Chapter 14: Arithmetic (Math)

Section 14.1: Constants

Constants	Description	Approximate
`Math.E`	Base of natural logarithm *e*	2.718
`Math.LN10`	Natural logarithm of 10	2.302
`Math.LN2`	Natural logarithm of 2	0.693
`Math.LOG10E`	Base 10 logarithm of *e*	0.434
`Math.LOG2E`	Base 2 logarithm of *e*	1.442
`Math.PI`	Pi: the ratio of circle circumference to diameter (π)	3.14
`Math.SQRT1_2`	Square root of 1/2	0.707
`Math.SQRT2`	Square root of 2	1.414
`Number.EPSILON`	Difference between one and the smallest value greater than one representable as a Number	2.2204460492503130808472633361816E-16
`Number.MAX_SAFE_INTEGER`	Largest integer `n` such that `n` and `n + 1` are both exactly representable as a Number	2^53 - 1
`Number.MAX_VALUE`	Largest positive finite value of Number	1.79E+308
`Number.MIN_SAFE_INTEGER`	Smallest integer `n` such that `n` and `n - 1` are both exactly representable as a Number	-(2^53 - 1)
`Number.MIN_VALUE`	Smallest positive value for Number	5E-324
`Number.NEGATIVE_INFINITY`	Value of negative infinity (-∞)	
`Number.POSITIVE_INFINITY`	Value of positive infinity (∞)	
`Infinity`	Value of positive infinity (∞)	

Section 14.2: Remainder / Modulus (%)

The remainder / modulus operator (`%`) returns the remainder after (integer) division.

```
console.log( 42 %  10); //  2
console.log( 42 % -10); //  2
console.log(-42 %  10); // -2
console.log(-42 % -10); // -2
console.log(-40 %  10); // -0
console.log( 40 %  10); //  0
```

This operator returns the remainder left over when one operand is divided by a second operand. When the first operand is a negative value, the return value will always be negative, and vice versa for positive values.

In the example above, `10` can be subtracted four times from `42` before there is not enough left to subtract again without it changing sign. The remainder is thus: `42 - 4 * 10 = 2`.

The remainder operator may be useful for the following problems:

1. Test if an integer is (not) divisible by another number:

```
x % 4 == 0 // true if x is divisible by 4
x % 2 == 0 // true if x is even number
```

```
x % 2 != 0 // true if x is odd number
```

Since `0 === -0`, this also works for `x <= -0`.

2. Implement cyclic increment/decrement of value within `[0, n)` interval.

Suppose that we need to increment integer value from `0` to (but not including) `n`, so the next value after `n-1` become `0`. This can be done by such pseudocode:

```
var n = ...; // given n
var i = 0;
function inc() {
    i = (i + 1) % n;
}
while (true) {
    inc();
    // update something with i
}
```

Now generalize the above problem and suppose that we need to allow to both increment and decrement that value from `0` to (not including) `n`, so the next value after `n-1` become `0` and the previous value before `0` become `n-1`.

```
var n = ...; // given n
var i = 0;
function delta(d) { // d - any signed integer
    i = (i + d + n) % n; // we add n to (i+d) to ensure the sum is positive
}
```

Now we can call `delta()` function passing any integer, both positive and negative, as delta parameter.

Using modulus to obtain the fractional part of a number

```
var myNum = 10 / 4;       // 2.5
var fraction = myNum % 1; // 0.5
myNum = -20 / 7;          // -2.857142857142857
fraction = myNum % 1;     // -0.857142857142857
```

Section 14.3: Rounding

Rounding

`Math.round()` will round the value to the closest integer using *half round up* to break ties.

```
var a = Math.round(2.3);       // a is now 2
var b = Math.round(2.7);       // b is now 3
var c = Math.round(2.5);       // c is now 3
```

But

```
var c = Math.round(-2.7);       // c is now -3
var c = Math.round(-2.5);       // c is now -2
```

Note how `-2.5` is rounded to `-2`. This is because half-way values are always rounded up, that is they're rounded to the integer with the next higher value.

