JavaScript Notes

Software tools....2
- Computers....2
- Low and high level programming languages. .3
- Compilers and interpreters....3
- Editors....4
- IDEs....4
- JavaScript tool setup....5
- The DOM....5
- Security and the sand-box....6

Basic ideas....6
- How to write JavaScript practically....6
- Using external script files....9
- Variables....9
- Assignment statements....10
- Expressions....10
- Input and output....11
- Conditional statements....12
- = and == and ===....13
- Loops....14

Debugging....17
- Strict mode....17
- Debugging strategies....18

Modularity....19
- Managing scale - decomposition....19
- JavaScript functions....19
- Pass by value....21
- Scope....22
- hoisting....23
- var let and const....23

Standards....25

How to get more help....26

Primitive types....26
- Type....26
- Strict typing....27
- Primitive types....27

Objects and protoypes....29
- How to make an object....30
- The object.property dot notation....30
- What is the point of OOP?....30
- Methods....31
- Objects as associative arrays....31
- Functions are objects....32
- Patterns of objects – using constructors....32
- Object property attributes....33
- The constructor property....35
- Inheritance....36
- Constructor prototype property....38
- Encapsulation....38
- References....39
- Static fields....41
- The built-in objects and the Object object 42
- Constructor....42
- The global object....43

Function objects and concepts....43
- Standard function definition and call....44
- Functions as data objects....44
- Function expressions....45
- Anonymous functions....45
- The Function object....45

Classes....46
- Class....46
- Methods....47
- Constructor....48
- Private fields....48
- Static....49
- Class inheritance....50

Arrays, maps and other data structures....51
- Associative arrays....51
- Typed arrays....51
- The Array object....52
- The Set object....53
- Map....54

The DOM - JavaScript in the web page....54
- Responding to a button click....55
- A basic calculator....56
- More JavaScript DOM use....57

Modules....58
- Basic idea....58
- Need server not file://....58
- Basic example....58
- Modules always strict mode....60
- Use of default....60
- Import and export formats....61
- Namespaces and importing objects....61

Software tools

Computers

This text is about programming computers. That includes mobile phones, tablets, laptops and desktop computers, and also devices like smart TVs, automobile engine management systems, printers, scanners and so on.

To think about all these different types of devices at the same time, it is useful to have an abstract model of computing. This means to just think about what they have in common, and leaving out the details of exactly how they work.

One abstract model is a Turing machine (Alan Turing, 1936). This has a ‘tape’, on which symbols are written, and a head which can read and write on it. The tape head is connected to a finite state machine. This is something which switches from its current state to another state, depending on its current state and the symbol under the tape head. We can think of the finite state machien as being like a processor and memory, and the tape is like a file store on a hard disc or cloud storage.

A different abstract model is the lambda calculus (Alonzo Church and Stephen Kleene, 1935). This sees a computer as a function, like a mathematical function such as square root. This has input (say, 9) and produces output (3). The lambda calculus is a totally general version of this. It can be proved that Turing machines and the lambda calculus are equivalent.

We will take a computer to be more practical. The stored program concept has the program, as a sequence of instructions, held in memory while it is being executed, alongside data also in memory. The memory is organised as a set of cells or locations, each with a different address, and holding one byte (eight bits, binary digits, 0s or 1s, like 0110 1010). That means all program instructions and all data must be encoded as binary patterns.

The computer also has a processor. This can read program instructions out of memory, decode and execute them, and move on to the next instruction. A processor has a small number of registers, which are very fast storage locations inside the processor, which program instructions often refer to.

In practice a computer may have a multi-core processor, containing several processors which can run in parallel, and a maybe a graphics processor and memory just used for graphics.

Low and high level programming languages

The instructions the processor can recognise and execute are called machine code or native code. Different processors have different machine code instruction sets. Typically a processor has around 200 instructions it can execute.

Those instructions are represented by binary patterns – so 1011 1010 might mean 'add'.

It is possible in theory to write programs in machine code, but is totally impractical. Instead we use languages like JavaScript or C, which use more natural language words like 'while' and 'if'.

Low and high level languages are a range. At the low level end we have native code and assembler, with instructions about processor registers and memory locations. At the high level end languages like JavaScript use a small set of English words as instructions (typically aound 50 words).

Compilers and interpreters

Only native code actually executes in a computer. So the high level language rogram we write must be changed into machine code. One software tool to help with this is called a compiler. This is software which reads in a high level language program (called source code) and outputs the equivalent, changed into a different language (called object code). Usually object code is native code. An example of a compiler in common use today is gcc, the open source Gnu Compiler Collection.

So the use of a compiler is a two-stage process:

1. Compile the source into native code

2. Execute the resulting object code.

Different things can happen at these two stages. The first is called compile-time. The second is run-time.

Another type of tool is an interpreter. This inputs a high level language program (often called a script) and executes a loop:

```
repeat for each instruction
.. work out how to do it
.. do it
until end of script reached
```

Different languages are usually treated in different ways. C, for example, is usually compiled to native code, while JavaScript is usually interpreted. But in theory any language can be interpreted or compiled.

Editors

Source code is just text – like for example

```
var x=5;
var y=4;
var z=x+y;
console.log(z);
```

We need a way of 'writing' this text, and also reading it in from a text file, editing it, and writing a new version to a text file. A software tool to do this is called a *text editor*.

This is different from a word processor. This includes formats like font styles, size, colour, layout and so on. But source code is text, and nothing else.

Text editors are pretty simple applications.On Windows, notepad is a text editor provided with the operating system. Other options are notepad++ and atom and textpad. Linux text editors include nano and geany and gedit.

IDEs

An IDE is an 'integrated development environment'. When programming we need to:

- write and edit source code

- compile it
- run it to test it out
- repeat until its time to go home

An IDE is a software tool which makes for quick and easy access to an editor and a compiler. It will also offer other features like

- managing a set of related source code files, and other resources such as images, in a project
- ability to handle different compilers and languages
- debugging support – like stepping through a program one instruction at a time
- syntax-colour – showing keywords in one colour, number in another and so on
- format source code – neat indentation

Common IDEs are Netbeans, Eclipse, Intellij IDEA, Microsoft Visual Studio and Android Design Studio.

JavaScript tool setup

JavaScript code is usually interpreted, using the interpreter inside a web browser. So all we need is a text editor and a web browser, used like this:

1. Use a text editor to edit the JavaScript source code
2. Open it as a local file in a web browser (how depends on the browser, but its CTRL-O on Chrome, Firefox and Opera) to see if it works
3. Go back to step 1 if needed

The DOM

The DOM is the Document Object Model.

Usually JavaScript runs in a browser, and needs to refer to the displayed web page. The DOM is how it can refer to the web page and the parts in it. We have a section about the DOM later.

Most of the examples here use the console, not the DOM - so they are not typical of how real JavaScript is used.

The idea is to keep things as simple as possible, to get an understanding of the JavaScript language, including OOP and functional programming.

Once that is understood, and html and CSS, it is possible to use JavaScript normally.

There are many 'frameworks' like JQuery. These are libraries of JavaScript code. It is a good plan to learn the language before trying to use libraries in that language.

Security and the sand-box

JavaScript code is usually embedded in a web page and executes when the user downloads it from the web. The user has no idea what the code is doing.

This is therefore a potential security risk. Suppose the code reads all the personal data stored on the user’s device and sends it somewhere? Or deletes everything on the user’s device?

To prevent this, JavaScript in a browser runs in a restricted sand-box. It cannot read or write anything on the user’s machine, on any drive or in the memory of other processes. An exception to this is html5 ‘local storage’, but this is restricted to storage for that particular domain, and it cannot send any data from it anywhere elsewhere.

This is not true for JavaScript executing outside the browser. For example, node.js can read and write local files.

