LUND UNIVERSITY

JavaScript

EDAF90 Web Programming

Per Andersson

JavaScript

- “the world’s most misunderstood programming language”
- working name at Netscape 1995: *LiveScript*
- syntax and name in common with Java
- object function oriented language
- interpreted
- dynamically typed
- run in any web browser and node.js

Interpreted

- no compilation → no compilation errors → you need to test more
- JS language design leads to:
  - many silent errors
  - weird and unexpected behaviour in some contexts
  - for example a miss spelled property name
- the programmer have more responsibility
- programmers needs extended language knowledge
- use jslint to check your code
- write test cases to catch compilation errors
- work with small increments

Types

Six data types that are primitives (immutable):

- `undefined`
- `boolean`
- `number`
- `bigint` - literal syntax: `42n`
- `string`
- `symbol` - unique and immutable

Structural type:

- `object`

Structural root:

- `null`

Types

The `typeof` operator returns a string indicating the type of the operand or `"function"`.

```
typeof null === "object"
typeof function(){} === "function"
```

Wrapper objects:

- `Boolean`, `Number`, `Bigint`, `String`, `Symbol`
- `Object`, `Function`

```
typeof "EDAF90"=== "string"
typeof new String("EDAF90")=== "object"
```

Dynamically Typed Language

JavaScript is dynamically typed.

- A declaration introduce a name.
- An assignment associate the name with a new `<value, type>` tuple.
- Type conversion only when values are used, never when assigned.
- This includes objects. You can add and remove properties.

valid JavaScript

```
let a = 'Per';
a = 0;
a = null;
a = undefined;
```

typeof

```
typeof 'Per' === 'string';
typeof 0 === 'number';
typeof null === 'object';
typeof undefined = 'undefined';
```

`typeof` returns a string. 8 possible values, all types except **`null`**.

Type coercion

- JavaScript will automatically convert values when needed.
- The type conversion algorithm have some non intuitive consequences.
- There is a strong preference to convert to `string`.
- This is the root of some of the *bad parts* of JavaScript.

automatic type conversion

```
3 + '42'; // '342'
null + 'Per'; // 'nullPer'
3 == '3' // true
```

Type Conversion

Enforce type conversion with expressions.

type converting expressions

```
typeof (+'42') // 'number'
typeof (!!null) // 'boolean'
```

Use type converting functions: `Number()`, `String()`, and `Boolean()`.

type converting expressions

```
typeof Number('42') // 'number'
typeof new Number('42') // 'object'
typeof Number('Per') // 'number (NaN)'
typeof Boolean('false') // 'boolean'
typeof String(42) // 'string'
```

Parameter Types

- variable and parameter declarations are untyped
- names get a `<value, type>` when assigned
- can not enforce argument types
- hard to write functions that can handle any value
- `typeof` can handle som cases

Strings

String literals and templates

- `'single quotation mark'`
- `"double quotation mark"`
- ```
  `string templates
  can span multiple lines
  and contain embedded expressions: 1+2=${1+2}`
  ```

Operations

- `'Per'+ ' ' + 'Andersson'`
- `'Per'.lenght`
- `'Per'.toUpperCase()` - return a new `string`
- `'Per'[0]` - read only

Note

strings are immutable
  ```

Truthy/Falsy

Falsy:

- `false`
- `0`
- `0n`
- `""`,`''`,`` ` ``
- `null`
- `undefined`
- `NaN`

no need for