Rounding up

`Math.ceil()` will round the value up.

```
var a = Math.ceil(2.3);        // a is now 3
var b = Math.ceil(2.7);        // b is now 3
```

`ceil`ing a negative number will round towards zero

```
var c = Math.ceil(-1.1);       // c is now 1
```

Rounding down

`Math.floor()` will round the value down.

```
var a = Math.floor(2.3);        // a is now 2
var b = Math.floor(2.7);        // b is now 2
```

`floor`ing a negative number will round it away from zero.

```
var c = Math.floor(-1.1);       // c is now -1
```

Truncating

Caveat: using bitwise operators (except `>>>`) only applies to numbers between `-2147483649` and `2147483648`.

```
2.3  | 0;                       // 2 (floor)
-2.3 | 0;                       // -2 (ceil)
NaN  | 0;                       // 0
```

Version ≥ 6

`Math.trunc()`

```
Math.trunc(2.3);                // 2 (floor)
Math.trunc(-2.3);               // -2 (ceil)
Math.trunc(2147483648.1);       // 2147483648 (floor)
Math.trunc(-2147483649.1);      // -2147483649 (ceil)
Math.trunc(NaN);                // NaN
```

Rounding to decimal places

`Math.floor`, `Math.ceil()`, and `Math.round()` can be used to round to a number of decimal places

To round to 2 decimal places:

```
 var myNum = 2/3;               // 0.6666666666666666
 var multiplier = 100;
 var a = Math.round(myNum * multiplier) / multiplier;  // 0.67
 var b = Math.ceil (myNum * multiplier) / multiplier;  // 0.67
 var c = Math.floor(myNum * multiplier) / multiplier;  // 0.66
```

You can also round to a number of digits:

```
 var myNum = 10000/3;           // 3333.3333333333335
 var multiplier = 1/100;
 var a = Math.round(myNum * multiplier) / multiplier;  // 3300
```

```
var b = Math.ceil (myNum * multiplier) / multiplier;  // 3400
var c = Math.floor(myNum * multiplier) / multiplier;  // 3300
```

As a more usable function:

```
// value is the value to round
// places if positive the number of decimal places to round to
// places if negative the number of digits to round to
function roundTo(value, places){
    var power = Math.pow(10, places);
    return Math.round(value * power) / power;
}
var myNum = 10000/3;    // 3333.3333333333335
roundTo(myNum, 2);  // 3333.33
roundTo(myNum, 0);  // 3333
roundTo(myNum, -2); // 3300
```

And the variants for `ceil` and `floor`:

```
function ceilTo(value, places){
    var power = Math.pow(10, places);
    return Math.ceil(value * power) / power;
}
function floorTo(value, places){
    var power = Math.pow(10, places);
    return Math.floor(value * power) / power;
}
```

Section 14.4: Trigonometry

All angles below are in radians. An angle `r` in radians has measure `180 * r / Math.PI` in degrees.

Sine

```
Math.sin(r);
```

This will return the sine of `r`, a value between -1 and 1.

```
Math.asin(r);
```

This will return the arcsine (the reverse of the sine) of `r`.

```
Math.asinh(r)
```

This will return the hyperbolic arcsine of `r`.

Cosine

```
Math.cos(r);
```

This will return the cosine of `r`, a value between -1 and 1

```
Math.acos(r);
```

This will return the arccosine (the reverse of the cosine) of `r`.

```
Math.acosh(r);
```

This will return the hyperbolic arccosine of `r`.

Tangent

```
Math.tan(r);
```

This will return the tangent of `r`.

```
Math.atan(r);
```

This will return the arctangent (the reverse of the tangent) of `r`. Note that it will return an angle in radians between `-π/2` and `π/2`.

```
Math.atanh(r);
```

This will return the hyperbolic arctangent of `r`.

```
Math.atan2(x, y);
```

This will return the value of an angle from `(0, 0)` to `(x, y)` in radians. It will return a value between `-π` and `π`, not including π.

Section 14.5: Bitwise operators

Note that all bitwise operations operate on 32-bit integers by passing any operands to the internal function ToInt32.