Basic ideas

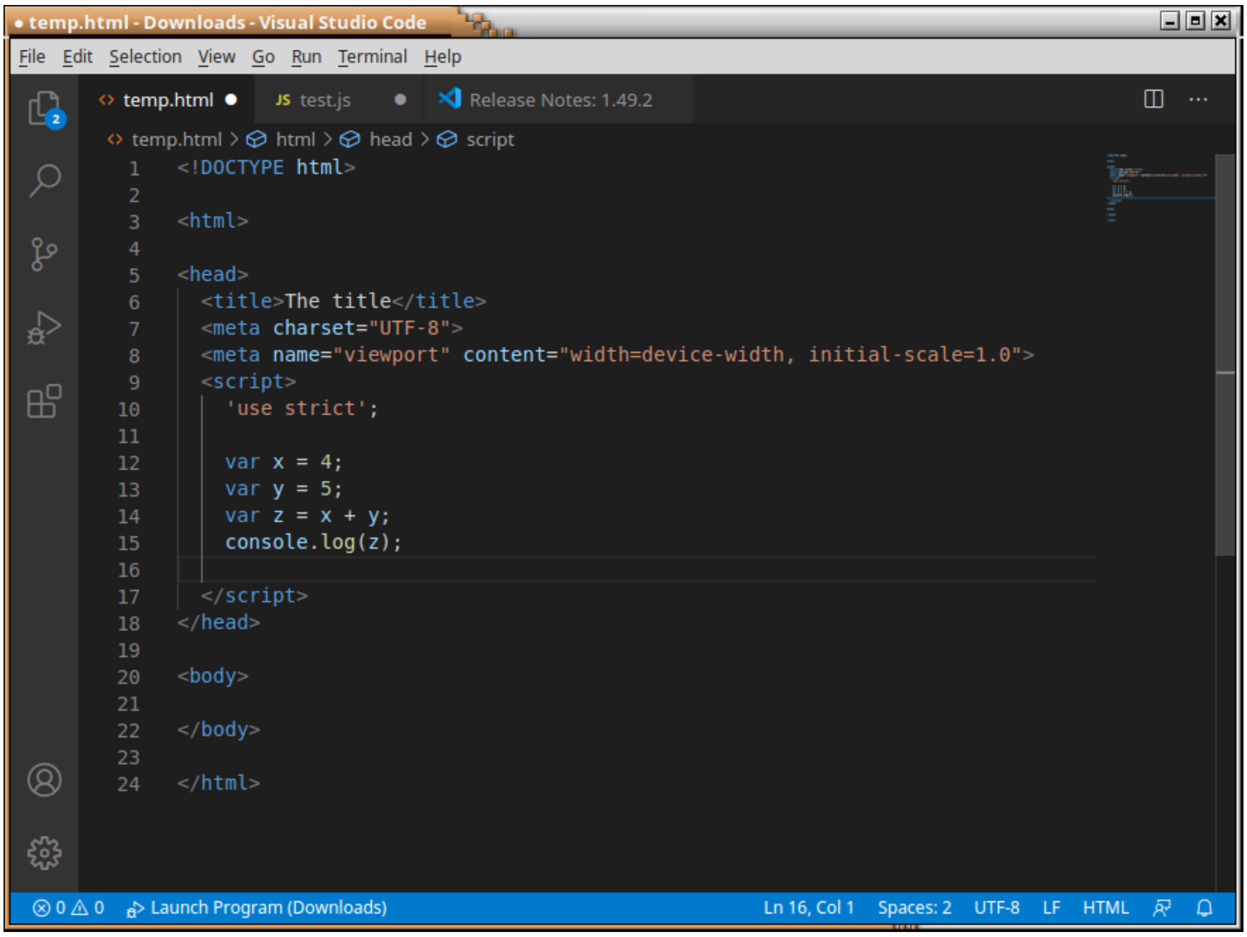
How to write JavaScript practically

Usually JavaScript code is executed by the interpreter in a web browser, and it is embedded in a web page. So you need to:

- Use a text editor to write a web page with JavaScript in it, and save it in a file named something.html

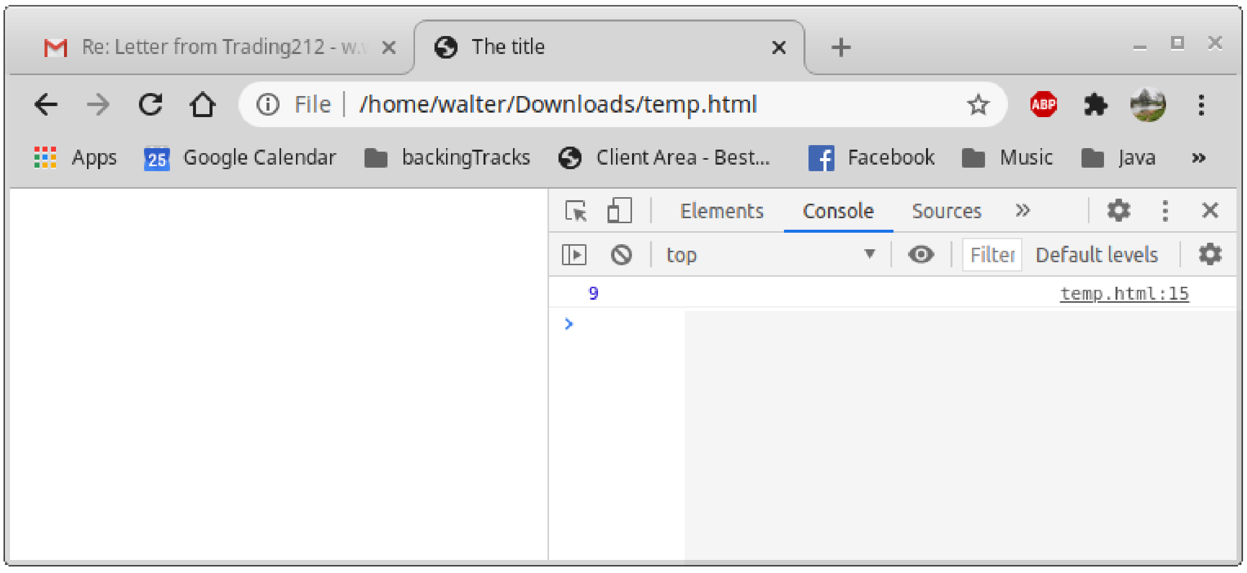
- In a web browser, open the file something.html locally (Ctrl-O)
- Output from the JavaScript is usually shown in the html of the web page. But to start with, we can get simple output to the console using console.log. So you must tell the browser to display console output

For example, here is a web page being edited in Visual Studio:


• temp.html - Downloads - Visual Studio Code
File Edit Selection View Go Run Terminal Help
temp.html
test.js
Release Notes: 1.49.2
temp.html > html > head > script
0 0
Launch Program (Downloads)
Ln 16, Col 1 Spaces: 2 UTF-8 LF HTML

```
1   <!DOCTYPE html>
2
3   <html>
4
5   <head>
6     <title>The title</title>
7     <meta charset="UTF-8">
8     <meta name="viewport" content="width=device-width, initial-scale=1.0">
9     <script>
10      'use strict';
11
12      var x = 4;
13      var y = 5;
14      var z = x + y;
15      console.log(z);
16
17    </script>
18  </head>
19
20  <body>
21
22  </body>
23
24  </html>
```

The JavaScript is lines 10 to 15. The rest is html for the web page. This is saved as temp.html



9 temp.html:15

Then this is opened locally in Chrome:

In Chrome the console is opened by clicking the 3 dots top right, then More tools.. Developers tools.

Do not attempt to type out that file. Copy-paste this:

```
<!DOCTYPE html>

<html>

<head>
  <title>The title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <script>
    'use strict';

    var x = 4;
    var y = 5;
    var z = x + y;
    console.log(z);

  </script>
</head>

<body>

</body>

</html>
```

Most of this is html – hypertext markup language. This is made of elements, which start and end with tags, like <head>..</head> and

<title>..</title>. One of these tags is <script>, enclosing JavaScript code. When the browser sees this, it executes it using the interpreter. We will usually just list the JavaScript, like

```
    'use strict';

    var x = 4;
    var y = 5;
    var z = x + y;
    console.log(z);
```

but in fact this needs to be embedded in the html web page.

Using external script files

We can keep the html web page, and JavaScript code, in separate file, like:

```
<!DOCTYPE html>

<html>

<head>
  <title>The title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <script src="myscript.js"></script>
</head>

<body>
.. web page here..
</body>

</html>
```

Then myscript.js just contains the JavaScript code in a separate file. This has the big advantage that we can use same code in many pages. But to keep things simple, here we embed the code in the web page.

Variables

A variable is

- some data held in memory as a program executes,
- which has a name, and

- usually changes in value during execution

  In our example script we have three variables, named x y and z. These variable names are examples of identifiers – names chosen by the programmer. Identifier names should be chosen to suggest the purpose or idea of it, or say clearly what it does. In this example the x y and z are just simple examples, and represent nothing except three data values.

Assignment statements

Programs mostly consist of statements, which are usually either instructions to the computer to do something, or supply some information needed.

In many languages (Java, C, C++, JavaScript) statements should be separated by a semi-colon – a ;

In JavaScript, if the programmer misses out the ; the interpreter will in effect insert it automatically. This is an unfortunate feature, since it encourages a habit which will not work in other languages.

One type of statement is an assignment, such as

```
var x=4;
```

This assigns the value 4 to the variable x. The direction of data movement is right to left – the 4 moves to x. Any previous value of x is lost.

In an expression, the terms left-hand and right hand side are used. In this example 4 is the right-hand side and x is the left. There are rules about what can appear on the left and right. These are mostly logical. For example

```
var 4 = x;
```

would mean take the value of x, and store it at 4. In other words take the value of 4 and change it to x. But this makes no sense, since we cannot change the value of 4.

Expressions

Often the right hand side is an *expression* – something for the interpreter to calculate the value of. For example in

```
var z = x+y;
```

x+y is an expression. The interpreter will calculate the value of this, at run-time.

An expression consists of *operators* and *operands*. An operator is something to do. Common ones are + for add, - for subtract, * for multiply and / for divide. The precedence rules – which to do first – are the same as in normal maths – sometimes called BODMAS. So for example 3*4+5 is 12+5 = 17, and not 3*9 = 27. We can use round brackets, so 3*(4+5) is 27.

We should not write assignments like

```
var z=3*(4+5);
```

but instead should write

```
var z=27;
```

The first version just wastes time and computation.

Input and output

Output means sending the results of a program somewhere. For JavaScript, that often means changing a web page and display a result that way – maybe as a number, but maybe changing the colour of something, altering page layout, moving to a new page location or whatever. Output might go to a printer, or to a file on local storage, or cloud storage or whatever.

JavaScript output often uses the DOM. For simplicity we will start by outputting values to the console, by statements like

```
console.log(x,y,z);
```

so you need the console open in the browser.

Input includes

- The user typing something in at runtime
- A mouse click or mouse movement
- A touch on a touch screen
- A temperature read from a sensor

- A value read in from a file

and so on.

Usually input in JavaScript comes through the DOM.

Input and output is usually referred to as I/O.

