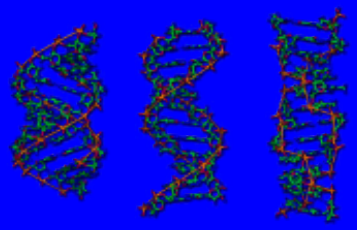
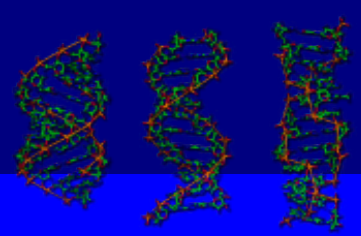


JavaScript types & constructors



JavaScript has types?

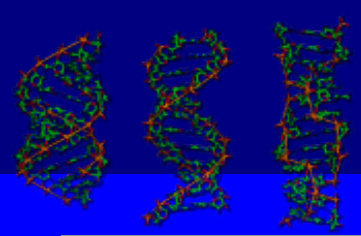
There is no compile-time type checking

In a sense, everything is an 'object'

But you can check at run-time what an object is like, and there is a lot of explicit/implicit run-time type checking

There is a crude `typeof` operator returning a string

Every object has a ***prototype***, which we will look at later



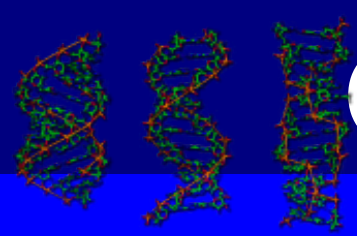
Objects and functions


```
"use strict";
var obj = {};

function f() { }
f.owner = "Pat";

console.log(typeof obj, typeof f);
console.log(f);
console.log(f.name, f.length, f);
```

```
js> node {f}
object function
{ [Function: f] owner: 'Pat' }
f 0 function f() { }
js>
```

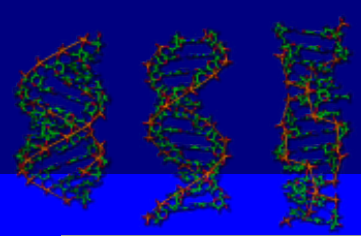


Object and function types

Objects, functions have types `"object"`, `"function"`

You can add properties to a function (if they don't clash with standard ones `name`, `length`, `call`, ...) and it is common to add properties to functions

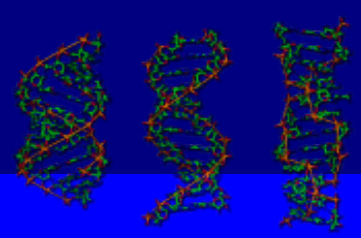
The `console.log` function sometimes gives different results for x depending on whether x is the first argument or not (print as object or call `toString`) because the first argument can be a format string (like `printf`) so the function inspects it differently



Strings and characters


```
"use strict";
var s = "x";
var c = 'x';
s[0] = 'z';   // (fails silently)
console.log(s, typeof s, s[0]);
console.log(c, typeof c, c==s, c===s);
// s.owner = "Pat"; (fails)
```

```
js> node {f}
x string x
x string true true
js>
```



The string type

String constants can be written with single or double quotes, e.g. `"don't"` or `'He said "Hi"'`

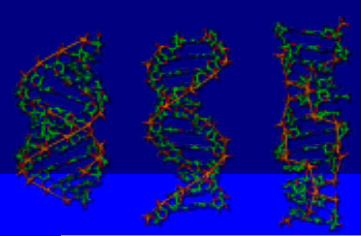
There is no character type, just one-character strings

You can use `s[i]` or (old) `s.charAt(i)`

Strings are immutable, e.g. `s = s.toUpperCase()`

You can't attach new fields/methods to a string (but you can add methods to its prototype)

The encoding is UTF16 (two bytes per character)

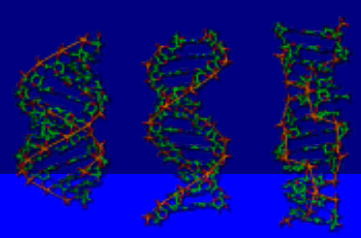


Numbers


```
"use strict";
var i = 42;
var f = 42.56789;
console.log(typeof i, typeof f);
console.log(i.toFixed(2), f.toFixed(2));

// f.owner = "Pat"; (fails)
console.log(f.owner);
```

```
js> node {f}
number number
42.00 42.57
undefined
js>
```



