无类继承

JavaScript面向对象的根基

周爱民 / aimingoo
aiming@gmail.com
https://aimingoo.github.io
https://github.com/aimingoo

```
rand = new Person("Rand McKinnon", ...
```

https://docs.oracle.com/cd/E19957-01/816-6408-10/object.htm#1193255

https://www.fi.muni.cz/~xdohnal/manuals/jscript/toc.html

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

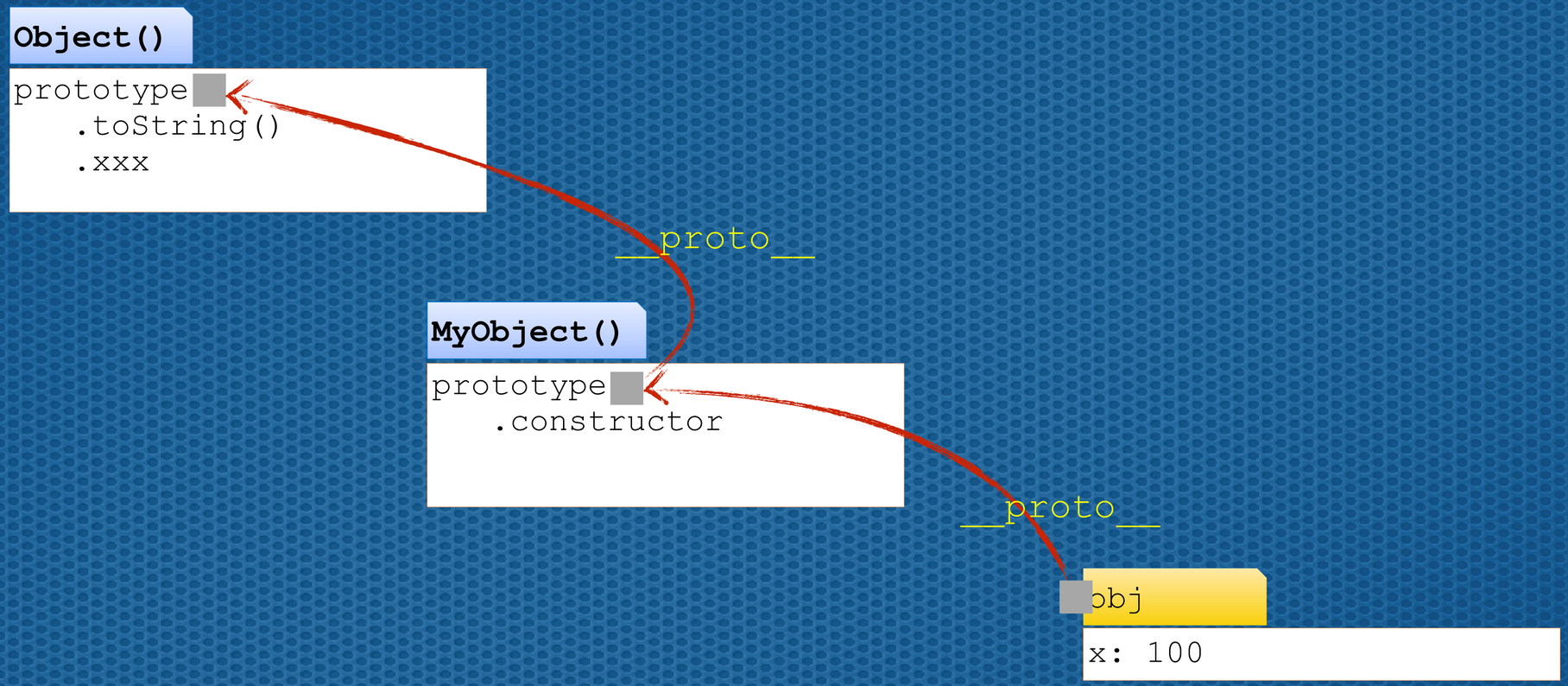
https://docs.oracle.com/cd/E19957-01/816-6408-10/object.htm#1193255

https://www.fi.muni.cz/~xdohnal/manuals/jscript/toc.html

JavaScript 1.1

prototype based

```
function MyObject() {
  this.xxx = ...
  ...
}
MyObject.prototype = new Object();
MyObject.prototype.constructor = MyObject;
...

obj = new MyObject();
obj.x = 100;
```




```
console.log(obj.x);
console.log(obj.xxx);
console.log(obj.toString());
```