```
if (name === null || name.length === 0){
  name = 'anonymous';
}
```

Short Circuit

Logic operations return the value of one operand.
Nullish coalescing operator (??), right hand side iff LHS is **`null`** or **`undefined`**

some expressions	evaluates to
`a = 'Per' || 'default value';`	`a = 'Per';`
`b = '' || 'default value';`	`b = 'default value';`
`c = 'Per' || null;`	`c = 'Per';`
`d = NaN || undefined;`	`d = undefined;`
`e = 'Per' && 'Andersson';`	`e = 'Andersson';`
`f = undefined && 'Andersson';`	`f = undefined;`
`g = 'Per' && NaN;`	`g = NaN;`
`h = ref && ref.value;`	`h = ref ? ref.value : ref;`
`i = '' ?? 'default value';`	`i = '';`

Optional Chaining operator

- `object?.property`
- access a property or calls a function
- short-circuit and return **`undefined`** if:
  - `object` is **`null`** or **`undefined`**, or
  - `property` is not a property of `object`

throws no exceptions

```
function myFunction(obj) {
  console.log( obj.?prop );
  console.log( obj.?[1]);
  console.log( obj.func.?());
  obj.func = 3;
  console.log( obj.func.?());
  // Uncaught TypeError: obj.func is not a function
  obj?.a?.b?.[0]?.()?.c;
}
```

Equality and sameness

There are four equality algorithms in ES2015:

- Abstract/Loose Equality: `==`, `!=`
  - triggers type conversion leading to unexpected behaviour
- Strict Equality: `===`, `!==`, compare type and value
  - conform to IEEE 754 (so **`NaN`** `!=` **`NaN`**, and `-0 == +0`)
- `Object.is()`: Same Value, as strict equality except for NaN, -0, and `+0`

evaluates to true

```
1 == '1';
[1, 2] == '1,2';
[1, 2] != '1, 2';
'true' != true;
```

evaluates to true

```
-0 === +0;
0 == false
1 !== '1';
null == undefined;
null !== undefined;
```

Check out the JavaScript Equality Table

Functions

- functions are values
  - `Function` objects
  - normal object, with the addition of being callable
  - `object` in the typesystem
  - `typeof` returns function
  - higher order functions
    - » a function can be passed as argument
    - » a function can return another function
- call by value - like in Java (objects are references)
- default return value:
  - **`undefined`**
  - **`this`** in constructors
- three ways to create functions:
  - function declaration
  - function expression
  - `Function` constructor (not recommended for security reasons)

Function Declaration

- is a statement
- no need to use semicolon after a function declaration
- creates
  - a `Function` object
  - a variable with the function name

function declaration

```
function calcRectArea(width, height) {
  return width * height;
}

console.log(calcRectArea(5, 6));
```

Function Expression

- is an expression
- creates a `Function` object
- the function name is optional, omitting it creates an anonymous function
- the name is stored in the `Function` object, can only be used inside the function
- you must store the value, pas it as argument, to use the function

function expression

```
const array1 = [1, 4, 9, 16];
const map1 = array1.map(function(x) { return x * 2});
```

Default Parameters

- function parameters default to **`undefined`**
- parameters can have other default values (ES2015)
- parameter values are available to later default parameters
- default parameters are evaluated at call time

rest parameters

```
function multiply(a, b = 1) {
  return a * b;
}

function greet(name,
               greeting,
               message = greeting + ' ' + name) {
   return [name, greeting, message];
}
```

Rest Parameters

- must be the last named parameter
- all remaining arguments are wrapped into an `Array`

rest parameters

```
function sloppySum(first, ...theRest) {
  return theRest.reduce((previous, current) => {
    return previous + current;
  });
}
```

Arguments Object

- `arguments` is an `Array`-like object
- contains all arguments
- doesn't have `Array`'s built-in methods like `forEach()` and `map()`
- properties
  - `arguments.callee`
  - `arguments.caller`
  - `arguments.length`
  - `arguments[@@iterator]`

arguments

```
function foo(a, b, c) {
  console.log(arguments[1]);
}
foo(1, 2, 3);
```

Arrow Function

- convenient syntax
- is an expression
- creates an anonymous function, can not use recursion
- without own bindings to the **`this`**, `arguments`, **`super`**, or **`new`**`.target`
- these values are retained from enclosing lexical context
- ill suited as methods, and they cannot be used as constructors

syntax

