Chapter 10: Comparison Operations

Section 10.1: Abstract equality / inequality and type conversion

The Problem

The abstract equality and inequality operators (`==` and `!=`) convert their operands if the operand types do not match. This type coercion is a common source of confusion about the results of these operators, in particular, these operators aren't always transitive as one would expect.

```
"" ==  0;     // true A
 0 == "0";    // true A
"" == "0";    // false B
false == 0;   // true
false == "0"; // true

"" !=  0;     // false A
 0 != "0";    // false A
"" != "0";    // true B
false != 0;   // false
false != "0"; // false
```

The results start to make sense if you consider how JavaScript converts empty strings to numbers.

```
Number("");    // 0
Number("0");   // 0
Number(false); // 0
```

The Solution

In the statement `false B`, both the operands are strings (`""` and `"0"`), hence there will be **no type conversion** and since `""` and `"0"` are not the same value, `"" == "0"` is `false` as expected.

One way to eliminate unexpected behavior here is making sure that you always compare operands of the same type. For example, if you want the results of numerical comparison use explicit conversion:

```
var test = (a,b) => Number(a) == Number(b);
test("", 0);        // true;
test("0", 0);       // true
test("", "0");      // true;
test("abc", "abc"); // false as operands are not numbers
```

Or, if you want string comparison:

```
var test = (a,b) => String(a) == String(b);
test("", 0);   // false;
test("0", 0);  // true
test("", "0"); // false;
```

Side-note: `Number("0")` and `new Number("0")` isn't the same thing! While the former performs a type conversion, the latter will create a new object. Objects are compared by reference and not by value which explains the results below.

```
Number("0") == Number("0");         // true;
```

```
new Number("0") == new Number("0"); // false
```

Finally, you have the option to use strict equality and inequality operators which will not perform any implicit type conversions.

```
"" ===  0;  // false
 0 === "0"; // false
"" === "0"; // false
```

Further reference to this topic can be found here:

Which equals operator (== vs ===) should be used in JavaScript comparisons?.

Abstract Equality (==)

Section 10.2: NaN Property of the Global Object

`NaN` ("**N**ot **a** **N**umber") is a special value defined by the *IEEE Standard for Floating-Point Arithmetic*, which is used when a non-numeric value is provided but a number is expected (`1 * "two"`), or when a calculation doesn't have a valid `number` result (`Math.sqrt(-1)`).

Any equality or relational comparisons with `NaN` returns `false`, even comparing it with itself. Because, `NaN` is supposed to denote the result of a nonsensical computation, and as such, it isn't equal to the result of any other nonsensical computations.

```
(1 * "two") === NaN  //false

NaN === 0;          // false
NaN === NaN;        // false
Number.NaN === NaN; // false

NaN < 0;            // false
NaN > 0;            // false
NaN > 0;            // false
NaN >= NaN;         // false
NaN >= 'two';       // false
```

Non-equal comparisons will always return `true`:

```
NaN !== 0;          // true
NaN !== NaN;        // true
```

Checking if a value is NaN

Version ≥ 6

You can test a value or expression for `NaN` by using the function Number.isNaN():

```
Number.isNaN(NaN);          // true
Number.isNaN(0 / 0);        // true
Number.isNaN('str' - 12);   // true

Number.isNaN(24);           // false
Number.isNaN('24');         // false
Number.isNaN(1 / 0);        // false
Number.isNaN(Infinity);     // false

Number.isNaN('str');        // false
Number.isNaN(undefined);    // false
```

```
Number.isNaN({});          // false
```

Version < 6

You can check if a value is `NaN` by comparing it with itself:

```
value !== value;    // true for NaN, false for any other value
```

You can use the following polyfill for `Number.isNaN()`:

```
Number.isNaN = Number.isNaN || function(value) {
    return value !== value;
}
```

By contrast, the global function `isNaN()` returns `true` not only for `NaN`, but also for any value or expression that cannot be coerced into a number:

```
isNaN(NaN);         // true
isNaN(0 / 0);       // true
isNaN('str' - 12);  // true

isNaN(24);          // false
isNaN('24');        // false
isNaN(Infinity);    // false

isNaN('str');       // true
isNaN(undefined);   // true
isNaN({});          // true
```