Conditional statements

A conditional statement executes another statement only if an expression is true. This is often called an *if statement*. Like this:

```
<!DOCTYPE html>

<html>

<head>
  <title>The title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <script>
    'use strict';

    var x = 4;
    var y = 5;
    if (x>y)
    {
      console.log("x is bigger");
    }
    else{
      console.log("x is smaller")
    }

  </script>
</head>

<body>

</body>

</html>
```

Check:

- The (round) brackets in if (x>y)
- No semi-colon after the if
- The curly brackets { and }. This makes a block, and we can have more than 1 statement, of any type, in the block

- The else part is optional.

= and == and ===

Do not use = to compare values. For example:

```
var x = 4;
var y = 5;
if (x=7)
{
console.log(x);
}
```

x=7 does not test if x is equal to 7. It is an *assignment*, to make x 7.

So this always prints out 7.

Better is to use ==

```
        var x = 4;
        var y = 5;
        if (x==7)
        {
        console.log(x);
        console.log("x is 7");
        }
        else
        {
        console.log(x);
        console.log("x is not 7");
        }
```

But JavaScript is unusual in that it uses *type coercion*

```
        if (4=='3+1')
        {
        console.log("same");
        }
        else
        {
        console.log("different");
        }
```

This means the interpreter will sometimes change the type of operands to make them comparable. In this example, the type of 4 is number, and '3+1' is a string, but type coercion converts them to the same type, and decides they are the same.

=== is better:

```
        if (4==='3+1')
        {
```

```
        console.log("same");
        }
        else
        {
        console.log("different");
        }
```

=== compares type *and* value, with no type coercion, so it outputs 'different'. Usually === is the one to use.

Loops

A loop is used to repeat code. This is often called iteration.

JavaScript has 5 types of loops, and 2 statements relating to loops

while

There is a *loop header* which starts 'while.' and this is followed by the *loop body*, which is the statements which are repeated. The loop body is often a block, enclosed in { curly brackets }.

After while there is a boolean expresion in (round brackets). The loop body repeats so long as the expression is true.

For example:

```
    var counter = 0;
    while (counter < 5) {
      console.log(counter, "Hello");
      counter++;
    }
```

It could be the expression is false the first time in, in which case the loop body will not be executed, even once:

```
    var counter = 7;
    while (counter < 5) {
      console.log(counter, "Hello");
      counter++;
    }
```

do.. while

This is similar to a while loop, except the expression comes at the end of the loop body:

```
    var counter = 0;
    do  {
      console.log(counter, "Hello");
      counter++;
    }while (counter < 5)
```

As a result, a do..while will always execute at least once.

for loop

Most loops have 3 parts - startup, when to carry on repeating, and what to change every time.

A for loop header has these parts:
for (initialisation; when to continue; what to change)

For example

```
    for (var counter=0; counter<5; counter++)
    {
      console.log(counter, "Hello");
    }
```

so this starts with counter=0

Each time around, it does counter++

and it repeats so long as counter<5

for..in

JavaScript *objects* have a set of attributes, and a for..in loop lets us iterate over them:

```
    var person=[];
    person.name="Fred";
    person.passport=72354;
    person.phone="07747 827 100";

    for (var attribute in person)
    {
      console.log(attribute, person[attribute]);
```

This outputs:

name Fred

passport 72354

phone 07747 827 100

for of

This iterates over the values of an object (value not value names):

```
var person=["Fred", 72354, "07747 827 100"];

for (var attribute of person)
{
  console.log(attribute);
}
```

break and continue

break breaks out of and ends a loop. It usually comes with an if.

continue means 'continue looping, but skip the rest of the body this time':

```
var i = 0;
while (i < 10) {
  i++;
  console.log(i);
  if (i % 2 === 1) continue; // skip the next part
  console.log("Even");
  if (i === 8) break; // stop at 8

}
```

This outputs

```
1

2

Even

3

4

Even

5

6

Even

7

8

Even
```

Debugging

Debugging is the process of finding and correcting errors in code. The ability to debug is just as important as the ability to write program code.

There are three broad classes of errors.

The first are errors that break the rules of the language – that is, syntax errors. These should be detected by the compiler or interpreter.

The second are errors that occur at runtime. These might produce an exception,

The third are logic errors, sometimes called algorithm errors. The method being used in the program is incorrect, and does not do what the program is intended to do.

Strict mode

In the earliest days of the web, when Javascript was new, code was being produced by people who were new to programming, so errors were very common. So most interpreters would attempt to ignore syntax errors. This is non-strict mode. In strict mode, the interpreter will signal errors.

A consequence of non-strict mode means that errors can be difficult to find. For example, in non-strict mode variables do not need to be declared. But that means if you mis-type a variable name, a new variable just appears. For example

```
var singleton=false;
..
simpleton=true;
```

We meant to have a variable named singleton. But we mis-typed it as ‘simpleton’. Now we have another variable named simpleton, and singleton is still false. There is no error message, and it will take us an hour to spot the mistake.

You tell the interpreter to use strict mode by putting ‘use strict’ as the first line in a script. So

```
          <script >
             'use strict';
             x = 1;
             console.log(x);
         </script>
```

works as expected – x is not declared, so we get a syntax error.

But

```
         <script >
             x = 1;
             'use strict';
             console.log(x);
         </script>
```

in fact is not in strict mode, and we get no syntax error.

You can also set an individual function to be in strict mode. But a better option is to make the entire script be strict.

Modules (see later) are automatically in strict mode.

Debugging strategies

1. *Read the error message* – what the error is and where it is. Try to make sense of it.

2. Be clear what *should* happen. What is the correct output, and how does it differ from the actual output?

3. *Print debugging*. This means outputting important variables at key points in the code. For JavaScript, this probably means console.log

4. *Comment debugging*. The idea is to find which part of the code causes the error, without deleting it, then re-typing it. So you comment out sections of code /* like this */ If the error remains, that code was OK, and you can remove the commenting

5. Copy the error message and *Google it*.

6. Use a *debugger*. This is software which lets you do things like stepping through codeone statement at a time, adding 'watch variables' to display values, run to cursor which means run at full speed up to a point in the code, and so on. Not commonly available in JavaScript.

7. Does the code *actually execute*? You have some code which does not do something – is it because it simply never executes? Put a console.log or an alert just to see if that code executes.

Modularity

Managing scale - decomposition

One of the tasks of software engineering is managing scale. Dealing with 10 or a 100 lines of code is not a problem – but a million is.

The usual solution is some type of *decomposition* – splitting a problem, and the code which solves it, into smaller problems. This means that

- The problems are simpler and the code is smaller
- The code units can be tested separately
- They can be written by separate team members, at the same time in parallel, so the over project is shorter
- They can be re-used in other projects.

Modularity is usually achieved in JavaScript by using functions, described in this section.

More recently, *modules* have become available - see later.

JavaScript functions

How code is split depends on the language. In JavaScript the standard technique is to use a function. For example:

```
<!DOCTYPE html>

<html>
    <head>
        <title>The title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <script>
            function average (a,b)
            {
                var result = (a+b)/2;
                return result;
            }

            var x=3;
            var y=4;
            var z=average(x,y);
            console.log(z); // outputs 3.5
        </script>
    </head>
    <body>

    </body>
</html>
```

Control flow

In other words, what happens when? Execution does not start at the function definition, which is

```
            function average (a,b)
            {
                var result = (a+b)/2;
                return result;
            }
```

It starts at the first code *not* in a function, which is

```
            var x=3;
```

Then at

```
            var z=average(x,y);
```

the function is *called*. That makes execution switch to the start of the function, and it carries on until it hits the return. The return instruction means ‘return to where you were called from’. Finally it does

```
console.log(z);
```

Data flow into the function

Data values are passed into a function through *parameters* – sometimes called arguments. So in

```
            function average (a,b)
```

the function named average has two formal parameters, named a and b. When the function is called, at

```
var z=average(x,y);
```

the actual parameters are x and y. These are matched up in order So the value of x (3) is copied to a, and the value of y (5) is copied to b. So inside the function,

```
                var result = (a+b)/2;
```

result will work out to be 3.5.

Data flow out

A value can flow out using the return statement. So

```
                return result;
```

does 2 things. It returns control flow to where the function was called from, and it passes back there its value – in this case, 3.5. Then in

```
var z=average(x,y);
```

the returned value is assigned to variable z.

We can call a function many times, such as:

```
            var x=3;
            var y=4;
            var z=average(x,y);
            console.log(z);
            z=average(8,9);
            var another = average(z,x);
```

control and data flow works as described.

Pass by value

Look at this code:

```
            function average (a,b)
            {
                var result = (a+b)/2;
                a=99;
                return result;
            }

            var x=3;
            var y=4;
            var z=average(x,y);
            console.log(x, z); // outputs 3, 3.5
```

Here the value of the parameter a is changed inside the function. But this does not alter the matching actual parameter, x. Why not?

These are numbers, and in JavaScript, numbers are primitives. And primitives are 'passed by value'. That means a *copy* is passed to the function. So a=99 alters the *copy* – it does not change the original, x.

Scope

Look:

```
            var x=3;
            var y=4;

            function average (a,b)
            {
                var x;
                var result = (a+b)/2;
                x=99;
                return result;
            }


            var z=average(x,y);
            console.log(x, z);
```

We declare 2 variables named x. We assign 3 to one, and 99 to the other. But the x inside the function does not change the x outside the function.

Scope means the part of a code sequence where a name has the same meaning – if any (if it is not declared, it has no meaning).