```
([param[, param]]) => {
  statements
}

param => expression
```

Arrow Function, examples

example of arrow functions

```
let sqr = x => x*x;

let calcRectArea = (width, height) => width * height;

let pi = _ => Math.PI;

let myLogger = (msg) => {
  console.log(new Date() + ': ' + msg);
};

let foo = (width, height) => { width * height };
```

Higher order functions

JavaScript has all features of a function oriented language.

function oriented programming

```
let list = [1, 2, 3, 4, 5];
let a = list.filter((x) => x % 2 === 0);
let b = a.map(x => x + 2);
b.forEach(console.log);
let c = b.reduce((sum, x) => sum + x, 0);
```

chaining

```
let sum = [1, 2, 3, 4, 5];
sum.filter((x) => x % 2 === 0)
.map(x => x + 2)
.reduce((sum, x) => sum + x, 0);
```

Closure

- lexical scope
- a closure gives you access to an outer function's scope from an inner function
- closures are created every time a function is created, at function creation time

closure

```
let name = 'Per Andersson';
let foo = function() {
  name = 'anonymous';
}
console.log(name);
foo();
console.log(name);
```

Closure

- remember, functions are values.
- inner functions can be returned from a function.

closure

```
function foo() {
  let cnt = 0;
  return (_ => cnt++);
}

let idGenerator = foo();

console.log(idGenerator());
some_async_function(idGenerator);
another_async_function(idGenerator);
```

Variables and Global Name Space

Variables

- reading an undeclared name throws a `ReferenceError`
- assigning to an undeclared name creates it as a global variable

Global name space

- shared by all JavaScript files
- high risk of name conflict
- do not use

Scope Rules

Two different kind of scopes:

- function scope
  - **`var`**
- block scope (ES2015)
  - **`let`**
  - **`const`**
  - works like scope in Java

Function Scope

- declare variables using **`var`**
- the scope is the current execution context
  - the function
  - the global context
- redeclaration of names are allowed
- considered bad practice today

Function Scope, example 1

```
function foo() {
  y = 1; // Throws a ReferenceError in strict mode.
  var x = 3;
  if (true) {
    var x = 2;
  }
  return x;
}

try {
  console.log(y);
} catch (e) { console.log('Oops'); }
foo();
console.log(y); // 1
```

Function Scope, example 2

```
function foo() {
  for (var i=0; i<2; i++) {
    for (var i=0; i<2; i++) {
      console.log(i);
    }
  }
  return x;
}
foo() // 0, 1
```

Function Scope, example 3

```
var a = [];
for (var i=0; i<3; i++) {
  a[i] = function() { console.log(i); };
}
a[0]();
a[1]();
a[2]();
```

Hoisting

- all declared variables are created before any code is executed
- variable and function declarations are lifted to top of function
- initialisation remain in place
- function declaration: name and body are hoisted
- function expression: is assignment, only the name is hoisted

Hoisting

```
function foo() {
  console.log(x); // undefined
  var x = 3;
  console.log(x); // 3
}
```

```
hoistedFun = _ => 'function declared by assignment';

function hoistedFun() {
  return 'function declaration';
}

console.log(hoistedFun());
```

JavaScript modules

Introduced in ES6

my-module.js

```
function cube(x) {
  return x * x * x;
}
const foo = Math.PI + Math.SQRT2;
const text = "private in module";
export { cube, foo };
```

some-code.js

```
import { cube, foo } from './my-module.js';

console.log(cube(3));
console.log(foo);
```

CommonJS modules

Common in environments not supporting JavaScript Modules, for example node.

my-module.js

```
function cube(x) {
  return x * x * x;
}
const foo = Math.PI + Math.SQRT2;
```

some-code.js