ECMAScript defines a “sameness” algorithm called `SameValue` which, since ECMAScript 6, can be invoked with `Object.is`. Unlike the `==` and `===` comparison, using `Object.is()` will treat `NaN` as identical with itself (and `-0` as not identical with `+0`):

```
Object.is(NaN, NaN)       // true
Object.is(+0, 0)          // false

NaN === NaN               // false
+0 === 0                  // true
```

Version < 6

You can use the following polyfill for `Object.is()` (from MDN):

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue algorithm
    if (x === y) { // Steps 1-5, 7-10
      // Steps 6.b-6.e: +0 != -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // Step 6.a: NaN == NaN
      return x !== x && y !== y;
    }
  };
}
```

Points to note

NaN itself is a number, meaning that it does not equal to the string "NaN", and most importantly (though perhaps unintuitively):

```
typeof(NaN) === "number"; //true
```

Section 10.3: Short-circuiting in boolean operators

The and-operator (`&&`) and the or-operator (`||`) employ short-circuiting to prevent unnecessary work if the outcome of the operation does not change with the extra work.

In `x && y`, `y` will not be evaluated if `x` evaluates to **`false`**, because the whole expression is guaranteed to be **`false`**.

In `x || y`, `y` will not be evaluated if `x` evaluated to **`true`**, because the whole expression is guaranteed to be **`true`**.

Example with functions

Take the following two functions:

```
function T() { // True
  console.log("T");
  return true;
}

function F() { // False
  console.log("F");
  return false;
}
```

Example 1

```
T() && F(); // false
```

Output:

> 'T'
> 'F'

Example 2

```
F() && T(); // false
```

Output:

> 'F'

Example 3

```
T() || F(); // true
```

Output:

> 'T'

Example 4

```
F() || T(); // true
```

Output:

> 'F'
> 'T'

Short-circuiting to prevent errors

```
var obj; // object has value of undefined
if(obj.property){ }// TypeError: Cannot read property 'property' of undefined
if(obj.property && obj !== undefined){}// Line A TypeError: Cannot read property 'property' of
undefined
```

Line A: if you reverse the order the first conditional statement will prevent the error on the second by not executing it if it would throw the error

```
if(obj !== undefined && obj.property){}; // no error thrown
```

But should only be used if you expect `undefined`

```
if(typeof obj === "object" && obj.property){}; // safe option but slower
```

Short-circuiting to provide a default value

The `||` operator can be used to select either a "truthy" value, or the default value.

For example, this can be used to ensure that a nullable value is converted to a non-nullable value:

```
var nullableObj = null;
var obj = nullableObj || {};  // this selects {}

var nullableObj2 = {x: 5};
var obj2 = nullableObj2 || {} // this selects {x: 5}
```

Or to return the first truthy value

```
var truthyValue = {x: 10};
return truthyValue || {}; // will return {x: 10}
```

The same can be used to fall back multiple times:

```
envVariable || configValue || defaultConstValue // select the first "truthy" of these
```

Short-circuiting to call an optional function

The `&&` operator can be used to evaluate a callback, only if it is passed:

```
function myMethod(cb) {
    // This can be simplified
    if (cb) {
       cb();
    }
```

```
    // To this
    cb && cb();
}
```

Of course, the test above does not validate that `cb` is in fact a **`function`** and not just an `Object/Array/String/Number`.

Section 10.4: Null and Undefined

The differences between `null` and `undefined`

`null` and **`undefined`** share abstract equality `==` but not strict equality `===`,

```
null == undefined   // true
null === undefined  // false
```

They represent slightly different things:

- **`undefined`** represents the *absence of a value*, such as before an identifier/Object property has been created or in the period between identifier/Function parameter creation and it's first set, if any.
- **`null`** represents the ***intentional*** *absence of a value* for an identifier or property which has already been created.

They are different types of syntax:

- **`undefined`** is a *property of the global Object*, usually immutable in the global scope. This means anywhere you can define an identifier other than in the global namespace could hide **`undefined`** from that scope (although things can still **be `undefined`**)
- **`null`** is a *word literal*, so it's meaning can never be changed and attempting to do so will throw an *Error*.

The similarities between `null` and `undefined`

`null` and **`undefined`** are both falsy.

