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 $\longrightarrow$ no compilation errors
- design decision: try to execute everything
  - many silent errors
  - weird and unexpected behaviour
  - 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

# Types

Six data types that are primitives (immutable):

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

Structural types:

- `object`
- `function`

Structural root:

- `null`

# Types

The `typeof` operator returns a string indicating the type of the unevaluated operand.

```
typeof null === "object"
```

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 variable have a value and a type. Both can change.
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`**.

# Automatic Type Conversion

JavaScript will automatically convert values when needed.
There is a strong preference to convert to `string`.
The type conversion algorithm have some non intuitive side effect. 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
- they get a 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()`
- `'Per'[0]`

**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.

| 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 = null && 0;` | `h = null;` |
| `i = ref && ref.value;` | `i = ref ? ref.value : ref;` |

# Optional Chaining operator

- `a?.b`
- evaluate to **`undefined`** if `a` is not an object

# Equality and sameness

There are four equality algorithms in ES2015:

- Abstract Equality Comparison: `==`, `!=`
- Strict Equality Comparison: `===`, `!==`
- SameValueZero: used by `%TypedArray%` and `ArrayBuffer`
- SameValue: used in all other places (`Object.is`)

**evaluates to true**

```
NaN != NaN;
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
  - assign functions to variables
  - use functions as arguments to other functions
- 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)

# 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 to use the function

function expression

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

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

# Default Parameters

- function parameters default to **`undefined`**
- parameters can have other default values (ES2015)
- parameters 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
- 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 };
```

# Function Oriented Programming

JavaScript have all features of a function oriented language.

**function oriented programming**

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

**chaining**

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

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

let idGenerator = foo();

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

# Name Spaces

- one name space for variables and parameters
- each object have their own name space for properties
- you must use **`this`** when accessing object properties

Global name space

- shared by all functions and modules
- high risk of name conflict
- do not use

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

# 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

# Name Hoisting

In function scope

- all declared variables are created before any code is executed
- name declarations are lifted to top of function
- initialisation remain in place
- function expressions are not hoisted

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

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

# Immediately Invoked Function Expressions

- creates a new scope level, do not pollute surrounding
- if inside a loop
  - one scope for each loop iteration
  - local variables are instantiated in each loop iteration
  - can be used to clone and freeze outer state (loop variable)

```
// outer scope
var x = 2;
(function() {
  // inner hidden scope
  var x = 3;
})();

console.log(x);
// more outer scope
```

# IIFE, example 2

```
var domElements = [...];
for (var i=0; i<10; i++) {
  (function () {
    var id = i;
    domElement[i].onClick(
      _ => { alert('clicked ' + id); }
    );
  })();
}
```

# JavaScript modules

Introduced in ES6

my-module.js

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

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 map: string → any value
- objects have properties - a (string, value) par in the map (attributes, methods)
- 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`

# Objects, example 1

basic object

```
let myObject = {
  givenName: 'Per',
  familyName: 'Andersson',
  selector: 'givenName',
  getValue: function () {
    return this[this.selector];
  }
}

console.log(myObject.getValue());
myObject.selector = 'familyName';
console.log(myObject.getValue());
```

# Object Literals

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

**object literal**

```
let myObject = {
  five: 2 + 3,
  '5': 'five',
  '+' : 'plus',
  'null': 'not a good idea name'
 }
```

# Object Literals

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

**object literals**

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

function foo(a, b, c, d, e, f) {
 console.log('d = '+ d);
}
function bar(arg) {
 console.log('d = '+ arg.d);
}
```

# Named Parameters

Remember, `foo` and `bar` prints parameter `d`.

**What is printed?**

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

Did you notice thet foo have one extra parameter compared to the arguments list?
Too few, or extra parameters do not raise errors in JavaScript.

# Constructor Functions

- The **`new`** operator lets developers create an instance of an object type that has a constructor function.
- **`new`** accepts any functions
- *constructor function* are like classes in Java: forms and initialises objects when used with **`new`**
- by convention: use leading capital letter in *constructor function* names (similar to class names in Java)
- **`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 result of **`new`** is the value returned by the *constructor function*
     remember: the default return value of functions called by **`new`** is **`this`**
- only call constructor functions using **`new`**:
- combine with closure and you have the power

# 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(null, 5);
```