```
const stuff = require('./my-module.js');

console.log(stuff.cube(3));
console.log(stuff.foo);
```

Objects

- an object is a dictionary: string → any value
- attributes and methods are also called properties
- properties can have any name, including reserved words and operations
- access properties using:
  - dot notation: `myObj.prop`
  - array index notation: `myObj['prop']`
- `typeof objRef === 'object'`
- add properties by writing to them `myObj.newProp = 'adding stuff';`
- remove properties by: `delete myObj.newProp`

Create Objects

- object literals {*prop* : *value*}
- `new ConstructorFunction(args);`

Object Literals

- superset of JSON
- comma separated list of properties inside `{ }`
- a property is defined by:
  - `property-name : value`
  - `method-name(parameters){ statements }`
- name in plain text, quotes if needed
- value is any JavaScript expression
- `{a:a}` is the same as `{a}`

Object Literals

object literal

```
const familyName = 'Andersson';
const myObject = {
  givenName: 'Per',
  familyName,
  selector: 'givenName',
  getValue: function () {
    return this[this.selector];
  },
  setValue(value) {
    this[this.selector] = value;
  },
  '+': 'plus'
 }
```

Object Literals

- object literals are cheap
- use them frequently
- they bring structure and readability to programs

object literals

```
let myPoints = [{x: 0, y: 0}, {x:10, y:15}];

function bar(x, y, options) {
 console.log('b = '+ options?.b);
}

function foo(x, y, a, b, c, d) {
 console.log('d = '+ d);
}
```

Named Parameters

Remember, `foo` and `bar` prints option `b`.

What is printed?

```
foo(0, 0, 0, 0, 1, undefined, 1);
bar(0, 0, {a: 0, b: 0, c:1, e: 1});
```

Did you notice thet foo have one extra argument compared to the parameter list? Too few, or extra parameters are silent in JavaScript.

Access to Undefined Names

Variables and properties have distinct name spaces.

Name Scope: Variables and Parameters

- read: throws ReferenceError
- write: creates a variable in the global scope

Objects: Properties

- read: evaluates to **`undefined`**
- write: adds the property to the object

Constructor Functions

- same purpose as classes in Java
  - initialises objects when used with **`new`**
- are function, intended use differs
  - **`function`** `ConstructorFunction(args){ ... }`
  - by convention: use Pascal Case
- arrow functions can not be used as constructor functions
- **`new`** `ConstructorFunction(args)` will:
  1. creates an empty object
  2. set up inheritance
  3. calls `ConstructorFunction(args)` with the new object as **`this`**
  4. the constructor function adds properties to **`this`** and assign them values
  5. the result of **`new`** is the object returned by the *constructor function*
     remember: the default return value of functions called by **`new`** is **`this`**

Constructor Function Example

class definition

```
function Point(x, y) {
  this.x = x || 0;
  this.y = y || 0;
  this.getX = function() {
    return this.x;
  }
}
```

create instances

```
let point1 = new Point(3, 6);
let point2 = new Point();
let point2 = new Point(5);
let point3 =
  new Point(undefined, 5);
```

this

- properties are not in the scope of methods, must use **`this`**
- **`this`** is defined in all functions
- its value depends on how the function is called:
  - function call: `foo()` - the global object
  - dot notation: `obj.foo()` - the object left of the dot
  - explicit: `Function.prototype.call()`
  - explicit: `Function.prototype.bind()` - creates a new function with a predefined value for **`this`**
  - as an DOM event handler - the element the event fired from (not all cases for all browsers)
  - as an inline DOM event handler - the DOM element on which the listener is placed
- arrow functions: **`this`** from the enclosing scope is used

self

When a function is a “object method”

- you do not know if **`this`** refers to the right object
- use closure to fix this
- or use arrow functions

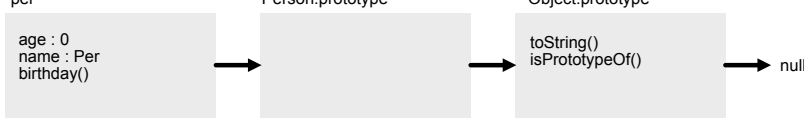
```
function Person() {
  const self = this; // Some choose `that` instead of `self`.
                     // Choose one and be consistent.
  self.age = 0;
  this.birthday = function() { self.age++; };
  this.birthday2 = _ => this.age++;
}
const per = new Person();
setInterval(per.birthday, 1000);
```

Prototype Based Inheritance

- all object inherit from another object or **`null`**
- default is `Object`
- objects forms a *prototype chain*
- property name lookup follows the prototype chain
- the chain ends with **`null`**
- you can access the prototype chain (but don't):
  - `Object.getPrototypeOf(object)`
  - `Object.setPrototypeOf(object, chain)`
- the prototype chain is initialised by **`new`** when the object is created
- the `prototype` property of the constructor function is used as the first link

Prototype Chain