In JavaScript there are 2 scopes – inside a function, which is called *local*, and outside, which is called *global.*

In this example we have a local x and a global x. They have the same name – but no connection. Assigning to one does not alter the other.

This is good, because if we are writing a function and want to use a local variable, we do not need to worry whether we have used the same variable name elsewhere. Even if we have, local variables will not alter each other.

Each function has its own, separate, scope:

```
            var x=3;
            var y=4;

            function average (a,b)
            {
                var x;
                var result = (a+b)/2;
                x=99;
                return result;
            }

            function another()
            {
                y=100; // the global y
                var x=89; // a different local x
            }


            var z=average(x,y);
            another(); // execute another
            console.log(x, y, z); // 3,100,3.5
```

hoisting

Variable declarations are treated as if they are 'hoisted' to the top of the code.

For example:

```
'use strict'

x=2;
```

fails, because we did not declare , and we would usually say:

```
'use strict'
var x;
x=2;
```

declaring x before we use it. But we can also say:

```
'use strict'
var x;
x=2;
```

because the var x is in effect hoisted to the top of the code.

var let and const

In strict mode we must declare variables. There are 3 ways to do that, using var let or const. This summarises the differences:

	scope	hoisted?	can be re-declared?	can be re-assigned?
var	local to function, or global	yes	yes	yes
let	local to block	no	no	yes
const	local to block	no	no	no

For example:

```
'use strict'
x=3;
var x; // hoisting happens
var x; // re-declaration OK
x=4;
function f()
{
  var x; // is local
  x=8;

}
f();
console.log(x); // 4
for (var i =0; i<10; i++)
{

  var x=9; // not local
}
console.log(x); // 9
```

compared with let:

```
// x=3; 'Cannot access 'x' before initialization'
let x; // so no hoisting
// let x; re-declaration not allowed
x=4;
function f()
{
  let x; // is local
  x=8;
   console.log(x); // 8
}
f();
console.log(x); // 4
for (var i =0; i<10; i++)
{

  let x=9; // local - block scope
}
console.log(x); // 4
```

and const:

```
'use strict'
// console.log(x); No hoisting
// const x;  Missing initializer in const declaration"
```

```
const x = 4; // OK
// x=4; cannot assign to constant
function f()
{
 const x=8; // is local

  console.log(x); // 8
}
f();
console.log(x); // 4
for (var i =0; i<10; i++)
{

 const x=9; // local - block scope
}
console.log(x); // 4
```

This is on Chrome 88.

A reasonable plan seems to be to use let for variables and const for constants.

Standards

A computing standard is a set of rules, ideas and definitions which many people agree with. Standards are agreed by committees working in organisations such as the ISO, IEEE, W3C and IETF. Standards are mostly open - everyone can access them, and chose to agree with them, or not (a closed standard is made up by one company). The idea is that the pressure to agree comes from the fact that products do not work together unless they use an open standard.

As an example, a web browser which does not support the current html5 standard would not display web pages correctly, and no-one would use it.

A language is a set of rules and definitions. Most languages, such as C, C++ and Java, have open standards.

For JavaScript, the standard is set by an organisation named *ECMA*. Versions are released every few years. The 2020 version is here.

How to get more help

You can Google or search youtube, but this is a bad idea. JavaScript can seem like a simple language, but it is not. There are many things on the web about JavaScript written by people who have little understanding of it. Many are mis-leading or simply incorrect. Be wary of simply becoming more confused. Here are some suggestions for checking your understanding

Write and run your own code snippets	You get to see actual output. But it might be different with different browsers or non-browser interpreters and so on.
Current ECMAScript standard	This sets out the actual rules of the language. But it is not a tutorial and does not try to teach the ideas, so for reference only. Also most browsers will not keep up with this latest standard
Previous language standards	5.1, 2016, 2017, 2018, 2019
https://esdiscuss.org/	ECMAScript discussion archive
https://es.discourse.group/	ECMAScript standard community discussion
https://caniuse.com/	Guidance about what you feature you can use, in what browser, in what version
https://developer.mozilla.org/en-US/docs/Web/JavaScript	From Mozilla, MDN provides reference material and tutorials at a range of levels
https://developer.mozilla.org/en-US/docs/Web	Mozilla MDN material on many aspects of web technology
https://html.spec.whatwg.org/	Html5 – 'the living standard'
https://developer.mozilla.org/en-US/docs/Web/CSS	Mozilla MDN on CSS

Primitive types

Type

Examples of data types are integers (whole numbers), numbers with fractional parts, characters, dates, times, colours, and so on. We can do different things with different types – for example we can multiply numbers but we cannot multiply colours. Programming languages use the idea of *data type*.

Some languages extend the idea of type to include *function type* – C does this. Other languages treat functions as data – JavaScript does this.

Strict typing

In some languages, variables have a type, and cannot change. In Java for example we say things like

int x;

which means the type of the variable x is an integer. These languages are strictly typed.

In JavaScript variables do not have type.

But values have type.

So we can change a variable's value, and so change the type:

```
        <script>
            var x;
            x=8;
            console.log(typeof x); // number
            x="Hello";
            console.log(typeof x); // string

        </script>
```

Primitive types

JavaScript has 6 *primitive types* : number, string, boolean, bigint, undefined and symbol. The primitive types are just values.

It also has some *object types*. There are some built-in objects, and the programmer can define others. An object has data parts, called fields, and code parts, called methods.

For each primitive type (like number) there is a matching wrapper type (like Number) which is an object. JavaScript is case sensitive, so number and Number are different.

Number

This primitive type is a number, positive or negative, whole number or not:

```
            var x;
            x=8;
            console.log(typeof x); // number - primitive
            x=new Number(8);
            console.log(typeof x); // object
            console.log(x instanceof Number);  // true
```

The Number object is said to be a wrapper object – it wraps a primitive value of type number, and adds relevant methods. The Number and other objects is examined later.

The JavaScript number primitive is held in 64 bits floating point IEEE 754 format. That means they are always stored in 64 bits, always have fractional parts, and are held in a format described in the IEEE 754 standard.

Strings

A string is a sequence of 1 or more characters. And a character is a like a b c A B C * ^ $ Σ п ± Ÿ and so on. We can 'add' strings by concatenating them – just joining them together:

```
            var x;
            x="A";
            x=x+1;
            console.log(x); // x1
            console.log(typeof x); // string
            x=x+" a b c d";
            console.log(x); // x1 a b c d
```

Note a *space character* is just another character. So "a b" is a string of 3 characters.

JavaScript uses *Unicode*:

```
            var x;
            x="αβγ ∈∃ℂ";
            console.log(x); // αβγ ∈∃ℂ
```

Boolean

The boolean type has just 2 possible values – true and false:

```
            var x;
            x=( 5 === 6);
            console.log(x); // false
```

BigInt

A BigInt primitive is whole number which can be indefinitely large. A BigInt literal – an actual value – is written with an n after it. So for example

```
            var x;
            x= 10n**300n;
            console.log(x);
```

That makes x to be 10^{300}, when the largest safe integer is about 10^{15} – safe in the sense that calculations with it will not go wrong.

BigInt may be slower than a number.

BigInt is a fairly new type, and old browsers will not understand it.

Undefined

The undefined type has just one value : undefined. This is the value variables have if they have not been assigned a value:

```
            var x;
            console.log(x); // undefined
```

Objects and protoypes

In computer science an object is a bundle (or a packet or group or bunch or collection) of data values and code. Usually objects have names, as identifiers, and are held in memory during program execution.

JavaScript is an object-oriented programming language (OOP) and can use an objected-oriented paradigm. That means it has a syntax for dealing with objects, and JavaScript code is often about objects.

ES6 introduced the keyword ‘class’. This section is about objects and prototypes, which are a key idea about how JavaScript objects work. The next section covers classes, which are much easier to use.

How to make an object

There are several ways to make a new object in JavaScript. Here is one way:

```
        var person=
        {
        name: "John",
        dateOfBirth: "1.3.94",
        payrollNumber: 423
        };
```

This declares a variable, named person. But this is not just one data value. It has three fields, named name, dateOfBirth and payrollNumber. These fields are string and number primitives.

This is called an *object literal*.

Representing a date as a string is not a good method – we do it here to keep things simple.

The object.property dot notation

We refer to one of the fields within an object by saying what the name of the object is, then a dot (.) then the field name. For example

```
        console.log(person.name);     // John
        console.log(person.payrollNumber);  // 423
```

What is the point of OOP?

OOP is not essential. Complete operating systems, like Unix, have been written without objects.

But OOP has several advantages. One is that we often want to code about things which have several aspects. The above example shows this – one employee has a name, a date of birth, a payroll number and so on. Primitive types can handle these as separate values, but an object can bring these together as a single thing.

Methods

An object is a bundle of data values and code. The code parts are called methods. We can tell an object to execute one of its methods. For example

```
        var person=
        {
        name: "John",
        dateOfBirth: "1.3.94",
        payrollNumber: 423,
        toString: function()
          {
          return "person: "+this.name+"\nDOB "+this.dateOfBirth+
          "\nPayroll number "+this.payrollNumber;
          }
        };
```

Now we have 3 data fields, plus another field named toString. This is different in that it is defined as a function. The syntax is similar to that used in our function definitions before, except that we are not giving it a name as function toString()... Instead we are saying toString: function()

Within the function we use ‘this’ to refer to the object which will execute it. So this.name means the name field of the object carrying out the toString method.

