Linguagens e Ambientes de Programação (2018/2019)

Teórica 19 (20/mai/2019)

Programação orientada pelos objetos em JavaScript.

Objetos literais

Em JavaScript, os objetos são praticamente simples dicionários, que mapeiam etiquetas em valores. Existe uma sintaxe própria para definir objetos diretamente.

Eis um exemplo de **objeto literal**, que define uma pessoa:

```
var p = {name: "Pedro", address: "Lisboa", age: 42};
```

Para aceder a uma componente dum objeto, há duas notações disponíveis:

```
var n = p.name;                    // Notação 1
var n = p["name"];                 // Notação 2
p.name = "Pedrinho";               // Muda nome
```

Se atribuirmos a um membro inexistente dum objeto, esse membro passa imediatamente a existir para esse objeto individual:

```
p.born = "Porto";
```

Para apagar um membro, usa-se a palavra `delete`:

```
delete p.born;
```

Eis um objeto mais complexo:

```
var myStructure = {
  name: {
    first: "Mel",
    last: "Smith"
    fullname: function() { return this.first + " " + this.last}
  },
  age: 33,
  hobbies: [ "chess", "jogging" ]
}
```

Agora uma definição mais rigorosa e completa de objeto: um objeto é um dicionário enriquecido por uma propriedade privada especial que se chama **prototype**.

Programação baseada em protótipos

Para criar múltiplos objetos semelhantes e para reutilizar código, o JavaScript original não usa classes mas sim **protótipos**. Esta técnica foi inventada em meados dos anos 1980 no contexto da linguagem Self. É uma técnica natural no contexto duma linguagem dinâmica.

Comecemos por falar um pouco da linguagem Self.

Em Self, a criação de novos objetos é efetuada a partir de objetos existentes. Sempre que um objeto P é usado como base para a criação de novos objetos, diz-se que P é um **protótipo**.

A criação dum novo objeto a partir dum protótipo P (designemos a operação por `copy(P)`) é muito simples: cria-se um objeto vazio (sem propriedades) e faz-se a propriedade especial `prototype` do novo objeto referir o protótipo P. Todos os objetos criados a partir dum protótipo P começam vazios e ficam a referir esse mesmo P.

Cada objeto **herda dinamicamente** do respetivo protótipo. A herança funciona assim: quando se tenta aceder a um membro dum objeto, se esse membro não estiver diretamente disponível no objeto, então a procura continua no respetivo protótipo. Se também não estiver diretamente disponível no protótipo, então procura-se no protótipo do protótipo. E assim sucessivamente, ao longo duma cadeia de protótipos.

Note que qualquer objeto pode ser usado como protótipo. Qualquer objeto passa a ser considerado um protótipo a partir do momento em que é usado para criar novos objetos.

Agora regressemos à linguagem JavaScript.

Em JavaScript, os objeto são idênticos aos do Self na medida em que contêm uma propriedade especial que identifica um protótipo, e cada objeto herda dinamicamente do seu protótipo. No entanto, o mecanismo disponível para a criação de objetos é mais complicado do que o do Self (veremos esse mecanismo na secção seguinte).

Em JavaScript, o protótipo dum objeto é guardado na seguinte propriedade privada:

```
[[Prototype]]
```

Algumas implementações de JavaScript expõem essa propriedade através da propriedade pública `__proto__`. Nas implementações de JavaScript usadas na nossa cadeira (Rhino e NodeJS), esta propriedade pública está disponível e é usada como se exemplifica:

```
obj.__proto__
```

De qualquer maneira, a maneira padronizada de testar se `proto` é protótipo de `obj`, é a seguinte:

```
proto.isPrototypeOf(obj)
```

Todos os objetos definidos através dum literal partilham automaticamente um protótipo predefinido que se escreve:

```
Object.prototype
```

Veja esta pequena sessão interativa que prova de duas maneiras diferentes que os objetos literais herdam realmente de `Object.prototype`:

```
> var a = {name: "Pedro", address: "Lisboa", age: 42}
> Object.prototype.isPrototypeOf(a)
true
```

```
> a.__proto__ === Object.prototype    // "===" verifica se dois objetos são o mesmo.
true
```

Programação orientada pelos objetos em JavaScript usando construtores

Em JavaScript, qual o mecanismo previsto para criar novos objetos e lhes associar o protótipo de onde herdam?

Convém começar por dizer que o mecanismo usado no JavaScript é um pouco complicado e foge ao que é tradicional nas linguagem baseadas em protótipos (que costumam imitar o Self). Foi provavelmente a pensar nos programadores de Java que se decidiu introduzir um mecanismo com alguma aparência de familiaridade.

O mecanismo usado em JavaScript para criar objetos é o mecanismo dos **construtores**. Um **construtor** serve para inicializar diversos objetos do mesmo tipo, que herdam do mesmo protótipo.