```
function Person(name) {
  this.age = 0;
  this.name = name;
  this.birthday = () => this.age++;
}
const per = new Person('Per');
```

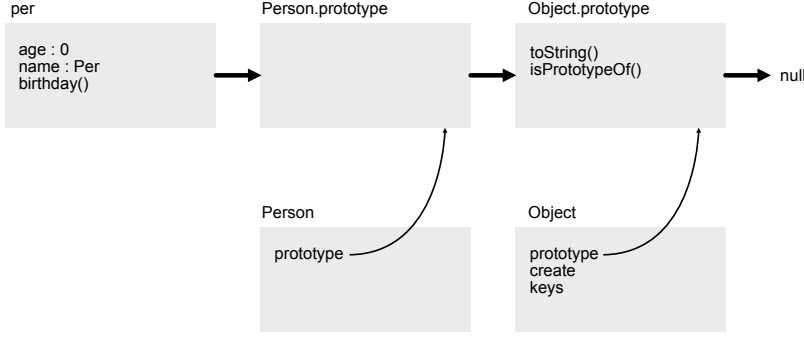


Function Object

Every functions is stored in a function object:

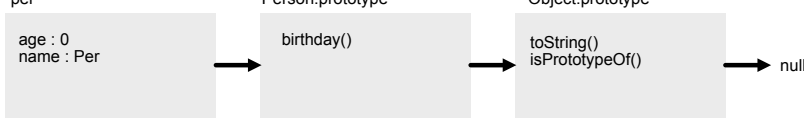
- Function object:
  - is callable, left hand side of ()
  - is an object, a `string` → `value` dictionary
  - constructor functions must have the property `prototype`
    - » all functions except: methods, arrow functions, or async functions
  - store static properties in the object/dictionary
- Prototype object:
  - added to the *prototype chain* by **`new`**
  - store inherited properties

Prototype Chain



prototype

```
function Person(name) {
  this.age = 0;
  this.name = name;
}
Person.prototype.birtday = function() { this.age++; };
const per = new Person();
```



Set up Prototype Chain

Setting up the prototype chain:

- **`new`** do the work for you
  - all constructor functions have the `prototype` property
  - **`new`**:
    - » creates an empty object
    - » **and** set its parent in the prototype chain to the `prototype` in the constructor function
  - all properties in the `prototype` of the constructor function are now in the prototype chain of the new object
- you can do it manually: `Object.create()`

Property Name Lookup

Property read:

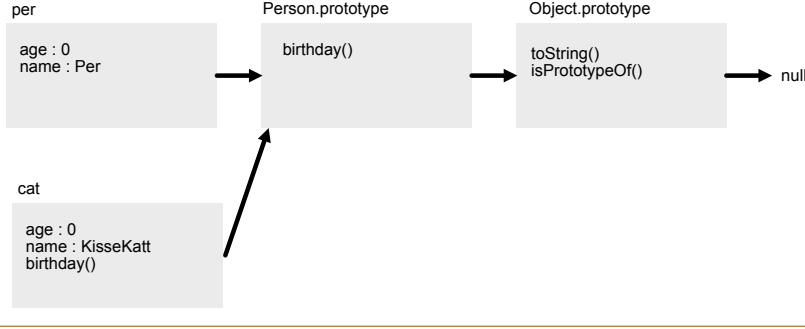
- follows the prototype chain
- return the first value found
- return **`undefined`** if the end of the prototype chain is reached

Property write:

- do not follows the prototype chain
- writes to the referenced object (left hand side of the dot)
- update if the name existed
- adds the property if the name did not exist

prototype