Bitwise or

```
var a;
a = 0b0011 | 0b1010; // a === 0b1011
// truth table
// 1010 | (or)
// 0011
// 1011  (result)
```

Bitwise and

```
a = 0b0011 & 0b1010; // a === 0b0010
// truth table
// 1010 & (and)
// 0011
// 0010  (result)
```

Bitwise not

```
a = ~0b0011; // a === 0b1100
// truth table
// 10 ~(not)
// 01  (result)
```

Bitwise xor (exclusive or)

```
a = 0b1010 ^ 0b0011; // a === 0b1001
// truth table
// 1010 ^ (xor)
// 0011
// 1001  (result)
```

Bitwise left shift

```
a = 0b0001 << 1; // a === 0b0010
a = 0b0001 << 2; // a === 0b0100
a = 0b0001 << 3; // a === 0b1000
```

Shift left is equivalent to integer multiply by `Math.pow(2, n)`. When doing integer math, shift can significantly improve the speed of some math operations.

```
var n = 2;
var a = 5.4;
var result = (a << n) === Math.floor(a) * Math.pow(2,n);
// result is true
a = 5.4 << n; // 20
```

Bitwise right shift `>>` (Sign-propagating shift) `>>>` (Zero-fill right shift)

```
a = 0b1001 >> 1; // a === 0b0100
a = 0b1001 >> 2; // a === 0b0010
a = 0b1001 >> 3; // a === 0b0001

a = 0b1001 >>> 1; // a === 0b0100
a = 0b1001 >>> 2; // a === 0b0010
a = 0b1001 >>> 3; // a === 0b0001
```

A negative 32bit value always has the left most bit on:

```
a = 0b11111111111111111111111111110111 | 0;
console.log(a); // -9
b = a >> 2;     // leftmost bit is shifted 1 to the right then new left most bit is set to on (1)
console.log(b); // -3
b = a >>> 2;    // leftmost bit is shifted 1 to the right. the new left most bit is set to off (0)
console.log(b); // 2147483643
```

The result of a `>>>` operation is always positive.
The result of a `>>` is always the same sign as the shifted value.

Right shift on positive numbers is the equivalent of dividing by the `Math.pow(2,n)` and flooring the result:

```
a = 256.67;
n = 4;
result = (a >> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >> n; //  16

result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >>> n; //  16
```

Right shift zero fill (`>>>`) on negative numbers is different. As JavaScript does not convert to unsigned ints when doing bit operations there is no operational equivalent:

```
a = -256.67;
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is false
```

Bitwise assignment operators

With the exception of not (~) all the above bitwise operators can be used as assignment operators:

```
a |= b;   // same as: a = a | b;
a ^= b;   // same as: a = a ^ b;
a &= b;   // same as: a = a & b;
a >>= b;  // same as: a = a >> b;
a >>>= b; // same as: a = a >>> b;
a <<= b;  // same as: a = a << b;
```

Warning: JavaScript uses Big Endian to store integers. This will not always match the Endian of the device/OS. When using typed arrays with bit lengths greater than 8 bits you should check if the environment is Little Endian or Big Endian before applying bitwise operations.

Warning: Bitwise operators such as `&` and `|` are **not** the same as the logical operators `&&` (and) and `||` (or). They will provide incorrect results if used as logical operators. The `^` operator is **not** the power operator (*ab*).

Section 14.6: Incrementing (++)

The Increment operator (`++`) increments its operand by one.

- If used as a postfix, then it returns the value before incrementing.
- If used as a prefix, then it returns the value after incrementing.

```
//postfix
var a = 5,    // 5
    b = a++,  // 5
    c = a     // 6
```

In this case, `a` is incremented after setting `b`. So, `b` will be 5, and `c` will be 6.

```
//prefix
var a = 5,    // 5
    b = ++a,  // 6
    c = a     // 6
```

In this case, `a` is incremented before setting `b`. So, `b` will be 6, and `c` will be 6.

The increment and decrement operators are commonly used in `for` loops, for example:

```
for(var i = 0; i < 42; ++i)
{
  // do something awesome!
}
```

Notice how the *prefix* variant is used. This ensures that a temporarily variable isn't needlessly created (to return the value prior to the operation).

Section 14.7: Exponentiation (Math.pow() or **)

Exponentiation makes the second operand the power of the first operand (ab).

```
var a = 2,
    b = 3,
    c = Math.pow(a, b);
```

`c` will now be 8

`Version > 6`

Stage 3 ES2016 (ECMAScript 7) Proposal:

```
let a = 2,
    b = 3,
    c = a ** b;
```

c will now be 8

Use Math.pow to find the nth root of a number.