Um construtor `C` é um função com as seguintes particularidades:

- Destina-se a ser chamada no contexto do operador `new`, com esta sintaxe **`new C()`**. O operador `new` cria um objeto vazio que depois é inicializado pelo construtor. Dentro do construtor, o objeto novo é conhecido por `this` e a sua inicialização envolve tipicamente apenas campos de dados; não campos funcionais.
- O construtor `C` é dono dum objeto vazio, que se tornará o protótipo de todos os objetos criados a partir desse construtor. Esse objeto escreve-se **`C.prototype`** e diz-se que é o *protótipo do construtor*.

Regra geral, os campos funcionais são adicionados no protótipo do construtor, para ficarem acessíveis por herança. Quando se adicionam novas funções ao protótipo, ele deixa de ser vazio. Repare bem: geralmente, os campos de dados pertencem a cada objeto e não são partilhados; os campos funcionais são metidos no protótipo e são partilhados por herança (pois não vale a pena repeti-los em cada objeto).

Abaixo define-se um construtor chamado Car para representar e inicializar automóveis. Neste exemplo, cada objeto fica com três campos de dados próprios. Os campos funcionais são seguidamente instalados no protótipo de Car e ficam disponíveis através de herança. Veja tudo com atenção:

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

Car.prototype.toString = function() {
    return "A Beautiful " + this.year + " " + this.make + " " + this.model;
}

Car.prototype.changeMake = function(make) {
    this.make = make;
}

var car1 = new Car("Toyota", "Corolla", 2002);
```

Para aceder ao construtor dum objeto `obj` escreve-se:

```
obj.constructor
```

Para aceder ao protótipo dum construtor `C` escreve-se:

```
C.prototype
```

Para testar se o construtor dum objeto `obj` é `C`, escreve-se:

```
obj instanceof C
```

Criação de hierarquias através dos construtores

Manipulando diretamente o membro `prototype` dos construtores é possível criar uma hierarquia de protótipos.

Eis um exemplo simples, que introduz um subtipo de Car. Repare que se muda o protótipo de FlyingCar para ser um objeto de tipo Car (em vez do habitual objeto vazio inicial).

```
function FlyingCar(make, model, year, maxFlightLevel) {
   this.make = make;
   this.model = model;
   this.year = year;
   this.maxFlightLevel = maxFlightLevel;
   this.flightLevel = 0;
}

FlyingCar.prototype = new Car("", "", 0); // muda-se aqui o protótipo!!!

FlyingCar.prototype.fly = function() {
    return "Flying level = " + this.flightLevel;
}

var flyingCar1 = new FlyingCar("Toyota", "CFly", 2099, 100);
```

Para tirar possível dúvidas sobre o mecanismo de herança, estudemos a seguinte chamada:

```
flyingCar1.fly();
```

Primeiro procura-se um campo `fly` no objeto `flyingCar1`. Não se encontra. Depois procura-se no protótipo desse objeto. Encontra-se!

Estudemos agora a chamada:

```
flyingCar1.toString();
```

Primeiro procura-se um campo `toString` no objeto no objeto `flyingCar1`. Não se encontra. Depois procura-se no protótipo desse objeto. Não se encontra. Depois procura-se no protótipo do protótipo. Encontra-se!

Programar em JavaScript, imitando as classes do Java

No ECMAScript 6, em 2015, foram introduzidas **classes** para trabalhar com objetos e herança. Essas classes fazem lembrar as classes do Java. Oferecem sintaxe familiar para lidar com os objetos do JavaScript. Estão disponíveis as mesmas palavras reservadas do Java, com significados absolutamente idênticos: **`class`**, **`extends`**, **`this`**, **`super`**, **`static`**, **`new`** e **`instanceof`**.

Usando estas classes é possível programar em JavaScript usando as abordagens habituais do Java e uma sintaxe bastante parecida. Mesmo assim, é importante listas as diferenças mais importantes:

- Nas classes do JavaScript não existe suporte direto para membros de dados privados. Os nossos membros de dados terão de ser públicos, mas tentaremos programar como se eles fossem privados.
- Em JavaScript não há interfaces.

- Em JavaScript os construtores são herdados, ao contrário do que acontece em Java. Em JavaScript, se uma subclasse não definir um construtor, ela herda o construtor da superclasse.
- Em JavaScript não é possível escrever tipos nas declarações.
- Em JavaScript existe a necessidade de escrever constantemente **this** para referir os campos do próprio objeto.

Repare que as classes do JavaScript são apenas **açúcar sintático**, porque internamente usam-se os objetos e os protótipos originais do JavaScript.

Os dois exemplos que se seguem ilustram a forma de programar em JavaScript usando classes, imitando fielmente o estilo do Java.

Exemplo 1

Hierarquia de classes para representar pontos a uma dimensão, duas dimensões e três dimensões. Fatoriza-se o código ao máximo, inclusivamente usando a construção `super` sempre que for aplicável.