We can tell the person object to do its toString method like this:

```
console.log(person.toString());
```

and this outputs

```
person: John
DOB 1.3.94
Payroll number 423
```

Objects often have a toString method, to produce a string version of the object for output and debugging purposes.

Objects as associative arrays

In computer science, an *associative array* is a set of key-value pairs. That means we have some data values, and each value is linked with some key. We can use the key to access the value. This is sometimes called a *map* – it connects each key with some value.

We can do that as follows in JavaScript:

```
        var person=[];  // create an empty associative array
        // then add fields:
        person['name']= "John";
        person['dateOfBirth']= "1.3.94";
        person['payrollNumber']=423;
        // access with dot notation..
        console.log(person.name);
        // or as associative array..
        console.log(person['name']);
```

The key must be a data value, or an identifier with a suitable value:

```
        var person=[];
        person['payrollNumber']= 123;
        // or
        var fieldName='payrollNumber';
        person[fieldName]=123;
        // but not
        person[payrollNumber]= 123; // unless payrollNumber is a
variable with a value
```

Functions are objects

In JavaScript, *a function is an object*. Function objects only differ from other objects in that they can be called:

```
        function MyFunction(a,b)
        {
        return a+b;
        }

        console.log(MyFunction(2,3)); // 5 - call the function
        MyFunction.x=27; // use it as an object
        console.log(MyFunction.x); // 27
```

If we invoke the function using the keyword new, this constructs a new object, as described next.

Patterns of objects – using constructors

The ways of making objects described above is fine for ‘one of a kind’ situations, where we only want one object of a type. But we often want several objects with the same structure – the same set of fields (but not values) and the same set of values. That means executing a similar sequence of instructions, repeatedly. That is what a function does.

We can do this using a constructor. This is a function, but invoked using the keyword new, instead of the usual function call.

In that situation the constructor creates a *new object*, referring to it as *this*. Fields can then be created on that object and values assigned to the fields. At the function end, the *this* object is returned:

```
function Person(n,dob, pn)
        {
        this.name=n;
        this.dateOfBirth=dob;
        this.payrollNumber=pn;
        }

        var person1=new Person("John","1.2.99",1234);
        var person2 =new Person("June","2.2.99",1235);
        var anotherPerson = new Person("Jane","3.2.99",1236);
```

We can include methods as well as data fields:

```
        function Person(n,dob, pn)
        {
        this.name=n;
        ..
        this.toString = function()
          {
          return "person: "+this.name+"\nDOB "+this.dateOfBirth+
          "\nPayroll number "+this.payrollNumber;
          }
        }

        ..
        var anotherPerson = new Person("Jane","3.2.99",1236);
        console.log(anotherPerson.toString());
```

Object property attributes

A JavaScript object is a collection of properties. These properties can have *attributes*.

We can control these directly using Object.defineProperty:

```
        function MyObject() { // constructor

            Object.defineProperty(this, 'b', {
                value: "Hello",
                writable: true, // we can change its value
                enumerable: false, // does not appear in a for..in
                configurable: true // can be deleted
            });
            Object.defineProperty(this, 'c', {
                value: 21,
                writable: false, // we cannot change its value
                enumerable: true, // does  appear in a for..in
                configurable: true // cannot be deleted
            });



            }

        var obj1 = new MyObject();
        console.log(obj1.b); // Hello
        obj1.b="bye";
        var p;
        for (p in obj1) // c, not b
        console.log(p);
        delete(obj1.c); // OK
```

An object can have two kinds of properties. A *data property* is as above, and can have attributes of value, writable, enumerable and configurable,

The other type of property is an *accessor property*. These would be called accessor methods in Java. A accessor property is a getter or setter function, (or both getter and setter). A setter function has one argument, and is used to set some value. A getter function has no arguments, and is used to read some value:

```
        function MyObject() {
            ..
            Object.defineProperty(this, 'accessA', {
                set(x) { // setter function
                    this.a = x;
                },
                get() { return this.a; } // getter

            })


            }

        var obj1 = new MyObject();

        obj1.accessA=77; // call the setter
        console.log(obj1.accessA); // call the getter
```

The constructor property

All JavaScript objects have a property named *constructor*. If the object has been made using the *new* syntax, the value of the constructor property is as expected:

```
        function CTR() {
            this.x = 99;
        }

        var obj = new CTR();
        console.log(obj.x); // 99
        console.log(obj.constructor); // ƒ CTR() { this.x=99; }
```

So the value of the constructor property of obj is CTR, which is a function.

Now a function is itself an object. So it in turn must have a constructor property:

```
        var obj = new CTR();
        console.log(obj.x); // 99
        console.log(obj.constructor); // ƒ CTR() { this.x=99; }
        console.log(obj.constructor.constructor); // ƒ Function()
{ [native code] }
```

So the constructor of CTR is Function. This is as if we had said:

```

var CTR = new Function..
```

which is the effect of the actual

```
        function CTR() {
```

We can do this explicitly, as:

```
        var CTR = new Function( 'this.x = 99;' );

        var obj = new CTR();
        console.log(obj.x); // 99
        console.log(obj.constructor);  // ƒ anonymous( ) { this.x =
99; }
```

Now *Function* is an intrinsic object, which can be used to construct functions, and other features related to functions. As an intrinsic object, it is there when the interpreter starts. It was not made by other JavaScript code, and this is why it says Function() { [native code] }

If we make an object using the object literal syntax, it still has a constructor property:

```
        var objLit =
        {
            x: 98
        }
        console.log(objLit.x); // 98
        console.log(objLit.constructor); // ƒ Object() { [native
code] }
```

so the constructor is Object, which is an intrinsic object.

Inheritance

In OOP the idea of inheritance is to avoid coding every new type of object from nothing. Instead we try to use an existing object as the basis for a new type. The idea is to re-use the fields in the old type in the new type. We might change them, or add extra fields as well.

JavaScript does this using the idea of a prototype. An object can have another object as its prototype. Then properties of the protoype are inherited by the other object.

As an example, we will have an object with a field a, and this will be the prototype of another object with a field b, and also the inherited field a:

```
        function Base() {
            this.a = 101;
        }

        function Sub() {
            this.b = 99;
            this.__proto__ = new Base();
        }

        var obj = new Sub();
        console.log(obj.a); // 101
        console.log(obj.b); // 99
```

JavaScript objects have a property __proto__ (caution -see below). The line

```
this.__proto__ = new Base();
```

sets that to be an object with Base as its constructor. So the a field from Base is inherited by the Sub object.

Each Sub object has its own, separate, a field:

```
        var obj = new Sub();
        var obj2=new Sub();
        obj2.a=200;
        console.log(obj2.a); // 200
        console.log(obj.a);  // 101 - the default value
```

obj2 has its own a field, different from obj's a field.

The name __proto__ for the property is a problem. Currently all browsers understand it (old Internet Explorer does not), but it is not part of the ECMAStandard, and it may stop working. A better alternative is Object.setPrototypeOf:

```
        function Base() {
            this.a = 101;
        }

        function Sub() {
            this.b = 99;
            Object.setPrototypeOf(this,  new Base());
        }
```

and a corresponding Object.getPrototypeOf.

The base object may itself have a prototype, so we have the idea of a *prototype chain*:

```
        function BaseBase() {
            this.q = 300;
        }

        function Base() {
            this.a = 101;
            Object.setPrototypeOf(this, new BaseBase());
        }

        function Sub() {
            this.b = 99;
            Object.setPrototypeOf(this, new Base());
        }

        var obj = new Sub();
        console.log(obj.q);  // 300
```

Constructor prototype property

Some objects – those which are functions, and so can be used as constructors using new, have a property named prototype. Confusingly, .prototype is different from .__proto__

```
         function Ctr() {
            this.b = 99;
        }

        var obj = new Ctr();
        console.log(obj.prototype); // undefined
        console.log(obj.__proto__); // {constructor: ƒ}
        console.log(Ctr.prototype); // {constructor: ƒ}
```

The idea of .prototype is that it is an object, belonging to a constructor, from which all its fields are inherited by an object constructed by that constructor. For example:

```
         function Ctr() {
            this.b = 99;

        }

        // give Ctr.prototype a function, named func, which sets b
        Ctr.prototype.func = new Function("x", "this.b=x");

        var obj = new Ctr(); //b will be 99
        obj.func(9); // call function from prototype, set b to 9
        console.log(obj.b); // 9
```

Encapsulation

One idea in OOP is that of encapsulation. The idea is that an object bundles together code and data, and access to the data is controlled, to make sure the data in the object is always valid and correct and that the object cannot ‘go wrong’.

This was intended to solve a problem in structured programming, where there would usually be a set of global data, and this would be accessed by code in functions. The problem was there was no way to control what code could alter what data. A bug in any one function could corrupt the global data, which might make other functions fail. Testing was then very difficult.

The OOP solution was to encapsulate all data into separate objects, so all data was a field in some object. And it was not directly accessible – in Java terms, it would be private. It can only be accessed through public getter and setter methods. And the setter methods would validate any data changes, to ensure the data could never 'go wrong'.