Finding the nth roots is the inverse of raising to the nth power. For example 2 to the power of 5 is 32. The 5th root of 32 is 2.

```
Math.pow(v, 1 / n); // where v is any positive real number
                    // and n is any positive integer

var a = 16;
var b = Math.pow(a, 1 / 2);  // return the square root of 16 = 4
var c = Math.pow(a, 1 / 3);  // return the cubed root of 16 = 2.5198420997897464
var d = Math.pow(a, 1 / 4);  // return the 4th root of 16 = 2
```

Section 14.8: Random Integers and Floats

```
var a = Math.random();
```

Sample value of a: 0.21322848065742162

Math.random() returns a random number between 0 (inclusive) and 1 (exclusive)

```
function getRandom() {
    return Math.random();
}
```

To use Math.random() to get a number from an arbitrary range (not [0,1)) use this function to get a random number between min (inclusive) and max (exclusive): interval of [min, max)

```
function getRandomArbitrary(min, max) {
  return Math.random() * (max - min) + min;
}
```

To use Math.random() to get an integer from an arbitrary range (not [0,1)) use this function to get a random number between min (inclusive) and max (exclusive): interval of [min, max)

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

To use Math.random() to get an integer from an arbitrary range (not [0,1)) use this function to get a random number between min (inclusive) and max (inclusive): interval of [min, max]

```
function getRandomIntInclusive(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Functions taken from
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Section 14.9: Addition (+)

The addition operator (+) adds numbers.

```
var a = 9,
    b = 3,
    c = a + b;
```

c will now be 12

This operand can also be used multiple times in a single assignment:

```
var a = 9,
    b = 3,
    c = 8,
    d = a + b + c;
```

d will now be 20.

Both operands are converted to primitive types. Then, if either one is a string, they're both converted to strings and concatenated. Otherwise, they're both converted to numbers and added.

```
null + null;      // 0
null + undefined; // NaN
null + {};        // "null[object Object]"
null + '';        // "null"
```

If the operands are a string and a number, the number is converted to a string and then they're concatenated, which may lead to unexpected results when working with strings that look numeric.

```
"123" + 1;        // "1231" (not 124)
```

If a boolean value is given in place of any of the number values, the boolean value is converted to a number (`0` for `false`, 1 for `true`) before the sum is calculated:

```
true + 1;         // 2
false + 5;        // 5
null + 1;         // 1
undefined + 1;    // NaN
```

If a boolean value is given alongside a string value, the boolean value is converted to a string instead:

```
true + "1";        // "true1"
false + "bar";     // "falsebar"
```

Section 14.10: Little / Big endian for typed arrays when using bitwise operators

To detect the endian of the device

```
var isLittleEndian = true;
(()=>{
    var buf = new ArrayBuffer(4);
    var buf8 = new Uint8ClampedArray(buf);
    var data = new Uint32Array(buf);
    data[0] = 0x0F000000;
    if(buf8[0] === 0x0f){
        isLittleEndian = false;
    }
})();
```

Little-Endian stores most significant bytes from right to left.

Big-Endian stores most significant bytes from left to right.

```
var myNum = 0x11223344 | 0;  // 32 bit signed integer
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = myNum; // store number in 32Bit array
```

If the system uses Little-Endian, then the 8bit byte values will be

```
console.log(data8[0].toString(16)); // 0x44
console.log(data8[1].toString(16)); // 0x33
console.log(data8[2].toString(16)); // 0x22
console.log(data8[3].toString(16)); // 0x11
```

If the system uses Big-Endian, then the 8bit byte values will be

```
console.log(data8[0].toString(16)); // 0x11
console.log(data8[1].toString(16)); // 0x22
console.log(data8[2].toString(16)); // 0x33
console.log(data8[3].toString(16)); // 0x44
```

Example where Endian type is important

