allenwb / 0Option3ConstructorSummary.md Secret
Created 6 years ago

Embed <script src="https://gist. Download ZIP

New ES6 constructor features and semantics: Alternative 3 Derived constructors default to ordinary object create

0Option3ConstructorSummary.md

New ES6 Constructor Semantics and Usage Examples

Default Ordinary Allocate: Alternative Design where subclass constructors do not automatically call superclass constructors. bkut instead create a new ordinary object as the initial value of `this`.

This Gist presents a new design of class-based object construction in ES6 that does not require use of the two-phase @@create protocol.

One of the characteristics of this proposal is that subclass constructors must explicitly **super** invoke their superclass's constructor if they wish to use the base class' object allocation and initialization logic. If they don't, a new ordinary object is automatically allocated. This is the same behavior that is used for basic constructor functions in all of the alternative proposals. So, this alterantive can be seen as a unifying the semantics of basic constructor functions and class constructors.

An alternative version of this design automatically invokes the base constructor in most situations.

In addition to the material below, there is a seperate design rationale covering several key features that are common to both designs.

1constructor-summary.md

Constructor Semantics Summary

1. Constructors may be invoked either as a "function call" or by a **new** operator
2. function call: Foo(arg)
3. **new** operator: **new** Foo(arg)
4. Within a constructor, the **new^** token can be used as a *PrimaryExpression* to determine how the construtor was invoked
5. If the value of **new^** is **undefined** then the current function invocation was as a function call.
6. If the value of **new^** is not **undefined** then the current function invocation was via the **new** operator and the value of **new^** is the constructor functon that **new** was originally applied to.
7. When a constructor is invoked using the **new** operator it has two responsibility
8. Allocate (or otherwise provide) a **this** object
9. Initialize the **this** object prior to returning it to the caller
10. The allocation step may be performed automatially prior to evaluating the constructor body or it may be manually performed by code within the consturctor body.
11. Automatic allocation is the default for all class constructors (and constructors created using function definitions). 1. The value of **this** is initialized to a newly allocated ordinary object immediately prior to evaluating the body of the

constructor.

12. If a constructor body contains an assignment of the form **this =** then automatic allocation is not performed and the constructor is expected to perform manual allocation. 1. The value of **this** is uninitialized (in its TDZ) upon entry to the body of a manually allocating constructor. 2. Referencing (either explicitly or implicitly) an unitialized **this** will throw a ReferenceError. 2. The first evaluation of an assignment to **this** initializes it. 3. **this** may be dyanmically asigned to only once within an evaluation of a manaully allocating constructor body. 4. When a constructor is invoked via a function call (ie, via [[Call]]), **this** is preinitialized using the normal function call rules. Any dynamic assignment to **this** during such an invocation will throw a ReferenceError.
13. Executing **new super** in a constructor (that was itself invoked using the **new** operator) invokes its superclass constructor using the same **new^** value as the invoking constructor.

2usage-patterns.md

Usage Pattern for a Proposed New ES6 Constructor Semantics

The patterns below progressively explain the key semantic features of this proposal and their uses. A more complete summary of the semantics for this proposal are in Proposed ES6 Constructor Semantics Summary.

Allocation Patterns

A base class with default allocation that needs no constructor logic

A class declaration without an **extends** clause and without a constructor method has a default constructor that allocates an ordinary object.

```
class Base0 {};

let b0 = new Base0;   //equivalent to: let b0 = Object.create(Base0.prototype);
```

A base class with default allocation and a simple constructor

The most common form of constructor automatically allocates an ordinary object, binds the object to **this**, and then evaluates the constructor body to initialize the object.

```
class Base1 {
  constructor(a) {
    this.a = a;
  }
}

let b1 = new Base1(1);   //equivalent to: let b1 = Object.create(Base1.prototype);
                         //               b1.a = 1;
```

A base class that manually allocates an exotic object

Any constructor body that contains an explicit assignment to **this** does not perform automatic allocation. Code in the constructor body must allocate an object prior to initializing it.

```
class Base2 {
  constructor(x) {
    this = [ ];  //create a new exotic array instance
    Object.setPrototypeOf(this, new^.prototype);
    this[0] = x;
  }
  isBase2 () {return true}
}

let b2 = new Base2(42);
assert(b2.isBase2()===true);
assert(Array.isArray(b2)===true);  //Array.isArray test for exotic array-ness.
assert(b2.length == 1);  //b2 has exotic array length property behavior
assert(b2[0] == 42);
```

A derived class with that adds no constructor logic

Often derived classes want to just use the inherited behavor of their superclass constructor. If a derieved class does not explicitly define a constructor method it automatially inherits its superclass constructor, passing all of its arguments..

```
class Derived0 extends Base1 {};

let d0 = new Derived0("b1");
assert(d0.a === "b1");
assert(d0.constructor === Derived0);
assert(Object.getPrototypeOf(d0) === Derived0.prototype);
```

The above definition of Derived0 is equivalent to:

```
class Derived0 extends Base1 {
  constructor() {
    this = new super(...arguments);
  }
}
```

Note that the **this** value of a constructor is the default value of the invoking **new** expression unless it is explicilty over-ridden by a **return** statement in the constructor body. (This is a backwards compatabible reinterpretation of legacy [[Construct]] semantics, updated to take into account that the initial allocation may be performed within the constructor body.)

