
console.log(a)
```

For..of

New type of iterator, an alternative to `for..in`. It returns the values instead of the `keys`.

```
let list = [4, 5, 6];

console.log(list)

for (let i in list) {
   console.log(i);
}
```

```
let list = [4, 5, 6];

console.log(list)


for (let i of list) {
   console.log(i);
}
```

Promises

The bread and butter for async programing.

```
var p1 = new Promise((resolve, reject) => {
```

```
  setTimeout(() => resolve("1"), 101)
})
var p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("2"), 100)
})

Promise.race([p1, p2]).then((res) => {
   console.log(res)
})

Promise.all([p1, p2]).then((res) => {
   console.log(res)
})
```