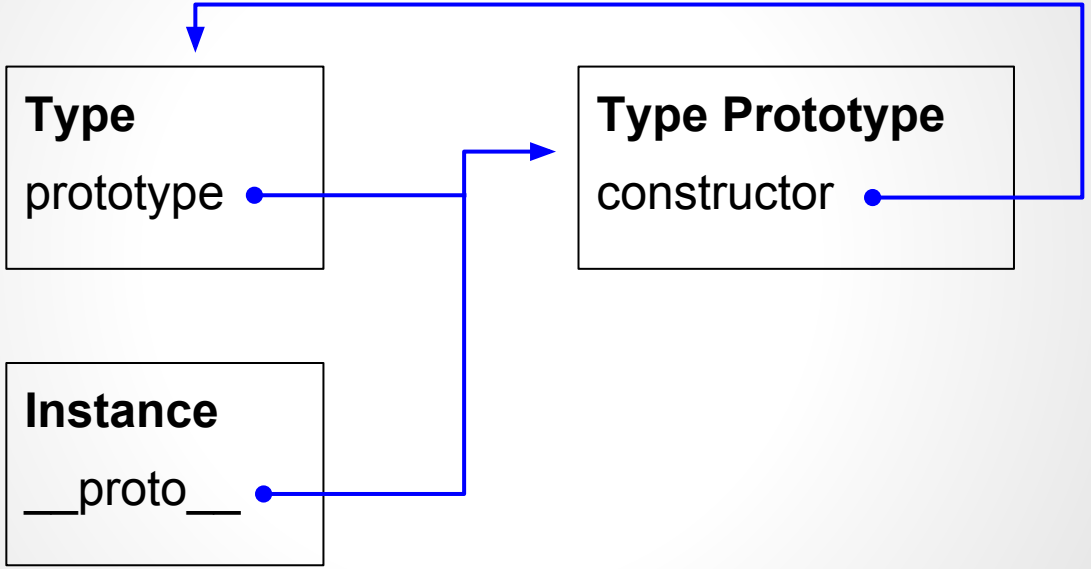
Object Oriented Javascript

Inheritance

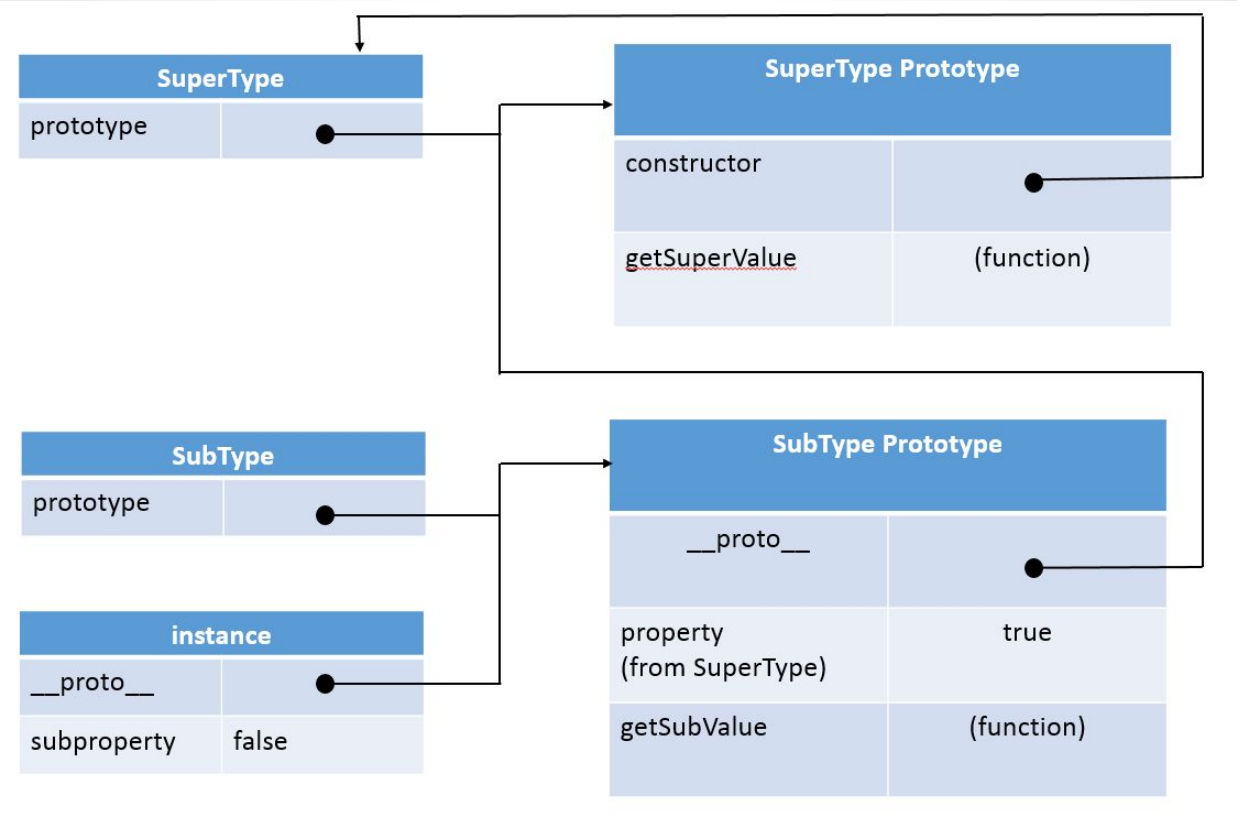
Javascript supports only “Implementation inheritance”

