JavaScript for ABAP Programmers

Syntax

Chris Whealy / The RIG

ABAP	JavaScript
Strongly typed	Weakly typed
Syntax similar to COBOL	Syntax derived from Java
Block Scope	Lexical Scope
No equivalent concept	Functions are 1st class citizens
OO using class based inheritance	OO using referential inheritance
Imperative programming	Imperative or Functional programming

Syntax

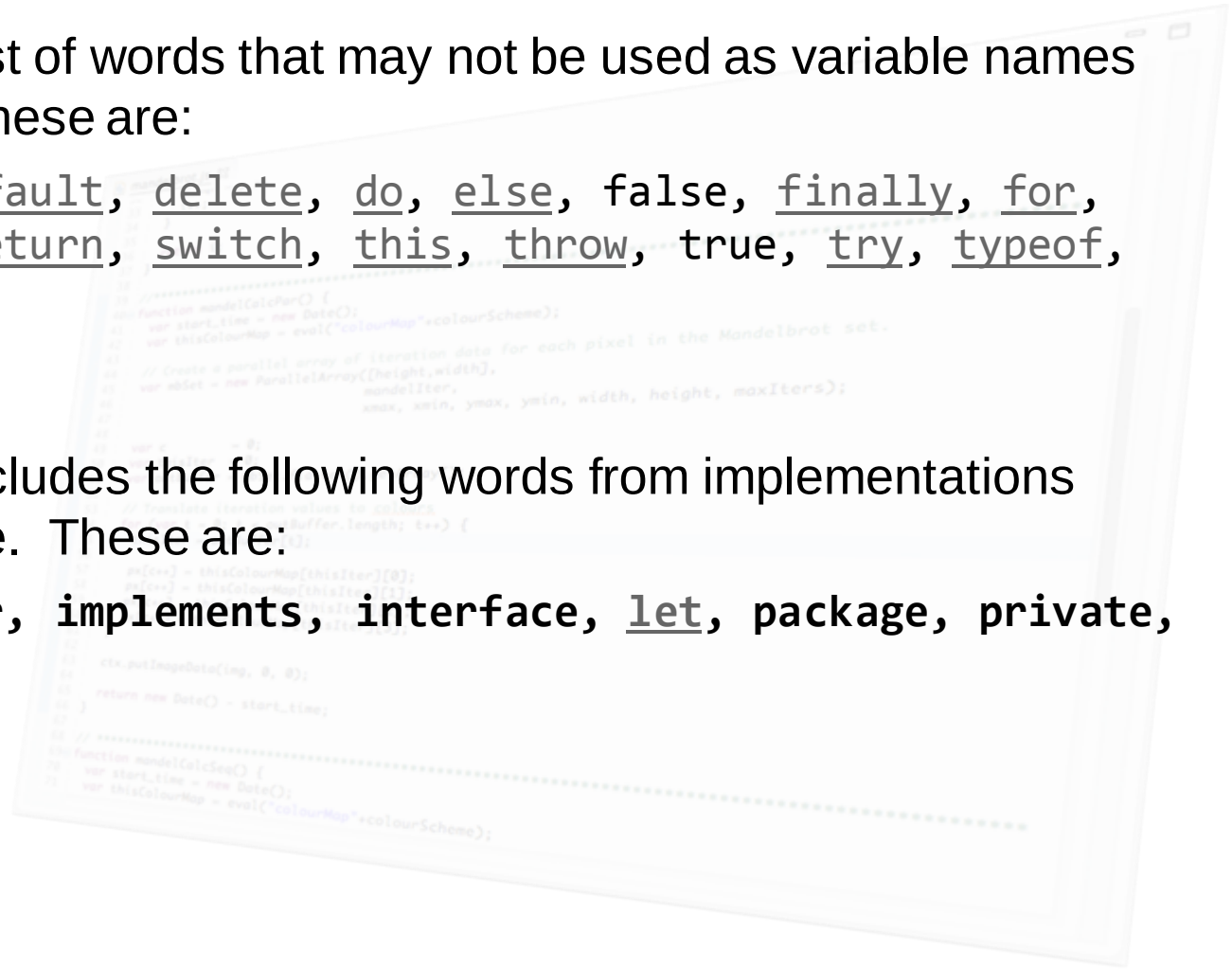
Syntax: Keywords and Reserved words

As with any other language, JavaScript has a list of words that may not be used as variable names because they are keywords of the language. These are:

```
break, case, catch, continue, debugger, default, delete, do, else, false, finally, for,
function, if, in, instanceof, new, null, return, switch, this, throw, true, try, typeof,
var, void, while, with
```

However, the ECMAScript specification also excludes the following words from implementations because they have been reserved for future use. These are:

```
class, enum, export, extends, import, super, implements, interface, let, package, private,
protected, public, static, yield
```

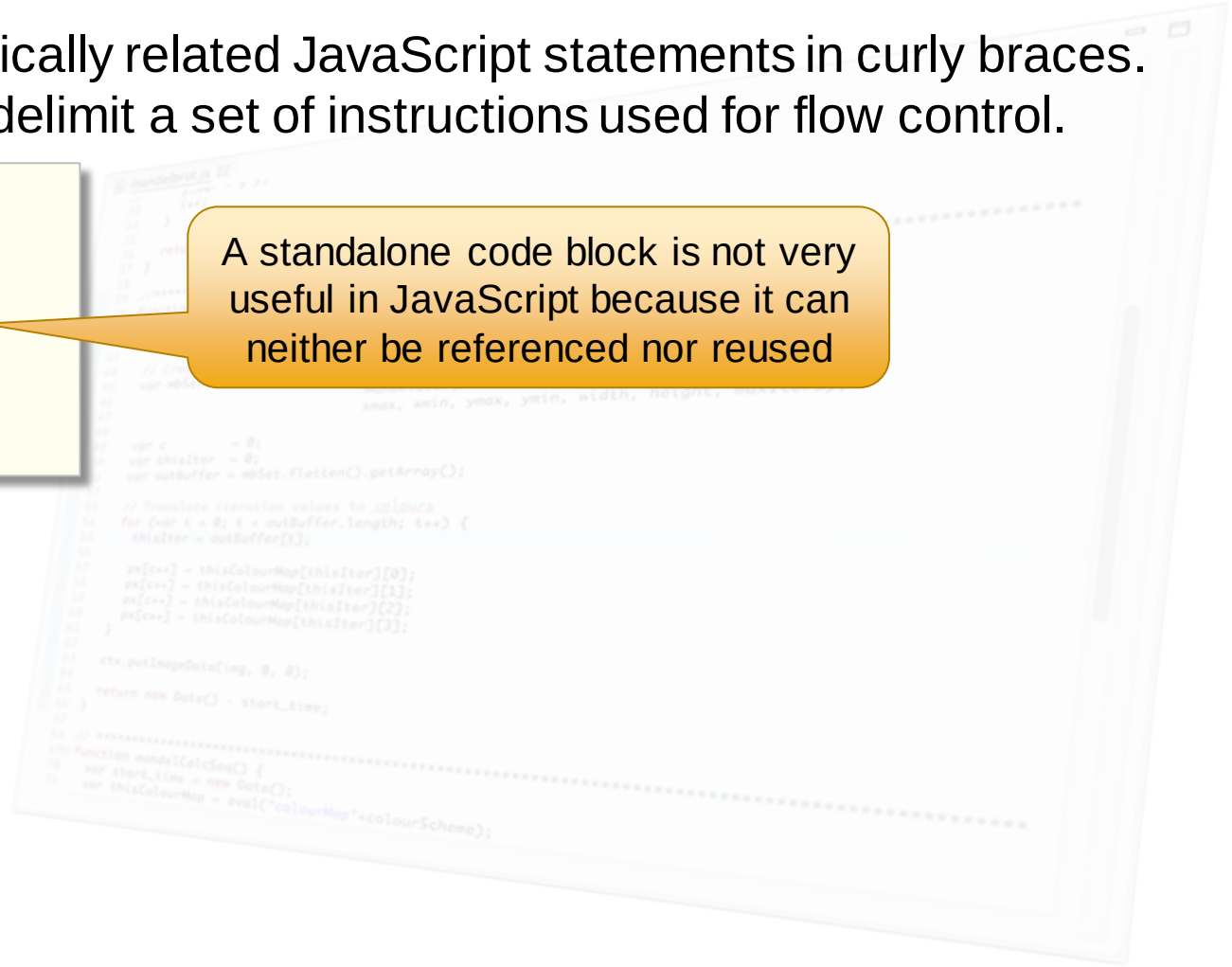


Syntax: Code Blocks

A code block is formed by enclosing a set of logically related JavaScript statements in curly braces. Code blocks are used to define functions, or to delimit a set of instructions used for flow control.

```
// A stand-alone code block
{
  // Everything inside the curlies belongs
  // to this code block
}
```

A standalone code block is not very useful in JavaScript because it can neither be referenced nor reused



Syntax: Code Blocks

A code block is formed by enclosing a set of logically related JavaScript statements in curly braces. Code blocks are used to define functions, or to delimit a set of instructions used for flow control.

```
// A stand-alone code block
{
  // Everything inside the curlies belongs
  // to this code block
}
```

A standalone code block is not very useful in JavaScript because it can neither be referenced nor reused

```
// Here's a more useful code block
function doThis(someParameter) {
  // This code block becomes the executable
  // part of a function object.
}

doThis("a value");
```

Preceding a code block with the keyword `function()` (and an optional name) creates a function object

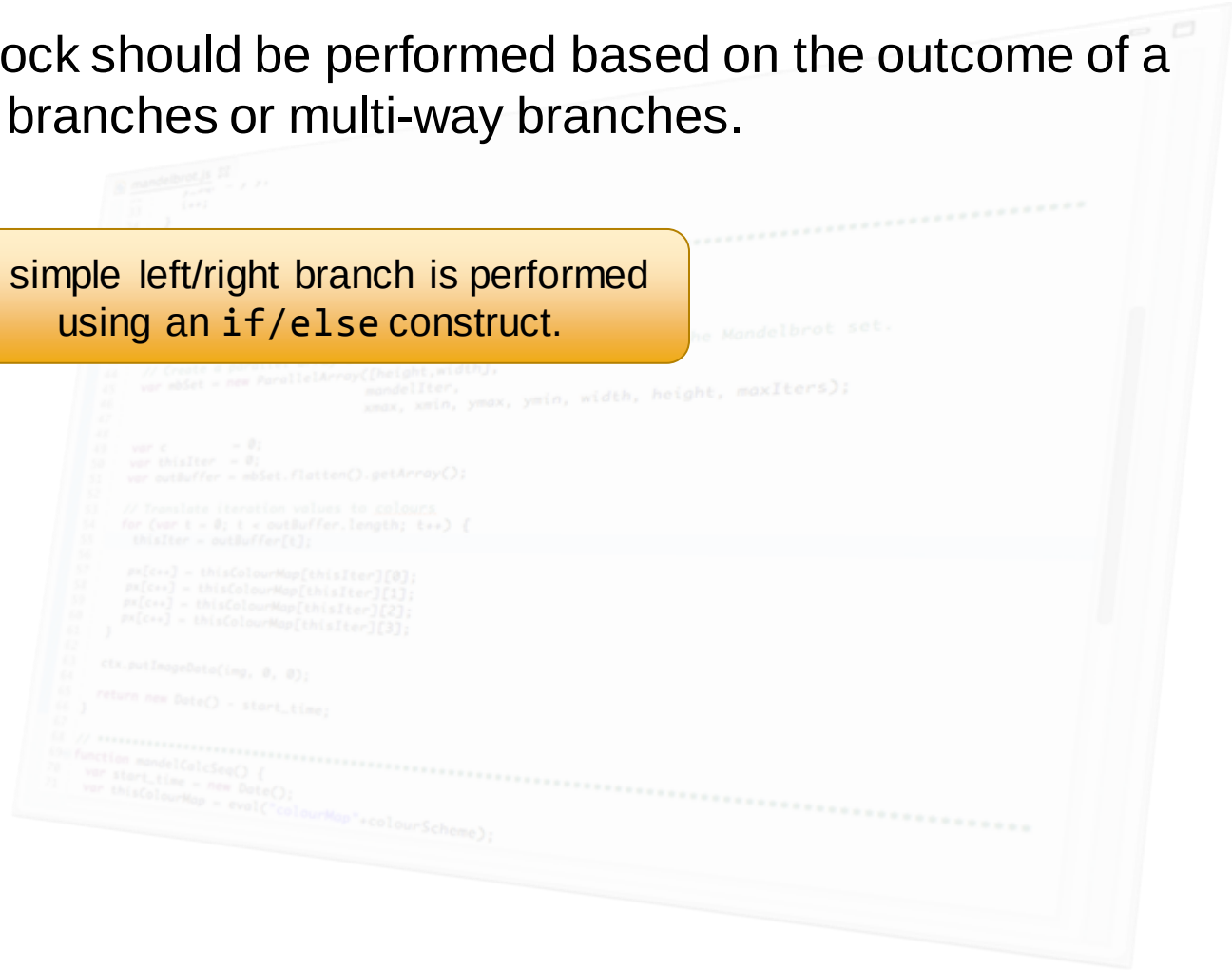
A function is executed by using the invocation operator `()` after the function name with zero or more parameters

Syntax: Flow Control: Selection

A selection statement determines which code block should be performed based on the outcome of a condition. Selections are either simple left/right branches or multi-way branches.

```
// Left/Right branch:
// Is this condition true?
if (someCondition == true) {
  // Yup, so do this code block
}
else {
 // Nope, do this code block instead
}
```

A simple left/right branch is performed using an `if/else` construct.



Syntax: Flow Control: Selection

A selection statement determines which code block should be performed based on the outcome of a condition. Selections are either simple left/right branches or multi-way branches.

```
// Left/Right branch:
// Is this condition true?
if (someCondition == true) {
  // Yup, so do this code block
}
else {
 // Nope, do this code block instead
}
```

A simple left/right branch is performed using an `if/else` construct.

```
// Multi-way branch:
switch (someValue) {
  case "a":
    alert("Bang!");
    break;
  case "b":
    alert("Fizz!");
    break;
  default:
    alert("None of the above!");
}
```

Multi-way branch is performed using a `switch/case` construct

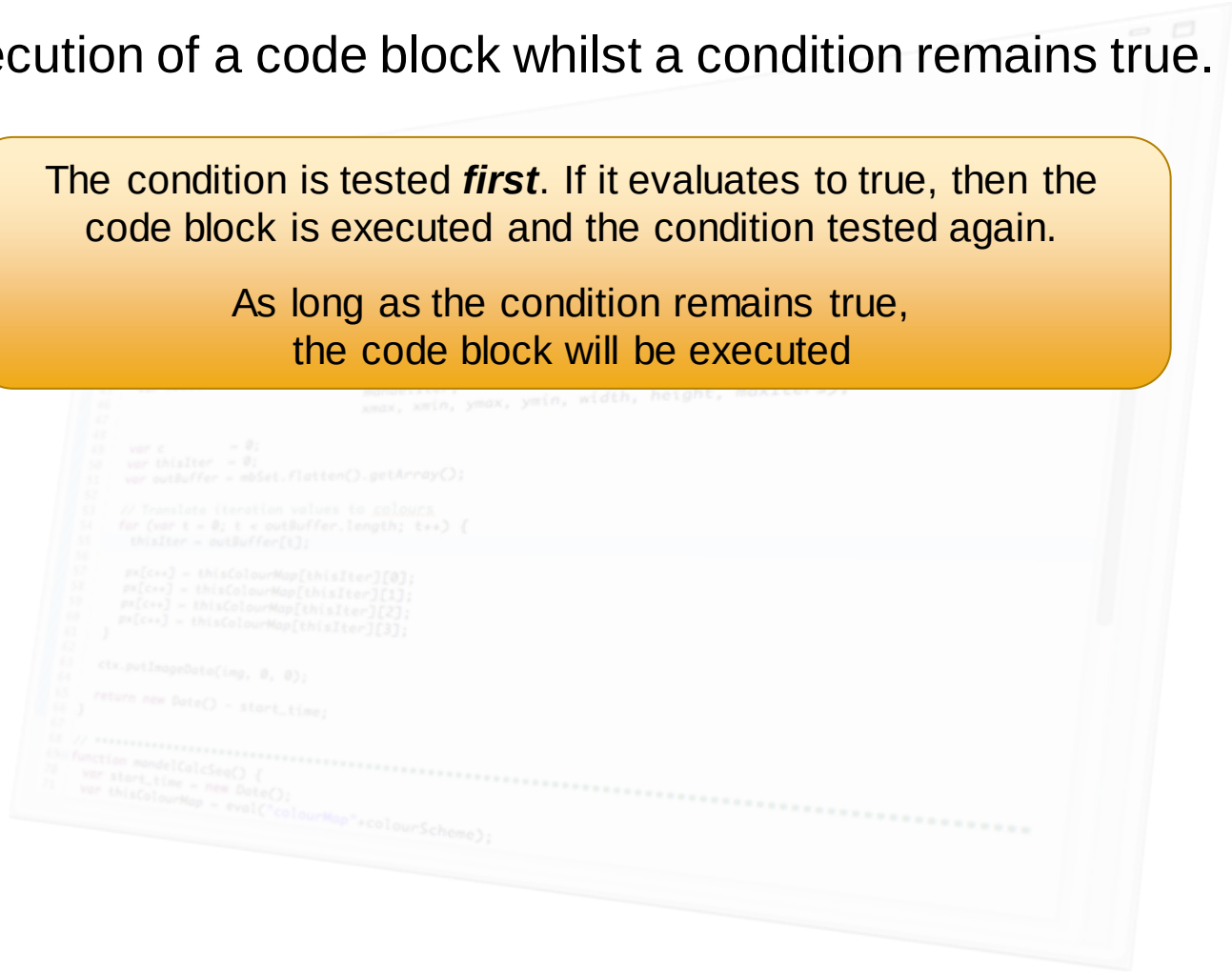
Syntax: Flow Control: Iteration

"Iteration" is the name given to the repeated execution of a code block whilst a condition remains true.

```
// Iterate using 'while'
while (someCondition == true) {
  // Do this code block
}
```

The condition is tested ***first***. If it evaluates to true, then the code block is executed and the condition tested again.