This is what JavaScript accessor properties offer. For example, suppose the value of 'a' in our object cannot be negative. We just need an if in the setter:

```
     function MyObject() {
           ..
           Object.defineProperty(this, 'accessA', {
               set(x) { // setter function
                   if (x>0) this.a = x;
               },
               get() { return this.a; } // getter

           })


           }
```

References

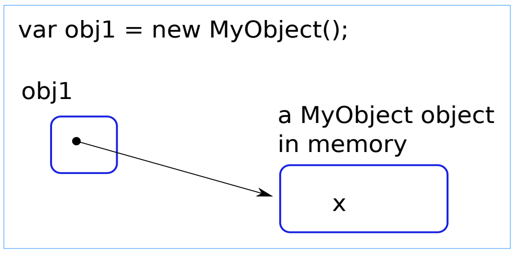
Why does changing one object alter another object?

```
        function MyObject() {

            this.x=2;

            }

        var obj1 = new MyObject();
        var obj2 = obj1;
        obj1.x=9;

        console.log(obj2.x); // 9
```

and if JavaScript is pass by value, how come this works?

```
        function swap(a, b) {
            var t = a.x;
            a.x = b.x;
            b.x = t;
        }
        function MyObject(v) {
            this.x = v;
        }

        var obj1 = new MyObject(1);
        var obj2 = new MyObject(2)
        swap(obj1, obj2);

        console.log(obj1.x, obj2.x); // 2  1
```

The answer is that if a variable names an object, the variable is a *reference* to that object. In effect, the variable is a pointer to the object, like this:



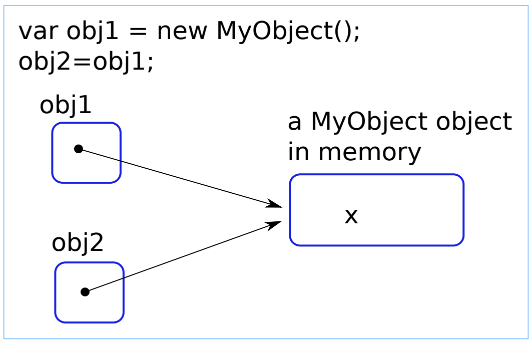
obj1 is a pointer to the object. We can think of this as being the address in RAM of where the object is held.

If we say

```
        var obj1 = new MyObject();
        var obj2 = obj1;
```

we have only said 'new' once, so we only have *one object*. We have two pointers, obj1 and obj2, which point to the *same object*.

So if we say obj1.x=9, this changes obj2.x – because they are the *same object*.



JavaScript is pass by value. This means when a parameter is passed to a function, a copy is passed. But when the parameter is an object, a *copy* of a *pointer to* the object is being passed.

So the function can follow that pointer and alter what is in the object.

So here:

```
function swap(a, b) {
            var t = a.x;
            a.x = b.x;
            b.x = t;
        }
```

the a and b are copies of pointers to two objects. The t=a.x follows one pointer and gets the x field in that object. Then b.x=t follows the b pointer, and writes that x value into the object that b points to.

In the calling code

```
        var obj1 = new MyObject(1);
        var obj2 = new MyObject(2)
        swap(obj1, obj2);
```

we have used new twice, so we just have 2 objects (not 4). And the swap function has exchanged their x fields.

If we had said

```
function swap(a, b) {
            t = a;
            a = b;
            b = t;
        }
```

it would have had no effect. The a and b are copies of pointers. Changing those copies (instead of the objects they point at) has no effect.

Static fields

In OOP, a static field is per object *type*, not *per object*.

So it is a single value shared between all objects of that type. Instead of all objects having their own value, there is just one static value belonging to the type. In JavaScript we can do this using a field of the constructor function object. A constructor is a function, and a function is an object, so it can have fields.

An example of the use of a static field is in a version of the auto-number feature of some databases:

```
        Person.lastID=0; // static field initialisation – a field of
the function object

        function Person(n,dob) // payrollNumber is auto-assigned
        {

        this.name=n;
        this.dateOfBirth=dob;
        this.payrollNumber=Person.lastID; // current last ID
        Person.lastID++; // move static field onto next ID
        this.toString = function()
          {
          return "person: "+this.name+"\nDOB "+this.dateOfBirth+
          "\nPayroll number "+this.payrollNumber;
          }
        }

        var person1=new Person("John","1.2.99");
        var person2 =new Person("June","2.2.99");
        var anotherPerson = new Person("Jane","3.2.99");
        console.log(anotherPerson.toString());
```

The anotherPerson is

```
person: Jane
DOB 3.2.99
Payroll number 2
```

The built-in objects and the Object object

JavaScript has a set of built-in objects, which already exist when the interpreter starts. They include:

Object and Function : the fundamental objects concerned with how OOP works

Number, BigInt, Boolean, String and Symbol : the wrapper classes that *wrap* primitive values. That means they have a primitive value as a property, and have methods useful in connection with that type

Math, Date, RegExp, JSON : useful utility objects

Array, Set and Map : basic data structures

The complete reference for all built-in objects is at MDN.

Object is one of those pre-built objects It provides several things useful for manipulating objects, and since JavaScript is an OOP language, is key:

Constructor

Creates a new empty object:

```
    var obj = new Object();
    obj.f = 99;
    console.log(obj); // {f: 99}
```

Other methods include:

Methods	
Object.assign(to, from);	copies enumerable properties from one object to another
Object.defineProperty(..)	add a property to an object, with given attributes
Object.create	Create an object with a given prototype
Object.freeze	Prevent an object from being changed
Object.getPrototypeOf	
Object.is	Compares two objects (almost like ===)
Object.keys	Return an array of enumerable own property names
Prototype methods	
toString	Return a string representation
valueOf	For a wrapper class, returns the value wrapped

The global object

When the interpreter starts, there is a global object in scope. A 'this' in global scope refers to it. Anything you declare, like

```
var x;
```

in fact becomes a property of that global object.

In a browser, the global object is named window. Th eidentifier 'globalThis' refers to the global object in any environment:

```
console.log(this); // Window
console.log(globalThis); // Window
var x=88;
console.log(globalThis.x); // 88
console.log(globalThis.Date); // ƒ Date() { [native code] }
console.log(Window===window); // false
console.log(window instanceof Window); // true
// var window = new Window(); no - Illegal constructor
var window;
console.log(window); // Window {parent: Window, opener: null, top: 
Window, length: 0, frames: Window, …}
```

Function objects and concepts

We can think about functions in two different ways – as code, or data:

1. In C and Pascal and Basic and Algol and Fortran, a function is a code unit. It either just does something (a procedure in Pascal, or a void function in C), or it returns a computed value. A function is a code block which executes. In JavaScript we often have functions which executes when te user clicks a button, for example.

2. Or, a function is a piece of data, like a number or a string. Code can input, process, store and retrieve, and output data – so it can do all of that with functions. Th eonly special thing about a function as a data type is that it can also be executed. Languages which can handle functions as data which can be manipulated are known as functional languages. JavaScript has an object named Function, which supports this.

Standard function definition and call

Like this:

```
function max(a,b)
{
  return a>b? a : b;
}

var bigger = max(7,9);
```

Functions as data objects

For example:

```
function max(a, b) {
  console.log(max.name); // max
  console.log(max.length); // 2 (number of arguments expected)

  return a > b ? a : b;
}

var x = max(8, 9); // calling the function

console.log(max instanceof Function); // true
var code = max.toString();
console.log(code); // as above!
```

Here max is a function – so it is an object, a bunch of data and methods, in an inheritance chain. Functions inherit from the built-in object Function, and this has properties of name and length, as shown above.

So max is an object, an instance of Function. From Function it inherits the toString()method, which produces the code of the function

Function expressions

An expression is code which evaluates to something. A function expression is code which evaluates to a function. This could be assigned to an identifier, and then called, like:

```
var bigger = function (a,b) { return a>b? a : b; };

var m = bigger(7,9);
```

Anonymous functions

If we say

var x=9;

we have an identifier (x) to which is assigned a value (9). A function expression evaluates to a value – a function object – and sometimes we do not need to assign that to an identifier. So it does not have a name. For example

```
function map(f,values) // apply function f to the array values
{
  for (var index=0; index<values.length; index++)
  values[index]=f(values[index]);
}

var data = [1,2,3];
map(function(x) { return 2*x;}, data);
console.log(data); // [2, 4, 6]
```

Here `function(x) { return 2*x;}` is a data value, which is a Function object. But it does not have a name – it has not been assigned to an identifier. This is like x+y is an expression, and sometimes we do not need to assign it to a name to use it. Here we apply it, as a function, to every element of an array.

The Function object

Function is a pre-built object which relates to functions.

It can be used as a constructor. The arguments are an optional list of strings, which are the arguments, and the final one is the source code of the function. For example:

```
var addUp = new Function('data','var result=0; var p; for(p=0;
p<data.length; p++) result+=data[p]; return result; ');

var values=[1,2,3];
var total=addUp(values);
console.log(total); // 6
```

In this example there is no advantage over a conventional function definition. But it would be possible to construct the function programmatically, and then call it. So JavaScript code can construct code, and run the result.

If this is done on the basis of user input, this might be a security risk.

The Function object has properties name and length, and the prototype has methods apply, call, bind and toString.

The difference between apply and call is that the first takes an array of arguments, and the second a list:

```
    function max(a, b) {
      return a > b ? a : b;
    }

    console.log(max.apply(null,[5,4]));
    console.log(max.call(null,5,4));
```

Classes

JavaScript implements OOP using prototypes, as described in the previous section. But some OOP languages, such as Java and C++, use the idea of a *class*, as a *type of object*. ES5 introduced class into JavaScript, as syntactic sugar – that is, to make code look better, while keeping prototypes as the actual way OOP is implemented.

