The **`new` operator** lets developers create an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

Syntax

```
new constructor[([arguments])]
```

Parameters

`constructor`
A class or function that specifies the type of the object instance.

`arguments`
A list of values that the *`constructor`* will be called with.

Description

The **`new`** keyword does the following things:

1. Creates a blank, plain JavaScript object;
2. Links (sets the constructor of) this object to another object;
3. Passes the newly created object from *Step 1* as the `this` context;
4. Returns `this` if the function doesn't return an object.

Creating a user-defined object requires two steps:

1. Define the object type by writing a function.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name and properties. An object can have a property that is itself another object. See the examples below.

When the code `new Foo(...)` is executed, the following things happen:

1. A new object is created, inheriting from `Foo.prototype`.
2. The constructor function `Foo` is called with the specified arguments, and with `this` bound to the newly created object. `new Foo` is equivalent to `new Foo()`, i.e. if no argument list is specified, `Foo` is called without arguments.
3. The object (not null, false, 3.1415 or other primitive types) returned by the constructor function becomes the result of the whole `new` expression. If the constructor function doesn't explicitly return an object, the object created in step 1 is used instead. (Normally constructors don't return a value, but they can choose to do so if they want to override the normal object creation process.)

You can always add a property to a previously defined object. For example, the statement `car1.color = "black"` adds a property `color` to `car1`, and assigns it a value of `"black"`. However, this does not affect any other objects. To add the new property to all objects of the same type, you must add the property to the definition of the `Car` object type.

You can add a shared property to a previously defined object type by using the `Function.prototype` property. This defines a property that is shared by all objects created with that function, rather than by just one instance of the object type. The following code adds a color property with value `"original color"` to all objects of type `Car`, and then overwrites that value with the string `"black"` only in the instance object `car1`. For more information, see prototype.

```
function Car() {}
car1 = new Car();
car2 = new Car();

console.log(car1.color);    // undefined

Car.prototype.color = 'original color';
console.log(car1.color);    // 'original color'

car1.color = 'black';
console.log(car1.color);    // 'black'

console.log(Object.getPrototypeOf(car1).color); // 'original color'
console.log(Object.getPrototypeOf(car2).color); // 'original color'
console.log(car1.color);   // 'black'
console.log(car2.color);   // 'original color'
```

If you didn't write the `new` operator, **the Constructor Function would be invoked like any Regular Function,** *without creating an Object.* In this case, the value of `this` is also different.

Examples

Object type and object instance

Suppose you want to create an object type for cars. You want this type of object to be called `Car`, and you want it to have properties for make, model, and year. To do this, you would write the following function:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Now you can create an object called `myCar` as follows:

```
var myCar = new Car('Eagle', 'Talon TSi', 1993);
```

This statement creates `myCar` and assigns it the specified values for its properties. Then the value of `myCar.make` is the string "Eagle", `myCar.year` is the integer 1993, and so on.

You can create any number of `car` objects by calls to `new`. For example:

```
var kensCar = new Car('Nissan', '300ZX', 1992);
```

Object property that is itself another object

Suppose you define an object called `Person` as follows:

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
```

And then instantiate two new `Person` objects as follows:

```
var rand = new Person('Rand McNally', 33, 'M');
var ken = new Person('Ken Jones', 39, 'M');
```

Then you can rewrite the definition of `Car` to include an `owner` property that takes a `Person` object, as follows:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
```

To instantiate the new objects, you then use the following:

```
var car1 = new Car('Eagle', 'Talon TSi', 1993, rand);
var car2 = new Car('Nissan', '300ZX', 1992, ken);
```

Instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the parameters for the owners. To find out the name of the owner of `car2`, you can access the following property:

```
car2.owner.name
```