As long as the condition remains true, the code block will be executed



Syntax: Flow Control: Iteration

"Iteration" is the name given to the repeated execution of a code block whilst a condition remains true.

```
// Iterate using 'while'
while (someCondition == true) {
  // Do this code block
}
```

The condition is tested ***first***. If it evaluates to true, then the code block is executed and the condition tested again.

As long as the condition remains true, the code block will be executed

```
// Iterate using 'for'
for (var i=0; i<someLimit; i++) {
  // Do this code block
}
```

A for loop executes a code block a specific number of times and is controlled by providing:

- A start value for the loop control variable
- A continuation condition
- An instruction to modify the loop control variable

Syntax: Flow Control: Iteration

"Iteration" is the name given to the repeated execution of a code block whilst a condition remains true.

```
// Iterate using 'while'
while (someCondition == true) {
  // Do this code block
}
```

The condition is tested ***first***. If it evaluates to true, then the code block is executed and the condition tested again.

As long as the condition remains true, the code block will be executed

```
// Iterate using 'for'
for (var i=0; i<someLimit; i++) {
  // Do this code block
}
```

A for loop executes a code block a specific number of times and is controlled by providing:

- A start value for the loop control variable
- A continuation condition
- An instruction to modify the loop control variable

```
// Iterate using 'for ... in'
for (var prop in someObject) {
  // Perform this code block on each
  // property belonging to someObject
}
```

The for in construct allows you to iterate through all the properties belonging to an object

Syntax: Statement Terminators

In JavaScript the statement terminator is a semi-colon. However, be careful; its use is ***optional!***

```
// Be careful with the optional statement terminator in JavaScript!
var someVariable = "Some value"  // The semi-colon is missing, but that's OK because there's nothing ambiguous here...
```

Syntax: Statement Terminators

In JavaScript the statement terminator is a semi-colon. However, be careful; its use is ***optional!***

```
// Be careful with the optional statement terminator in JavaScript!
var someVariable = "Some value"  // The semi-colon is missing, but that's OK because there's nothing ambiguous here...

// This is also OK. The invisble carriage return after the 'return n * n' statement automatically terminates the expression.
function good_sqr(n) {
  return n * n
}
```

Syntax: Statement Terminators

In JavaScript the statement terminator is a semi-colon. However, be careful; its use is ***optional!***

```
// Be careful with the optional statement terminator in JavaScript!
var someVariable = "Some value"  // The semi-colon is missing, but that's OK because there's nothing ambiguous here...

// This is also OK. The invisble carriage return after the 'return n * n' statement automatically terminates the expression.
function good_sqr(n) {
  return n * n
}

// NASTY SURPRISE!! This function always returns the special value 'undefined'!
function bad_sqr(n) {
  // The 'return' statement used with no parameters means 'return undefined'. If 'return' is immediately followed by a
  // carriage return, then JavaScript assumes that you meant to write 'return undefined'.
  // The expression 'n * n;' on its own is valid, but will never be executed because the function has already terminated
  return
    n * n;
}
```

Syntax: Statement Terminators

In JavaScript the statement terminator is a semi-colon. However, be careful; its use is ***optional!***

```
// Be careful with the optional statement terminator in JavaScript!
var someVariable = "Some value"  // The semi-colon is missing, but that's OK because there's nothing ambiguous here...

// This is also OK. The invisble carriage return after the 'return n * n' statement automatically terminates the expression.
function good_sqr(n) {
  return n * n
}

// NASTY SURPRISE!! This function always returns the special value 'undefined'!
function bad_sqr(n) {
  // The 'return' statement used with no parameters means 'return undefined'. If 'return' is immediately followed by a
  // carriage return, then JavaScript assumes that you meant to write 'return undefined'.
  // The expression 'n * n;' on its own is valid, but will never be executed because the function has already terminated
  return
    n * n;
}
```

IMPORTANT!

A missing statement terminator ***sometimes*** results in a carriage return being misinterpreted as the end of the statement. Therefore:

- JavaScript statements should always be terminated explicitly
- Only split JavaScript statements across multiple lines at points where statement termination is not syntactically possible

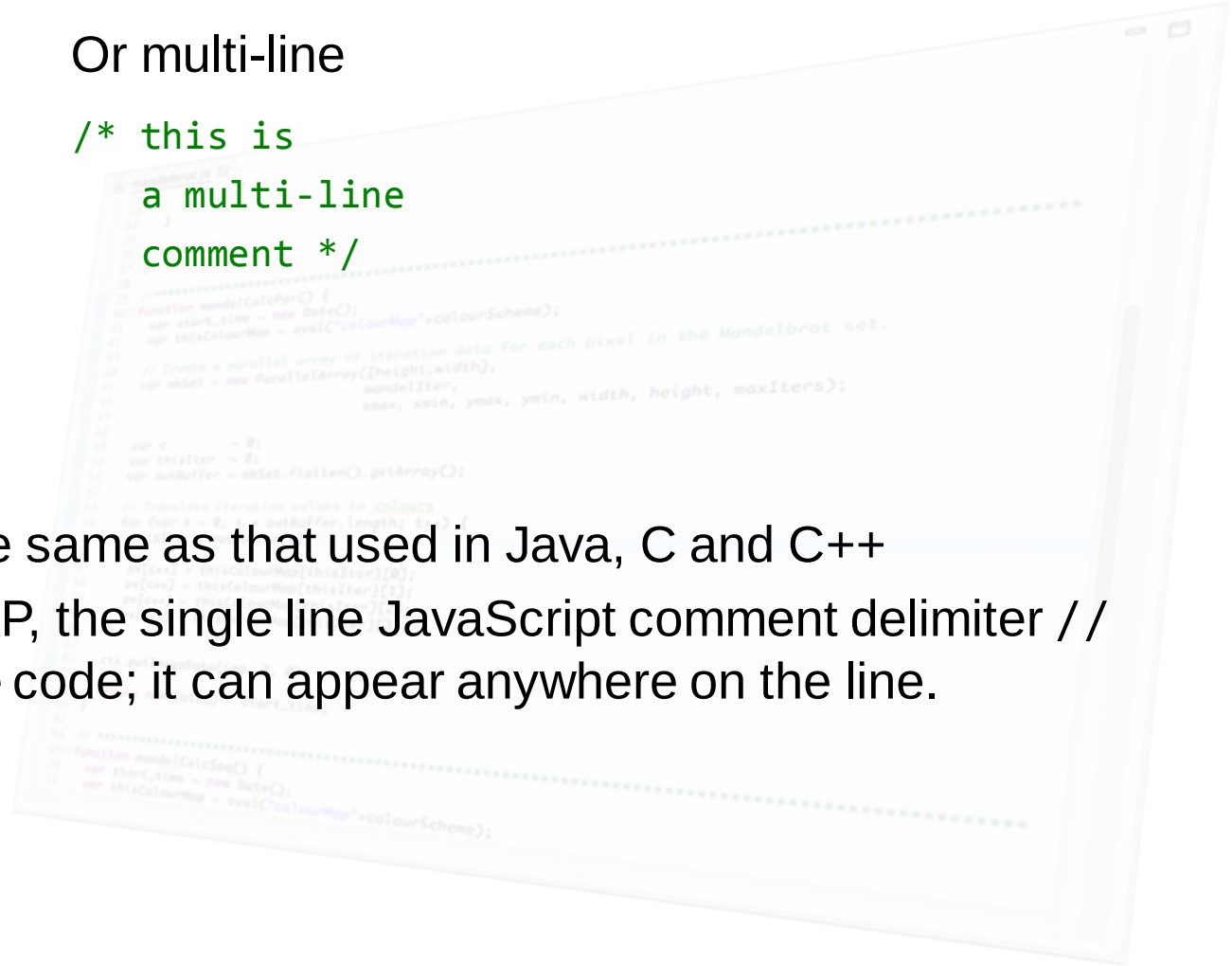
Syntax: Comments

Comments can be single line

```
// this is a single line comment
```

Or multi-line

```
/* this is
   a multi-line
   comment */
```



The comment syntax in JavaScript is exactly the same as that used in Java, C and C++

Unlike the full line comment delimiter "*" in ABAP, the single line JavaScript comment delimiter // does not need to start in column 1 of the source code; it can appear anywhere on the line.

Basic Operators

Unary Operators 1/4

A unary operator is an operator that requires only one operand.

Some unary operators can be specified either before (prefix) or after (postfix) their operand.

```
var value1 = 1;
var value2 = 0;
```

Unary Operators 1/4

A unary operator is an operator that requires only one operand.

Some unary operators can be specified either before (prefix) or after (postfix) their operand.

```
var value1 = 1;
var value2 = 0;

// Unary increment ++
// Unary decrement --
// The (in/de)crement operator can be used as either a prefix or a postfix!
value2 = value1++;    // First, assign value1 to value2 then increment value1
value1;               // 2
value2;               // 1
```

Unary Operators 1/4

A unary operator is an operator that requires only one operand.

Some unary operators can be specified either before (prefix) or after (postfix) their operand.

```
var value1 = 1;
var value2 = 0;

// Unary increment ++
// Unary decrement --
// The (in/de)crement operator can be used as either a prefix or a postfix!
value2 = value1++;    // First, assign value1 to value2 then increment value1
value1;               // 2
value2;               // 1

value2 = ++value1;    // First, increment value1 then assign the result to value2
value1;               // 3
value2;               // 3
```

Unary Operators 2/4

The arithmetic operators for addition and subtraction can be used as prefix unary operators.

```
var value1 = -1;
var value2 = "cat";

-value1; // → 1.   Return the negative numeric value
+value1; // → -1.  Return the numeric value, not the positive numeric value!
         //         Notice, the minus sign is not flipped!

-value2; // → NaN. A string has no numeric representation, so the result is 'Not a number'
+value2; // → NaN. A string has no numeric representation, so the result is 'Not a number'
```

Unary Operators 3/4

There are two unary logical NOT operators.
"!" performs a Boolean NOT operation and "~" performs a bitwise NOT operation.

```
var value1 = false;
var value2 = 2;


// "!" coerces its operand to type Boolean (if necessary), then negates it
!value1;            // → true.
!value2;            // → false.  Any non-zero number equates to true, !true is false
```

Unary Operators 3/4

There are two unary logical NOT operators.
"!" performs a Boolean NOT operation and "~" performs a bitwise NOT operation.

```
var value1 = false;
var value2 = 2;
var value3 = 0x0F; // A hexadecimal value is prefixed by 0x. 0F is the binary value 00001111

// "!" coerces its operand to type Boolean (if necessary), then negates it
!value1;           // → true.
!value2;           // → false.  Any non-zero number equates to true, !true is false

// "~" performs a bitwise logical NOT. Each bit in the binary value is flipped
~value3;           // → 0xF0 (binary value 00001111 becomes 11110000)
```

Unary Operators 4/4 (And just for fun…)

What would happen if you applied the bitwise NOT operator to a Boolean value?

```
// Please explain the following behaviour:
// Hint: You need to understand how the "2's compliment" binary representation works
~true;                  // Bitwise NOT of Boolean true  → -2
~false;                 // Bitwise NOT of Boolean false → -1
```

Unary Operators 4/4 (And just for fun…)

What would happen if you applied the bitwise NOT operator to a Boolean value?

```
// Please explain the following behaviour:
// Hint: You need to understand how the "2's compliment" binary representation works
~true;                  // Bitwise NOT of Boolean true  → -2
~false;                 // Bitwise NOT of Boolean false → -1

// Think about the binary values used to represent TRUE and FALSE (hint → 1 and 0)
// So 'true' is interpreted as numeric 1 (which as a single byte is 00000001) and 'false' is
// interpreted as numeric 0 (which as a single byte is 00000000).
```

Unary Operators 4/4 (And just for fun…)

What would happen if you applied the bitwise NOT operator to a Boolean value?

```
// Please explain the following behaviour:
// Hint: You need to understand how the "2's compliment" binary representation works
~true;                  // Bitwise NOT of Boolean true  → -2
~false;                 // Bitwise NOT of Boolean false → -1

// Think about the binary values used to represent TRUE and FALSE (hint → 1 and 0)
// So 'true' is interpreted as numeric 1 (which as a single byte is 00000001) and 'false' is
// interpreted as numeric 0 (which as a single byte is 00000000).

// Now perform the bitwise NOT operation: so all the bits in 00000001 are flipped to give
// 11111110, and all the bits in 00000000 are flipped to give 11111111
```

Unary Operators 4/4 (And just for fun…)

What would happen if you applied the bitwise NOT operator to a Boolean value?

```
// Please explain the following behaviour:
// Hint: You need to understand how the "2's compliment" binary representation works
~true;                  // Bitwise NOT of Boolean true  → -2
~false;                 // Bitwise NOT of Boolean false → -1

// Think about the binary values used to represent TRUE and FALSE (hint → 1 and 0)
// So 'true' is interpreted as numeric 1 (which as a single byte is 00000001) and 'false' is
// interpreted as numeric 0 (which as a single byte is 00000000).

// Now perform the bitwise NOT operation: so all the bits in 00000001 are flipped to give
// 11111110, and all the bits in 00000000 are flipped to give 11111111

// Now look at these values using the 2's complement format (where the senior bit has a
// value of -128, not +128). Therefore, 11111110 = -2 and 11111111 = -1
```

Binary Operators 1/3

A binary operator is one that requires two operands; one on the left and one on the right.
Binary operators that separate their operands are also known as “infix” operators.

```
// The familiar arithmetic operations such as add, subtract, divide and multiply are binary
// operators
var x = 12;
var y = 3;

// No surprises here I hope...
x + y;    // → 15
x - y;    // → 9
x * y;    // → 36
x / y;    // → 4

// Another less familiar binary operator is modulus division
x % y;    // → 0. Divide x by y, return the remainder (12 / 3 = 4 remainder 0)
```

Binary Operators 2/3

Binary operators to perform logical operations at the bit-level and bitwise shifting.
Bit shifting effectively multiplies or divides a value by some power of 2.

```
var x = 0x0F;    // Decimal 15 or 00001111
var y = 0xAA;    // Decimal 170 or 10101010
var z = 3;

// These operators manipulate the individual bits of their operands
x & y;   // → 0x0A  00001010  Bitwise AND
x | y;   // → 0xAF  10101111  Bitwise OR
x ^ y;   // → 0xA5  10100101  Bitwise Exclusive OR (XOR)

y << z;  // → 1360  Shift y left by z places, zero padding on right (y * 2^z)
y >> z;  // → 0x15  Shift y right by z places, preserving the sign bit
y >>> z; // → 0x15  Shift y right by z places, zero padding on left
```

Binary Operators 3/3

Binary operators to perform Boolean logical operations

```
// These operators treat their operands as Boolean values
x && y;   // Boolean AND  If x is false, the condition fails immediately without testing y
x || y;   // Boolean OR   If x is true, the condition passes immediately without testing y
```

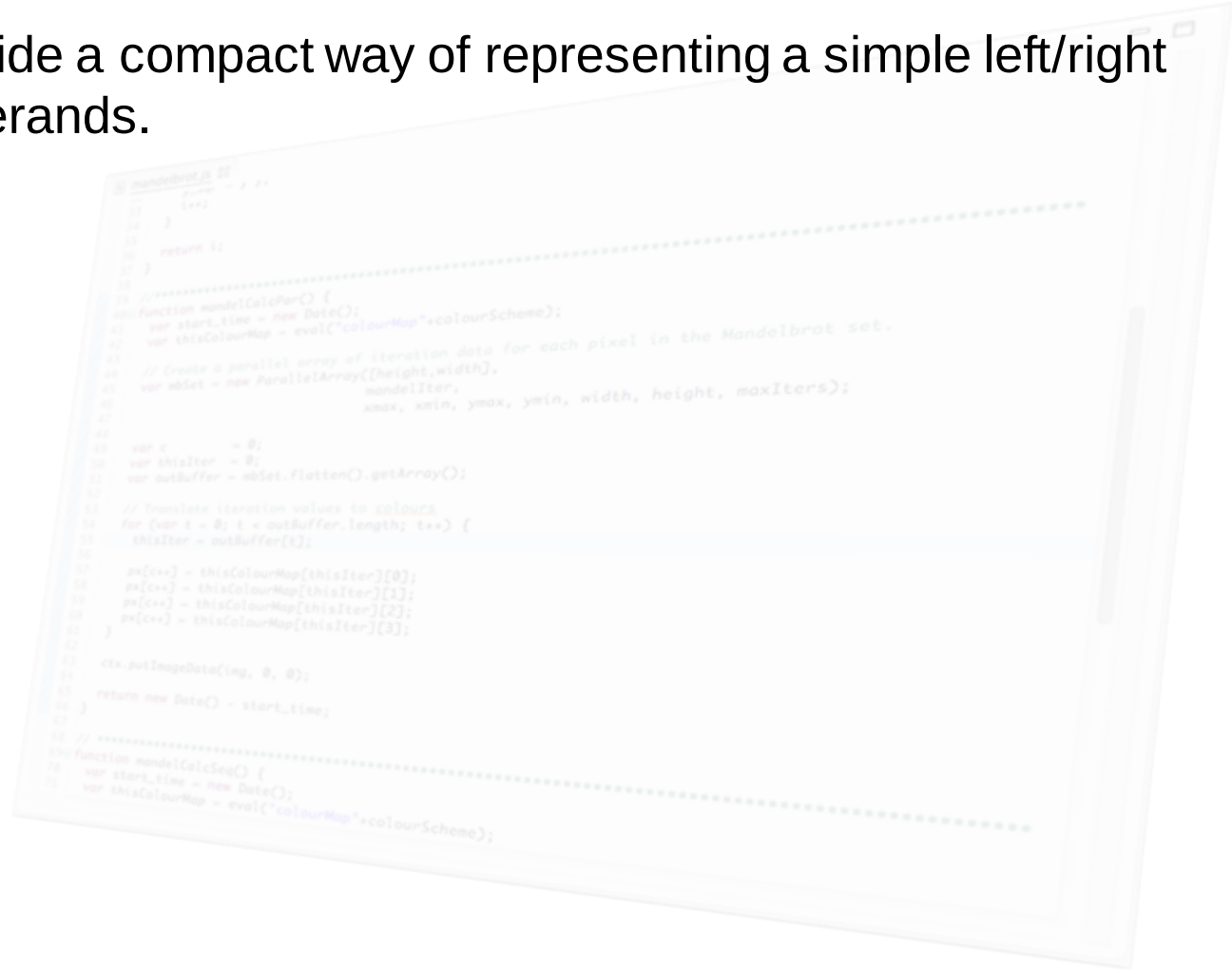
As with all modern languages, the JavaScript Boolean operators AND and OR use a performance enhancement called "early bail out".