```
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// To speed up read and write from the image buffer you can create a buffer view that is
// 32 bits allowing you to read/write a pixel in a single operation
var buf32 = new Uint32Array(imgData.data.buffer);
// Mask out Red and Blue channels
var mask = 0x00FF00FF; // bigEndian pixel channels Red,Green,Blue,Alpha
if(isLittleEndian){
    mask = 0xFF00FF00; // littleEndian pixel channels Alpha,Blue,Green,Red
}
var len = buf32.length;
var i = 0;
while(i < len){  // Mask all pixels
    buf32[i] &= mask; //Mask out Red and Blue
}
ctx.putImageData(imgData);
```

Section 14.11: Get Random Between Two Numbers

Returns a random integer between `min` and `max`:

```
function randomBetween(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}
```

Examples:

```
// randomBetween(0, 10);
Math.floor(Math.random() * 11);

// randomBetween(1, 10);
Math.floor(Math.random() * 10) + 1;

// randomBetween(5, 20);
Math.floor(Math.random() * 16) + 5;

// randomBetween(-10, -2);
Math.floor(Math.random() * 9) - 10;
```

Section 14.12: Simulating events with different probabilities

Sometimes you may only need to simulate an event with two outcomes, maybe with different probabilities, but you may find yourself in a situation that calls for many possible outcomes with different probabilities. Let's imagine you want to simulate an event that has six equally probable outcomes. This is quite simple.

```
function simulateEvent(numEvents) {
    var event = Math.floor(numEvents*Math.random());
    return event;
}

// simulate fair die
console.log("Rolled a "+(simulateEvent(6)+1));  // Rolled a 2
```

However, you may not want equally probable outcomes. Say you had a list of three outcomes represented as an array of probabilities in percents or multiples of likelihood. Such an example might be a weighted die. You could rewrite the previous function to simulate such an event.

```
function simulateEvent(chances) {
    var sum = 0;
    chances.forEach(function(chance) {
        sum+=chance;
    });
    var rand = Math.random();
    var chance = 0;
    for(var i=0; i<chances.length; i++) {
        chance+=chances[i]/sum;
        if(rand<chance) {
            return i;
        }
    }

    // should never be reached unless sum of probabilities is less than 1
    // due to all being zero or some being negative probabilities
    return -1;
}

// simulate weighted dice where 6 is twice as likely as any other face
// using multiples of likelihood
console.log("Rolled a "+(simulateEvent([1,1,1,1,1,2])+1));  // Rolled a 1

// using probabilities
console.log("Rolled a "+(simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1));  // Rolled a 6
```

As you probably noticed, these functions return an index, so you could have more descriptive outcomes stored in an array. Here's an example.

```
var rewards = ["gold coin","silver coin","diamond","god sword"];
var likelihoods = [5,9,1,0];
// least likely to get a god sword (0/15 = 0%, never),
// most likely to get a silver coin (9/15 = 60%, more than half the time)

// simulate event, log reward
console.log("You get a "+rewards[simulateEvent(likelihoods)]);  // You get a silver coin
```

Section 14.13: Subtraction (-)

The subtraction operator (`-`) subtracts numbers.

```
var a = 9;
var b = 3;
var c = a - b;
```

`c` will now be 6

If a string or boolean is provided in place of a number value, it gets converted to a number before the difference is calculated (`0` for **`false`**, `1` for **`true`**):

```
"5" - 1;     // 4
7 - "3";     // 4
"5" - true;  // 4
```

If the string value cannot be converted into a Number, the result will be **`NaN`**:

```
"foo" - 1;      // NaN
100 - "bar";    // NaN
```

Section 14.14: Multiplication (*)

The multiplication operator (`*`) perform arithmetic multiplication on numbers (literals or variables).

```
console.log( 3 *  5); //  15
console.log(-3 *  5); // -15
console.log( 3 * -5); // -15
console.log(-3 * -5); //  15
```

Section 14.15: Getting maximum and minimum

The `Math.max()` function returns the largest of zero or more numbers.

```
Math.max(4, 12);   //  12
Math.max(-1, -15); // -1
```

The `Math.min()` function returns the smallest of zero or more numbers.