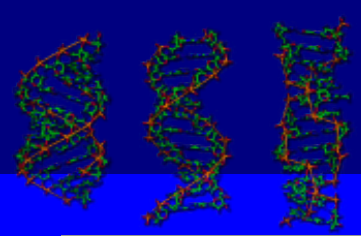
The number type

There is only one number type, `"number"`, represented using double precision

An integer is a double which happens to have a zero fractional part, and division with `/` is exact (use `Math.floor` to throw away the remainder)

Remainder with `%` works, but may have peculiar effects, some 'integer' operations have rounding error problems

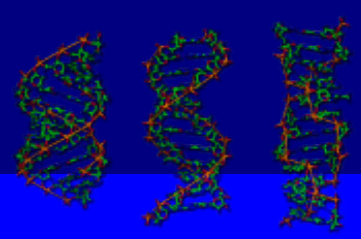
Numbers are immutable, and you can't add new fields



Booleans


```
"use strict";
console.log(typeof true);
console.log(0 ? true : false);
console.log(0 == false, 0 === false);
var x = true;
// x.owner = "Pat"; (fails)
console.log(x.owner);
```

```
js> node {f}
boolean
false
true false
undefined
js>
```



The boolean type

The `boolean` type is the weirdest in JavaScript

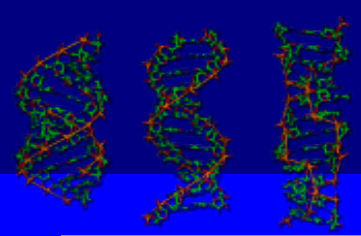
`0`, `""`, `NaN`, `null`, `undefined` are treated as false

Everything else (probably) is true, including `"false"`

This leniency can be handy, e.g. `if (x)...` can mean if x exists or if x succeeds

But it can also be extremely unpredictable

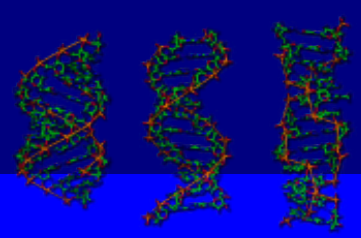
There is a lenient `==` operator and a safer `===` operator



Arrays = lists


```
"use strict";
var a = [4, 5, 6];
console.log(a, a[1], typeof a);
a.push(7);
console.log(a);
delete a[1];
a.owner = "Pat";
console.log(a, [] == []);
```

```
js> node {f}
[ 4, 5, 6 ] 5 'object'
[ 4, 5, 6, 7 ]
[ 4, , 6, 7, owner: 'Pat' ] false
js>
```



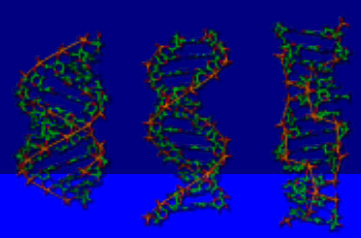
The array type?

There is ***almost*** a separate type for arrays, and you can initialise with square brackets

But `typeof` returns `object`, arrays can have holes in, and you can add extra fields or methods

`[] == []` gives false, because these are two different empty-array objects (a consequence of not having a 'proper' array type) so test `list.length == 0`, not `list == []`

Arrays ***are*** dynamic lists, you can use `push`, `pop`, `shift`, `join`, `slice`, `splice`, `concat`, `sort` etc



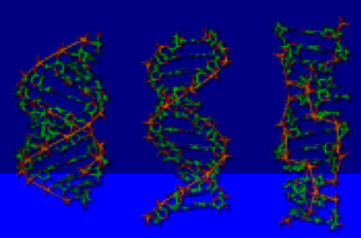
Miscellaneous types

There are dates, regular expression patterns, global JSON and Math facilities, errors and exceptions

The values `undefined` and `null` are almost identical, except that `typeof null` is `object`

Fortunately, `if (x == null)...` works if x is undefined, or you can just write `if (x)...`