This means that if the first operand of the AND operator evaluates to false, then the whole condition has failed and there is no point testing the second operand, so the test bails out early.

Similarly, if the first operand of the OR operator evaluates to true, then the whole condition has passed and there is no point testing the second operand, so the test bails out early.

Ternary Operator: Test A Condition Without Using if

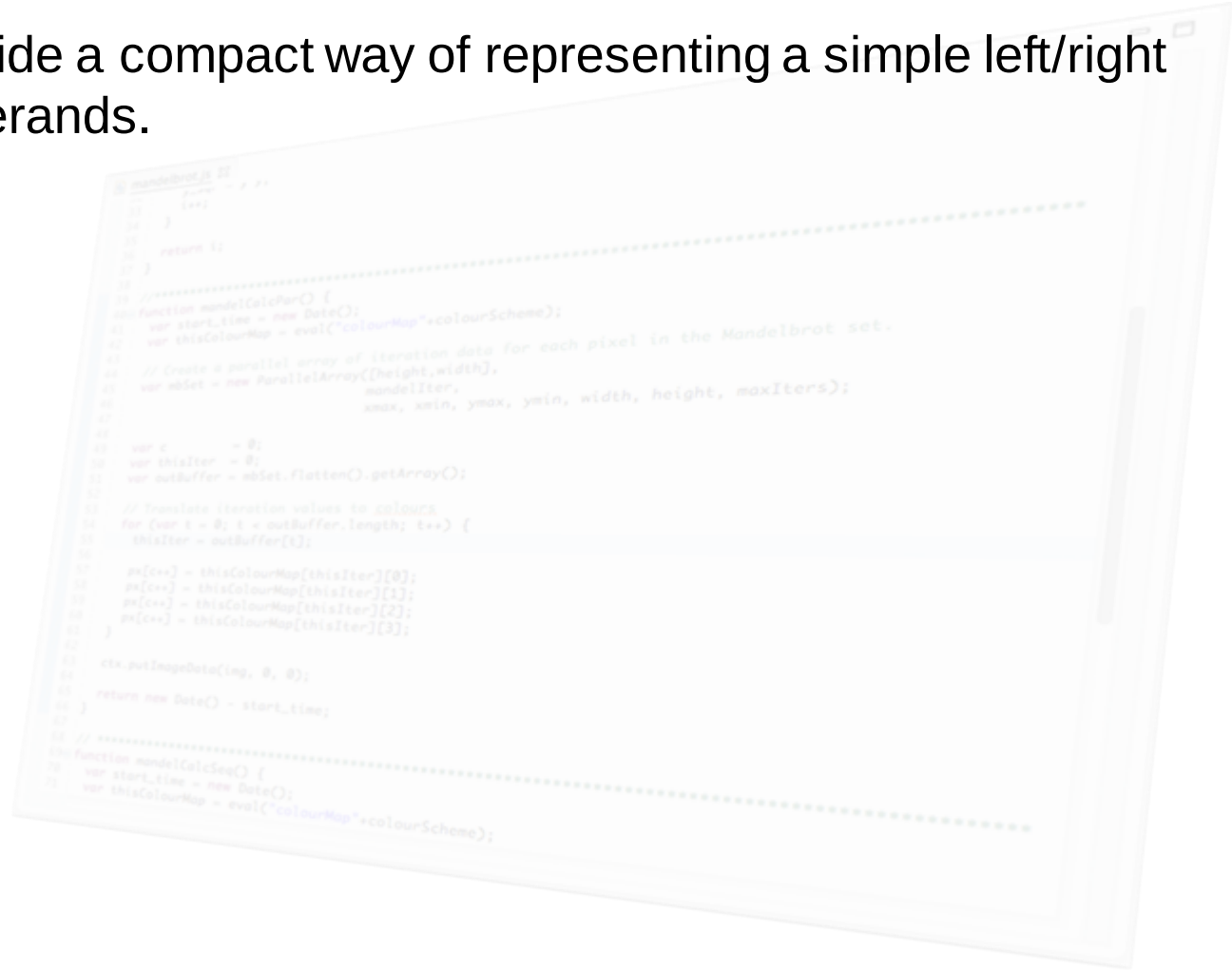
JavaScript has a single ternary operator to provide a compact way of representing a simple left/right selection. A ternary operator requires ***three*** operands.



Ternary Operator: Test A Condition Without Using if

JavaScript has a single ternary operator to provide a compact way of representing a simple left/right selection. A ternary operator requires ***three*** operands.

```
// Doing it the long way
function catSays(animal) {
  var result;

  if (animal == "cat" ||
      animal == "kitten" ) {
    result = "Meow!";
  }
  else {
    result "Not a cat!";
  }

  return result;
}
```



This is perfectly correct, but it is unnecessarily verbose...

Ternary Operator: Test A Condition Without Using if

JavaScript has a single ternary operator to provide a compact way of representing a simple left/right selection. A ternary operator requires ***three*** operands.

```
// Doing it the long way
function catSays(animal) {
  var result;

  if (animal == "cat" ||
      animal == "kitten" ) {
    result = "Meow!";
  }
  else {
    result "Not a cat!";
  }

  return result;
}
```

```
// The ternary operator allows you to write an if
// statement without needing to use the "if" keyword
function catSays(animal) {
  return (animal == "cat" ||
          animal == "kitten") ? "Meow!" : "Not a cat!";
}
```

This is perfectly correct, but it is unnecessarily verbose...

Ternary Operator: Test A Condition Without Using if

JavaScript has a single ternary operator to provide a compact way of representing a simple left/right selection. A ternary operator requires ***three*** operands.

```
// Doing it the long way
function catSays(animal) {
  var result;

  if (animal == "cat" ||
      animal == "kitten" ) {
    result = "Meow!";
  }
  else {
    result "Not a cat!";
  }

  return result;
}
```

```
// The ternary operator allows you to write an if
// statement without needing to use the "if" keyword
function catSays(animal) {
  return (animal == "cat" ||
          animal == "kitten") ? "Meow!" : "Not a cat!";
}
```

If this condition is true...

This is perfectly correct, but it is unnecessarily verbose...

Ternary Operator: Test A Condition Without Using if

JavaScript has a single ternary operator to provide a compact way of representing a simple left/right selection. A ternary operator requires ***three*** operands.

```
// Doing it the long way
function catSays(animal) {
  var result;

  if (animal == "cat" ||
      animal == "kitten" ) {
    result = "Meow!";
  }
  else {
    result "Not a cat!";
  }

  return result;
}
```

```
// The ternary operator allows you to write an if
// statement without needing to use the "if" keyword
function catSays(animal) {
  return (animal == "cat" ||
          animal == "kitten") ? "Meow!" : "Not a cat!";
}
```

Then the result is whatever value follows the "?" character

This is perfectly correct, but it is unnecessarily verbose...

Ternary Operator: Test A Condition Without Using if

JavaScript has a single ternary operator to provide a compact way of representing a simple left/right selection. A ternary operator requires ***three*** operands.

```
// Doing it the long way
function catSays(animal) {
  var result;

  if (animal == "cat" ||
      animal == "kitten" ) {
    result = "Meow!";
  }
  else {
    result "Not a cat!";
  }

  return result;
}
```

This is perfectly correct, but it is unnecessarily verbose...

```
// The ternary operator allows you to write an if
// statement without needing to use the "if" keyword
function catSays(animal) {
  return (animal == "cat" ||
          animal == "kitten") ? "Meow!" : "Not a cat!";
}
```

Else the result is whatever value follows the ":" character

Ternary Operator: Test A Condition Without Using if

JavaScript has a single ternary operator to provide a compact way of representing a simple left/right selection. A ternary operator requires ***three*** operands.

```
// Doing it the long way
function catSays(animal) {
  var result;

  if (animal == "cat" ||
      animal == "kitten" ) {
    result = "Meow!";
  }
  else {
    result "Not a cat!";
  }

  return result;
}
```

```
// The ternary operator allows you to write an if
// statement without needing to use the "if" keyword
function catSays(animal) {
  return (animal == "cat" ||
          animal == "kitten") ? "Meow!" : "Not a cat!";
}
```

The value returned by the function

…is the outcome of the ternary operator

This is perfectly correct, but it is unnecessarily verbose...

Type Coercion

Type Coercion to String: Using the Unary + Operator

"Type coercion"* is the process by which a value of one data type is transformed into a value of another data type. This will occur ***automatically*** whenever an operator is supplied with a value that is not immediately suitable.

```
var value1 = "-1";
var value2 = "cat";



+value1; // → -1.          Unary "+" returns the numeric representation of a string
+value2; // → NaN.         Numeric representation of a non-numeric string is 'Not a number'
```

* To "coerce" someone into doing something means that you have forced them to do something they would not otherwise willingly do.

Type Coercion to String: Using the Unary + Operator

"Type coercion" is the process by which a value of one data type is transformed into a value of another data type. This will occur ***automatically*** whenever an operator is supplied with a value that is not immediately suitable.

```
var value1 = "-1";
var value2 = "cat";
var value3 = false;
var value4 = true;

+value1; // → -1.          Unary "+" returns the numeric representation of a string
+value2; // → NaN.         Numeric representation of a non-numeric string is 'Not a number'

+value3; // → numeric 0.   Boolean false is coerced to numeric 0
+value4; // → numeric 1.   Boolean true is coerced to numeric 1
```

The unary + operator can be used (if possible) to coerce character strings or Boolean values into numbers.

Type Coercion to String: Overloading the Binary + Operator

The binary plus operator can also be used for string concatenation. This is called “overloading” an operator. ***Be careful though!*** The behaviour changes depending on the data types of the operands!

```
// The plus operator performs a normal arithmetic operation because both operands are numeric
1 + 2;    // → 3  (No surprises here, I hope)
```

Type Coercion to String: Overloading the Binary + Operator

The binary plus operator can also be used for string concatenation. This is called “overloading” an operator. ***Be careful though!*** The behaviour changes depending on the data types of the operands!

```
// The plus operator performs a normal arithmetic operation because both operands are numeric
1 + 2;    // → 3  (No surprises here, I hope)

// If any one of the operands passed to the binary plus operator is of type string, then the
// operator has been "overloaded" and string concatenation will take place instead of
// arithmetic addition.
1 + "2";  // → "12"
"3" + 4;  // → "34"
```

Type Coercion to String: Overloading the Binary + Operator

The binary plus operator can also be used for string concatenation. This is called “overloading” an operator. ***Be careful though!*** The behaviour changes depending on the data types of the operands!

```
// The plus operator performs a normal arithmetic operation because both operands are numeric
1 + 2;    // → 3  (No surprises here, I hope)

// If any one of the operands passed to the binary plus operator is of type string, then the
// operator has been "overloaded" and string concatenation will take place instead of
// arithmetic addition.
1 + "2";  // → "12"
"3" + 4;  // → "34"

// Caution – type coercion takes place based on the data types of the operands!
1 + 2 + "3" + 4; // → "334" '1+2' is a numeric operation, so add. The rest is concatenation
```

Type Coercion to Boolean: Using the Boolean NOT Operator

The Boolean NOT operator can be used to coerce a value to be treated either as `true` or `false`.

```
var value1 = false;
var value2 = 2;

// "!" coerces its operand to Boolean (if necessary), then negates it
!value1;            // → true.
!value2;            // → false.  Any non-zero number equates to true, !true is false
```

Type Coercion to Boolean: Using the Boolean NOT Operator

The Boolean NOT operator can be used to coerce a value to be treated either as `true` or `false`.

```
var value1 = false;
var value2 = 2;

// "!" coerces its operand to Boolean (if necessary), then negates it
!value1;            // → true.
!value2;            // → false.  Any non-zero number equates to true, !true is false

// Here's a useful trick when minimising JavaScript code. The keywords "true" and "false" use
// 4 and 5 characters each. Rewriting these as !0 and !1 reduces the count to 2
!0;                 // → true.   0 is coerced to Boolean false, !false is true
!1;                 // → false.  1 is coerced to Boolean true, !true is false
```

Type Coercion to Boolean: Truthy and Falsy 1/2

Since all JavaScript data types can be coerced to a Boolean value, this leads to the concept that all JavaScript values are either “truthy” or “falsy”.

```
// Values that evaluate to Boolean true are said to be 'truthy', and those that evaluate to
// Boolean false are said to be 'falsy'.

// For example, create an object whose properties hold a value of each data type
var datatypes = {
      zero:0, one:1, "null":null, "undefined":undefined, "NaN":NaN,
      "function":function() {}, "true":true, "false":false,
      emptyString:"", emptyObject:{}, emptyArray:[]
    }
```

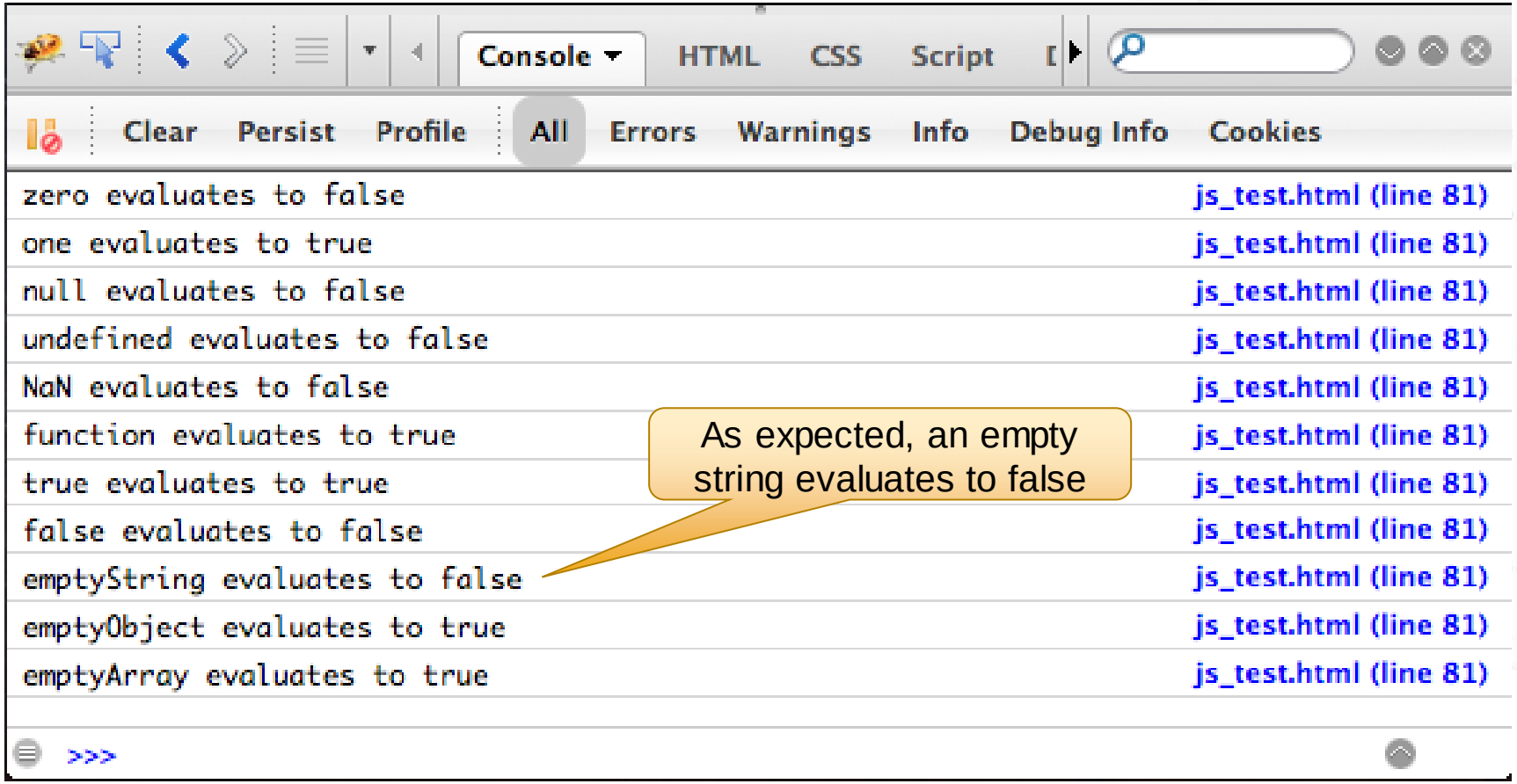
Type Coercion to Boolean: Truthy and Falsy 1/2

