Appendix A: Reserved Keywords

Certain words - so-called *keywords* - are treated specially in JavaScript. There's a plethora of different kinds of keywords, and they have changed in different versions of the language.

Section A.1: Reserved Keywords

JavaScript has a predefined collection of *reserved keywords* which you cannot use as variables, labels, or function names.

ECMAScript 1

Version = 1

A — E	*E — R*	*S — Z*
break	export	super
case	extends	**switch**
catch	**false**	**this**
class	**finally**	**throw**
const	**for**	**true**
continue	**function**	**try**
debugger	if	**typeof**
default	import	**var**
delete	in	**void**
do	**new**	while
else	**null**	with
enum	**return**	

ECMAScript 2

Added **24** additional reserved keywords. (New additions in bold).

Version = 3 Version = E4X

A — F	*F — P*	*P — Z*
abstract	**final**	**public**
boolean	**finally**	**return**
break	**float**	**short**
byte	**for**	**static**
case	**function**	super
catch	**goto**	**switch**
char	if	**synchronized**
class	**implements**	**this**
const	import	**throw**
continue	in	**throws**
debugger	**instanceof**	**transient**
default	**int**	**true**
delete	**interface**	**try**
do	**long**	**typeof**
double	**native**	**var**
else	**new**	**void**
enum	**null**	**volatile**
export	**package**	while
extends	**private**	with

false	protected

ECMAScript 5 / 5.1

There was no change since *ECMAScript 3*.

ECMAScript 5 removed int, byte, char, **goto**, long, final, float, short, double, native, throws, boolean, abstract, volatile, transient, and synchronized; it added **let** and yield.

A — F	*F — P*	*P — Z*
break	**finally**	public
case	**for**	**return**
catch	**function**	**static**
class	if	super
const	implements	**switch**
continue	import	**this**
debugger	in	**throw**
default	**instanceof**	**true**
delete	interface	**try**
do	**let**	**typeof**
else	**new**	**var**
enum	**null**	**void**
export	package	while
extends	private	with
false	protected	**yield**

> implements, **let**, private, public, interface, package, protected, **static**, and yield are **disallowed in strict mode only**.

> eval and arguments are not reserved words but they act like it in **strict mode**.

ECMAScript 6 / ECMAScript 2015

A — E	*E — R*	*S — Z*
break	export	super
case	extends	**switch**
catch	**finally**	**this**
class	**for**	**throw**
const	**function**	**try**
continue	if	**typeof**
debugger	import	**var**
default	in	**void**
delete	**instanceof**	while
do	**new**	with
else	**return**	yield

Future reserved keywords

The following are reserved as future keywords by the ECMAScript specification. They have no special functionality at present, but they might at some future time, so they cannot be used as identifiers.

`enum`

The following are only reserved when they are found in strict mode code:

`implements`	`package`	`public`
`interface`	`private`	\`static'
`let`	`protected`	

Future reserved keywords in older standards

The following are reserved as future keywords by older ECMAScript specifications (ECMAScript 1 till 3).

`abstract`	`float`	`short`
`boolean`	`goto`	`synchronized`
`byte`	`instanceof`	`throws`
`char`	`int`	`transient`
`double`	`long`	`volatile`
`final`	`native`	

Additionally, the literals null, true, and false cannot be used as identifiers in ECMAScript.

From the Mozilla Developer Network.

Section A.2: Identifiers & Identifier Names

With regards to reserved words there is a small distinctions between the *"Identifiers"* used for the likes of variable or function names and the *"Identifier Names"* allowed as properties of composite data types.

For example the following will result in an illegal syntax error:

```
var break = true;
```

> Uncaught SyntaxError: Unexpected token break

However the name is deemed valid as a property of an object (as of ECMAScript 5+):

```
var obj = {
    break: true
};
console.log(obj.break);
```

To quote from this answer:

From the ECMAScript® 5.1 Language Specification:

> Section 7.6
>
> Identifier Names are tokens that are interpreted according to the grammar given in the "Identifiers" section of chapter 5 of the Unicode standard, with some small modifications. An `Identifier` is an `IdentifierName` that is not a `ReservedWord` (see 7.6.1).

> **Syntax**
>
> ```
> Identifier ::
>   IdentifierName but not ReservedWord
> ```

By specification, a `ReservedWord` is:

> Section 7.6.1
>
> A reserved word is an `IdentifierName` that cannot be used as an `Identifier`.
>
> ```
> ReservedWord ::
>   Keyword
>   FutureReservedWord
>   NullLiteral
>   BooleanLiteral
> ```

This includes keywords, future keywords, `null`, and boolean literals. The full list of keywords are in Sections 7.6.1 and literals are in Section 7.8.

The above (Section 7.6) implies that `IdentifierName`s can be `ReservedWord`s, and from the specification for object initializers:

> Section 11.1.5
>
> **Syntax**
>
> ```
> ObjectLiteral :
>   { }
>   { PropertyNameAndValueList }
>   { PropertyNameAndValueList , }
> ```

Where `PropertyName` is, by specification:

> ```
> PropertyName :
>   IdentifierName
>   StringLiteral
>   NumericLiteral
> ```

As you can see, a `PropertyName` may be an `IdentifierName`, thus allowing `ReservedWord`s to be `PropertyName`s. That conclusively tells us that, *by specification*, it is allowed to have `ReservedWord`s such as `class` and `var` as `PropertyName`s unquoted just like string literals or numeric literals.

To read more, see Section 7.6 - Identifier Names and Identifiers.

Note: the syntax highlighter in this example has spotted the reserved word and still highlighted it. While the example is valid JavaScript developers can get caught out by some compiler / transpiler, linter and minifier tools that argue otherwise.