You can find out more by looking at tutorials or the standard

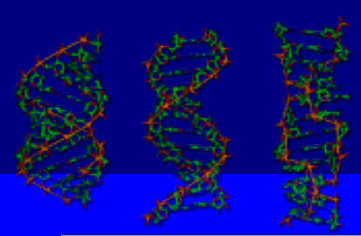


Adventure

It is time to get our adventure game working, all except the interactive input

Instead, we will put commands at the bottom of the program

```
...
obey("go south");
obey("dig sand");
```



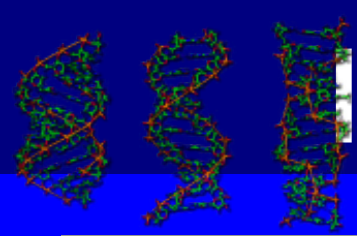
Splitting commands


```
"use strict";

function obey(command) {
  var words = command.split(" ");
  console.log(words);
}

obey("go south");
```

```
js> node {f}
[ 'go', 'south' ]
js>
```



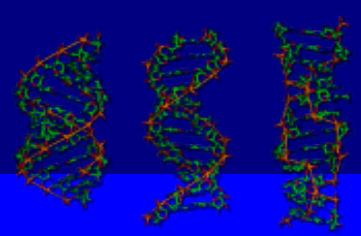
Implementing obey and go


```
"use strict";
var tree = { where: "tree", go: go };
var beach = { where: "beach", go: go };
tree.south = beach;

var here = tree;

function go() {
  here = this;
  console.log(here.where);
}

function obey(command) {
  var words = command.split(" ");
  var verb = words[0], noun = words[1];
  var object = here[noun];
  object[verb].call(object);
}
```

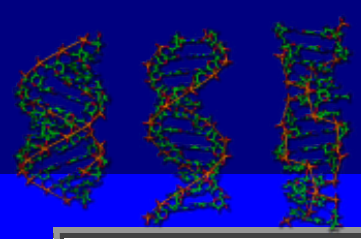


Commands

In our adventure game, a `"verb noun"` command has a simple meaning

Use the noun to find an object attached to the current place, use the verb to find a method attached to the object, call it

We can give a standard error message if this fails



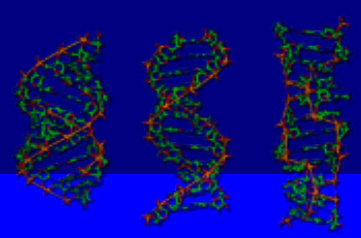
The complete adventure


```
// The island game, without interaction
"use strict";

// Set up the world and current place
var spade = { what:"a spade", take:take };
var sand = { what:"sand", dig:dig };
var tree = { where:"near a tree", go:go, spade:spade };
var beach = { where:"on a beach", go:go, north:tree, sand:sa
tree.south = beach;
var here = tree;

// Test if the player is carrying a given thing
function isBeingCarried(thing) {
  return (thing.drop != undefined);
}

// Define the actions
function take() {
```

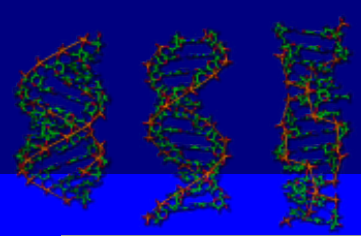


Details

Things on the ground have a `take` method, things being carried have a `drop` method

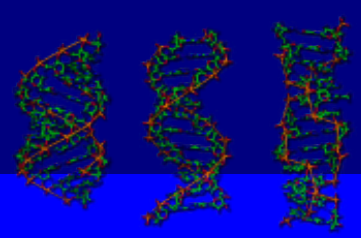
The only new JavaScript feature in the program is the `for..in` loop

`for (var x in obj)` makes `x` run through the names of the properties of `obj`



Playing the game

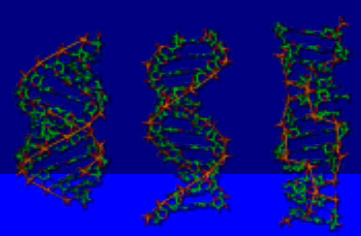

```
js> node p20.js
You are near a tree
There is a spade here
> go south
You are on a beach
There is sand here
> dig sand
You have no spade
> go north
You are near a tree
There is a spade here
> take spade
You are carrying a spade
> go south
You are on a beach
There is sand here
> dig sand
You find your holiday return ticket and go home
js>
```