- ★ Prototype Chaining
- ★ Constructor Stealing
- ★ Prototypal Inheritance
- ★ Parasitic Inheritance

Prototype Chaining



Prototype Chaining



Prototype Chaining

```
function SuperType(){
    this.property = true;
}

SuperType.prototype.getSuperValue = function(){
    return this.property;
}

function SubType(){
    this.subproperty = true;
}
```

Prototype Chaining

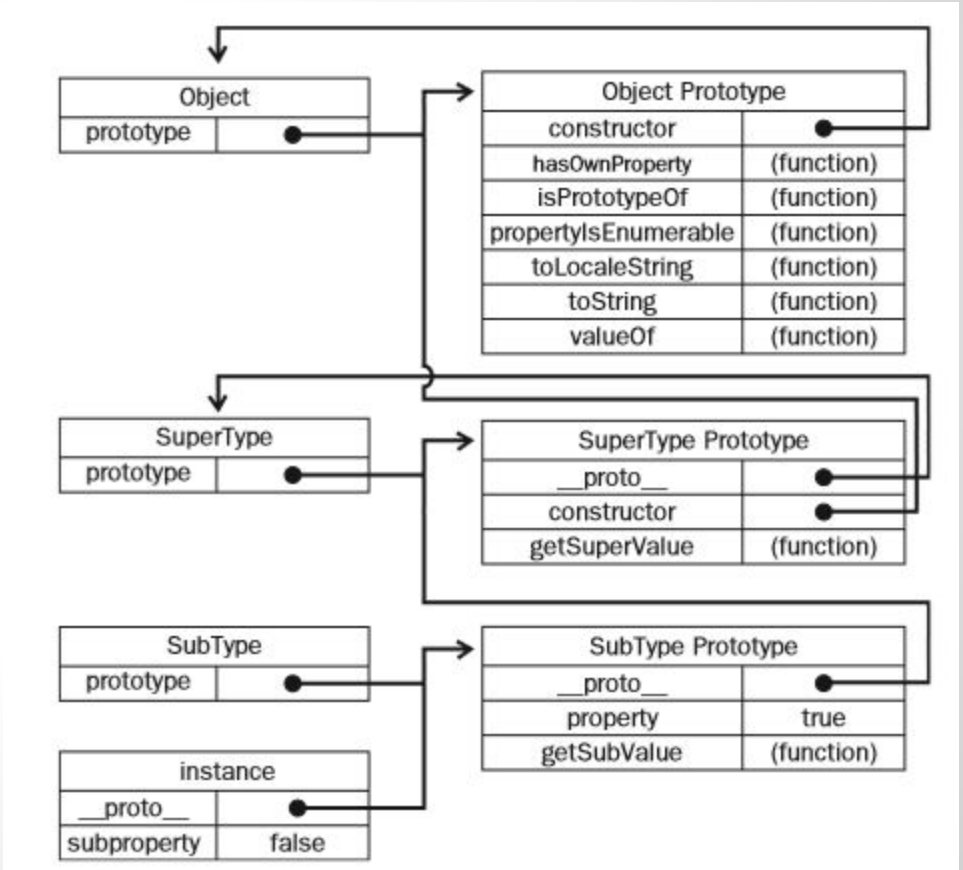
```
//inherit from SuperType
SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function(){
     return this.subproperty;
}

var instance = new SubType();
alert(instance.getSuperValue()); //true
```