- 原型继承
- 类抄写
- 声明：类/原型/对象...
- 类继承

js1.1 ~ 1.x
js2
ECMAScript edition 1~4

ECMAScript Ed5

Non-classes based

```
Object.hasOwnProperty(obj, ...)
Object.defineProperty(obj, ...)
Object.freeze()
Object.XXXX()
...
```

```
obj = Object.create(X, {
      ...
})
```

```
Object.setPrototypeOf(obj, X)
```

```
X.isPrototypeOf(obj)
```

```
X.isPrototypeOf(obj)
```

vs.

```
obj instanceof Object
```

1. 对象就是属性包

① 所以原型继承就是属性包的链式访问

② 所以可以没有类继承（例如构造器和new）

an object is a collection of properties

JavaScript 2015
ECMAScript Ed 6 / ES6

该来的总会来的。—— 崔椭圆@2020

```
let obj = {
    name: 'obj',
    foo() {
        ...
    }
}
```

```
homeObj = Method[[HomeObject]];

super = Object.getPrototypeOf(homeObj)
```

```
let obj = {
 foo() {  //（所以这里是静态声明，而非初始器赋值）
    console.log(super.name, this.name);
  }
}
Object.setPrototypeOf(obj, {name: 'Super'});
obj.name = 'obj';
obj.foo();  // Super, obj

let obj2 = Object.create(obj, {name: 'I am obj2'});
obj2.foo(); // Super, I am obj2
```

```
let foo = obj.foo;
foo();  // ???, ???
```

Class

is appearances

```
class MyClass extends X {
  constructor() {
    ..
  }
  static foo() {
    ..
  }
  more() {
...
```

```
class MyClass extends X {
  constructor() {
    super();
    ..
  }

  static foo() {
    ..
  }

  more() {
    ..
  ...
```

=

```
// bind className and set prototypes
MyClass = asClassConstructor(constructor);
Object.setPrototypeOf(MyClass, X);
Object.setPrototypeOf(MyClass.prototype,
    X.prototype);

// for static methods
setHomeObject(foo, MyClass);

// for constructor and instance methods
setHomeObject(constructor, MyClass.prototype);
setHomeObject(more, MyClass.prototype);
```

```
class MyClass {}
class MyClass extends X {}
```

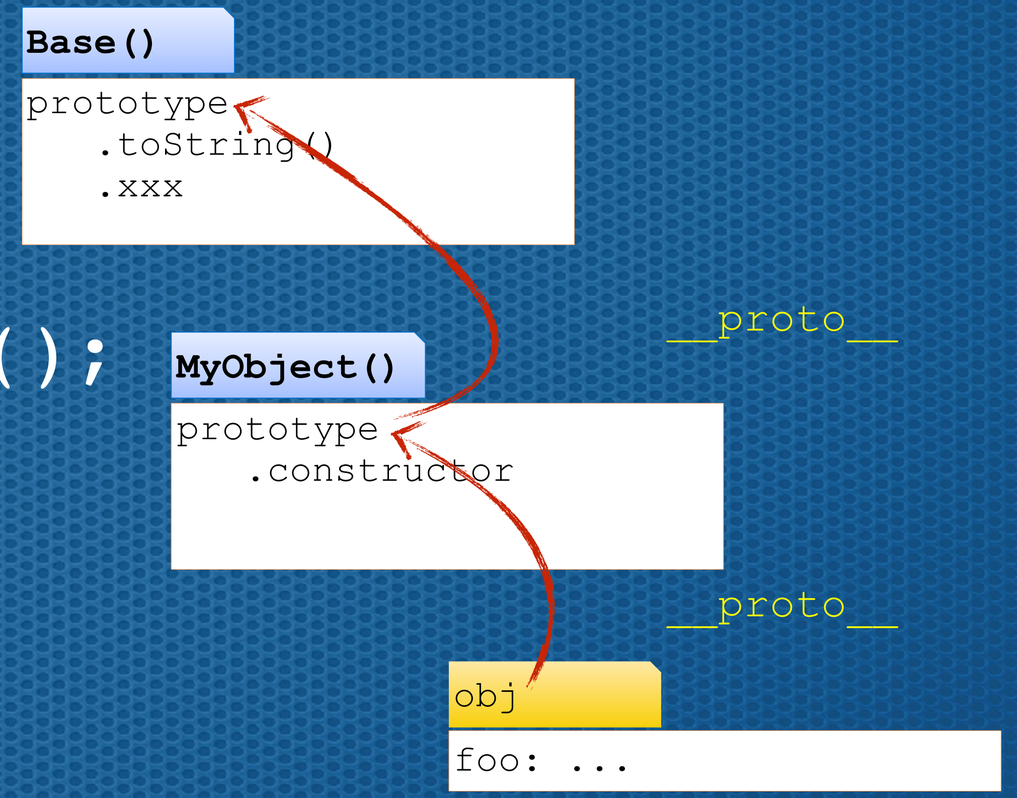
```
class MyClass {
  constructor() { }   // no "super()"
}

class MyClass extends X {
  constructor(...args) {  // default
    super(...args);
  }
}
```

```
> class MyClass extends null {}

> obj = new MyClass;
TypeError: Super constructor null of …

> Object.setPrototypeOf(MyClass, Object)

> typeof(obj = new MyClass)
'object'
```