```
Math.min(4, 12);   //  4
Math.min(-1, -15); // -15
```

Getting maximum and minimum from an array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max.apply(Math, arr),
    min = Math.min.apply(Math, arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

ECMAScript 6 spread operator, getting the maximum and minimum of an array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max(...arr),
    min = Math.min(...arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

Section 14.16: Restrict Number to Min/Max Range

If you need to clamp a number to keep it inside a specific range boundary

```
function clamp(min, max, val) {
    return Math.min(Math.max(min, +val), max);
}

console.log(clamp(-10, 10, "4.30")); // 4.3
console.log(clamp(-10, 10, -8));     // -8
console.log(clamp(-10, 10, 12));     // 10
console.log(clamp(-10, 10, -15));    // -10
```

Use-case example (jsFiddle)

Section 14.17: Ceiling and Floor

`ceil()`

The `ceil()` method rounds a number *upwards* to the nearest integer, and returns the result.

Syntax:

```
Math.ceil(n);
```

Example:

```
console.log(Math.ceil(0.60)); //  1
console.log(Math.ceil(0.40)); //  1
console.log(Math.ceil(5.1));  //  6
console.log(Math.ceil(-5.1)); // -5
console.log(Math.ceil(-5.9)); // -5
```

`floor()`

The `floor()` method rounds a number *downwards* to the nearest integer, and returns the result.

Syntax:

```
Math.floor(n);
```

Example:

```
console.log(Math.ceil(0.60)); //  0
console.log(Math.ceil(0.40)); //  0
console.log(Math.ceil(5.1));  //  5
console.log(Math.ceil(-5.1)); // -6
console.log(Math.ceil(-5.9)); // -6
```

Section 14.18: Getting roots of a number

Square Root

Use `Math.sqrt()` to find the square root of a number

```
Math.sqrt(16)   #=> 4
```

Cube Root

To find the cube root of a number, use the `Math.cbrt()` function

Version ≥ 6

```
Math.cbrt(27)  #=> 3
```

Finding nth-roots

To find the nth-root, use the `Math.pow()` function and pass in a fractional exponent.

```
Math.pow(64, 1/6) #=> 2
```

Section 14.19: Random with gaussian distribution

The `Math.random()` function should give random numbers that have a standard deviation approaching 0. When picking from a deck of card, or simulating a dice roll this is what we want.

But in most situations this is unrealistic. In the real world the randomness tends to gather around an common normal value. If plotted on a graph you get the classical bell curve or gaussian distribution.

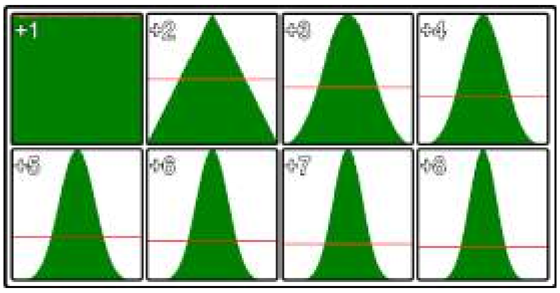
To do this with the `Math.random()` function is relatively simple.

```
var randNum = (Math.random() + Math.random()) / 2;
var randNum = (Math.random() + Math.random() + Math.random()) / 3;
var randNum = (Math.random() + Math.random() + Math.random() + Math.random()) / 4;
```

Adding a random value to the last increases the variance of the random numbers. Dividing by the number of times you add normalises the result to a range of 0–1

As adding more than a few randoms is messy a simple function will allow you to select a variance you want.

```
// v is the number of times random is summed and should be over >= 1
// return a random number between 0-1 exclusive
function randomG(v){
    var r = 0;
    for(var i = v; i > 0; i --){
        r += Math.random();
    }
    return r / v;
}
```



The image shows the distribution of random values for different values of v. The top left is standard single `Math.random()` call the bottom right is `Math.random()` summed 8 times. This is from 5,000,000 samples using Chrome

This method is most efficient at `v<5`

Section 14.20: Math.atan2 to find direction

If you are working with vectors or lines you will at some stage want to get the direction of a vector, or line. Or the direction from a point to another point.

`Math.atan(yComponent, xComponent)` return the angle in radius within the range of `-Math.PI` to `Math.PI` (`-180` to `180` deg)

Direction of a vector

```
var vec = {x : 4, y : 3};
var dir = Math.atan2(vec.y, vec.x);  // 0.6435011087932844
```

Direction of a line