Class

A class is a *type of object*. It provides a template, a recipe or cookie cutter for making sets of objects of the same type:

```
        class Person {
            firstName;
            lastName;
        }
        var p1 = new Person;
        p1.firstName = 'John';
        p1.lastName = 'Smith';
        console.log(p1.firstName, p1.lastName);
        var p2 = new Person;
        p2.firstName = 'Jane';
        p2.lastName = 'Jones';
        console.log(p1.firstName, p1.lastName);
```

It is a convention to have class names starting Upper Case. This is to make it clearer which identifiers are classes and what are not.

Here the p1 object is an *instance* of the class Person.

Methods

As well as data fields, a class can also define *methods* bundled into an object:

```
        class Person {
            firstName;
            lastName;

            setFields(f, l) {
                this.firstName = f;
                this.lastName = l;
            }

        }

        var p1 = new Person;
        p1.setFields('John', 'Smith');
        console.log(p1.firstName, p1.lastName);
```

So now a Person instance knows how to do the setFields method.

In

```
            setFields(f, l) {
                this.firstName = f;
                this.lastName = l;
            }
```

The this is a refernce to teh object executing the method. So if we invoked it by

```
 p1.setFields('John', 'Smith');
```

then 'this' is a refernce to the p1 object. So this.firstName is teh firstName field of the p1 object.

A method can contain any code, not just assignment to fields.

Constructor

As in the previous section, a constructor is code used to initialize an object:

```
        class Person {
            firstName;
            lastName;

            constructor(f, l) {
                this.firstName = f;
                this.lastName = l;
            }

        }

        var p1 = new Person('John','Smith');
        console.log(p1.firstName, p1.lastName);
```

Constructors cannot be overloaded, meaning a class can only have one constructor (Unlike Java, where classes often have several constructors).

Private fields

By default, class fields are public, meaning they can be accessed from outside the class. As a result of accidental bugs, this can produce invalid data:

```
        class Person {
            firstName;
            lastName;

            constructor(f, l) {
                this.firstName = f;
                this.lastName = l;
            }
        }

        var p1 = new Person('John','Smith');
        p1.lastName=''; // no last name after this
        console.log(p1.firstName, p1.lastName);
```

We can prevent this by making data fields private, by having them start with a #. Then they cannot be accessed from outside the class.

But then - how can we use them?

We must write public *getter* and *setter* methods to access them, and the setter can *validate* any changes:

```
          class Person {
             #firstName; // private fields
             #lastName;

             constructor(f, l) {
                 this.#firstName = f;
                 this.#lastName = l;
             }

             setLastName(l)
             {
                 if (l.length>0) // do not allow zero length last
names
                 this.#lastName=l;
             }

             display()
             {
                console.log(this.#firstName, this.#lastName);
             }
         }

         var p1 = new Person('John','Smith');
         // not allowed - p1.#lastName='anything';
         p1.setLastName('Ali');
         p1.display();
```

Currently (2020) private fields do not work in Firefox and Internet Explorer. They work in Chrome.

Static

Static members are *per class not per object*. In other words they are data fields or methods for which there is just one copy, shared by all instances of the class, and so common to all objects of that type. For example, suppose we want to express the fact that people have two legs, as distinct from horses that have four and spiders that have eight:

```
         class Person {
             #firstName;
```

```
            #lastName;

            constructor(f, l) {
                this.#firstName = f;
                this.#lastName = l;
            }

            static legCount=2;

            setLastName(l)
            {
                if (l.length>0)
                this.#lastName=l;
            }

            display()
            {
               console.log(this.#firstName, this.#lastName);
            }
        }

        var p1 = new Person('John','Smith');
        console.log(Person.legCount);
```

p1.legCount does not work – a static field can only be accessed through the class, not an instance (unlike Java).

Many sites say the purpose of static members is to 'save memory'. This is not true. The purpose is to express the idea that a static member is *about the class* and not about the objects.

Class inheritance

A class can *extend* another class. The original class might be called a base class, and the extending class a sub-class. Instances of the sub-class *inherit* properties of the base class:

```
class Base
{
    baseField=3;
}

class Sub extends Base
{
    subField=4;
}

var obj=new Sub();

console.log(obj.baseField, obj.subField); // 3 4
```

The idea of this is to re-use code. If you need a class, and there already is a class that does a lot of what you want, you can define your class to extend the base class, and use its code.

We can get a class hierarchy, one base class, extended by some sub-classes, which in turn are extended by others, and so on.

Some people say inheritance should be used with care, because of the 'fragile base class' problem. This is if the base class definition changes, sub-classes might stop working correctly.

Arrays, maps and other data structures

Associative arrays

We have seen that all objects are *associative arrays* or *maps*. So they are key-value pairs:

```
var obj=new Object();
obj['x']=99; // write key-value x-99
console.log(obj['x']); // get value with key x
```

Typed arrays

These have either 8, 16, 32 or 64 bits per element, and are fixed length. The idea is to enable javaScript to be able to manipulate data in large binary blocks, like sound files or videos, at high speed.

For example:

```
var arr = new Int8Array(5);
for (var index = 0; index < arr.length; index++)
  arr[index]=index+10;
for (var index = 0; index < arr.length; index++)
  console.log(arr[index]); // 10, 11.. 14
```

We can iterate as an enumerable:

```
for (var p in arr)
  console.log(p, arr[p]); // 10, 11.. 14
```

There are signed and unsigned versions. An Int8Array is signed 8 bits, and so the elements can range for -128 to +127:

```
 var arr = new Int8Array(5);
arr[2]=-1;
console.log(arr[2]); // -5
```

If the value is outside this range, the result is 'wrong':

```
var arr = new Int8Array(5);
// range is -128 to +127
arr[2]=+128;
console.log(arr[2]); // -128
```

but a Uint8Array is unsigned, so it ranges from 0 to 255:

```
var arr = new Uint8Array(5);
arr[2]=+128;
console.log(arr[2]); // 128
arr[2]=257;
console.log(arr[2]); // 1
arr[2]=-1
console.log(arr[2]); // 255
```

There is a corresponding Int16Array, Uint32Array and so on.

There are also an 8 bit unsigned 'clamped' version, such that if the range is exceeded you just get the maximum or minimum:

```
var arr = new Uint8ClampedArray(5);
arr[2]=+300;
console.log(arr[2]); // 255
```

The Array object

In most languages an array is a data structure with elements which are accessed using an index, which is an integer. The Array object provides something similar to this:

```
var arr = new Array(3); // make an array with 3 slots
arr[0] = 27; arr[1] = 15; arr[2] = -9; // put values in
for (var index = 0; index < 3; index++)
  console.log(arr[index]); // 27 15 9
```

But there are differences. An array can contain a mix of data types:

```
var arr = new Array(3);
arr[0] = 27; arr[1] = "Hello"; arr[2] = false;
for (var index = 0; index < 3; index++)
  console.log(arr[index]);
```

Array indexes start at 0. *Array.length* is the number of elements in the array. But unlike most languages (C, Java, C++) an array is not fixed in size.

Neither are they dense. An array which has had a large index written to will be extended to that size if required, meaning there may be elements as some indexes which are undefined:

```
var arr = new Array(3); // so 0 to 4

arr[5]="Surprise";
for (var index = 0; index < arr.length; index++)
  console.log(index, arr[index]);
```

outputs

```
0 undefined
1 undefined
2 undefined
3 undefined
4 undefined
5 Surprise
```

The Set object

The special aspect of this is that a Set will not contain duplicates. For example:

```
var mySet=new Set();
for (var c of "What a fine day!") // iterate through a string
{
mySet.add(c);
}

for (var c of mySet) // get elements of the set
console.log(c);
```

Output is W h a t <space> f i n e d y !

This differsrom a mathematical set in that it is mutable (you cannot add or remove elements froma real set) and it is ordered as the insertion order (sets are not ordered).

A Set is fairly recent and old browsers will not understand.

The March 2021 ECMAScript specification says "Set objects must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection." So it treats a Set as an ADT, and requires access time better than O[n] - which is what a hash set would usually give.

Map

A Map object is a collection of name-value pairs:

```
var myMap = new Map();
myMap.set(1,"abc"); // insert 4 key-value pairs
myMap.set(2,"def");
myMap.set(4,"jkl");
myMap.set(3,"ghi");
console.log(myMap.get(2)); // def
console.log(myMap.has(3)); // true - it has a key 3
for (var node of myMap)
console.log(node); // come out in insertion order
```

The key methods are set, get and has.

We can say

```
var myMap = new Map();
myMap["test"]="result";
console.log(myMap["test"]); // result
```

but this just treats myMap as an associative array, which all JavaScript objects are. get and has would fail unless the data is added by set.

Like Set, Map is fairly new.

The DOM - JavaScript in the web page

JavaScript code usually runs inside a web browser, and is intended for client-side scripting - making things happen in a web page (as opposed to server-side scripting such as PHP, running code in a server before the web page is sent).

The DOM is the document object model. This is how the things in the web page can be referred to in JavaScript code.

There are 2 main issues. These are *when* the code is run, and how it refers to *web page components*.

The when is controlled by *event-handlers*. An event is something that happens in the web page, such as the user clicking a button. We set up JavaScript code as an event-handler - code which will be run when the event happens.

Web page components can be picked out by various techniques. One uses their html *id attribute*.