```
class Point1 {

    constructor(x) {
        this.x = x;
    }

    static zero() {
        return new Point1(0);
    }

    equals(that) {
        return this.x == that.x;
    }

    shift(deltax) {
        this.x += deltax;
    }

    show() {
        console.log("(" + this.x + ")");
    }

    a() {
       console.log(1);
    }
}

class Point2 extends Point1 {

    constructor(x, y) {
        super(x);
        this.y = y;
    }

    static zero() {
        return new Point2(0, 0);
    }

    equals(that) {
        return super.equals(that) && this.y == that.y;
    }

    shift(deltax, deltay) {
        super.shift(deltax);
        this.y += deltay;
    }
```

```
    show() {
        console.log("(" + this.x + ", " + this.y + ")");
    }

    a() {
       console.log(2);
    }

    b() {
       super.a();
    }
}

class Point3 extends Point2 {

    constructor(x, y ,z) {
        super(x, y);
        this.z = z;
    }

    static zero() {
        return new Point3(0, 0, 0);
    }

    equals(that) {
        return super.equals(that) && this.z == that.z;
    }

    shift(deltax, deltay, deltaz) {
        super.shift(deltax, deltaz);
        this.z += deltaz;
    }

    show() {
        console.log("(" + this.x + ", " + this.y + ", " + this.z + ")");
    }

    a() {
       console.log(3);
    }

    test() {
        this.a();
        super.a();
        this.b();
        super.b();
    }
}

class Tests {
    static testInheritance() {
        var a = new Point3(1,2,3);
        var b = new Point3(6,7,8);
        a.show(); b.show();
        a.shift(1, 1, 1); a.show(); b.show();
        console.log("---");
        b.shift(1, 1, 1); a.show(); b.show();
    }

    static testMixedTypes() {  // a parte inválida das operações não tem efeito.
        var a = new Point1(1);
        var b = new Point2(6,7);
        a.show(); b.show();
        a.shift(1, 1, 1); a.show(); b.show();
        console.log("---");
        b.shift(1, 1, 1); a.show(); b.show();
    }

    static run() {
```

```
        this.testInheritance();
        console.log("+++++++++++++++");
        this.testMixedTypes();
        console.log("+++++++++++++++");
        var a = Point3.zero();
        a.test();
    }
}

Tests.run()

// Run tests

> Tests.run()
(1, 2, 3)
(6, 7, 8)
(2, 3, 4)
(6, 7, 8)
---
(2, 3, 4)
(7, 8, 9)
+++++++++++++++
(1)
(6, 7)
(2)
(6, 7)
---
(2)
(7, 8)
+++++++++++++++
3
2
1
1
```

Exemplo 2

Representação de expressões algébricas, onde pode ocorrer uma variável "x".

A representação natural usa uma árvore com nós de tipos variados.

Fatoriza-se o código ao máximo usando classes abstratas e `super`.

Por exemplo, a classe `BinNode`, consideramo-la abstrata porque captura a noção abstrata de nó binário e porque não tencionamos criar objetos desse tipo. Algum do código das classes concretas `AddNode` e `MultNode` é fatorizado na classe `BinNode`.

```
class ExpNode {
    constructor() {}
    big() { return size() > 1000; }
}

class BinNode extends ExpNode {
    constructor(l, r) { super(); this.l = l; this.r = r; }
    size() { return 1 + this.l.size() + this.r.size(); }
    height() { return 1 + Math.max(this.l.height(), this.r.height()); }
}

class UnaryNode extends ExpNode {
    constructor(e) { super(); this.e = e; }
    size() { return 1 + this.e.size(); }
    height() { return 1 + this.e.height(); }
}

class ZeroNode extends ExpNode {
```

```
    constructor() { super(); }
    size() { return 1; }
    height() { return 1; }
}

class AddNode extends BinNode {
    constructor(l, r) { super(l, r); }
    eval(v) { return this.l.eval(v) + this.r.eval(v); }
    deriv() { return new AddNode(this.l.deriv(), this.r.deriv()); }
}

class MultNode extends BinNode {
    constructor(l, r) { super(l, r); }
    eval(v) { return this.l.eval(v) * this.r.eval(v); }
    deriv() { return new AddNode(new MultNode(this.l, this.r.deriv()),
                                       new MultNode(this.l.deriv(), this.r)); }
}

class SimNode extends UnaryNode {
    constructor(e) { super(e); }
    eval(v) { return -this.e.eval(v); }
    deriv() { return newSimNode(this.e.deriv()); }
}

class ConstNode extends ZeroNode {
    constructor(c) { super(); this.c = c; }
    eval(v) { return this.c; }
    deriv() { return new ConstNode(0); }
}

class VarNode extends ZeroNode {
    constructor() { super(); }
    eval(v) { return v; }
    deriv() { return new ConstNode(1); }
}

class Tests2 {
    static test1() {
        var e = new MultNode(
                        new AddNode(new ConstNode(4), new VarNode),
                        new ConstNode(6.5));

        console.log("((4 + x) * 6.5)' (3) = " + e.deriv().eval(3));
    }
}

// Run tests

> Tests2.test1()
((4 + x) * 6.5)' (3) = 6.5
```

#80