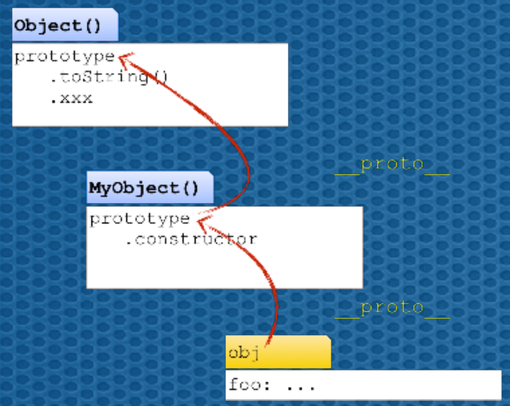
new X

New or create

```
function MyObject() {}
MyObject.prototype = new Base();
MyObject.prototype.foo = ...

obj = new MyObject;
```



```
function MyObject() {}
MyObject.prototype = new Base();
MyObject.prototype.foo = ...

obj = new MyObject;
```




```
class MyClass extends Base {
  constructor() { ...
  foo() { ...
...
obj2 = new MyClass;
```

Base()

```
constructor {
    _this = new_instance()
    setPrototypeOf(_this, new.target.prototype);
    // binding <this> or return
    ...
}
```

MyClass()

```
constructor {
    super(); // with new.target
    ...
...
```

obj2 is <_this>

obj2

```
obj2 = new MyClass;
obj2.foo();
```

```
// fail
MyFunction.prototype = Function;
MyDate.prototype = new Date();

...
```

```
// success
class MyDate extends Date {

  ...

}

// success
class MyFunction extends Function {

  ...

}
```

```
class MyClass extends X;
```

① 父类*X*是可以通过写类的原型来重置的

② super总是通过计算指向*X*，并受上述重置影响

```
obj = new MyClass;
```

① obj总是由祖先类(Base)创建的实例

Atom

Meta based

```
X = Object.create(null);
Object.setPrototypeOf(X, null);
```

```
function isAtom(obj) {
  let types = {object: true, function: true};

  if (types[typeof obj]) {
    return !(obj instanceof Object);
  }

  return false;
}
```

arguments
namespace

```
// for arguments
args = Function('return arguments')();
isAtom(args); // true

// for namespace
import * as namespace from "./exports.mjs";
isAtom(namespace); // true
```

atom – 原子

meta – 元：能产生原子的一个过程

- Meta - 能产生meta的一个过程，称为元类型
- MetaClass - 如果所产生的meta是一个类，则这个Meta称为元类类型

```
// 如下代码意味着Atom()是元函数或元构造器
atom = new Atom();
```

```
// a meta/atom-constructor, derived from null, it's meta style
class meta extends null {
  ...
```

```
// Atom() for meta system
class Atom extends null {
  constructor() {
    return Object.create(new.target.prototype);
  }
}
```

```
// Atom() for meta system
class Atom ...

// Meta
class Meta extends null {
  constructor() {
    return Object.setPrototypeOf(
      class extends new.target{}, Atom);
  }
}
```