```
let cat = new Person("KisseKatt");
cat.birtday = function() { this.age += 7; }
```



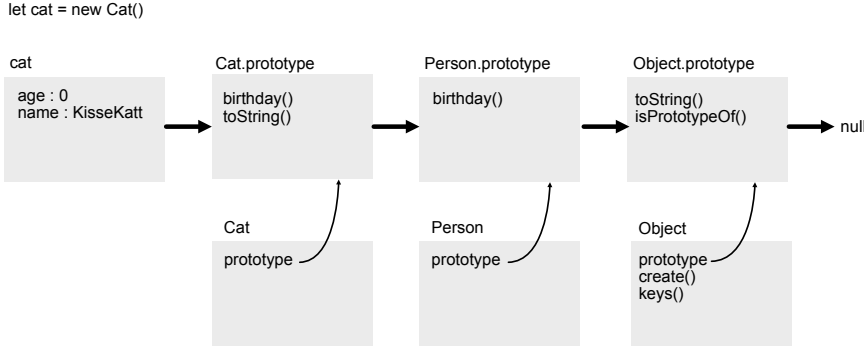
Inheritance

- `Object.create()` creates an object with a given prototype chain
- store it as the `prototype` property in the constructor function
- explicit call the constructor of the superclass

Cat extends Person

```
function Cat(name) {
  return Person.call(this, name);
}
Cat.prototype = Object.create(Person.prototype);
Cat.prototype.birthday = function() { this.age += 7; }
Cat.prototype.toString = function() {
  return 'I am a cat of age ' + this.age';
}
```

prototype


let cat = new Cat()
cat
age : 0
name : KisseKatt
Cat.prototype
birthday()
toString()
Person.prototype
birthday()
Object.prototype
toString()
isPrototypeOf()
null
Cat
prototype
Person
prototype
Object
prototype
create()
keys()

Class

a "Java class" corresponds to two objects in JavaScript

- a constructor function:
  - its name is part of the variable name space
  - place static stuff here
- a prototype object
  - the object to add to the prototype chain
  - methods are placed here

`Class` was introduced in ECMAScript 2015

- syntactical sugar, set up the prototype chin as outlined above
- access is **`public`** or **`#private`**
- **`static`** will add the property to the constructor function object
- methods are place int he prototype of the constructor function
- attributes are place in the created object

Class Example

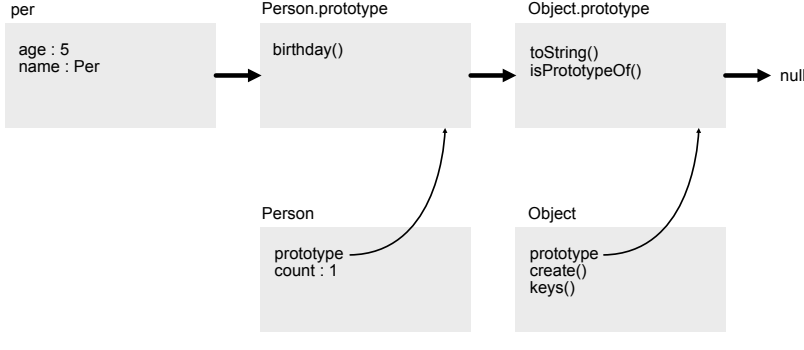
```
class Person {
  static count = 0;
  static #defaultName = "Anonymous";

  constructor(name, age) {
    this.name = name || Person.#defaultName;
    this.age = age || 0;
    Person.count = Person.count + 1;
  }

  birthday() {
    this.age++;
  }
}
```

prototype

```
const per = new Person("Per", 5);
```



Class Extends

The constructor:

- in a derived class must call **`super()`** before you can access **`this`**
- in a base class may not call **`super()`**