A derived class that extends the behavior of its base class constructor

A derived class that wishes to use and extend its base class constructor must explicitly invoke the base constructor and may assign the resulting value to **this**. Such classes typically will first invoke its base class constructor using the **new** operator, to obtain a new instance. It then evaluates the remainder of the derived class constructor using the value returned from the base constructor as the **this** value.

```
class Derived1 extends Base1 {
  constructor(a, b) {
    this = new super(a); //note only a passed to super constructor
    this.b = b;
  }
};

let d1 = new Derived1(1, 2);
assert(d1.a === 1);   //defined in Base1 constructor
assert(d1.b === 2);   //defined in Derived1 constructor
```

A derived class that invokes its base constructor with permutated arguments

A derived class may perform arbitrary computations prior to calling its base constructor, as long as the computations don't reference **this**. For example, if the derived constructor needs to modify the arguments passed to the base constructor, it must perform the necessary computations prior to invoking the base constructor and assigning the result to **thus**.

```
class Derived2 extends Derived1 {
  constructor(a, b, c) {
    this = new super(b, a);
    this.c = c;
  }
};

let d2 = new Derived2(1, 2);
assert(d2.a === 2);
assert(d2.b === 1);
assert(d2.c === 3);
```

A derived class that invokes its base constructor with computed arguments

```
class Derived3 extends Derived1 {
  constructor(c) {
    let a = computeA(), b = computeB(); //note, can't reference 'this' here.
    this = new super(a, b);
    this.c = c;
  }
};

let d3 = new Derived3(3);
assert(d3.c === 3);
```

A derived class that doesn't use its base class constuctor

```
class Derived4 extends Base2 {
   constructor (a) {
      this.a = a;
    }
  isDerived4() {return true};
}

let d4 = new Derived4(42);
assert(d4.isBase2()===true);  //inherited from Base2
assert(d4.isDerived4()===true);; //from Derived4
assert(Array.isArray(d4)===false); //not an exotic array object.
assert(d4.hasOwnProperty("length")===false);
assert(d4.a == 42);
```

A derived class that wraps a Proxy around the object produced by its base constructor

```
class Derived5 extends Derived1 {
  constructor() {return Proxy(new super(...arguments), new^.handler())};
  static handler() {
    return {
      get(target, key, receiver) {
        console.log(`accessed property: ${key}`);
        return Reflect.get(target, key, receiver);
      }
    };
  };
};
```

Use of **return** is appropiate here because we don't need to reference **this** within the body of the constructor. Using **return** replaces the default ordinary object that was allocated.

Additional classes that use different handlers can be easily defined:

```
class Derived6 extends Derived5 {
  static handler() {
    return Object.assign({
      set(target, key, value, receiver) {
        console.log(`set property: ${key} to: ${value}`);
        return Reflect.get(target, key, value, receiver)
      }
    }, super.handler());
  };
};
```

Note that Derived6 doesn't need an explicit constructor method definition. Its automatically generated constructor performs `this=new super(...arguments);`. ####An Abstract Base class that can't be instantiated using new

```
class AbstractBase {
  constructor() {
      this = undefined;
  }
  method1 () {console.log("instance method inherited from AbstractBase");
  static smethod () {console.log("class method inherited from AbstractBase")};
};

let ab;
try {ab = new AbstractBase} {catch(e) {assert(e instance of TypeError)};
assert(ab===undefined);
```

Classes derived from AbstractBase must explicitly allocate their instance objects.

```
class D7 extends AbstractBase {
  constructor () {
    return this; //be explicitly that we are using the automatically allocated ordinary object
  }
}

let d7 = new D7();
d7.method1();  //logs message
D7.smethod();  //logs message
assert(d7 instanceof D7);
assert(d7 instanceof  AbstractBase);

class D8 extends AbstractBase {};
new D8;  //throws TypeError because result of new is undefined
```

Constructor Called as Function Patterns

A unique feature of ECMAScript is that a constructor may have distinct behaviors depending whether it is invoke by a **new** expression or by a regular function call expression.

Detecting when a constructor is called as a function

When a constructor is called using a call expression, the token **new^** has the value **undefined** within the constructor body.

```
class F0 {
  constructor() {
    if (new^) console.log('Called "as a constructor" using the new operator.');
    else console.log('Called "as a function" using a function expression.');
  }
}
```

```
new F0; //logs: Called "as a constructor" using the new operator.
F0();   //logs: Called "as a function" using a function expression.
```

A constructor that creates new instances when called as a function

```
class F1 {
  constructor() {
    if (!new^) return new F1;
  }
}

assert((new F1) instanceof F1);
assert(F1() instanceof  F1);
```

A constructor that refuses to be called as a function

```
class NonCallableConstructor {
  constructor () {
    if (!new^) throw Error("Not allowed to call this constructor as a function");
  }
}
```