Construction

So far, objects have been created without constructors

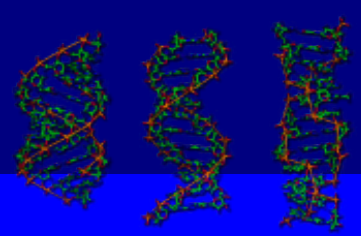
```
var tree = { where:"...", go:go, ... };
var beach = { where:"...", go:go, ... };
function go() { ... this.where ... }
```



Some problems

Some of the problems with this approach are:

- the code is not DRY
- the methods are accessible as functions
- the method names are global and could clash
- the fields are public
- the methods need `this.` to access fields



Odie: classical

Each pet has its own preferred way of being created

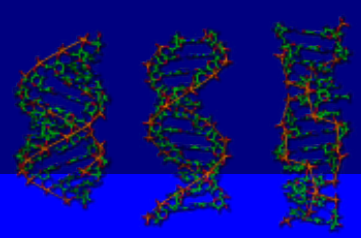
Odie prefers classical constructors

```
var odie = new Pet("woof");

function Pet(text) {
  this.sound = text;
  this.speak = speak;
}

function speak() { console.log(this.sound); }
```

'Classical' means 'old' and 'class-related'

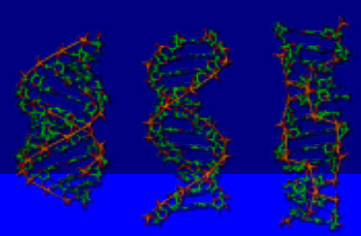


Classical constructors

The `Pet` constructor is *classical*; their names usually start with a capital letter

The `new` keyword does most of the work: it creates a blank object `{}` and calls the constructor as if it were a method, i.e. `this` is set to `{}`, and it returns the object after the call, roughly as if:

```
new(con,arg,...) {
  var x = {};
  con.call(x,arg,...);
  return x;
}
```

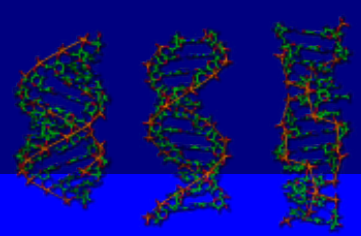


Comparison with Java

A constructor call `new X()` in Java does roughly the same thing: it creates a blank object of the given class, executes the constructor method with `this` set to the fresh object, and returns the object

The `new` keyword in JavaScript deliberately looks like Java, but it becomes different and confusing when we add details about prototypes

So this is one reason for avoiding `new` in JavaScript



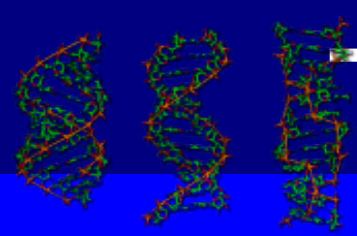
Inflexibility

Imagine `Pet` is in a library and there are millions of lines of code out there which call `new Pet(...)`

You are the author of the library, and you change your mind about how `Pet` should work, e.g. you sometimes want to give back an existing object from a cache rather than create a new object

You can't, because you can't tell all the programmers out there to stop using the `new` keyword

There is an implementation detail ("new object") which is in the calling code instead of inside the class/module



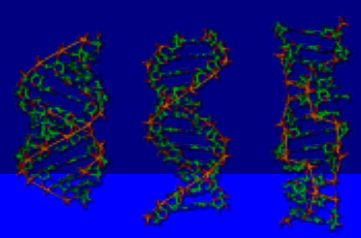
The factory design pattern

The factory ('method') design pattern is:

factories

To create an object,
don't use a constructor,
use an ordinary function

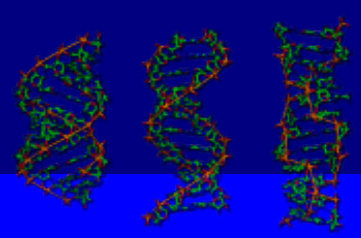
As usual, you can regard this as a rough idea of what to do if the inflexibility of new becomes a problem
(or you can regard is a fault in the contructor idea)