Since all JavaScript data types can be coerced to a Boolean value, this leads to the concept that all JavaScript values are either “truthy” or “falsy”.

```
// Values that evaluate to Boolean true are said to be 'truthy', and those that evaluate to
// Boolean false are said to be 'falsy'.

// For example, create an object whose properties hold a value of each data type
var datatypes = {
      zero:0, one:1, "null":null, "undefined":undefined, "NaN":NaN,
      "function":function() {}, "true":true, "false":false,
      emptyString:"", emptyObject:{}, emptyArray:[]
    }

// Loop around each property in the object writing the coerced Boolean value to the console
for (var i in datatypes) {
  // See whether the above properties are truthy or falsy
  // The "double not" (!!) is a trick that coerces any data type to its Boolean equivalent
  console.log(i + " evaluates to " + !!datatypes[i]);
}
```

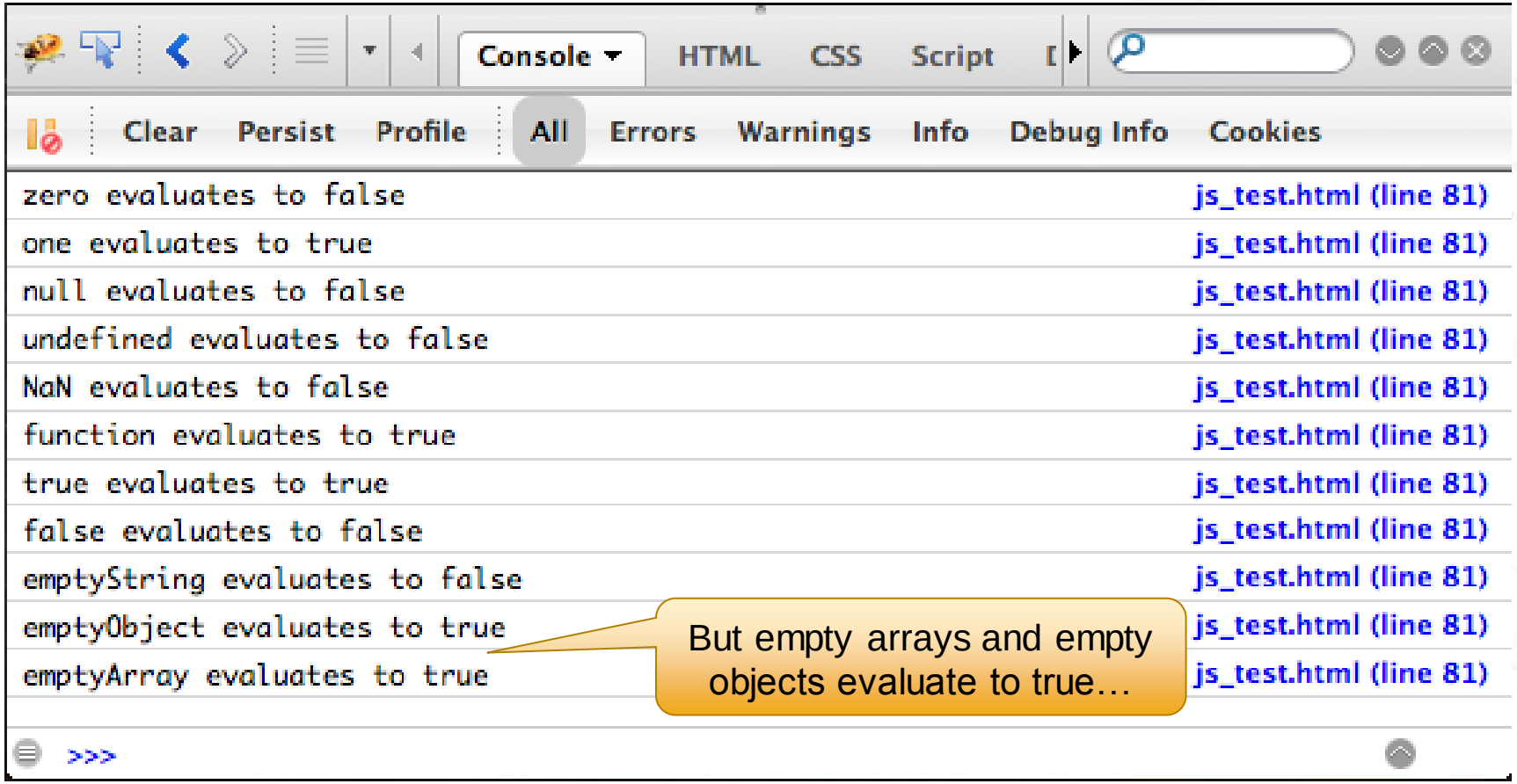
Type Coercion to Boolean: Truthy and Falsy 2/2

The general idea is this: any value that can be thought of as “empty” will evaluate to `false` such as zero or `null` etc. However, **BE CAREFUL!** Not all evaluations work the way you might expect!



Type Coercion to Boolean: Truthy and Falsy 2/2

The general idea is this: any value that can be thought of as “empty” will evaluate to `false` such as zero or `null` etc. However, **BE CAREFUL!** Not all evaluations work the way you might expect!



Type Coercion to Boolean: Overloading the OR Operator

As a consequence of type coercion to Boolean, we can overload the logical OR operator (||) to check whether a parameter has been passed to a function, and if not, substitute a default value.

```
// An overloaded OR operator is used to provide a default value if a function is not passed
// a required parameter
var person = function(fName,lName,DoB) {
  this.firstName = fName || "Not specified";
  this.lastName  = lName || "Not specified";
  this.dateOfBirth = DoB;
}
```

Type Coercion to Boolean: Overloading the OR Operator

As a consequence of type coercion to Boolean, we can overload the logical OR operator (||) to check whether a parameter has been passed to a function, and if not, substitute a default value.

```
// An overloaded OR operator is used to provide a default value if a function is not passed
// a required parameter
var person = function(fName,lName,DoB) {
  this.firstName = fName || "Not specified";
  this.lastName  = lName || "Not specified";
  this.dateOfBirth = DoB;
}
```

Here we rely on three aspects of JavaScript's behaviour:

1. Early bail out means that the second operand of the OR operator will only be evaluated if the first operand equals (or is coerced to) **`false`**
2. If a function parameter is not supplied, the parameter value will be **`null`**
3. Type coercion causes **`null`** to evaluate to **`false`**.

Type Coercion to Boolean: Overloading the OR Operator

As a consequence of type coercion to Boolean, we can overload the logical OR operator (||) to check whether a parameter has been passed to a function, and if not, substitute a default value.

```
// An overloaded OR operator is used to provide a default value if a function is not passed
// a required parameter
var person = function(fName,lName,DoB) {
  this.firstName = fName || "Not specified";
  this.lastName  = lName || "Not specified";
  this.dateOfBirth = DoB;
}
```

Here we rely on three aspects of JavaScript's behaviour:

1. Early bail out means that the second operand of the OR operator will only be evaluated if the first operand equals (or is coerced to) **`false`**
2. If a function parameter is not supplied, the parameter value will be **`null`**
3. Type coercion causes **`null`** to evaluate to **`false`**.

If either parameter `fName` or `lName` is supplied, then their non-null value will coerce to `true` causing OR to bail out early and return the parameter value. However, if either of these parameters are not supplied, then their values will be **`null`**, forcing the OR operator to return the value of its second operand – which is the default value.

Type Coercion to Boolean: Counting Down to Zero in Loops

Here's another example of where type coercion can be use to simplify the coding.

```
// Use number → Boolean type coercion to
// control a loop
var n = 5;

while (n--) {
  console.log("Stop when n = 0. n is now "+n);
}
```

```
Stop when n = 0. n is now 4
Stop when n = 0. n is now 3
Stop when n = 0. n is now 2
Stop when n = 0. n is now 1
Stop when n = 0. n is now 0
```

Here, we are relying on the fact that all non-zero numeric values are truthy and zero is falsy.

Type Coercion to Boolean: Counting Down to Zero in Loops

Here's another example of where type coercion can be use to simplify the coding.

```
// Use number → Boolean type coercion to
// control a loop
var n = 5;

while (n--) {
  console.log("Stop when n = 0. n is now "+n);
}
```

```
Stop when n = 0. n is now 4
Stop when n = 0. n is now 3
Stop when n = 0. n is now 2
Stop when n = 0. n is now 1
Stop when n = 0. n is now 0
```

Here, we are relying on the fact that all non-zero numeric values are truthy and zero is falsy.

The decrement operator `--` is specifically used in the postfix position. This means that the value of `n` is tested ***before*** being decremented.

Type Coercion to Boolean: Counting Down to Zero in Loops

Here's another example of where type coercion can be use to simplify the coding.

```
// Use number → Boolean type coercion to
// control a loop
var n = 5;

while (n--) {
  console.log("Stop when n = 0. n is now "+n);
}
```

```
Stop when n = 0. n is now 4
Stop when n = 0. n is now 3
Stop when n = 0. n is now 2
Stop when n = 0. n is now 1
Stop when n = 0. n is now 0
```

Here, we are relying on the fact that all non-zero numeric values are truthy and zero is falsy.

The decrement operator `--` is specifically used in the postfix position. This means that the value of `n` is tested ***before*** being decremented.

We now enter the body of the **`while`** loop with the decremented value of `n` – hence when the value of `n` is first written to the console it is 4, not 5. All non-zero values of `n` are truthy, so the loop continues.

Type Coercion to Boolean: Counting Down to Zero in Loops

Here's another example of where type coercion can be use to simplify the coding.

```
// Use number → Boolean type coercion to
// control a loop
var n = 5;

while (n--) {
  console.log("Stop when n = 0. n is now "+n);
}
```

```
Stop when n = 0. n is now 4
Stop when n = 0. n is now 3
Stop when n = 0. n is now 2
Stop when n = 0. n is now 1
Stop when n = 0. n is now 0
```

Here, we are relying on the fact that all non-zero numeric values are truthy and zero is falsy.

The decrement operator `--` is specifically used in the postfix position. This means that the value of `n` is tested ***before*** being decremented.

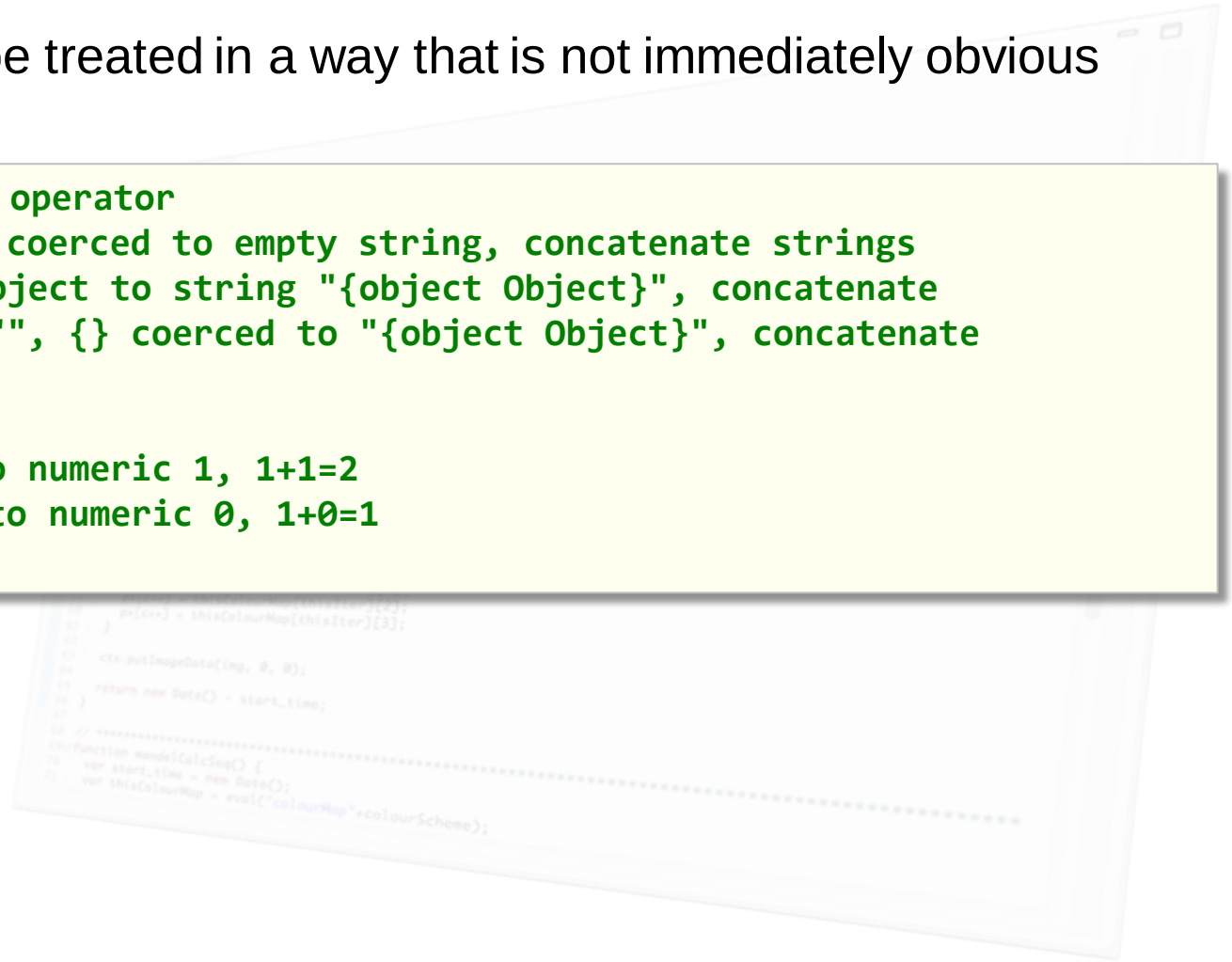
We now enter the body of the **`while`** loop with the decremented value of `n` – hence when the value of `n` is first written to the console it is 4, not 5. All non-zero values of `n` are truthy, so the loop continues.

The last time around the loop, `n` now equals 0. 0 is falsy, so the condition fails and the loop terminates.

Type Coercion: Now Things Start To Get A Little Strange…

Type coercion can cause certain data types to be treated in a way that is not immediately obvious

```
// Type coercion to string by overloading the plus operator
1 + [];    // → "1"                Empty array is coerced to empty string, concatenate strings
1 + {};    // → "1{object Object}" Coerce empty object to string "{object Object}", concatenate
[] + {};   // → "{object Object}"  [] coerced to "", {} coerced to "{object Object}", concatenate

// Type coercion of Boolean to numeric
1 + true;  // → 2       Boolean true is coerced to numeric 1, 1+1=2
1 + false; // → 1       Boolean false is coerced to numeric 0, 1+0=1
```



Type Coercion: OK, That's Just Weird…

And as if that were not strange enough, here are some even stranger edge cases - and as is written at the edges of all good Medieval maps: *Here be dragons*

```
// Type coercion can produce some bizarre results!
{} + [];      // → 0
{} + 1;       // → 1
({}) + 1;     // → "{object Object}1"

!+[];         // → true
!+[]+!![];    // → 2

!+[]+[];      // → "true"
!+[]+!![]+[]; // → "2"

[][[]];       // → undefined
[][[]]+[];    // → "undefined"
```

This strange behaviour is the basis of the Github obfuscation project called Hieroglyphy

The typeof Operator

Type Determination (Another Unary Operator)

The **typeof** operator allows you to discover the data type of a variable's value – well mostly…

```
// Declare some variables of various data types
var aNumber    = 123;
var aString    = "Nothing to see here, move along";
var anObject   = { aProperty : 0 };
var aFunction  = function() { };
var notANumber = NaN;
var anArray    = [1,2,3,4,5];
var nullValue  = null;

// What data types do we have here?
typeof aNumber;        // → 'number'
typeof aString;        // → 'string'
typeof anObject;       // → 'object'
typeof aFunction;      // → 'function'
```

Type Determination (Another Unary Operator)

The **typeof** operator allows you to discover the data type of a variable's value – well mostly…

```
// Declare some variables of various data types
var aNumber    = 123;
var aString    = "Nothing to see here, move along";
var anObject   = { aProperty : 0 };
var aFunction  = function() { };
var notANumber = NaN;
var anArray    = [1,2,3,4,5];
var nullValue  = null;

// What data types do we have here?
typeof aNumber;        // → 'number'
typeof aString;        // → 'string'
typeof anObject;       // → 'object'
typeof aFunction;      // → 'function'
typeof notANumber;     // → 'number'  This isn't as weird as you might think (See ECMAScript spec)
```

Type Determination (Another Unary Operator)

The **typeof** operator allows you to discover the data type of a variable's value – well mostly…

```
// Declare some variables of various data types
var aNumber    = 123;
var aString    = "Nothing to see here, move along";
var anObject   = { aProperty : 0 };
var aFunction  = function() { };
var notANumber = NaN;
var anArray    = [1,2,3,4,5];
var nullValue  = null;

// What data types do we have here?
typeof aNumber;        // → 'number'
typeof aString;        // → 'string'
typeof anObject;       // → 'object'
typeof aFunction;      // → 'function'
typeof notANumber;     // → 'number'  This isn't as weird as you might think (See ECMAScript spec)

// So far, so good – the typeof operator is being well-behaved...
typeof anArray;        // → 'object'   Hmmm, this answer isn't wrong, but then neither is it helpful
```

Type Determination (Another Unary Operator)