Prototype Chaining

- ★ All reference types inherit from Object by default
- ★ Accomplished using Prototype Chaining
- ★ They inherit the default methods



Prototype Chaining

```
// Instance Of ?

alert(instance instanceof Object);    //true
alert(instance instanceof SuperType); //true
alert(instance instanceof SubType);   //true

// Prototype Of ?

alert(Object.prototype.isPrototypeOf(instance));    //true
alert(SuperType.prototype.isPrototypeOf(instance)); //true
alert(SubType.prototype.isPrototypeOf(instance));   //true
```

```
// Will this work ??
function SuperType(){
    this.property = true;
}
SuperType.prototype.getSuperValue = function(){
    return this.property;
}
function SubType(){
    this.subproperty = true;
}
//override existing method
SubType.prototype.getSuperValue = function(){
    return false;
}
//inherit from SuperType
SubType.prototype = new SuperType();
```

```
// Yup, this works for me :)
function SuperType(){
    this.property = true;
}
SuperType.prototype.getSuperValue = function(){
    return this.property;
}
function SubType(){
    this.subproperty = true;
}
//first always assign the prototype (inherit)
SubType.prototype = new SuperType();

//override existing method
SubType.prototype.getSuperValue = function(){
    return false;
}
```

Prototype Chaining

```
//Never use object literal to create prototype methods
//inherit from SuperType
SubType.prototype = new SuperType();

//overrides the prototype chain
SubType.prototype = {
    getSuperValue: function(){
        return false;
    }
}
```

Prototype Chaining

```
function SuperType(){
     this.names = [“Sergi”, “Diego”];
}
function SubType(){}
//inherit from SuperType
SubType.prototype = new SuperType();

var instance1 = new SubType();
instance1.names.push(“Eric”);
alert(instance1.names); // ”Sergi,Diego,Eric”

var instance2 = new SubType();
alert(instance2.names); // ”Sergi,Diego,Eric”
```

Prototype Chaining

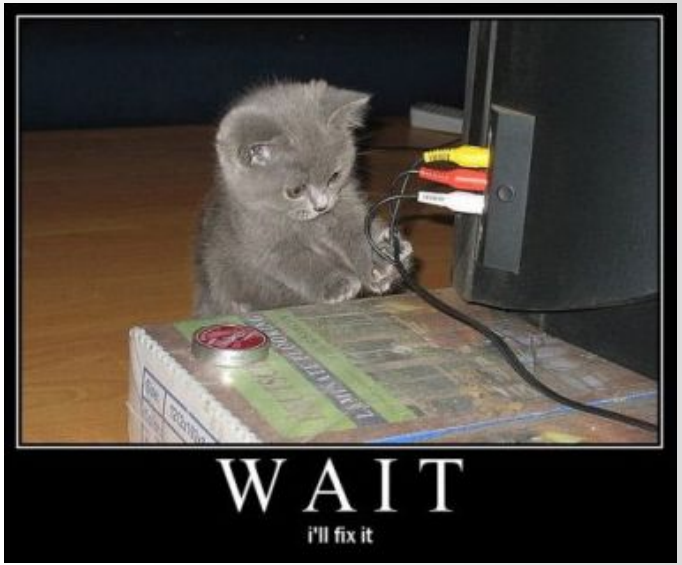
```
/* PROS
 # Easy implementation of inheritance
 # Long inheritance chains
 # Benefits of prototype pattern function reuse
 */

/* CONS
 # Shouldn't contain reference values
 # Cannot pass arguments into the SuperType constructor
 */
```

Prototype Chaining

```
/* PROS
 # Easy implementation of inheritance
 # Long inheritance chains
 # Benefits of prototype pattern function reuse
 */

/* CONS
 # Shouldn't contain reference values
 # Cannot pass arguments into the SuperType constructor
 */
```



Constructor Stealing

With **Constructor Stealing** of inheritance we can fix the 2 contras of **Prototype Chaining**