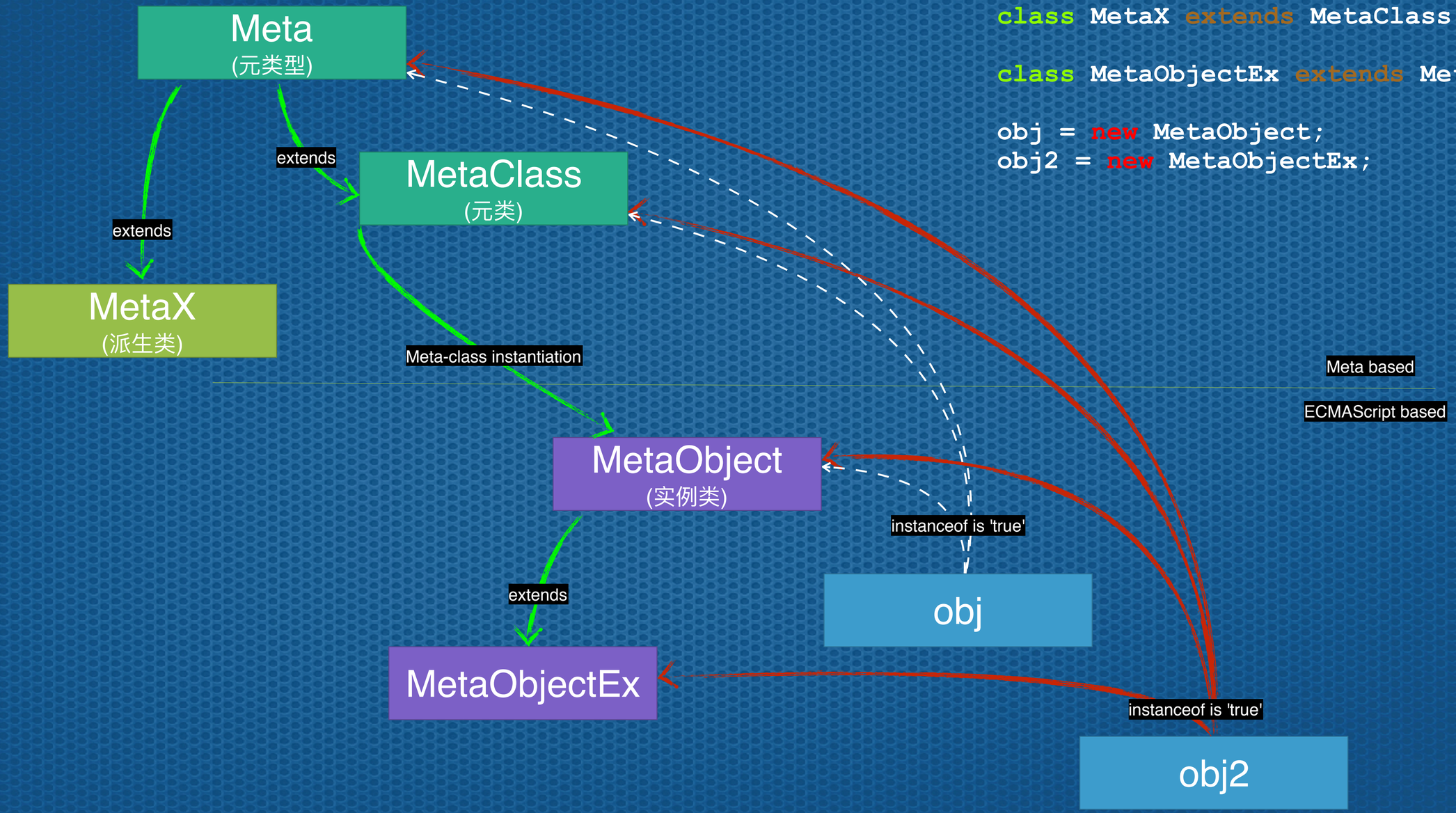
```
// Atom() for meta system
class Atom ...

// Meta
class Meta ...

// to atom style
void Object.setPrototypeOf(Atom, null);
void Object.setPrototypeOf(Meta, null);
```

```
// Atom() for meta system
class Atom ...
// Meta
class Meta ...
// to atom style
...

// Meta Classes
class MetaClass extends Meta {
  static isClassOf(x) {
    return x instanceof this ||
      x.prototype instanceof this;
  }
}
```

```
// Atom() for meta system
class Atom ...
// Meta
class Meta ...
// to atom style
...

// Meta Classes
class MetaClass extends Meta { ...

class MetaObject extends new MetaClass {}
x = new MetaObject;
```

```
> [MetaObject, MetaClass, Meta, Atom, x]
    .forEach(x=>console.log(isAtom(x)));
true
true
true
true
true

> typeof x
'object'
```