```
if (null) console.log("won't be logged");
if (undefined) console.log("won't be logged");
```

Neither **`null`** or **`undefined`** equal **`false`** (see this question).

```
false == undefined   // false
false == null        // false
false === undefined  // false
false === null       // false
```

Using `undefined`

- If the current scope can't be trusted, use something which evaluates to *undefined*, for example **`void 0`**;.
- If **`undefined`** is shadowed by another value, it's just as bad as shadowing `Array` or `Number`.
- Avoid *setting* something as **`undefined`**. If you want to remove a property *bar* from an *Object* `foo`, **`delete`** `foo.bar;` instead.
- Existence testing identifier `foo` against **`undefined`** **could throw a Reference Error**, use **`typeof`** `foo` against `"undefined"` instead.

Section 10.5: Abstract Equality (==)

Operands of the abstract equality operator are compared *after* being converted to a common type. How this

conversion happens is based on the specification of the operator:

Specification for the == operator:

> **7.2.13 Abstract Equality Comparison**
> The comparison `x == y`, where `x` and `y` are values, produces **`true`** or **`false`**. Such a comparison is performed as follows:
>
> 1. If `Type(x)` is the same as `Type(y)`, then:
>
> - **a.** Return the result of performing Strict Equality Comparison `x === y`.
>
> 2. If `x` is **`null`** and `y` is **`undefined`**, return **`true`**.
> 3. If `x` is **`undefined`** and `y` is **`null`**, return **`true`**.
> 4. If `Type(x)` is `Number` and `Type(y)` is `String`, return the result of the comparison `x == ToNumber(y)`.
> 5. If `Type(x)` is `String` and `Type(y)` is `Number`, return the result of the comparison `ToNumber(x) == y`.
> 6. If `Type(x)` is `Boolean`, return the result of the comparison `ToNumber(x) == y`.
> 7. If `Type(y)` is `Boolean`, return the result of the `comparison x == ToNumber(y)`.
> 8. If `Type(x)` is either `String`, `Number`, or `Symbol` and `Type(y)` is `Object`, return the result of the comparison `x == ToPrimitive(y)`.
> 9. If `Type(x)` is Object and `Type(y)` is either `String`, `Number`, or `Symbol`, return the result of the comparison `ToPrimitive(x) == y`.
> 10. Return **`false`**.

Examples:

```
1 == 1;                     // true
1 == true;                  // true  (operand converted to number: true => 1)
1 == '1';                   // true  (operand converted to number: '1' => 1 )
1 == '1.00';                // true
1 == '1.00000000001';       // false
1 == '1.00000000000000001'; // true  (true due to precision loss)
null == undefined;          // true  (spec #2)
1 == 2;                     // false
0 == false;                 // true
0 == undefined;             // false
0 == "";                    // true
```

Section 10.6: Logic Operators with Booleans

```
var x = true,
    y = false;
```

AND

This operator will return true if both of the expressions evaluate to true. This boolean operator will employ short-circuiting and will not evaluate `y` if `x` evaluates to **`false`**.

```
x && y;
```

This will return false, because `y` is false.

OR

This operator will return true if one of the two expressions evaluate to true. This boolean operator will employ

short-circuiting and `y` will not be evaluated if `x` evaluates to **`true`**.

```
x || y;
```

This will return true, because `x` is true.

NOT

This operator will return false if the expression on the right evaluates to true, and return true if the expression on the right evaluates to false.

```
!x;
```

This will return false, because `x` is true.

Section 10.7: Automatic Type Conversions

Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type:

```
var x = "Hello";     // typeof x is a string
x = 5;               // changes typeof x to a number
```

When doing mathematical operations, JavaScript can convert numbers to strings:

```
var x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number
var x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string
var x = "5" + 7;     // x.valueOf() is 57,  typeof x is a string
var x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number
var x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number
var x = "5" - 7;     // x.valueOf() is -2,  typeof x is a number
var x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number
```

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number):

```
"Hello" - "Dolly"    // returns NaN
```

Section 10.8: Logic Operators with Non-boolean values (boolean coercion)

Logical OR (`||`), reading left to right, will evaluate to the first *truthy* value. If no *truthy* value is found, the last value is returned.