Power constructors

In JavaScript, an ordinary function used to create objects instead of a constructor, is called a *power constructor*

It is becoming common among some JavaScript programmers to avoid classical constructors and the keyword new altogether



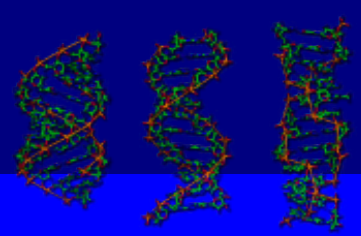
Wanda: powerful

Wanda prefers simple power constructors:

```
var wanda = pet("bloop");

function pet(text) {
  var x = { sound: text, speak: speak };
  function speak() { console.log(this.sound); }
  return x;
}
```

The fields are still public, and the methods still need to use `this.` to access the fields



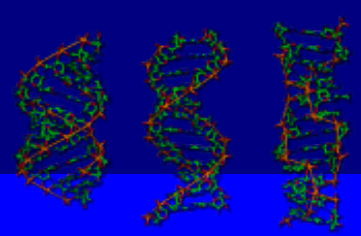
Garfield: closurise

Garfield likes power constructors with local variables:

```
var garfield = pet("meow");

function pet(text) {
  var sound = text;
  var x = { speak: speak };
  function speak() { console.log(sound); }
  return x;
}
```

The fields are private and `this.` is not needed in the methods – this relies on *closures*, which can be space-hungry, and it can be shortened

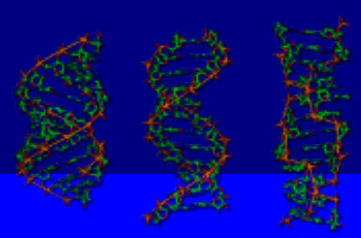


Shortening

Garfield's `pet` function can be written:

```
function pet(sound) {
  var x = { speak: speak };
  function speak() { console.log(sound); }
  return x;
}
```

Arguments to a function ***are*** local variables of the function, so they don't need to be renamed if they have the right name already



Closures

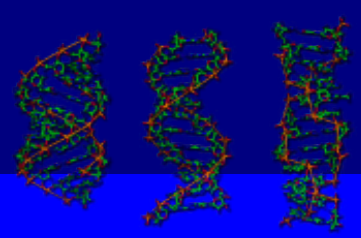
Closures are absolutely essential in JavaScript – they are used in input/output and in writing servers

An inner function can access the local variables, including args, of the outer function

This works even after the outer function returns

A closure object is created on each outer call to remember them

- a personal view



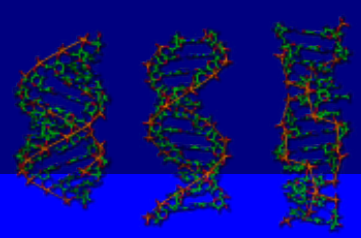
A personal view

Closures are a historical legacy left over from before object oriented programming

In OO, the object a method is attached to should be its closure, as in Java

Compared to the owning object, closures are implicit, invisible, uncontrollable and (experience suggests) poorly understood

Nevertheless, many features of JavaScript are attractive so, given JavaScript as it is, closures have to be tolerated and used well



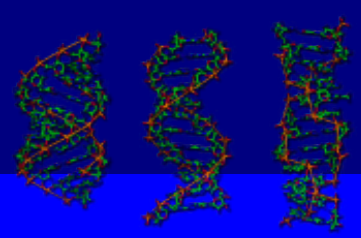
Minnie: prototypical

Minnie likes power constructors with protoypes:

```
var minnie = pet("squeak");

function pet(text) {
  var x = Object.create(prototype);
  x.sound = text;
  var prototype = { sound: null; speak: speak };
  function speak() { console.log(sound); }
  return x;
}
```

An object's prototype provides defaults for its methods and constants



Choice

Most JavaScript programming is done without using constructors at all but if you care, the Garfield and Minnie approaches are the best

Unfortunately, the two don't mix

Advice: follow Garfield if you like the convenience, readability, flexibility, and don't mind the extra memory or lack of inhertance, otherwise use Minnie's approach