Responding to a button click

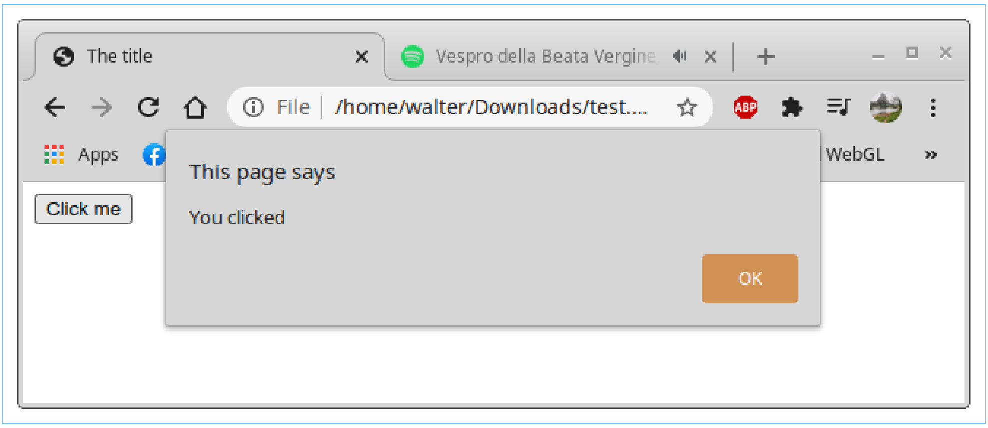
```
<!DOCTYPE html>

<html>

<head>
  <title>The title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <script>
    'use strict';

    function greet() {
      alert("You clicked");
    }
  </script>
</head>

<body>
  <button onclick="greet()">Click me</button>

</body>

</html>
```

The html button element has an onclick attribute. The value of this is JavaScript code which will be called when the event happens. In this example

```
<button onclick="greet()">
```

means the greet() function will be called when the button is clicked.

That function just calls the alert() function, which display a message like:


The title
Vespro della Beata Vergine
File /home/walter/Downloads/test....
Apps
WebGL
Click me
OK

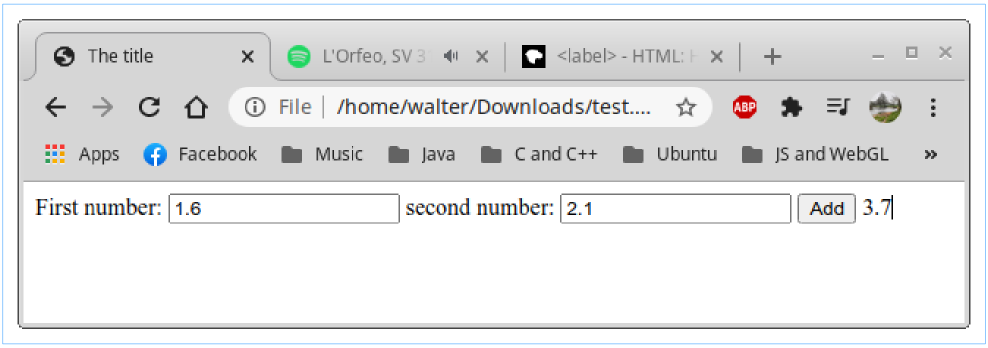
This page says

You clicked

html has a set of possible events for each element. For the body, the onload event occurs after the body has finished loading and being displayed, and is often used for initialising code.

A basic calculator

To do this:



First number: 1.6 second number: 2.1 3.7

we do this:

```
<!DOCTYPE html>

<html>

<head>
  <title>The title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <script>
    'use strict';

    function add() {
      // get elements
      var text1 = document.getElementById("num1");
      var text2 = document.getElementById("num2");
      // get value, and convert from string to number
      var num1 = parseFloat(text1.value);
      var num2 = parseFloat(text2.value);
      // add
      var result = num1 + num2;
      // get output element
      var display = document.getElementById("result");
      // set value
      display.innerHTML = result;


    }
  </script>
</head>

<body>
  <label for="num1">First number:</label>
  <input type="text" id="num1" name="num1" value="1.6">
  <label for="num2">second number:</label>
  <input type="text" id="num2" name="num2" value="2.1">
  <button onclick="add()">Add</button>
  <label id="result"></label>

</body>

</html>
```

We set the onclick event handler to be the add function.

This uses getElementById to get references to the 2 input elements. Then it uses .value to get the text in them, and parseFloat to convert the text to a number.

Then we add the numbers, get the output label, and set its innerHTML property to be result.

More JavaScript DOM use

We can also use JavaScript to add and remove web page elements after the page was been loaded.

We can change any elements properties. This includes its style attributes, and since this includes the display attribute, whether it is hidden or displayed.

A good set of notes is here.

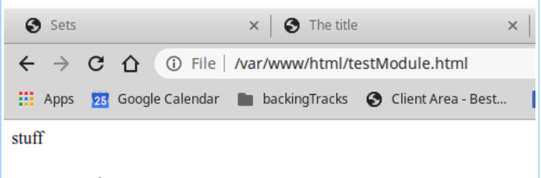
Modules

Basic idea

A module is like a script, but global scope is not shared between modules. Instead variables and functions must be explicitly exported from and imported into modules.

Need server not file://

We often develop html css and Javascript loading web pages directly into a browser, from a local file. In the URL bar the browser displays:

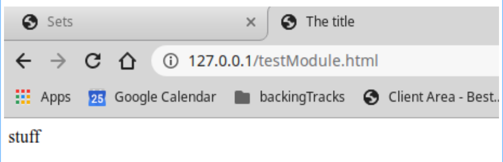


Using file protocol

and the protocol in use is file://. So the browser is simply reading the html page, and anything else, just like a word processor loads a document from a disk file.

But JavaScript modules will not allow that. A different protocol (usually http://) is required. The browser sends a request to a server. The server finds the html file, and sends it back to the browser, which displays it.

A simple way to achieve that is to download, install and start a local http server, such as Apache (link). Then if we fetch pages from the 'local loop' with IP address 127.0.0.1, we are using our local server, like this:



Using http:// protocol

Basic example

In a JavaScript file testModule.js we have

```
var x=99;
var y=101;

export {x};
```

and in a web page:

```
<html>
    <head>
        <title>The title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">

        <script type="module">
           import {x}  from "./testModule.js";
           y=4; // error -not found
        </script>
    </head>
    <body>
        <div>stuff</div>
    </body>
</html>
```

The relative path "./testModule.js" is for the file testModule.js to be in the same folder as the web page. It could be in a sub-folder, like "./modules/ testModule.js". Just "testModule.js" does not work. Some systems use filename extension .mjs not .js for modules, but some servers (like Apache 2.4.18) do not think .js is a script, and fail to load .mjs as being the wrong mime type.

As well as var, we can import const, let and functions:

```
 /*
 Filename testModule.js JavaScript file
 */

function add(a, b) { return a+b; };

var x=1;
const y=2;

export {add,x,y};
```

and

```
        <script type="module">
           import {add,x,y}  from "./testModule.js";

           var z=add(x,y);
           console.log(z); // 3
        </script>
```

Imports and exports are effectively ‘hoisted’:

```
 /*
 Filename testModule.js JavaScript file
 */
export {add,x,y};

function add(a, b) { return a+b; };
```

```
var x=1;
const y=2;
```

and

```
        <script type="module">
           var z=add(x,y);
           console.log(z); // still works
           import {add,x,y}  from "./testModule.js";
        </script>
```

We cannot re-declare an imported variable:

```
        <script type="module">
           var x; // Uncaught SyntaxError: Identifier 'x' has already
been declared
           var z=add(x,y);
           console.log(z);
           import {add,x,y}  from "./testModule.js";
        </script>
```

Modules always strict mode

Without saying 'use strict'; a module is executed in strict mode – another advantage.

Use of default

A module can export one (only) thing as default. Then another module can import the default and re-name it:

```
/*
 Filename testModule.js JavaScript file
 */
var x=1;
var p=2;
export {p};
export default x;
```

and

```
        <script type="module">
           import {default as y}  from "./testModule.js";
           import {p}  from "./testModule.js";
           console.log(y, p); // 1, 2
        </script>
```

Import and export formats

The import and export statements can take various forms, optionally renaming items:

Import Statement Form	Filename imported	Imported name	Name as used	Does what?
import v from "mod";	"mod"	"default"	"v"	Import and rename a default export
import * as ns from "mod";	"mod"	"*"	"ns"	Import a namespace object – see below
import {x} from "mod";	"mod"	"x"	"x"	Import, not re-named
import {x as v} from "mod";	"mod"	"x"	"v"	Import re-named
import "mod";	No actual import – for side-effects			

And

Export Statement Form	What is exported	File requested	What is imported?	Exported as..
export var v;	"v"			"v"
export default function f(){}	"default"			"f"
export default function() {}	"default"			"*default*"
export default 42;	"default"			"*default*"
export {x};	"x"			"x"
export {v as x};	"x"			"v"
export {x} from "mod";	"x"	"mod"	"x"	
export {v as x} from "mod";	"x"	"mod"	"v"	
export * from "mod";		"mod"	"*"	

The export .. from..; enables one module to aggregate exports from several files into a single exporting file.

Namespaces and importing objects

There are two namespaces in JavaScript – global and local to a function. So if we have a global variable x in one script, we cannot have another, different global x in another.

Importing items into an object fixes this. For example:

```
/*
 Filename testModule.js JavaScript file
 */
var a=1;
var b=2;
export {a,b};
```

and

```
        <script type="module">
           import  * as MyModule from "./testModule.js" ;

           console.log(MyModule.a, MyModule.b); // 1, 2
        </script>
```

Items are imported as fields of an object (here, MyModule) and referred to as such (like MyModule.a). This way we do not have to worry that about a global name in one script clashing with the same name in another we use the object as a namespace.