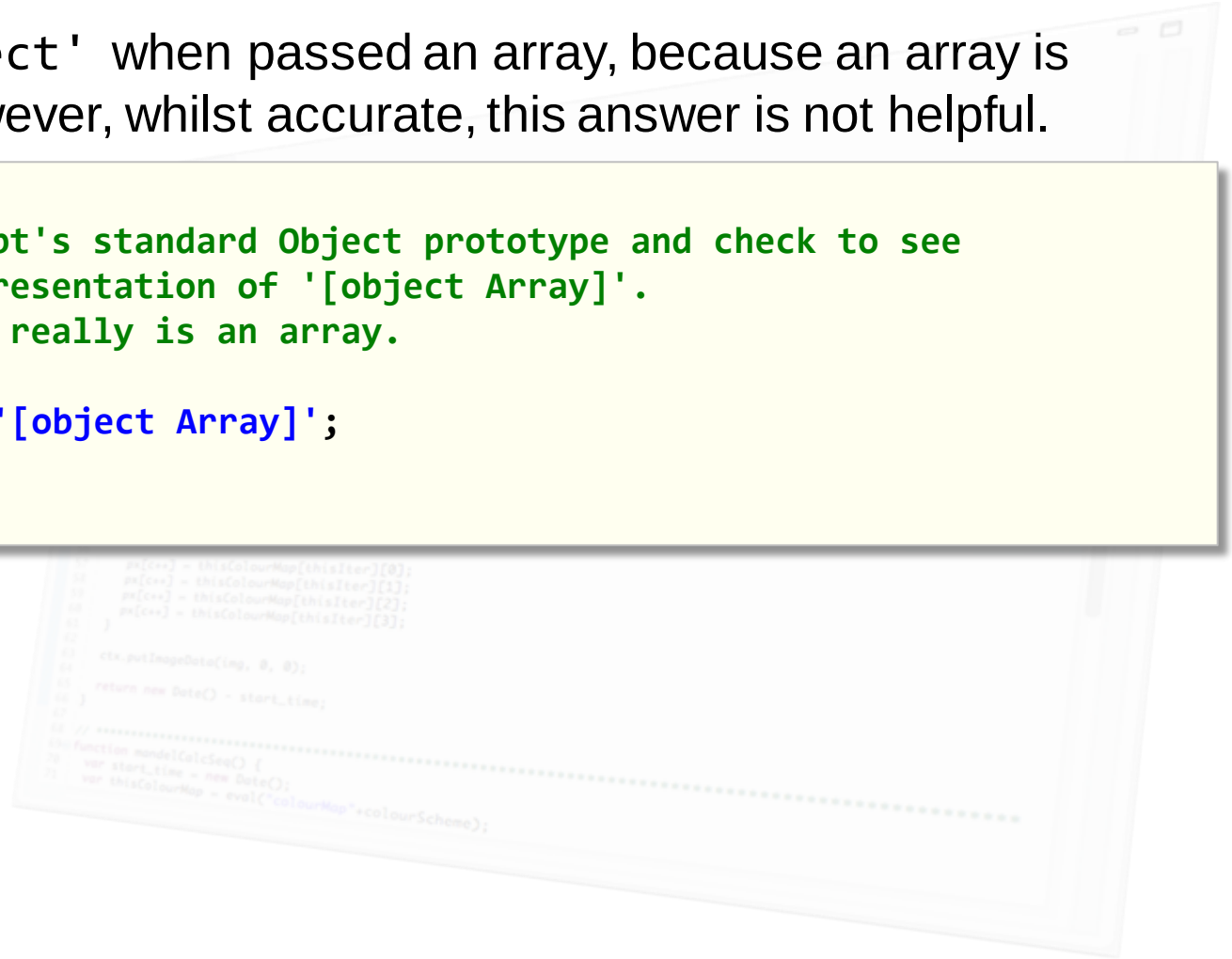
The **typeof** operator allows you to discover the data type of a variable's value – well mostly…

```
// Declare some variables of various data types
var aNumber    = 123;
var aString    = "Nothing to see here, move along";
var anObject   = { aProperty : 0 };
var aFunction  = function() { };
var notANumber = NaN;
var anArray    = [1,2,3,4,5];
var nullValue  = null;

// What data types do we have here?
typeof aNumber;        // → 'number'
typeof aString;        // → 'string'
typeof anObject;       // → 'object'
typeof aFunction;      // → 'function'
typeof notANumber;     // → 'number'  This isn't as weird as you might think (See ECMAScript spec)

// So far, so good – the typeof operator is being well-behaved...
typeof anArray;        // → 'object'   Hmmm, this answer isn't wrong, but then neither is it helpful
typeof nullValue;      // → 'object'   Sorry, but this is just wrong!
```

Fixing the `typeof` operator 1/3

`typeof` is not entirely wrong for returning `'object'` when passed an array, because an array is simply an object with array-like properties. However, whilst accurate, this answer is not helpful.

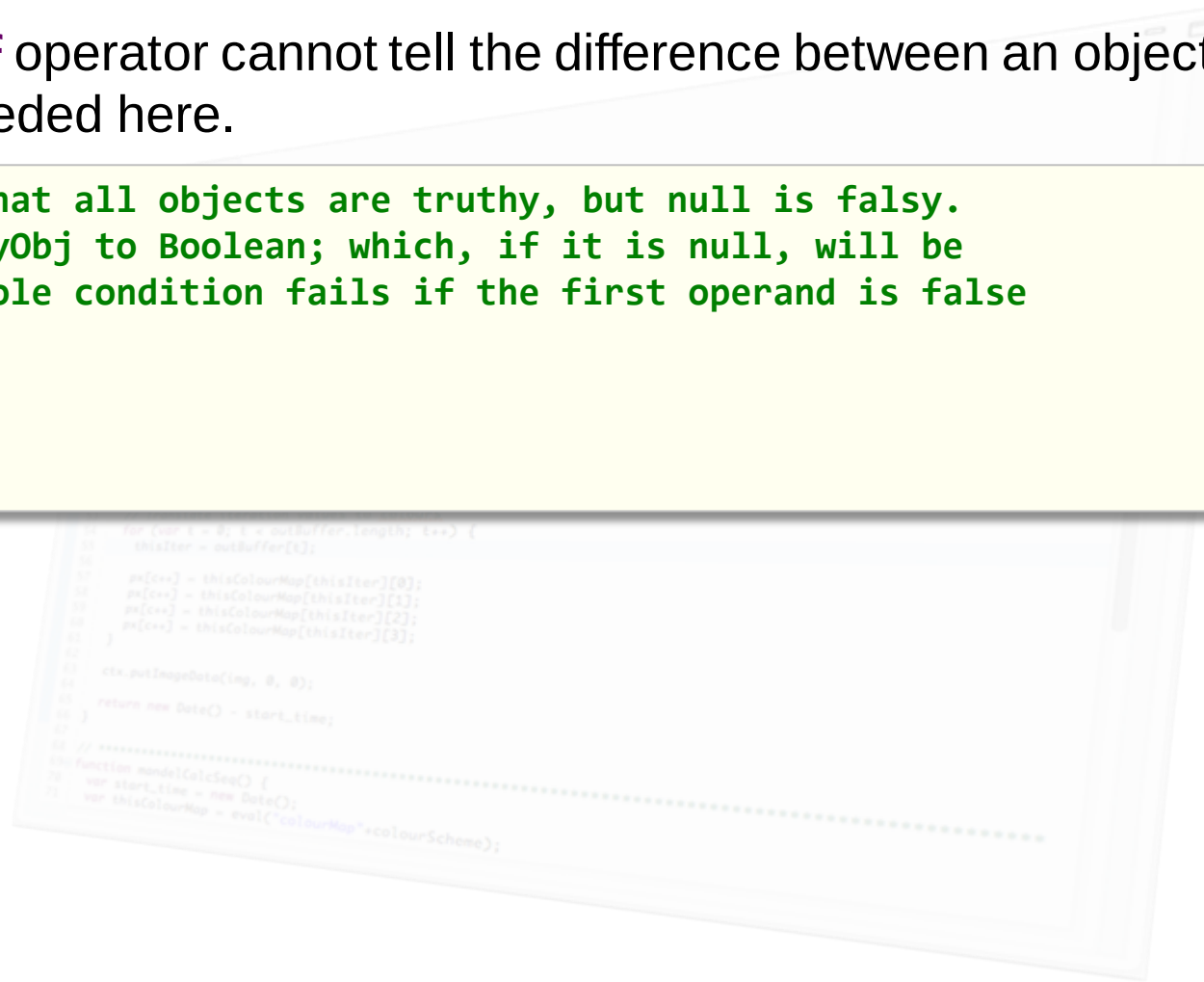
```
// Create our own isArray() function
// Use the toString() method belonging to JavaScript's standard Object prototype and check to see
// whether the object in question has a string representation of '[object Array]'.
// If it does, then we can be sure that the object really is an array.
var isArray = function(obj) {
  return Object.prototype.toString.apply(obj) === '[object Array]';
};
```



Fixing the `typeof` operator 2/3

What is much more annoying is that the **`typeof`** operator cannot tell the difference between an object and a **`null`** value. So some improvement is needed here.

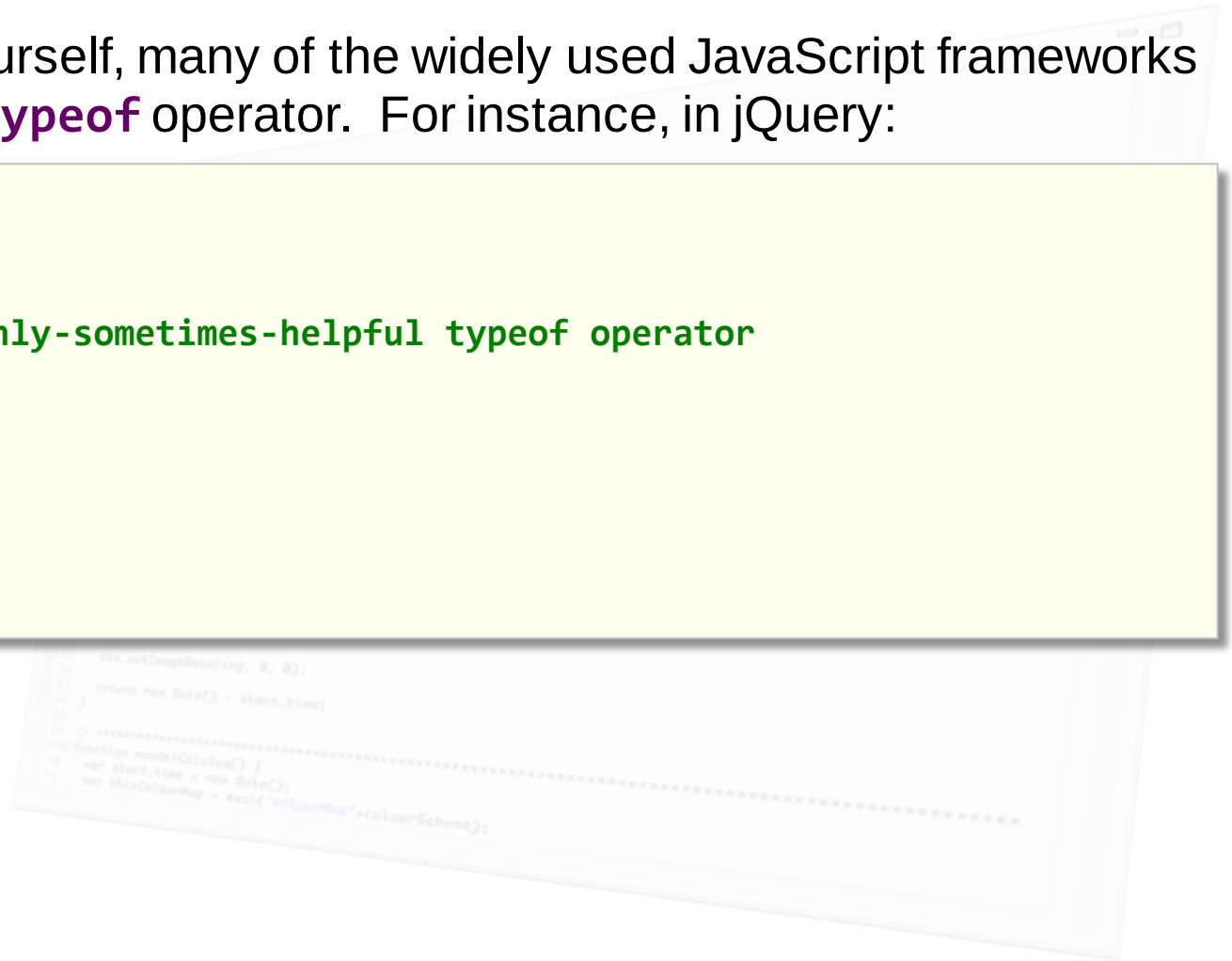
```
// This improvement works by exploiting the fact that all objects are truthy, but null is falsy.
// The first expression in the condition coerces myObj to Boolean; which, if it is null, will be
// false. Because this is an AND condition, the whole condition fails if the first operand is false
if (myObj && typeof myObj === 'object') {
  // Yup, this is an object and not null value
};
```



Fixing the `typeof` operator 3/3

Rather than having to implement these fixes yourself, many of the widely used JavaScript frameworks provide their own replacement function for the **`typeof`** operator. For instance, in jQuery:

```
var anArray   = [1,2,3,4,5];
var nullValue = null;

// jQuery provides a robust fix for JavaScript's only-sometimes-helpful typeof operator
jQuery.type(anArray);        // → 'array'
jQuery.type(nullValue);      // → 'null'

// Or simply
jQuery.isArray(anArray);     // → true
```



Using Operators

Operators: Basic Assignment

Any value can be assigned to any variable using the assignment operator "="

```
var char1 = "a";    // The value "a" is assigned to the local variable char1
var char2 = "b";    // The value "b" is assigned to the local variable char2

// An object containing the single property "propertyName" is assigned to the variable myObj
var myObj = {
      propertyName: "Some value"
    };

// A regular expression is assigned to the variable parseURL
var parseURL =
  /^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([^?#]*))?(?:\?([^#]*))?(?:#(.*))?$/;
```

Operators: Shortcut Assignments

JavaScript has various shortcut operators designed to make operations more compact.

```
x += y;    // Add y to x and store the result in x:        x = x + y
x -= y;    // Subtract y from x and store the result in x: x = x - y
x *= y;    // Multiply x by y and store the result in x:   x = x * y
x /= y;    // Divide x by y and store the result in x:     x = x / y

x %= y;    // Remainder of x divided by y is stored in x:  x = x % y

x <<= y;   // Bitwise left shift:                  x = x << y
x >>= y;   // Bitwise right shift (preserve sign): x = x >> y
x >>>= y;  // Bitwise right shift (ignore sign):   x = x >>> y

x &= y;    // Bitwise AND: x = x & y
x |= y;    // Bitwise OR:  x = x | y
x ^= y;    // Bitwise XOR: x = x ^ y
```

Operators: Basic Comparison

Any two values can be compared:

```
// >   Greater than
// >=  Greater than or equal to
// <   Less than
// <=  Less than or equal to
// !=  Not equal to

3.14 <= 2.717;    // → False
"b" > "a";        // → True   Because "b" has a larger ASCII value than "a"
```

Operators: Don't Confuse Assignment and Comparison

Do not confuse the assignment operator with the comparison operator!

```
var char1 = "a";   // Declare a variable called 'char1' and assign to it the value "a"
var char2 = "b";   // Declare a variable called 'char2' and assign to it the value "b"

// Beginner error...
if (char1 = char2) {
  // OOPS!
  // The above condition actually evaluates to true because:
  // 1) The value of char2 is assigned to char1, so char1 now equals "b"
  // 2) Since "b" is a non-empty string, it is truthy, so the value of char1 is coerced to true
  // 3) Therefore the condition always passes
}
```

Operators: Comparison With Type Coercion: ==

Any test for (in)equality must first check whether the data types of the operands are the same.
If they are not, then type coercion is performed:

```
// "=="  If data types are not equal, first perform type coercion, then test for equality
// "!="  If data types are not equal, first perform type coercion, then test for inequality

"1" == 1;       // → True  Numeric 1 is converted to string "1". Strings are equal
"2" != 1 + 2;   // → True  Perform addition → 1 + 2 = 3, convert result to string "3". Strings are not equal

2 == "1" + "1"; // False - Convert numeric 2 to string "2", concatenate "1" + "1" → "11". Strings are not equal

// Comparison operators
// ">" Greater than    ">=" Greater than or equal to
// "<" Less than       "<=" less than or equal to

"a" > "b";  // → False  "a" comes before "b" in the ASCII collation sequence
"4" < 5;    // → True   Convert numeric 5 to string "5", "4" comes before "5" in the ASCII collation sequence
```