# this

- **`this`** is defined in all functions
- arrow functions: **`this`** from the enclosing scope is used
- 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

# self

When a function is a “object method”

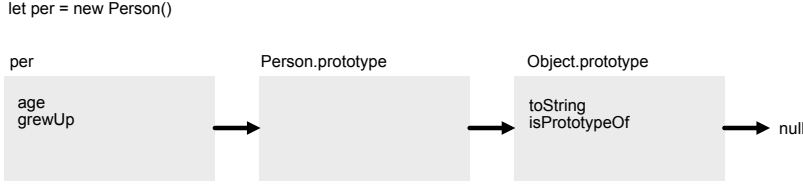
- you do not know if **`this`** refers to the right object
- use closure to fix this
- or use arrow functions

```
function Person() {
  let self = this; // Some choose `that` instead of `self`.
                   // Choose one and be consistent.
  self.age = 0;
  this.growUp = function() { self.age++; };
  this.growUp2 = _ => this.age++;
}
let per = new Person();
setInterval(per.growUp, 1000);
```

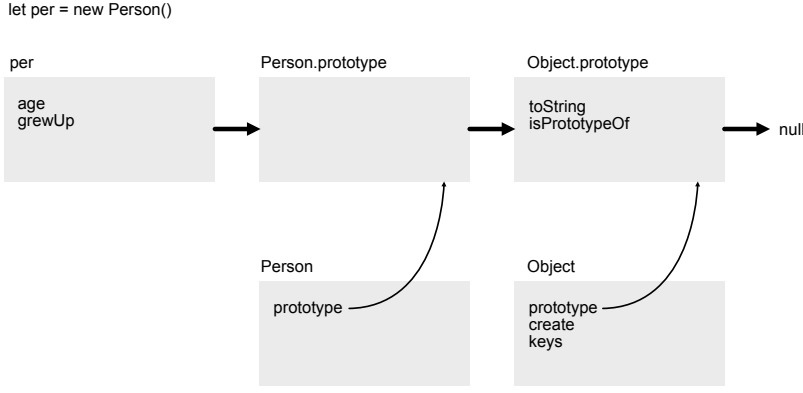
# Prototype Based Inheritance

- all object inherit from another object or **`null`**
- default is `Object`
- objects forms a *prototype chain*
- property name lookup follows the prototype chain
- you can access the prototype chain (but don't):
  - `Object.getPrototypeOf(object)`
  - `Object.setPrototypeOf(object, chain)`
- the chain ends with **`null`**

# Prototype Chain

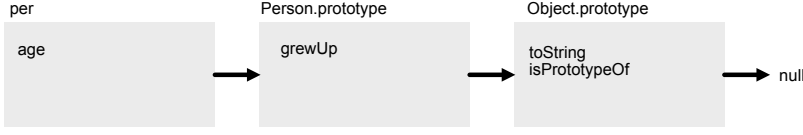


# Prototype Chain



# prototype

```
function Person() {
  this.age = 0;
}
Person.prototype.growUp = function() { this.age++; };
let per = new Person();
```



# Set up Prototype Chain

Setting up the prototype chain:

- **`new`** do the work for you
  - all `Function` objects have a property `prototype`
  - remember, constructor functions are instance of `Function`
  - **`new`**:
    - » creates an empty object
    - » **and** set its parent in the prototype chain to the `prototype` in the constructor function
  - all names 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:

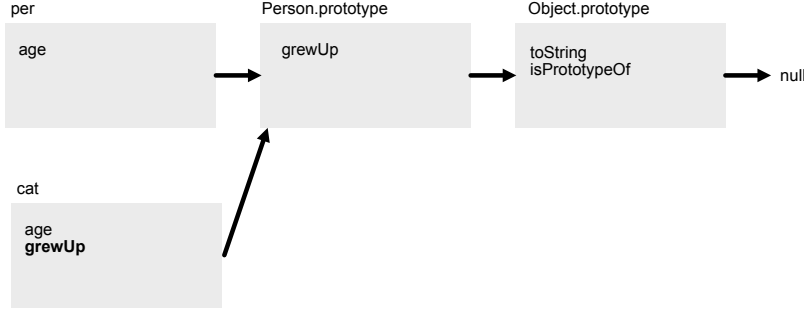
- follows the prototype chain
- return the first value found
- return **`undefined`** if the end of the prototype chain is reached