```
var line = {
    p1 : { x : 100, y : 128},
    p2 : { x : 320, y : 256}
}
// get the direction from p1 to p2
var dir = Math.atan2(line.p2.y - line.p1.y, line.p2.x - line.p1.x); // 0.5269432271894297
```

Direction from a point to another point

```
var point1 = { x: 123, y : 294};
var point2 = { x: 354, y : 284};
// get the direction from point1 to point2
var dir = Math.atan2(point2.y - point1.y, point2.x - point1.x); // -0.04326303140726714
```

Section 14.21: Sin & Cos to create a vector given direction & distance

If you have a vector in polar form (direction & distance) you will want to convert it to a cartesian vector with a x and y component. For reference the screen coordinate system has directions as 0 deg points from left to right, 90 (PI/2) point down the screen, and so on in a clock wise direction.

```
var dir = 1.4536; // direction in radians
var dist = 200; // distance
var vec = {};
vec.x = Math.cos(dir) * dist; // get the x component
vec.y = Math.sin(dir) * dist; // get the y component
```

You can also ignore the distance to create a normalised (1 unit long) vector in the direction of `dir`

```
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.cos(dir); // get the x component
vec.y = Math.sin(dir); // get the y component
```

If your coordinate system has y as up then you need to switch cos and sin. In this case a positive direction is in a counterclockwise direction from the x axis.

```
// get the directional vector where y points up
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.sin(dir); // get the x component
vec.y = Math.cos(dir); // get the y component
```

Section 14.22: Math.hypot

To find the distance between two points we use pythagoras to get the square root of the sum of the square of the component of the vector between them.

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.sqrt(x * x + y * y); // 11.180339887498949
```

With ECMAScript 6 came `Math.hypot` which does the same thing

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.hypot(x,y); // 11.180339887498949
```

Now you don't have to hold the interim vars to stop the code becoming a mess of variables

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var distance = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

`Math.hypot` can take any number of dimensions

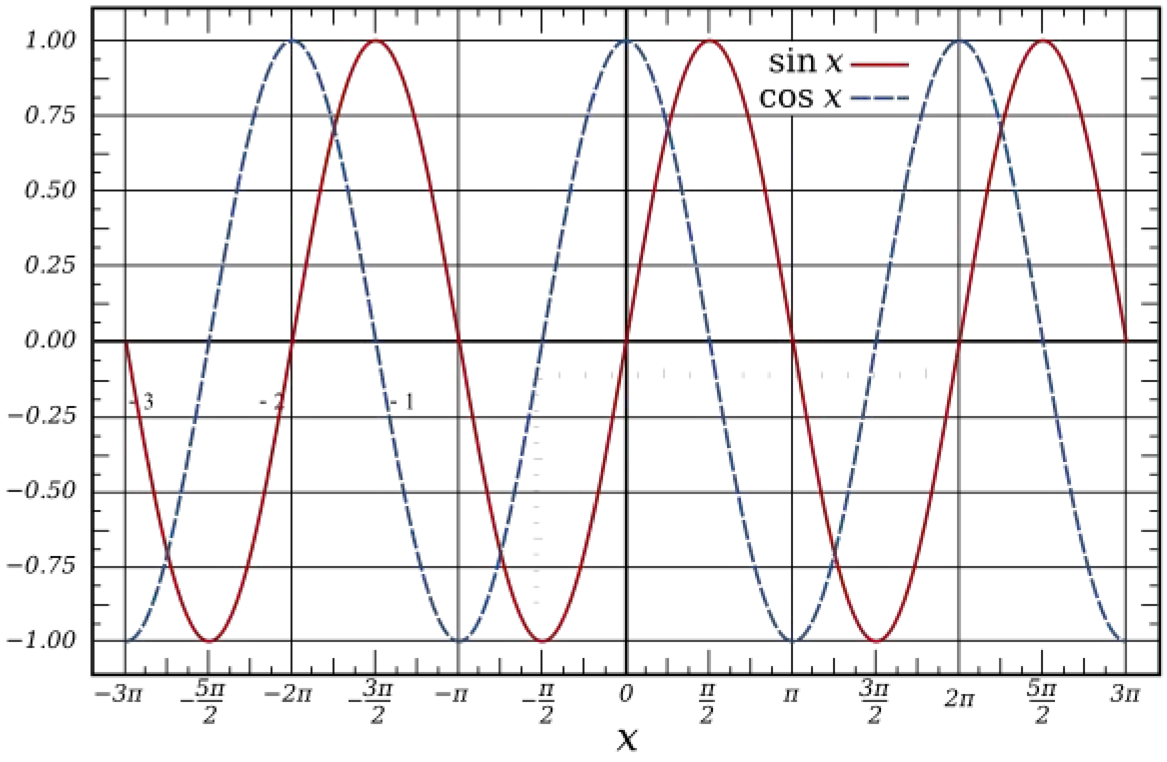
```
// find distance in 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325