- ★ Usage of reference values

Constructor Stealing

```
function SuperType(){
     this.names = [“Sergi”, “Diego”];
}
function SubType(){
     //inherit from SuperType
     SuperType.call(this);
}
var instance1 = new SubType();
instance1.names.push(“Eric”);
alert(instance1.names); // ”Sergi,Diego,Eric”

var instance2 = new SubType();
alert(instance2.names); // ”Sergi,Diego”
```

Constructor Stealing

With **Constructor Stealing** of inheritance we can fix the 2 contras of **Prototype Chaining**

- ★ Usage of reference values
- ★ Pass arguments into the SuperType constructor

Constructor Stealing

```
function SuperType(name){
     this.name = name;
}
function SubType(){
     //inherit from SuperType passing in an argument
     SuperType.call(this, "Natalio");
     //instance property
     this.age = 29;
}
var instance = new SubType();
alert(instance.name); // "Natalio"
alert(instance.age); // 29
```

Combination Inheritance

But **Constructor Stealing** is not enough flexible so we combine it with **Prototype Chaining** to obtain the best of every type of Inheritance.

Combination Inheritance

```
function SuperType(name){
      this.name = name;
      this.colors = [“red”, ”green”];
}
SuperType.prototype.sayName = function(){
      alert(this.name);
}
function SubType(name, age){
      //inherit from SuperType passing in an argument
      SuperType.call(this, name);
      this.age = age;
}
SubType.prototype = new SuperType();
SubType.prototype.sayAge = function(){
      alert(this.age);
}
```

```
var instance1 = new SubType(“Natalio”, 29);
instance1.colors.push(“blue”);
alert(instance1.colors) // ”red,green,blue”
instance1.sayName();    // ”Natalio”
instance1.sayAge();     // 29

var instance2 = new SubType(“Natasha”, 28);
alert(instance2.colors) // ”red,green”
instance2.sayName();    // ”Natasha”
instance2.sayAge();     // 28
```

Prototypal Inheritance by Douglas Crockford

```
function object(o){
     function F(){}
     F.prototype = o;
     return new F();
}
var person = {
     name: “Issel”,
     friends: [“Ruben”, “Elena”]
};
var anotherPerson = object(person);
anotherPerson.name = “Toni”;
anotherPerson.friends.push(“Toni”);
alert(person.friends); // ”Ruben,Elena,Toni”
```

Parasitic Inheritance by Douglas Crockford

```
function createAnother(original){
     var clone = object(original); //Create a new object
     clone.sayHi = function(){     //Augment the object in some way
          alert("Hi");
     }
     return clone;
}
var person = {
     name: "Issel",
     friends: ["Ruben", "Elena"]
};
var anotherPerson = createAnother(person);
anotherPerson.sayHi(); // "Hi"
```

Parasitic Combination Inheritance

```
function SuperType(name){
      this.name = name;
      this.colors = [“red”, “yellow”];
}
SuperType.prototype.sayName = function(){
      alert(this.name);
};
function SubType(name, age){
      SuperType.call(this, name); //Second call to SuperType()
      this.age = age;
}
SubType.prototype = new SuperType(); //First call to SuperType()
SubType.prototype.sayAge = function(){
      alert(this.age);
};
```

Parasitic Combination Inheritance

```
/**
 * To avoid 2 calls to the constructor, you should
 * only call the constructor from the child constructor and
 * instead of calling the constructor to build the child's prototype
 * you can only clone it using parasitic inheritance
 */

function inheritPrototype(subType, superType){
    var prototype = object(superType.prototype); //Create object
    prototype.constructor = subType;             //Augment object
    subType.prototype = prototype;               //Assign object
}
```

Parasitic Combination Inheritance

```
function SuperType(name){
      this.name = name;
      this.colors = [“red”, “yellow”];
}
SuperType.prototype.sayName = function(){
      alert(this.name);
};
function SubType(name, age){
      SuperType.call(this, name);
      this.age = age;
}
inheritPrototype(SubType, SuperType);
SubType.prototype.sayAge = function(){
      alert(this.age);
};
```

Parasitic Combination Inheritance

```
function SuperType(name){
      this.name = name;
      this.colors = ["red", "yellow"];
}
SuperType.prototype.sayName = function(){
      alert(this.name);
};
function SubType(name, age){
      SuperType.call(this, name);
      this.age = age;
}
inheritPrototype(SubType, SuperType);
SubType.prototype.sayAge = function(){
      alert(this.age);
};
```

Thankz

Any question?