Operators: Comparison Without Type Coercion: ===

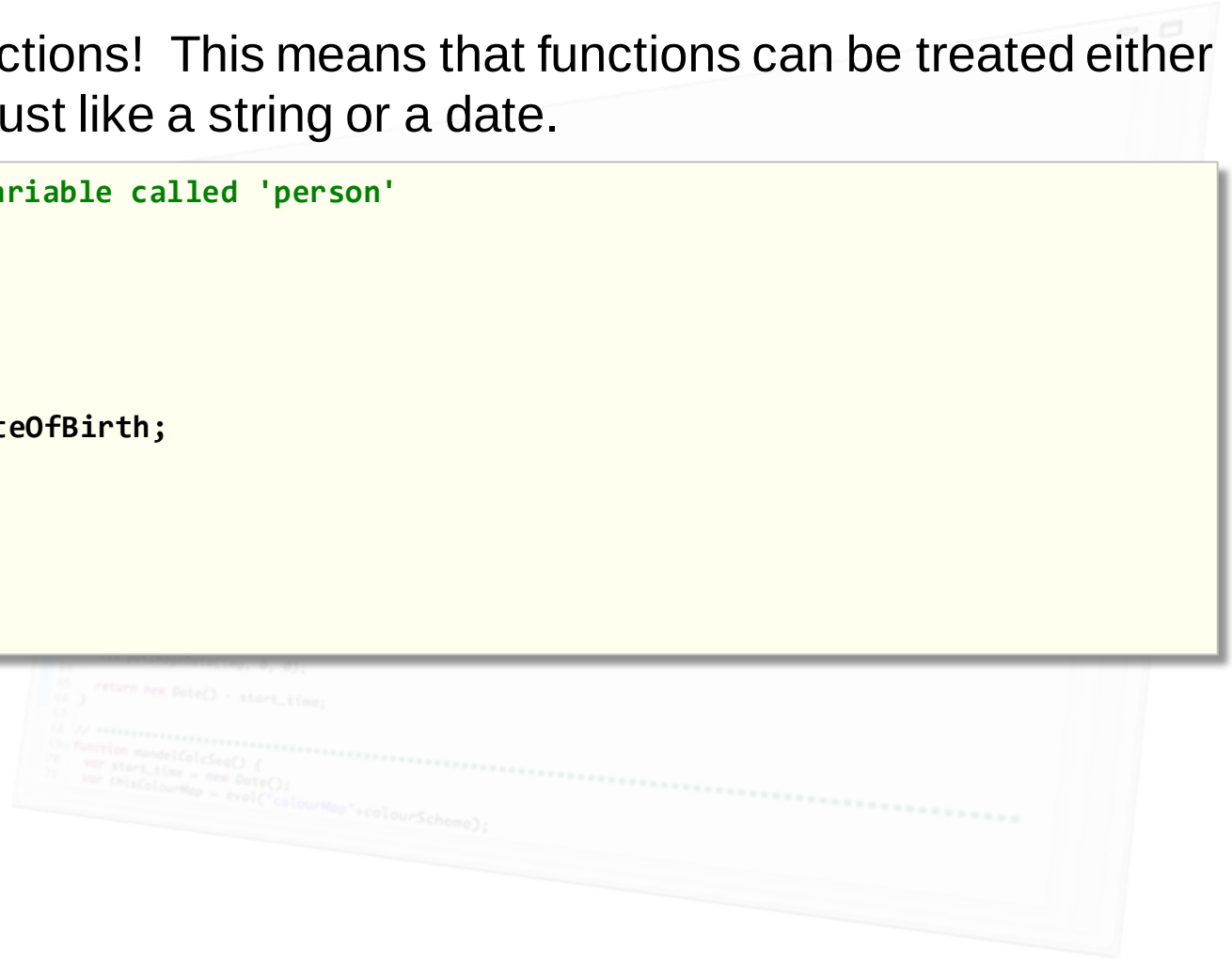
If your comparison is required to check the equality not only of the values, but also the data types, then you must use a different comparison operator:

```
// "==="  Test for exact equality. First test data types for equality, then test values for equality
// "!=="  Test for inequality.     First test data types for equality, then test values for inequality

"1" === 1;               // → False  Fails immediately because string and number are different data types

1 + 1 === "2";           // → False  1+1 = numeric 2. Number and string do not match
1 + 1 === parseInt("2"); // → True   1+1 = numeric 2. parseInt("2") → numeric 2. Both data type & value are equal

// Be careful with the equality operators and type coercion!
1 + 1 !== "2";           // → True   Because number and string are different data types
1 + 1 != "2";            // → False  Because number is coerced to string. Strings are equal
```

Operators: Invocation

Everything in JavaScript is an object – even functions! This means that functions can be treated either as executable units of code, or as data objects just like a string or a date.

```
// Define an anonymous function object and store it in a variable called 'person'
var person = function() {
  var firstName = "Harry";
  var lastName  = "Hawk";
  var dateOfBirth = "03 Aug 1976";

  return firstName + " " + lastName + " was born on " + dateOfBirth;
}
```



Operators: Invocation

Everything in JavaScript is an object – even functions! This means that functions can be treated either as executable units of code, or as data objects just like a string or a date.

```
// Define an anonymous function object and store it in a variable called 'person'
var person = function() {
  var firstName = "Harry";
  var lastName  = "Hawk";
  var dateOfBirth = "03 Aug 1976";

  return firstName + " " + lastName + " was born on " + dateOfBirth;
}
```

We can distinguish between a "*function as an executable unit of code* " and a "*function as a data object*" by the use of the invocation operator ( ). This operator can take zero or more parameters inside the parentheses.

Operators: Invocation

Everything in JavaScript is an object – even functions! This means that functions can be treated either as executable units of code, or as data objects just like a string or a date.

```
// Define an anonymous function object and store it in a variable called 'person'
var person = function() {
  var firstName = "Harry";
  var lastName  = "Hawk";
  var dateOfBirth = "03 Aug 1976";

  return firstName + " " + lastName + " was born on " + dateOfBirth;
}

person;      // → function()  This is just a reference to the object called person that happens to be a function
```

We can distinguish between a "*function as an executable unit of code* " and a "*function as a data object*" by the use of the invocation operator ( ). This operator can take zero or more parameters inside the parentheses.

Without the invocation operator, a function reference is simply that – a reference to an object that happens to have an executable part.

Operators: Invocation

Everything in JavaScript is an object – even functions! This means that functions can be treated either as executable units of code, or as data objects just like a string or a date.

```
// Define an anonymous function object and store it in a variable called 'person'
var person = function() {
  var firstName = "Harry";
  var lastName  = "Hawk";
  var dateOfBirth = "03 Aug 1976";

  return firstName + " " + lastName + " was born on " + dateOfBirth;
}

person;      // → function()  This is just a reference to the object called person that happens to be a function
person();    // → "Harry Hawk was born on 03 Aug 1976"    Now the function is invoked
```

We can distinguish between a "*function as an executable unit of code* " and a "*function as a data object*" by the use of the invocation operator ( ). This operator can take zero or more parameters inside the parentheses.

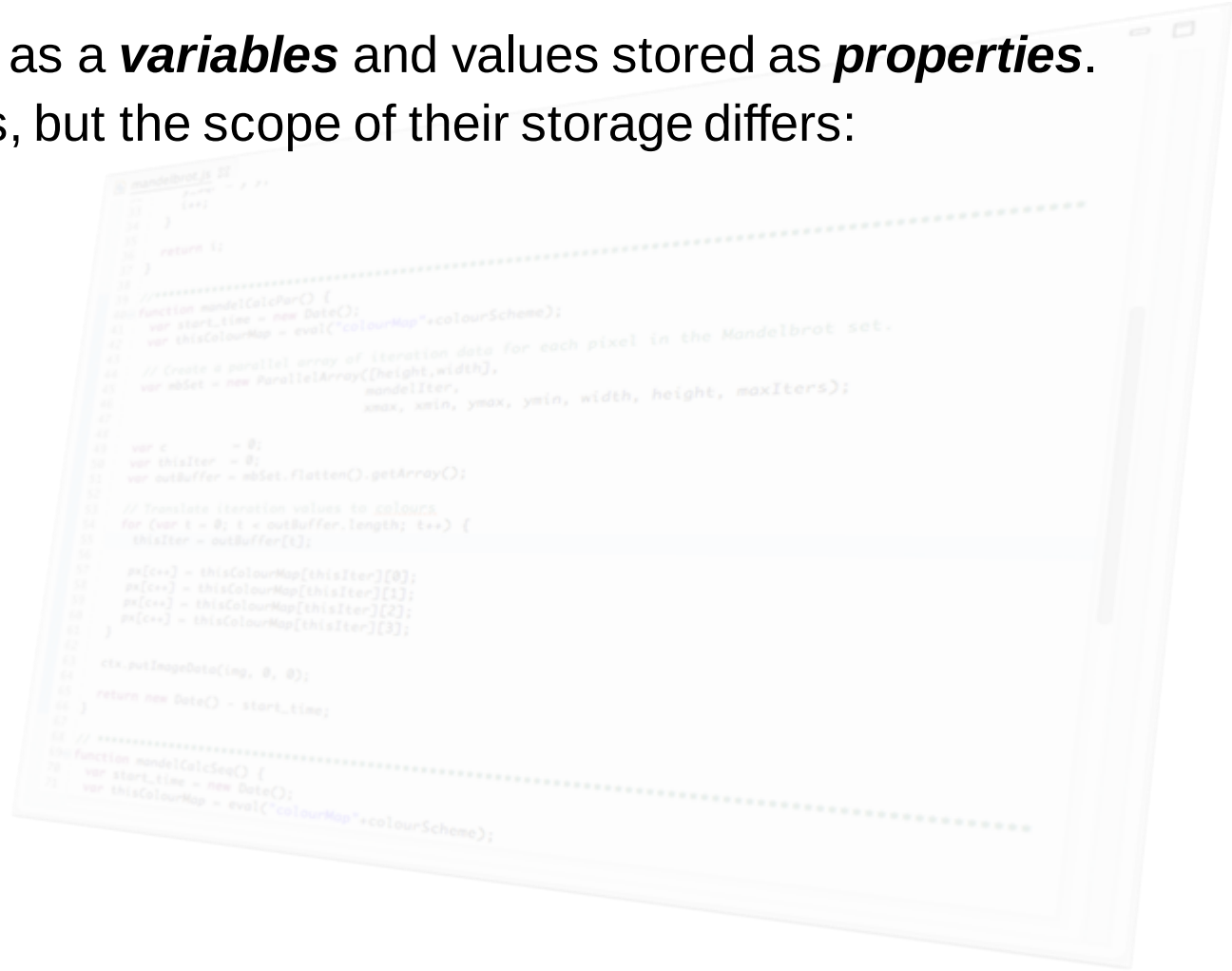
Without the invocation operator, a function reference is simply that – a reference to an object that happens to have an executable part.

Conversely, if the invocation operator is used, the function is treated as an executable unit of code and invoked.

Declarations & JavaScript Objects

Declarations: Variables and Properties

JavaScript distinguishes between values stored as a ***variables*** and values stored as ***properties***. Both properties and variables are named values, but the scope of their storage differs:



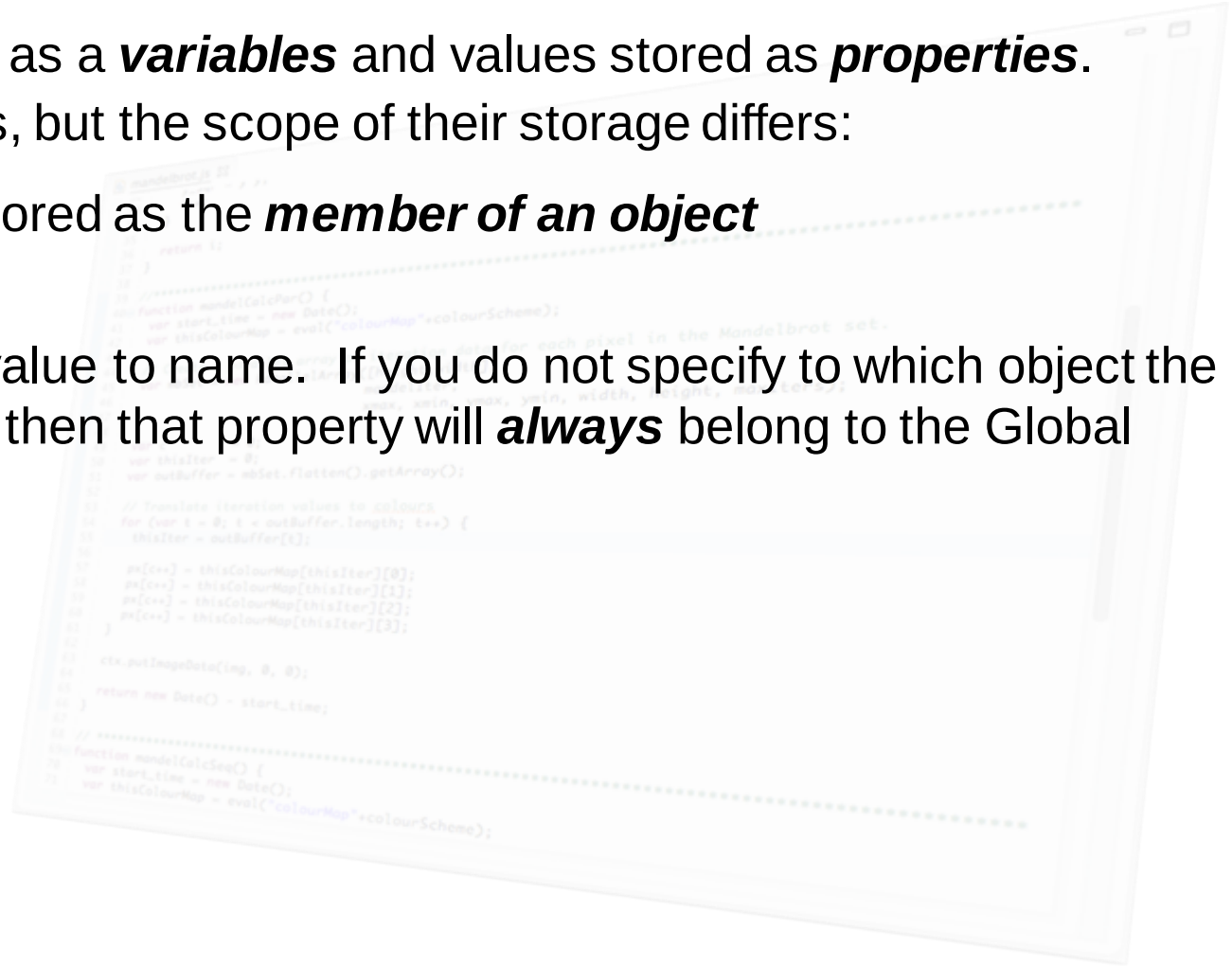
Declarations: Variables and Properties

JavaScript distinguishes between values stored as a ***variables*** and values stored as ***properties***.
Both properties and variables are named values, but the scope of their storage differs:

- A property is a named value that is always stored as the ***member of an object***

```
foobar = "Phluff 'n' stuff";
```

  A property is created simply by assigning a value to name. If you do not specify to which object the property belongs (as in the above example), then that property will ***always*** belong to the Global Object.



Declarations: Variables and Properties

JavaScript distinguishes between values stored as a ***variables*** and values stored as ***properties***.
Both properties and variables are named values, but the scope of their storage differs:

- A property is a named value that is always stored as the ***member of an object***

```
foobar = "Phluff 'n' stuff";
```

  A property is created simply by assigning a value to name. If you do not specify to which object the property belongs (as in the above example), then that property will ***always*** belong to the Global Object.

- A variable is a named value stored within an ***execution context***

```
var barfoo = "Chicken soup";
```

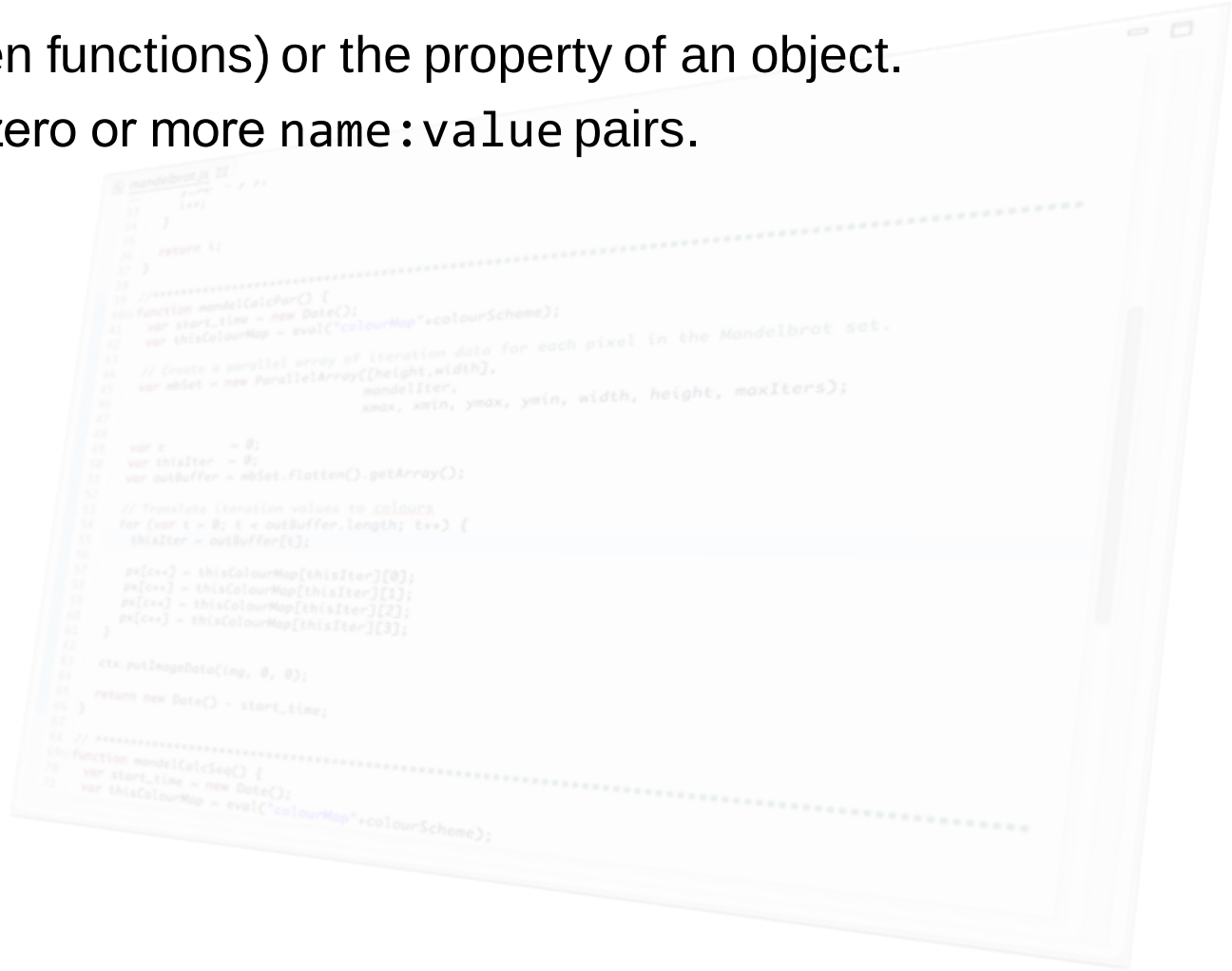
  A variable is created by use of the `var` keyword in front of the assignment.