// find length of 11th dimensional vector
var v = [1,3,2,6,1,7,3,7,5,3,1];
var i = 0;
dist = Math.hypot(v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++]);
```

Section 14.23: Periodic functions using Math.sin

`Math.sin` and `Math.cos` are cyclic with a period of 2*PI radians (360 deg) they output a wave with an amplitude of 2 in the range -1 to 1.

Graph of sine and cosine function: *(courtesy Wikipedia)*



They are both very handy for many types of periodic calculations, from creating sound waves, to animations, and even encoding and decoding image data

This example shows how to create a simple sin wave with control over period/frequency, phase, amplitude, and offset.

The unit of time being used is seconds.
The most simple form with control over frequency only.

```
// time is the time in seconds when you want to get a sample
// Frequency represents the number of oscillations per second
function oscillator(time, frequency){
    return Math.sin(time * 2 * Math.PI * frequency);
}
```

In almost all cases you will want to make some changes to the value returned. The common terms for modifications

- Phase: The offset in terms of frequency from the start of the oscillations. It is a value in the range of 0 to 1 where the value 0.5 move the wave forward in time by half its frequency. A value of 0 or 1 makes no change.
- Amplitude: The distance from the lowest value and highest value during one cycle. An amplitude of 1 has a range of 2. The lowest point (trough) -1 to the highest (peak) 1. For a wave with frequency 1 the peak is at 0.25 seconds, and trough at 0.75.
- Offset: moves the whole wave up or down.

To include all these in the function:

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    var t = time * frequency * Math.PI * 2; // get phase at time
    t += phase * Math.PI * 2; // add the phase offset
    var v = Math.sin(t); // get the value at the calculated position in the cycle
    v *= amplitude; // set the amplitude
    v += offset; // add the offset
    return v;
}
```

Or in a more compact (and slightly quicker form):

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    return Math.sin(time * frequency * Math.PI * 2 + phase * Math.PI * 2) * amplitude + offset;
}
```

All the arguments apart from time are optional

Section 14.24: Division (/)

The division operator (/) perform arithmetic division on numbers (literals or variables).

```
console.log(15 / 3); // 5
console.log(15 / 4); // 3.75
```

Section 14.25: Decrementing (--)

The decrement operator (--) decrements numbers by one.

- If used as a postfix to n, the operator returns the current n and *then* assigns the decremented the value.
- If used as a prefix to n, the operator assigns the decremented n and *then* returns the changed value.

```
var a = 5,    // 5
    b = a--,  // 5
    c = a     // 4
```

In this case, b is set to the initial value of a. So, b will be 5, and c will be 4.

```
var a = 5,    // 5
    b = --a,  // 4
    c = a     // 4
```

In this case, b is set to the new value of a. So, b will be 4, and c will be 4.

Common Uses

The decrement and increment operators are commonly used in **for** loops, for example:

```
for (var i = 42; i > 0; --i) {
  console.log(i)
}
```

Notice how the *prefix* variant is used. This ensures that a temporarily variable isn't needlessly created (to return the value prior to the operation).

> **Note:** Neither -- nor ++ are like normal mathematical operators, but rather they are very concise operators for *assignment*. Notwithstanding the return value, both

> `x--`
>
> and
>
> `--x`
>
> reassign to x such that `x = x - 1`.

```
const x = 1;
console.log(x--)  // TypeError: Assignment to constant variable.
console.log(--x)  // TypeError: Assignment to constant variable.
console.log(--3)  // ReferenceError: Invalid left-hand size expression in prefix operation.
console.log(3--)  // ReferenceError: Invalid left-hand side expression in postfix operation.
```