A constructor with distinct behavior when called as a function

```
class F2 {
  constructor(x) {
    if (new^) {
      //called as a constructor
      this.x = x;
    } else {
      //called as a function
      return x.reverse();
    }
  }
};

let f2c = new F2("xy");
let f2f = F2("xy");
assert(typeof f2c == "object") && f2c.x ==="xy");
assert(f2f === "yx");
```

super calls to constructors as functions

The distinction between "called as a function" and "called as a constructor" also applies to **super** invocations.

```
class F3 extends F2 {
  constructor(x) {
    if (new^) {
      this = new super(x+x);  //calls F2 as a constructor
    } else {
      return super(x) + super(x); //calls F2  as a function (twice)
    }
};

let f3c = new F3("xy");
let f3f = F3("xy");
assert(typeof f3c == "object") && f3c.x ==="xyxy");
assert(f3f === "yxyx");
```

Calling a superclass constructor to perform instance initialization.

A base class constructor that is known to perform automatic allocation may be called (as a function) by a derived constructor in order to apply the base initialization behavior to an instance allocated by the derived constructor.

```
class D8 extends Base1 {
  constructor(x) {
    //note, an ordinary object was automatically allocated and assigned to this
    super(x);  //note calling super "as a function", passes this,
               // and does not do any automatic allocation
  }
}

let d8 = new D8(8);
assert(d8.x==8);
```

However, care must be taken that the base constructor does not assign to **this** when "called as a function".

Patterns for Alternative Construction Framework

Two phase construction using @@create method

This construction framework breaks object construction into two phase, an allocation phase and an instance initializaiton phase. This framework design essentially duplicates the @@create design originally proposed for ES6. The design of this framework uses a static "@@create" method to perform the allocation phase. The @@create method may be over-ridden by subclass to change allocation behavior. This framework expects subclasses to place instance initialization logic into the consturctor body and performs top-down initializaiton.

```
Symbol.create = Symbol.for("@@create");
class BaseForCreate{
  constructor( ) {
    this = new^[Symbol.create]();
  }
  static [Symbol.create]() {
    // default object allocation
    return Object.create(this.prototpe);
  }}

class DerivedForCreate1 extends BaseForCreate {
  //A subclass that over rides instance initialization phase
  constructor(x) {
     this = new super();
    // instance initialization logic goes into the constructor
    this.x = x;
  }
}

class DerivedForCreate2 extends BaseForCreate{
  //A subclass that over rides instance allocation phase
  static [Symbol.create]() {
    // instance allocation logic goes into the @@create method body
    let obj = [ ];
    Object.setPrototypeOf(obj, this.prototype);
    return obj;
  }
}
```

Two phase construction using initialize method

This construction framework also breaks object construction into two phase, an allocation phase and an instance initializaiton phase. The design of this framework uses the constructor method to perform the allocation phase and expects subclasses to provide a seperate initializaiton method to peform instance initialization. The initialize methods control whether initialization occur in a top-down or buttom-up manner.

```
class BaseForInit {
  constructor(...args) {return this.initialize(...args)}
  initialize () {return this}
}

class DerivedForInit1 extends BaseForInit {
  //A subclass that over rides instance initialization phase
  initialize(x) {
    // instance initialization logic goes into an initialize method
    this.x = x;
  }
}

class DerivedForInit2 extends BaseForInit {
  //A subclass that over rides instance allocation phase
  constructor(...args) {
    // instance allocation logic goes into the constructor body
    this = [ ];
    Object.setPrototypeOf(this, new^.prototype);
    return this.initialize(...args);
  }
}

class DerivedForInit3T extends DerivedForInit1 {
  //A subclass that over rides instance initialization phase
  //and performs top-down super initialization
  initialize(x) {
    super.initialize();    //first perform any superclass instance initization
    this.x = x;
  }
}

class DerivedForInit3B extends DerivedForInit1 {
  //A subclass that over rides instance initialization phase
  //and performs bottom-up super initialization
  initialize(x) {
    this.x = x; //first initialize the state of this instance
    super.initialize();    //then perform superclass initization
  }
}
```

Some AntiPatterms

Using return instead of this to replace default allocation

```
class Anti1 {
  constructor() {
    return Object.create(new^.prototype);
    //better: this = Object.create(new^.prototype);
    //or: this = {__proto__: new^.prototype};
  }
}
```

JavaScript has always allowed a constructor to over-ride its autmatically allocated instance by returning a different object value. That remains the case with this design. However, using **return** in this manner (instead of assigning to **this**) may be less efficient because the constructor is specified to still automatically allocates a new instance.

Calling super() instead of invoking super() with new

```
class Derived2Bad extends Derived1 {
  constructor(a, b, c) {
    this = /*new*/ super(b, a);  //what if we forget to put in new
    this.c = c;
  }
};

new Derived2Bad(1,2,3); //ReferenceError
```

This constructor contains an assignment to **this** so it doesn't perform automatic allocation and **this** is initially uninitialized. It assigns to **this** but the **super()** call in its first statement implicitly references **this** before it is initialized so a ReferenceError exception will be thrown.