  More details will be given later about exactly what an “execution context” is.

Declarations: Objects

In JavaScript, everything is either an object (even functions) or the property of an object.

An ‘object’ is simply an unordered collection of zero or more `name:value` pairs.



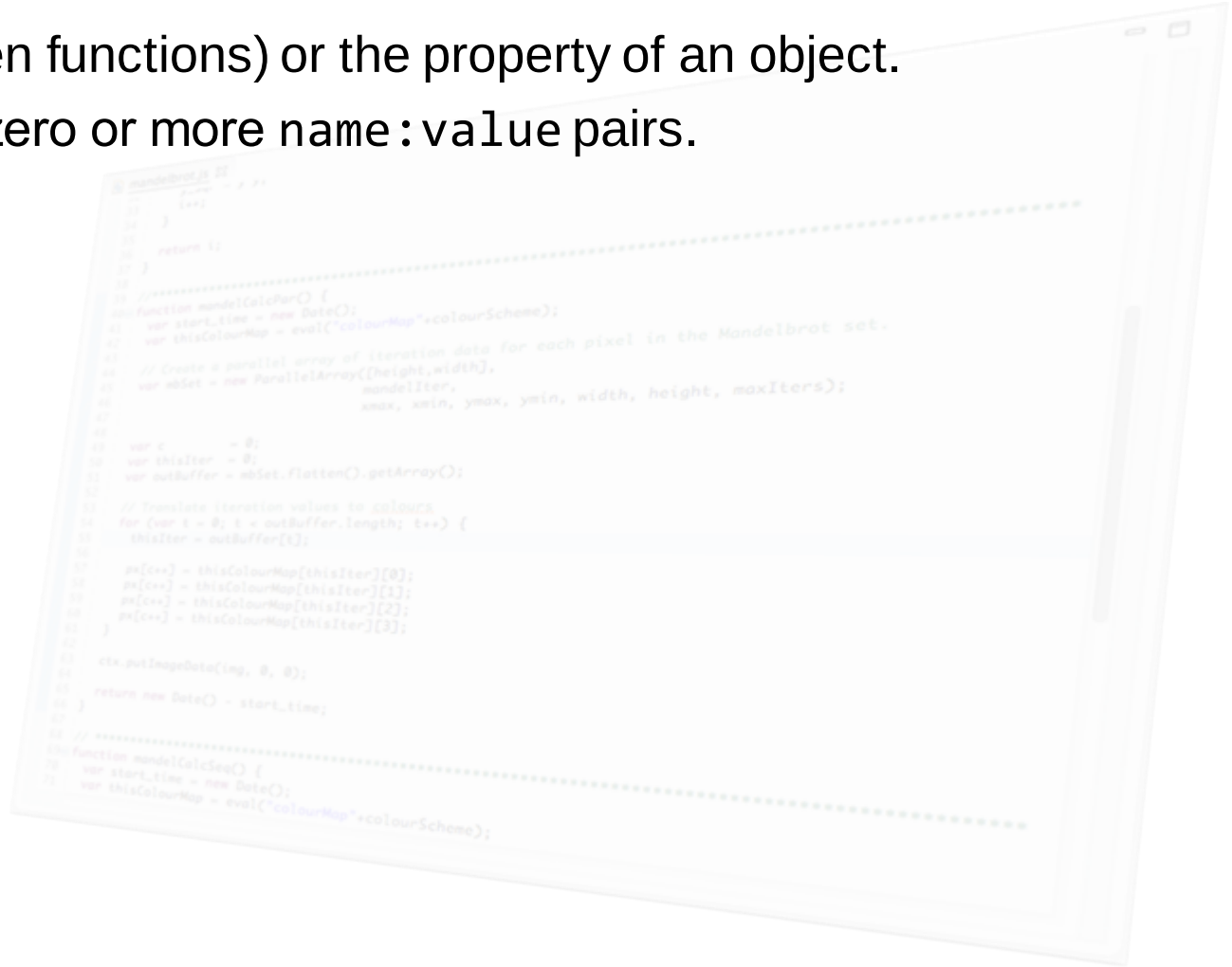
Declarations: Objects

In JavaScript, everything is either an object (even functions) or the property of an object.

An ‘object’ is simply an unordered collection of zero or more `name:value` pairs.

The simplest way to create an object is first to define an empty object, then assign it some arbitrary property values:

```
// Create an empty object
var person1 = {};

person1.firstName = "Harry";
person1.lastName  = "Hawk";
person1.hobbies   = ["swimming","cycling"];

person1.listHobbies = function() {
  return this.hobbies.join(" ");
}
```



Declarations: Objects

In JavaScript, everything is either an object (even functions) or the property of an object.

An 'object' is simply an unordered collection of zero or more `name:value` pairs.

The simplest way to create an object is first to define an empty object, then assign it some arbitrary property values:

```
// Create an empty object
var person1 = {};

person1.firstName = "Harry";
person1.lastName  = "Hawk";
person1.hobbies   = ["swimming","cycling"];

person1.listHobbies = function() {
  return this.hobbies.join(" ");
}
```

Alternatively, you could use the array syntax:

```
// Create an empty object
var person2 = {};

person2["firstName"] = "Harry";
person2["Last-Name"] = "Hawk";
person2["hobbies"]   = ["swimming","cycling"];

person2["listHobbies"] = function() {
  return this.hobbies.join(" ");
}
```

Declarations: Objects

In JavaScript, everything is either an object (even functions) or the property of an object.

An 'object' is simply an unordered collection of zero or more `name:value` pairs.

The simplest way to create an object is first to define an empty object, then assign it some arbitrary property values:

```
// Create an empty object
var person1 = {};

person1.firstName = "Harry";
person1.lastName  = "Hawk";
person1.hobbies   = ["swimming","cycling"];

person1.listHobbies = function() {
  return this.hobbies.join(" ");
}
```

Alternatively, you could use the array syntax:

```
// Create an empty object
var person2 = {};

person2["firstName"] = "Harry";
person2["Last-Name"] = "Hawk";
person2["hobbies"]   = ["swimming","cycling"];

person2["listHobbies"] = function() {
  return this.hobbies.join(" ");
}
```

The array syntax must be used either if you want to create an object property containing a character not permitted in a variable name, or if the property name is a reserved word.

Two Ways To Access Object Properties

The properties of a JavaScript object are accessed by specifying the object name and then using either the refinement operator "**.**" or the array element syntax. All object properties are ***public***!

```
// The refinement operator: place a dot "." between the object name and the property name
person1.firstName;        // → "Harry"
person1.hobbies[1];       // → "cycling"
```

Two Ways To Access Object Properties

The properties of a JavaScript object are accessed by specifying the object name and then using either the refinement operator "**.**" or the array element syntax. All object properties are ***public***!

```
// The refinement operator: place a dot "." between the object name and the property name
person1.firstName;        // → "Harry"
person1.hobbies[1];       // → "cycling"

// The refinement operator is useful if the property name is both known at design time and
// does not contain any illegal characters such as "-" or ";". However, if either of these
// conditions are not met, then object properties can be accessed as array elements.
person2["Last-Name"];     // → "Hawk"
person2["hobbies"][0];    // → "swimming"
```

Two Ways To Access Object Properties

The properties of a JavaScript object are accessed by specifying the object name and then using either the refinement operator "**.**" or the array element syntax. All object properties are ***public***!

```
// The refinement operator: place a dot "." between the object name and the property name
person1.firstName;        // → "Harry"
person1.hobbies[1];       // → "cycling"

// The refinement operator is useful if the property name is both known at design time and
// does not contain any illegal characters such as "-" or ";". However, if either of these
// conditions are not met, then object properties can be accessed as array elements.
person2["Last-Name"];     // → "Hawk"
person2["hobbies"][0];    // → "swimming"

// Both forms of property access can be chained together
person1.propertyObject.someProperty;
person2["propertyObject"]["someProperty"];
```

Two Ways To Access Object Properties

The properties of a JavaScript object are accessed by specifying the object name and then using either the refinement operator "**.**" or the array element syntax. All object properties are ***public***!

```
// The refinement operator: place a dot "." between the object name and the property name
person1.firstName;        // → "Harry"
person1.hobbies[1];       // → "cycling"

// The refinement operator is useful if the property name is both known at design time and
// does not contain any illegal characters such as "-" or ";". However, if either of these
// conditions are not met, then object properties can be accessed as array elements.
person2["Last-Name"];     // → "Hawk"
person2["hobbies"][0];    // → "swimming"

// Both forms of property access can be chained together
person1.propertyObject.someProperty;
person2["propertyObject"]["someProperty"];

// Any object property that is of type 'function' is known as a method and can be invoked
person1.listHobbies();    // → "swimming cycling"
```

Deleting Object Properties

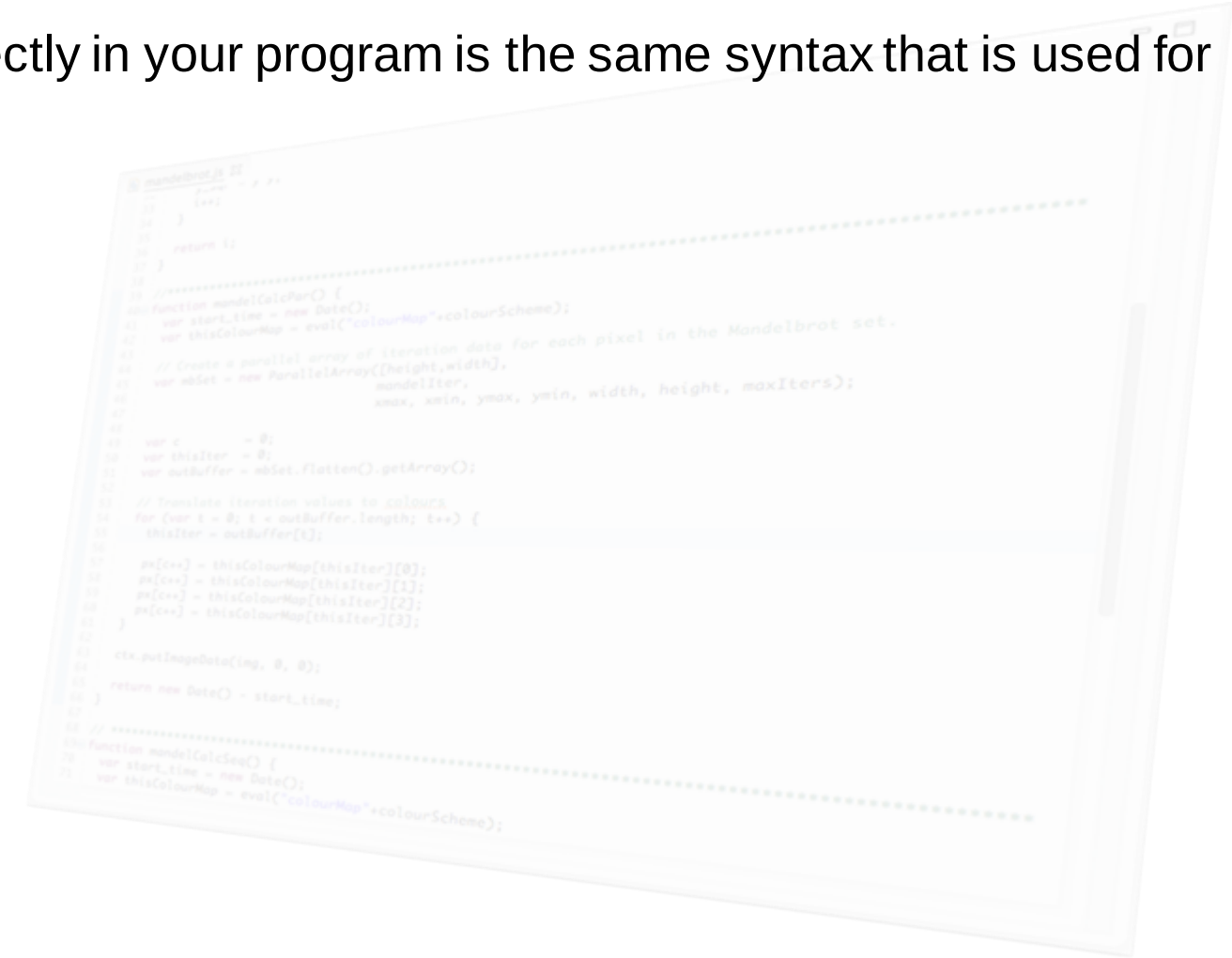
The properties of a JavaScript object can be deleted using the **`delete`** keyword.

```
// Property deletion
var aGlobalVariable = "I'm a global variable"; // Global variable, but not a property of the global object
aGlobalProperty     = "I'm a global property"; // A property belonging to the global object

delete aGlobalVariable;    // → false. Delete can only operate on properties, not variables
delete aGlobalProperty;    // → true. The property is deleted because it belongs to the global object

aGlobalVariable;           // → "I'm a global variable"
aGlobalProperty;           // → undefined
```

JavaScript Object Notation (JSON)

The syntax for creating a JavaScript object directly in your program is the same syntax that is used for serialising a JavaScript object as a text string:

```
// Create an inline JavaScript object
var person = {
  firstName : "Harry",
  lastName  : "Hawk",
  dateOfBirth : "1976 Aug 03"
}
```



JavaScript Object Notation (JSON)

The syntax for creating a JavaScript object directly in your program is the same syntax that is used for serialising a JavaScript object as a text string:

```
// Create an inline JavaScript object
var person = {
  firstName : "Harry",
  lastName  : "Hawk",
  dateOfBirth : "1976 Aug 03"
}
```

```
// The same object serialised in JSON
{ firstName : "Harry",
  lastName  : "Hawk",
  dateOfBirth : "1976 Aug 03" }
```

JavaScript Object Notation (JSON)

The syntax for creating a JavaScript object directly in your program is the same syntax that is used for serialising a JavaScript object as a text string:

```
// Create an inline JavaScript object
var person = {
  firstName : "Harry",
  lastName  : "Hawk",
  dateOfBirth : "1976 Aug 03"
}
```

```
// The same object serialised in JSON
{ firstName : "Harry",
  lastName  : "Hawk",
  dateOfBirth : "1976 Aug 03" }
```

The convenience of this symmetry makes JSON the ideal output format for data from a web server.

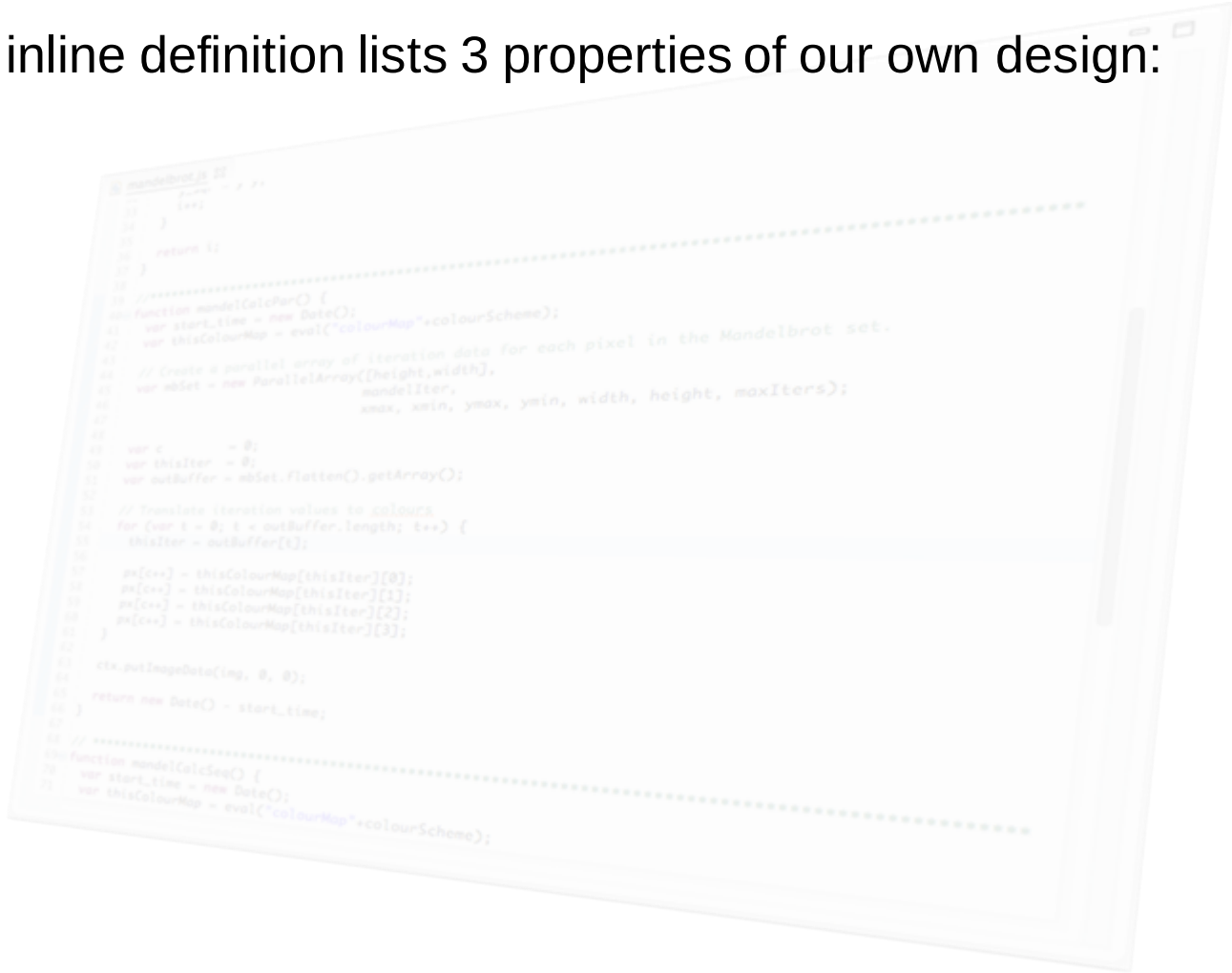
Once assigned to a JavaScript variable, a JSON string is parsed automatically making the internal structure of the object accessible to the rest of the program.

```
// Replacing the inline object definition with a function call that obtains a JSON response from a web
// server does not alter the above functionality unchanged
var person = getJSONFromBackend(someKeyValue);
```

Default Object Methods and Properties

When a simple JavaScript object is created, the inline definition lists 3 properties of our own design: `firstName`, `lastName` and `dateOfBirth`.