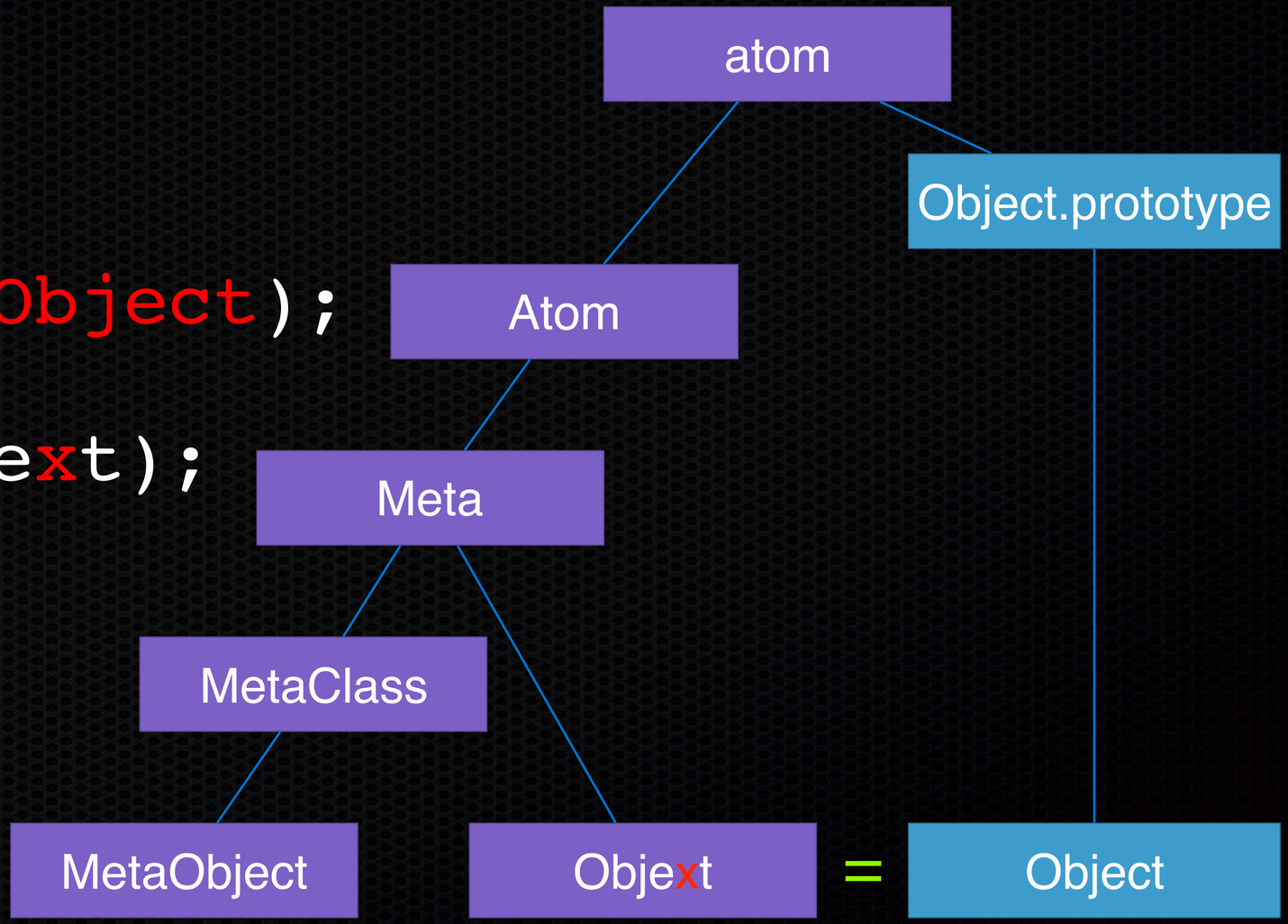
```
class Meta …
class MetaClass …

class MetaObject extends new MetaClass {}
class MetaX extends MetaClass {}

class MetaObjectEx extends MetaObject {}

obj = new MetaObject;
obj2 = new MetaObjectEx;
```

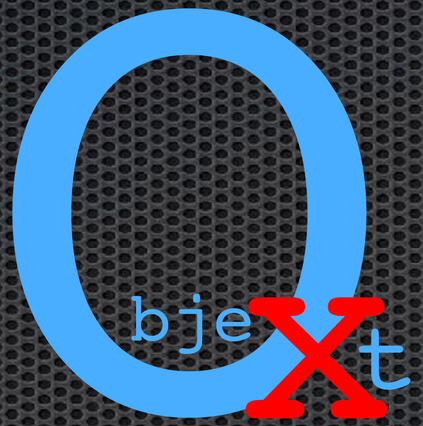


```
> Meta.isAtom(null);
true
> Meta.isAtom(Object.prototype);
true
```

```
> Objext = Meta.from(Object);
```

```
> Objext = Meta.from(Object);

> Objext.keys(new Objext);
[]
```





github.com/aimingoo/metameta