```
class Cat extends Person {
  constructor(age) {
    super(age);
  }
  birthday() {
    this.age += 7;
  }
  toString() {
    return 'I am a cat of age ' + this.age';
  }
}
```

Standard Classes

In JavaScript there are many standard classes. Some important:

- `Object` - default base class for all objects
- `Function` **`extends`** `Object` - base class for all functions
- `Array` - base class for array litterals

Property Descriptors

Distinction between

- *own properties*
- *inherited properties*

Object properties have descriptors (metadata)

- *value*
- *writable*
- *configurable*
- *enumerable*

Iteration

Iterating over object property names and values

- **`for ... in`** — all enumerable string properties (all keys, include inherited)
- `Object.keys()` — own enumerable
- `Object.values()` — own enumerable
- `Object.entries()` — own enumerable
- `Object.getOwnPropertyNames()` — own
- `...`, spread — own enumerable

More to learn

The JavaScript syntax only give you access to a subset of the language. . .

```
Object.defineProperty(obj, "prop", {
   value: "test",
   writable: false
});
```

This is however out of scope for this course.

Arrays

- variable size and type
- `myArray = [1, 'two', new Number(3)]`
- index must be number
- size is managed by JavaScript
- reading an undefined index returns **`undefined`**
- `myArray['per'] = 3` - adds a property to the array object
- `push(), pop(), slice()`
- `map(), reduce(), forEach()`
- **`for ... of`** - iterates over elements
- **`for ... in`** - iterates over enumerable object properties

Destructuring assignment

- unpack arrays and objects
- use:
  - left hand side of assignment
  - function parameters
- can have default values
- can be nested
- the tail of an array can be stored in a variable: `...remainingValues`

```
const foo = ['red', 'green'];
const [one, two, three = 'blue'] = foo;
console.log(one); // "red"
console.log(three); // "blue"
const [one, ...rest] = foo;
```

Destructuring assignment

```
const user = {
  id: 42,
  displayName: 'jdoe',
  fullName: {
    firstName: 'John',
    lastName: 'Doe'
  }
};

const {id:selectedId} = user;

function whoIs({displayName, fullName: {firstName: name}}) {
  return `${displayName} is ${name}`;
}
```

Spread Syntax

The spread syntax `...` can be used on
An iIterable, such as an array or string, can be expanded instead of:

- zero or more arguments (for function calls)
- elements (for array literals)

An object expression to be expanded instead of

- zero or more key-value pairs (for object literals)

```
function sum(x, y, z) {
  return x + y + z;
}

const numbers = [1, 2, 3];

const total = sum(...numbers);
```

Spread Syntax

```
const parts = ['shoulders', 'knees'];
const lyrics = ['head', ...parts, 'and', 'toes'];

const obj1 = { foo: 'bar', x: 42 };
const obj2 = { foo: 'baz', y: 13 };

const clonedObj = { ...obj1 };

const augmentedObj = { ...obj1, name: 'Per' };

const mergedObj = { ...obj1, ...obj2 };
```

Automatic Semicolon Insertion

Some JavaScript statements' syntax definitions require semicolons (;) at the end. If missing, a semicolon is added at the end of a line.

returns undefined

```
function() { return
1; }
```

Common to use minify to minimise script download size. All white spaces are removed.

works

```
let myVar = 9
if (myVar === 9) {
}
```

syntax error after minify

```
var myVar = 9 if (myVar === 9) {}
```

Use jslint to detect these problems.

Strict mode

Converting mistakes into errors.

Whole-script strict mode syntax

```
'use strict';
var v = "Hi! I'm a strict mode script!";
```

Function-level strict mode syntax

```
function strict() {
  'use strict';
  function nested() { return 'And so am I!'; }
  return "Hi! I'm a strict mode function! " + nested();
}
function notStrict() { return "I'm not strict."; }
```