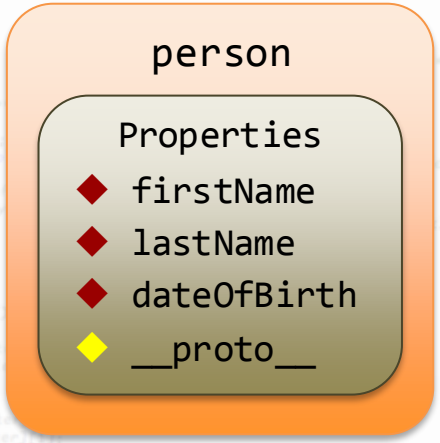
```
// Create an inline JavaScript object
var person = {
  firstName : "Harry",
  lastName  : "Hawk",
  dateOfBirth : "1976 Aug 03"
}
```



Default Object Methods and Properties

When a simple JavaScript object is created, the inline definition lists 3 properties of our own design: `firstName`, `lastName` and `dateOfBirth`.

```
// Create an inline JavaScript object
var person = {
  firstName : "Harry",
  lastName  : "Hawk",
  dateOfBirth : "1976 Aug 03"
}
```



However, the `person` object also has a default property called `__proto__` that cannot be deleted.

The `__proto__` property is not actually part of the ECMAScript specification, so in that sense, it is a non-standard property. However, all modern browsers (except Internet Explorer) implement this property.

More details will be given about this property later.

Default Object Methods and Properties 1/2

Using the NodeJS command prompt, a simple inline JavaScript object is created containing 3 properties: `firstName`, `lastName` and `dateOfBirth`.

```
> var person = {
...   firstName : "Harry",
...   lastName  : "Hawk",
...   dateOfBirth : "1976 Aug 03"
... }
undefined
>
```

Default Object Methods and Properties 2/2

If we type the object name followed by a dot character and then press tab, we are shown a list of all the properties available in this object. Notice that there are several default methods available.

```
> var person = {
...   firstName : "Harry",
...   lastName  : "Hawk",
...   dateOfBirth : "1976 Aug 03"
... }
undefined
> person.
person.__defineGetter__      person.__defineSetter__      person.__lookupGetter__      person.__lookupSetter__      person.constructor
person.hasOwnProperty        person.isPrototypeOf         person.propertyIsEnumerable  person.toLocaleString        person.toString
person.valueOf

person.dateOfBirth           person.firstName             person.lastName
```

Default Object Methods and Properties 2/2

If we type the object name followed by a dot character and then press tab, we are shown a list of all the properties available in this object. Notice that there are several default methods available.

```
> var person = {
...   firstName : "Harry",
...   lastName  : "Hawk",
...   dateOfBirth : "1976 Aug 03"
... }
undefined
> person.
person.__defineGetter__      person.__defineSetter__      person.__lookupGetter__      person.__lookupSetter__      person.constructor
person.hasOwnProperty        person.isPrototypeOf         person.propertyIsEnumerable  person.toLocaleString        person.toString
person.valueOf

person.dateOfBirth           person.firstName             person.lastName
```

Methods inherited from the JavaScript object called Object

Default Object Methods and Properties 2/2

If we type the object name followed by a dot character and then press tab, we are shown a list of all the properties available in this object. Notice that there are several default methods available.

```
> var person = {
...   firstName : "Harry",
...   lastName  : "Hawk",
...   dateOfBirth : "1976 Aug 03"
... }
undefined
> person.__proto__.
person.__proto__.__defineGetter__      person.__proto__.__defineSetter__      person.__proto__.__lookupGetter__
person.__proto__.__lookupSetter__      person.__proto__.constructor           person.__proto__.hasOwnProperty
person.__proto__.isPrototypeOf         person.__proto__.propertyIsEnumerable  person.__proto__.toLocaleString
person.__proto__.toString              person.__proto__.valueOf
```

Even though the `__proto__` property is not explicitly listed, if you enter its name followed by a dot and then press tab, you will see the methods and properties inherited from the object acting as `person`'s prototype.

More about prototypes later…

JavaScript Arrays

Arrays 1/4

Remember! Everything in JavaScript is an object; therefore, a JavaScript array is simply a regular object that has extra array-like functionality built in.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];

// Access the elements in the array
listOfThings[1];        // → "cup"
listOfThings.length;    // → 4
listOfThings[4];        // → undefined  All array indices are zero based, not one based
```

Arrays 1/4

Remember! Everything in JavaScript is an object; therefore, a JavaScript array is simply a regular object that has extra array-like functionality built in.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];

// Access the elements in the array
listOfThings[1];        // → "cup"
listOfThings.length;    // → 4
listOfThings[4];        // → undefined  All array indices are zero based, not one based

// JavaScript arrays are not typesafe, their elements can be of a mixture of datatypes
var differentThings = ["Chicken Soup", 12.34, null, ["a","b","c","d"],
                       undefined, NaN, {prop: 123}, true, false];
```

Arrays 1/4

Remember! Everything in JavaScript is an object; therefore, a JavaScript array is simply a regular object that has extra array-like functionality built in.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];

// Access the elements in the array
listOfThings[1];        // → "cup"
listOfThings.length;    // → 4
listOfThings[4];        // → undefined  All array indices are zero based, not one based

// JavaScript arrays are not typesafe, their elements can be of a mixture of datatypes
var differentThings = ["Chicken Soup", 12.34, null, ["a","b","c","d"],
                       undefined, NaN, {prop: 123}, true, false];
```

IMPORTANT

When you declare an Array, you are creating a ***regular JavaScript object*** that contains an extra set methods such as `.push()` and `.pop()`, and a `.length` property.

These methods give this JavaScript object ***array-like*** behaviour.

Arrays 2/4

Using the NodeJS command prompt, a new array is created called `myArray`. As before, type the object name followed by a dot and press the tab key to display the possible methods and properties…

```
> var listOfThings = ["ball","cup","pen","car"];
undefined
> listOfThings.
listOfThings.__defineGetter__     listOfThings.__defineSetter__     listOfThings.__lookupGetter__     listOfThings.__lookupSetter__
listOfThings.constructor          listOfThings.hasOwnProperty       listOfThings.isPrototypeOf        listOfThings.propertyIsEnumerable
listOfThings.toLocaleString       listOfThings.toString             listOfThings.valueOf

listOfThings.concat               listOfThings.every                listOfThings.filter               listOfThings.forEach
listOfThings.indexOf              listOfThings.join                 listOfThings.lastIndexOf          listOfThings.length
listOfThings.map                  listOfThings.pop                  listOfThings.push                 listOfThings.reduce
listOfThings.reduceRight          listOfThings.reverse              listOfThings.shift                listOfThings.slice
listOfThings.some                 listOfThings.sort                 listOfThings.splice               listOfThings.unshift

listOfThings.0                                                      listOfThings.2                    listOfThings.3
```

Methods inherited from the standard JavaScript object called Object

Arrays 2/4

Using the NodeJS command prompt, a new array is created called `myArray`. As before, type the object name followed by a dot and press the tab key to display the possible methods and properties…

```
> var listOfThings = ["ball","cup","pen","car"];
undefined
> listOfThings.
listOfThings.__defineGetter__     listOfThings.__defineSetter__     listOfThings.__lookupGetter__     listOfThings.__lookupSetter__
listOfThings.constructor          listOfThings.hasOwnProperty       listOfThings.isPrototypeOf        listOfThings.propertyIsEnumerable
listOfThings.toLocaleString       listOfThings.toString             listOfThings.valueOf

listOfThings.concat               listOfThings.every                listOfThings.filter               listOfThings.forEach
listOfThings.indexOf              listOfThings.join                 listOfThings.lastIndexOf          listOfThings.length
listOfThings.map                  listOfThings.pop                  listOfThings.push                 listOfThings.reduce
listOfThings.reduceRight          listOfThings.reverse              listOfThings.shift                listOfThings.slice
listOfThings.some                 listOfThings.sort                 listOfThings.splice               listOfThings.unshift

listOfThings.0                    listOfThings.1                    listOfThings.2                    listOfThings.3
```

Extra built-in methods that give this object array-like capabilities

Arrays 2/4

Using the NodeJS command prompt, a new array is created called `myArray`. As before, type the object name followed by a dot and press the tab key to display the possible methods and properties…

```
> var listOfThings = ["ball","cup","pen","car"];
undefined
> listOfThings.
listOfThings.__defineGetter__     listOfThings.__defineSetter__     listOfThings.__lookupGetter__     listOfThings.__lookupSetter__
listOfThings.constructor          listOfThings.hasOwnProperty       listOfThings.isPrototypeOf        listOfThings.propertyIsEnumerable
listOfThings.toLocaleString       listOfThings.toString             listOfThings.valueOf

listOfThings.concat               listOfThings.every                listOfThings.filter               listOfThings.forEach
listOfThings.indexOf              listOfThings.join                 listOfThings.lastIndexOf          listOfThings.length
listOfThings.map                  listOfThings.pop                  listOfThings.push                 listOfThings.reduce
listOfThings.reduceRight          listOfThings.reverse              listOfThings.shift                listOfThings.slice
listOfThings.some                 listOfThings.sort                 listOfThings.splice               listOfThings.unshift

listOfThings.0                    listOfThings.1                    listOfThings.2                    listOfThings.3
```

Automatically created object properties.

Each array element is created as a property whose name is the text string of the corresponding index number

Arrays 3/4

All array elements are treated as a special type of object property. JavaScript uses the string representation of the numeric index as the property name.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];

// Array elements can be accessed either by their numeric index, or by the string
// representation of the numeric index.
listOfThings[1];        // → "cup"
```

Arrays 3/4

All array elements are treated as a special type of object property. JavaScript uses the string representation of the numeric index as the property name.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];

// Array elements can be accessed either by their numeric index, or by the string
// representation of the numeric index.
listOfThings[1];        // → "cup"
listOfThings["1"];      // → "cup"
```

Arrays 3/4

All array elements are treated as a special type of object property. JavaScript uses the string representation of the numeric index as the property name.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];

// Array elements can be accessed either by their numeric index, or by the string
// representation of the numeric index.
listOfThings[1];        // → "cup"
listOfThings["1"];      // → "cup"

// Create a new element using a numeric string as the index
listOfThings["4"] = "tree";
```

Arrays 3/4

All array elements are treated as a special type of object property. JavaScript uses the string representation of the numeric index as the property name.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];

// Array elements can be accessed either by their numeric index, or by the string
// representation of the numeric index.
listOfThings[1];        // → "cup"
listOfThings["1"];      // → "cup"

// Create a new element using a numeric string as the index
listOfThings["4"] = "tree";
listOfThings[4];        // → "tree"
listOfThings;           // → ["ball","cup","pen","car","tree"]
```

Arrays 3/4

All array elements are treated as a special type of object property. JavaScript uses the string representation of the numeric index as the property name.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];

// Array elements can be accessed either by their numeric index, or by the string
// representation of the numeric index.
listOfThings[1];        // → "cup"
listOfThings["1"];      // → "cup"

// Create a new element using a numeric string as the index
listOfThings["4"] = "tree";
listOfThings[4];        // → "tree"
listOfThings;           // → ["ball","cup","pen","car","tree"]

// Alternatively, a new element can be appended to the array using the push() method
listOfThings.push("dog");
listOfThings;           // → ["ball","cup","pen","car","tree","dog"]
```

Arrays 4/4

Since JavaScript arrays are just objects with array-like behaviour, you can add new ***properties*** to the array object using any names you like.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];
listOfThings.length;    // → 4

// Create an array element using a non-numeric property name
listOfThings["first"] = "Some value";
listOfThings["first"];  // → "Some value"
```

Arrays 4/4

Since JavaScript arrays are just objects with array-like behaviour, you can add new ***properties*** to the array object using any names you like.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];
listOfThings.length;    // → 4

// Create an array element using a non-numeric property name
listOfThings["first"] = "Some value";
listOfThings["first"];  // → "Some value"

listOfThings.length;    // → 4  Uh?! Didn't we just add a new element to the array?
listOfThings;           // → ["ball","cup","pen","car"] No, we added an object property,
                        //                              not an array element...
```

Arrays 4/4

Since JavaScript arrays are just objects with array-like behaviour, you can add new ***properties*** to the array object using any names you like.

```
// Create an array variable.
var listOfThings = ["ball","cup","pen","car"];
listOfThings.length;    // → 4

// Create an array element using a non-numeric property name
listOfThings["first"] = "Some value";
listOfThings["first"];  // → "Some value"

listOfThings.length;    // → 4  Uh?! Didn't we just add a new element to the array?
listOfThings;           // → ["ball","cup","pen","car"] No, we added an object property,
                        //                              not an array element...
```

Be careful!

If the property name cannot be interpreted as an integer, then that new property will ***not*** be treated as an "element" of the array! It will simply be an object property with no special relevance to the set of elements making up the "array".

Array `.length` Property (and What it Doesn't Mean!)

The value of an array's `.length` property doesn't always return the value you might expect.
It ***does not necessarily*** represent the number of elements that actually exist in the array!

```
// Create an empty array
var someArray = [];

someArray.length;  // → 0. Everything is behaving as expected...

// Add a new element
someArray[4] = "Surprise!";
```

Array `.length` Property (and What it Doesn't Mean!)

The value of an array's `.length` property doesn't always return the value you might expect.
It ***does not necessarily*** represent the number of elements that actually exist in the array!

```
// Create an empty array
var someArray = [];

someArray.length;  // → 0. Everything is behaving as expected...

// Add a new element
someArray[4] = "Surprise!";
someArray.length;  // → 5  Uh!?
```

Array `.length` Property (and What it Doesn't Mean!)

The value of an array's `.length` property doesn't always return the value you might expect.
It ***does not necessarily*** represent the number of elements that actually exist in the array!

```
// Create an empty array
var someArray = [];

someArray.length;  // → 0. Everything is behaving as expected...

// Add a new element
someArray[4] = "Surprise!";
someArray.length;  // → 5  Uh!?

someArray;         // →    [undefined, undefined, undefined, undefined, "Surprise!"]
```

Array `.length` Property (and What it Doesn't Mean!)

The value of an array's `.length` property doesn't always return the value you might expect.
It ***does not necessarily*** represent the number of elements that actually exist in the array!

```
// Create an empty array
var someArray = [];

someArray.length;  // → 0. Everything is behaving as expected...

// Add a new element
someArray[4] = "Surprise!";
someArray.length;  // → 5  Uh!?

someArray;         // →    [undefined, undefined, undefined, undefined, "Surprise!"]
```

The `.length` property will always return a value 1 higher than the value of the current highest index – irrespective of whether any of the intervening elements exist or not!

Regular Expressions

A Regular Expression is a tool for defining a pattern of text and then locating instances of that pattern within a larger string.

```
// Create a regular expression for identifying RGB colour triples in the form of either
// "#RRGGBB" or "#RGB"
var regEx = /#([a-f0-9]{6}|[a-f0-9]{3})/i;

var colour1 = "The background colour is #3300ff";
var colour2 = "The foreground colour is sky blue pink";

var results1 = regEx.exec(colour1);  // → ["#3300ff", "3300ff"]
```

Regular Expressions

A Regular Expression is a tool for defining a pattern of text and then locating instances of that pattern within a larger string.

```
// Create a regular expression for identifying RGB colour triples in the form of either
// "#RRGGBB" or "#RGB"
var regEx = /#([a-f0-9]{6}|[a-f0-9]{3})/i;

var colour1 = "The background colour is #3300ff";
var colour2 = "The foreground colour is sky blue pink";

var results1 = regEx.exec(colour1);  // → ["#3300ff", "3300ff"]
var results2 = regEx.exec(colour2);  // → null  The string does not contain an RGB triple
```

Regular Expressions

A Regular Expression is a tool for defining a pattern of text and then locating instances of that pattern within a larger string.

```
// Create a regular expression for identifying RGB colour triples in the form of either
// "#RRGGBB" or "#RGB"
var regEx = /#([a-f0-9]{6}|[a-f0-9]{3})/i;

var colour1 = "The background colour is #3300ff";
var colour2 = "The foreground colour is sky blue pink";

var results1 = regEx.exec(colour1);  // → ["#3300ff", "3300ff"]
var results2 = regEx.exec(colour2);  // → null  The string does not contain an RGB triple

results1.index;  // → 25   The index at which the first text pattern match was found
```

Regular Expressions

A Regular Expression is a tool for defining a pattern of text and then locating instances of that pattern within a larger string.

```
// Create a regular expression for identifying RGB colour triples in the form of either
// "#RRGGBB" or "#RGB"
var regEx = /#([a-f0-9]{6}|[a-f0-9]{3})/i;

var colour1 = "The background colour is #3300ff";
var colour2 = "The foreground colour is sky blue pink";

var results1 = regEx.exec(colour1);  // → ["#3300ff", "3300ff"]
var results2 = regEx.exec(colour2);  // → null  The string does not contain an RGB triple

results1.index;  // → 25   The index at which the first text pattern match was found
```

See https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Regular_Expressions for more details on the syntax and use of regular expressions.