Class basic syntax

In practice, we often need to create many objects of the same kind, like users, or goods or whatever.

As we already know from the chapter Constructor, operator "new", `new function` can help with that.

But in the modern JavaScript, there's a more advanced "class" construct, that introduces great new features which are useful for object-oriented programming.

The "class" syntax

The basic syntax is:

```
class MyClass {
  // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

Then `new MyClass()` creates a new object with all the listed methods.

The `constructor()` method is called automatically by `new`, so we can initialize the object there.

For example:

```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    alert(this.name);
  }
}
// Usage:
let user = new User("John");
user.sayHi();
```

When `new User("John")` is called:

1. A new object is created.
2. The `constructor` runs with the given argument and assigns `this.name` to it.

…Then we can call methods, such as `user.sayHi`.

No comma between class methods

A common pitfall for novice developers is to put a comma between class methods, which would result in a syntax error.

The notation here is not to be confused with object literals. Within the class, no commas are required.

What is a class?

So, what exactly is a `class`? That's not an entirely new language-level entity, as one might think.

Let's unveil any magic and see what a class really is. That'll help in understanding many complex aspects.

In JavaScript, a class is a kind of a function.

Here, take a look:

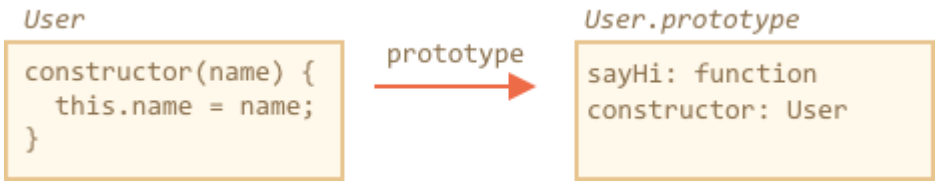
```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// proof: User is a function
alert(typeof User); // function
```

What `class User {...}` construct really does is:

1. Creates a function named `User`, that becomes the result of the class declaration.
   - The function code is taken from the `constructor` method (assumed empty if we don't write such method).
2. Stores all methods, such as `sayHi`, in `User.prototype`.

Afterwards, for new objects, when we call a method, it's taken from the prototype, just as described in the chapter F.prototype. So `new User` object has access to class methods.

We can illustrate the result of class User as:



Here's the code to introspect it:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// class is a function
alert(typeof User); // function

// ...or, more precisely, the constructor method
alert(User === User.prototype.constructor); // true

// The methods are in User.prototype, e.g:
alert(User.prototype.sayHi); // alert(this.name);

// there are exactly two methods in the prototype
alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi
```

Not just a syntax sugar

Sometimes people say that class is a "syntax sugar" in JavaScript, because we could actually declare the same without classkeyword at all:

```
// rewriting class User in pure functions

// 1. Create constructor function
function User(name) {
  this.name = name;
}
// any function prototype has constructor property by default,
// so we don't need to create it

// 2. Add the method to prototype
User.prototype.sayHi = function() {
  alert(this.name);
```

```
};

// Usage:
let user = new User("John");
user.sayHi();
```

The result of this definition is about the same. So, there are indeed reasons why `class` can be considered a syntax sugar to define a constructor together with its prototype methods.

Although, there are important differences.

1. First, a function created by `class` is labelled by a special internal property `[[FunctionKind]]:"classConstructor"`. So it's not entirely the same as creating it manually.

   Unlike a regular function, a class constructor can't be called without `new`:

   ```
   class User {
     constructor() {}
   }

   alert(typeof User); // function
   User(); // Error: Class constructor User cannot be invoked without 'new'
   ```

   Also, a string representation of a class constructor in most JavaScript engines starts with the "class…"

   ```
   class User {
     constructor() {}
   }

   alert(User); // class User { ... }
   ```

2. Class methods are non-enumerable A class definition sets `enumerable` flag to `false` for all methods in the `"prototype"`.

   That's good, because if we `for..in` over an object, we usually don't want its class methods.

3. Classes always `use strict` All code inside the class construct is automatically in strict mode.

   Also, in addition to its basic operation, the `class` syntax brings many other features with it which we'll explore later.

Class Expression

Just like functions, classes can be defined inside another expression, passed around, returned, assigned etc.

Here's an example of a class expression:

```
let User = class {
  sayHi() {
    alert("Hello");
  }
};
```

Similar to Named Function Expressions, class expressions may or may not have a name.

If a class expression has a name, it's visible inside the class only:

```
// "Named Class Expression" (alas, no such term, but that's what's going on)
let User = class MyClass {
  sayHi() {
    alert(MyClass); // MyClass is visible only inside the class
  }
};

new User().sayHi(); // works, shows MyClass definition

alert(MyClass); // error, MyClass not visible outside of the class
```

We can even make classes dynamically "on-demand", like this:

```
function makeClass(phrase) {
  // declare a class and return it
  return class {
    sayHi() {
      alert(phrase);
  };
  };
}
// Create a new class
let User = makeClass("Hello");
new User().sayHi(); // Hello
```

Getters/setters, other shorthands

Classes also include getters/setters, generators, computed properties etc.

Here's an example for `user.name` implemented using `get/set`:

```
class User {

 constructor(name) {
  // invokes the setter
  this._name = name;
 }

 get name() {
  return this._name;
 }

 set name(value) {
  if (value.length < 4) {
   alert("Name is too short.");
   return;
  }
  this._name = value;
 }

}

let user = new User("John");
alert(user.name); // John

user = new User(""); // Name too short.
```

Internally, getters and setters are created on `User.prototype`, like this:

```
Object.defineProperties(User.prototype, {
 name: {
  get() {
   return this._name
  },
  set(name) {
   // ...
```

```
    }
  }
});
```

Here’s an example with computed properties:

```
function f() { return "sayHi"; }

class User {
  [f()]() {
    alert("Hello");
  }
}
new User().sayHi();
```