```
var a = 'hello' || '';              // a = 'hello'
var b = '' || [];                   // b = []
var c = '' || undefined;            // c = undefined
var d = 1 || 5;                     // d = 1
var e = 0 || {};                    // e = {}
var f = 0 || '' || 5;               // f = 5
var g = '' || 'yay' || 'boo';       // g = 'yay'
```

Logical AND (`&&`), reading left to right, will evaluate to the first *falsy* value. If no *falsey* value is found, the last value is returned.

```
var a = 'hello' && '';                  // a = ''
var b = '' && [];                       // b = ''
var c = undefined && 0;                 // c = undefined
var d = 1 && 5;                         // d = 5
var e = 0 && {};                        // e = 0
var f = 'hi' && [] && 'done';           // f = 'done'
var g = 'bye' && undefined && 'adios';  // g = undefined
```

This trick can be used, for example, to set a default value to a function argument (prior to ES6).

```
var foo = function(val) {
    // if val evaluates to falsey, 'default' will be returned instead.
    return val || 'default';
}

console.log( foo('burger') );  // burger
console.log( foo(100) );       // 100
console.log( foo([]) );        // []
console.log( foo(0) );         // default
console.log( foo(undefined) ); // default
```

Just keep in mind that for arguments, `0` and (to a lesser extent) the empty string are also often valid values that should be able to be explicitly passed and override a default, which, with this pattern, they won't (because they are *falsy*).

Section 10.9: Empty Array

```
/* ToNumber(ToPrimitive([])) == ToNumber(false) */
[] == false; // true
```

When `[].toString()` is executed it calls `[].join()` if it exists, or `Object.prototype.toString()` otherwise. This comparison is returning **`true`** because `[].join()` returns `''` which, coerced into `0`, is equal to false ToNumber.

Beware though, all objects are truthy and `Array` is an instance of `Object`:

```
// Internally this is evaluated as ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'
```

Section 10.10: Equality comparison operations

JavaScript has four different equality comparison operations.

SameValue

It returns **`true`** if both operands belong to the same Type and are the same value.

Note: the value of an object is a reference.

You can use this comparison algorithm via `Object.is` (ECMAScript 6).

Examples:

```
Object.is(1, 1);           // true
Object.is(+0, -0);         // false
Object.is(NaN, NaN);       // true
Object.is(true, "true");   // false
Object.is(false, 0);       // false
```

```
Object.is(null, undefined); // false
Object.is(1, "1");          // false
Object.is([], []);          // false
```

This algorithm has the properties of an equivalence relation:

- Reflexivity: `Object.is(x, x)` is **`true`**, for any value `x`
- Symmetry: `Object.is(x, y)` is **`true`** if, and only if, `Object.is(y, x)` is **`true`**, for any values `x` and `y`.
- Transitivity: If `Object.is(x, y)` and `Object.is(y, z)` are **`true`**, then `Object.is(x, z)` is also **`true`**, for any values `x`, `y` and `z`.

SameValueZero

It behaves like SameValue, but considers `+0` and `-0` to be equal.

You can use this comparison algorithm via `Array.prototype.includes` (ECMAScript 7).

Examples:

```
[1].includes(1);            // true
[+0].includes(-0);          // true
[NaN].includes(NaN);        // true
[true].includes("true");    // false
[false].includes(0);        // false
[1].includes("1");          // false
[null].includes(undefined); // false
[[]].includes([]);          // false
```

This algorithm still has the properties of an equivalence relation:

- Reflexivity: `[x].includes(x)` is **`true`**, for any value `x`
- Symmetry: `[x].includes(y)` is **`true`** if, and only if, `[y].includes(x)` is **`true`**, for any values `x` and `y`.
- Transitivity: If `[x].includes(y)` and `[y].includes(z)` are **`true`**, then `[x].includes(z)` is also **`true`**, for any values `x`, `y` and `z`.

Strict Equality Comparison

It behaves like SameValue, but

- Considers `+0` and `-0` to be equal.
- Considers **`NaN`** different than any value, including itself

You can use this comparison algorithm via the `===` operator (ECMAScript 3).

There is also the `!==` operator (ECMAScript 3), which negates the result of `===`.

Examples:

```
1 === 1;            // true
+0 === -0;          // true
NaN === NaN;        // false
true === "true";    // false
false === 0;        // false
1 === "1";          // false
null === undefined; // false
[] === [];          // false
```

This algorithm has the following properties:

- Symmetry: x `===` y is **`true`** if, and only if, y === `xistrue, for any valuesxandy`.
- Transitivity: If x `===` y and y `===` z are **`true`**, then x `===` z is also **`true`**, for any values x, y and z.

But is not an equivalence relation because

- **`NaN`** is not reflexive: **`NaN`** `!==` **`NaN`**

Abstract Equality Comparison

If both operands belong to the same Type, it behaves like the Strict Equality Comparison.

Otherwise, it coerces them as follows:

- **`undefined`** and **`null`** are considered to be equal
- When comparing a number with a string, the string is coerced to a number
- When comparing a boolean with something else, the boolean is coerced to a number
- When comparing an object with a number, string or symbol, the object is coerced to a primitive

If there was a coercion, the coerced values are compared recursively. Otherwise the algorithm returns **`false`**.

You can use this comparison algorithm via the `==` operator (ECMAScript 1).

There is also the `!=` operator (ECMAScript 1), which negates the result of `==`.

Examples:

```
1 == 1;            // true
+0 == -0;          // true
NaN == NaN;        // false
true == "true";    // false
false == 0;        // true
1 == "1";          // true
null == undefined; // true
[] == [];          // false
```

This algorithm has the following property:

- Symmetry: x `==` y is **`true`** if, and only if, y `==` x is **`true`**, for any values x and y.

But is not an equivalence relation because

- **`NaN`** is not reflexive: **`NaN`** `!=` **`NaN`**
- Transitivity does not hold, e.g. `0 == ''` and `0 == '0'`, but `'' != '0'`

Section 10.11: Relational operators (<, <=, >, >=)

When both operands are numeric, they are compared normally:

```
1 < 2        // true
2 <= 2       // true
3 >= 5       // false
true < false // false (implicitly converted to numbers, 1 > 0)
```

When both operands are strings, they are compared lexicographically (according to alphabetical order):

```
'a' < 'b'    // true
'1' < '2'    // true
```

```
'100' > '12' // false ('100' is less than '12' lexicographically!)
```

When one operand is a string and the other is a number, the string is converted to a number before comparison:

```
'1' < 2      // true
'3' > 2      // true
true > '2'   // false (true implicitly converted to number, 1 < 2)
```

When the string is non-numeric, numeric conversion returns `NaN` (not-a-number). Comparing with `NaN` always returns `false`:

```
1 < 'abc'    // false
1 > 'abc'    // false
```

But be careful when comparing a numeric value with `null`, `undefined` or empty strings:

```
1 > ''         // true
1 < ''         // false
1 > null       // true
1 < null       // false
1 > undefined  // false
1 < undefined  // false
```

When one operand is a object and the other is a number, the object is converted to a number before comparison.So `null` is particular case because `Number(null);//0`

```
new Date(2015)< 1479480185280            // true
null > -1                                //true
({toString:function(){return 123}}) > 122  //true
```

Section 10.12: Inequality

Operator `!=` is the inverse of the `==` operator.
Will return `true` if the operands aren't equal.
The JavaScript engine will try and convert both operands to matching types if they aren't of the same type. **Note:** if the two operands have different internal references in memory, then `false` will be returned.

Sample:

```
1 != '1'      // false
1 != 2        // true
```

In the sample above, `1 != '1'` is `false` because, a primitive number type is being compared to a `char` value. Therefore, the JavaScript engine doesn't care about the datatype of the R.H.S value.

Operator: `!==` is the inverse of the `===` operator. Will return true if the operands are not equal or if their types do not match.

Example:

```
1 !== '1'     // true
1 !== 2       // true
1 !== 1       // false
```

Section 10.13: List of Comparison Operators

Operator	Comparison	Example
`==`	Equal	`i == 0`
`===`	Equal Value and Type	`i === "5"`
`!=`	Not Equal	`i != 5`
`!==`	Not Equal Value or Type	`i !== 5`
`>`	Greater than	`i > 5`
`<`	Less than	`i < 5`
`>=`	Greater than or equal	`i >= 5`
`<=`	Less than or equal	`i <= 5`

Section 10.14: Grouping multiple logic statements

You can group multiple boolean logic statements within parenthesis in order to create a more complex logic evaluation, especially useful in if statements.

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {
  console.log('You can enter our club');
}
```

We could also move the grouped logic to variables to make the statement a bit shorter and descriptive:

```
var isLegal = age >= 18;
var tall = height >= 5.11;
var suitable = isLegal && tall;
var isRoyalty = status === 'royalty';
var specialCase = isRoyalty && hasInvitation;
var canEnterOurBar = suitable || specialCase;

if (canEnterOurBar) console.log('You can enter our club');
```

Notice that in this particular example (and many others), grouping the statements with parenthesis works the same as if we removed them, just follow a linear logic evaluation and you'll find yourself with the same result. I do prefer using parenthesis as it allows me to understand clearer what I intended and might prevent for logic mistakes.

Section 10.15: Bit fields to optimise comparison of multi state data

A bit field is a variable that holds various boolean states as individual bits. A bit on would represent true, and off would be false. In the past bit fields were routinely used as they saved memory and reduced processing load. Though the need to use bit field is no longer so important they do offer some benefits that can simplify many processing tasks.

For example user input. When getting input from a keyboard's direction keys up, down, left, right you can encode the various keys into a single variable with each direction assigned a bit.

Example reading keyboard via bitfield

```
var bitField = 0;  // the value to hold the bits
const KEY_BITS = [4,1,8,2]; // left up right down
const KEY_MASKS = [0b1011,0b1110,0b0111,0b1101]; // left up right down
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
```

```
        if(e.type === "keydown"){
            bitField |= KEY_BITS[e.keyCode - 37];
        }else{
            bitField &= KEY_MASKS[e.keyCode - 37];
        }
    }
}
```

Example reading as an array

```
var directionState = [false,false,false,false];
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        directionState[e.keyCode - 37] = e.type === "keydown";
    }
}
```

To turn on a bit use bitwise *or* `|` and the value corresponding to the bit. So if you wish to set the 2nd bit `bitField |= 0b10` will turn it on. If you wish to turn a bit off use bitwise *and* `&` with a value that has all by the required bit on. Using 4 bits and turning the 2nd bit off `bitfield &= 0b1101;`

You may say the above example seems a lot more complex than assigning the various key states to an array. Yes, it is a little more complex to set but the advantage comes when interrogating the state.

If you want to test if all keys are up.

```
// as bit field
if(!bitfield) // no keys are on

// as array test each item in array
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
```

You can set some constants to make things easier

```
// postfix U,D,L,R for Up down left right
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // up left
const KEY_UR = KEY_U + KEY_R; // up Right
const KEY_DL = KEY_D + KEY_L; // down left
const KEY_DR = KEY_D + KEY_R; // down right
```

You can then quickly test for many various keyboard states

```
if ((bitfield & KEY_UL) === KEY_UL) { // is UP and LEFT only down
if (bitfield  & KEY_UL) {             // is Up left down
if ((bitfield & KEY_U) === KEY_U) {   // is Up only down
if (bitfield & KEY_U) {               // is Up down (any other key may be down)
if (!(bitfield & KEY_U)) {            // is Up up (any other key may be down)
if (!bitfield ) {                     // no keys are down
if (bitfield ) {                      // any one or more keys are down
```

The keyboard input is just one example. Bitfields are useful when you have various states that must in combination be acted on. JavaScript can use up to 32 bits for a bit field. Using them can offer significant performance increases. They are worth being familiar with.