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();
cat.grewUp = function() { this.age +=7; }
```



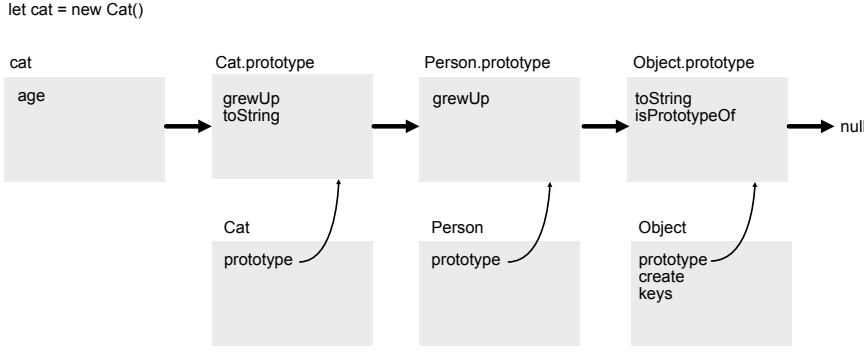
# Inheritance

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

**Cat extends Person**

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

# prototype


let cat = new Cat()
cat
age
Cat.prototype
grewUp
toString
Person.prototype
grewUp
Object.prototype
toString
isPrototypeOf
null
Cat
prototype
Person
prototype
Object
prototype
create
keys

# Class

a "Java class" corrsponds 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
  - place any stuff to be inherited by the instances here

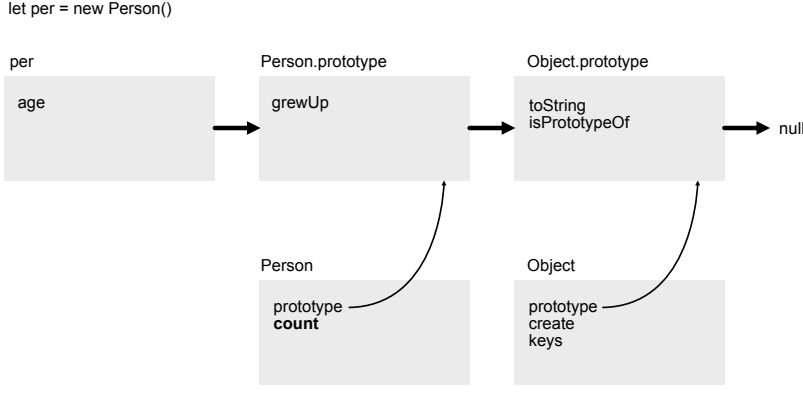
`Class` was introduced in ECMAScript 2015

- syntactical sugar, set up the prototype chin as outlined above
- **`static`** will add the property to the constructor function object
- no **`protected`** or **`private`**

# Class Example

```
class Person {
  constructor(age) {
    this.age = age || 0;
    Person.count = Person.count + 1;
  }
  static count = 0;
}
```

# prototype


let per = new Person()
per
age
Person.prototype
grewUp
Object.prototype
toString
isPrototypeOf
null
Person
prototype
count
Object
prototype
create
keys

# 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);
  }
  toString() {
    return 'I am a cat of age ' + this.age';
  }
}
```

# Access to Undefined Names

Variables and properties have distinct name spaces.

Variables:

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

Properties:

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

# 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 Properties

Object properties have metadata (properties)

- *Own property* - true for all names in **`this`**, false for inherited names
- *Enumerable*
- *Writable*

Iteration

- `Object.getOwnPropertyNames()` - a given object's own property names, but not `Symbols`
- `Object.keys()` - a given object's own enumerable property names
- **`for`** `(`**`const`** `property` **`in`** `object)` - object's own enumerable properties

# 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
- `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); // "default three"
const [one, ...rest] = foo;
```

# Destructuring assignment

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

const userId = ({id}) => id;

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 expression 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];

console.log(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

If needed, a semicolon is added at the end of a line.

the function will return 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."; }
```