System- och webbutvecklare JavaScript

(http://suw16-lernia.nodehill.se/)

Repetition: Klasser i JavaScript

February 23, 2017 at 5:45 am
JavaScript (http://suw16-lernia.nodehill.se/category/javascript/), OOP (http://suw16-lernia.nodehill.se/category/oop/)
Författare: admin

Grunder

En minimal klass

Det här är en klassdefinition i JavaScript:

```
class Person {
}
```

Vi kan nu skapa instanser av den så här:

```
var aPerson = new Person();
var anotherPerson = new Person();
```

En klass med en konstruktor

Konstruktorn är en speciell metod. Den tar emot inargumenten som skickas in från vid instantiering (**new**). Vi låter normalt sätt konstruktorn föra över inargumenten till olika instansegenskaper:

```
class Person {

  constructor(firstName = 'John', lastName = 'Doe', age = 25){
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

}

var aPerson = new Person('Anna', 'Andersson', 32);
```

Ett bättre alternativ:
En konstruktor med *ett enda* inargument

Ett alternativ till att skicka flera inargument till konstruktorn är att skicka ett settings-objekt till konstruktorn:

```
class Person {

  constructor(settings){
    var defaults = {
      firstName: 'John',
      lastName: 'Doe',
      age: 25
    };
    Object.assign(this, defaults, settings);
  }

}

var aPerson = new Person({
   firstName: 'Anna',
   lastName: 'Andersson',
   age: 32
});
```

Detta ger oss fördelen att instansiering blir tydligare att läsa och att man inte behöver komma ihåg ordningen på inargumenten.

Egenskaper och metoder

Egenskaper kallar vi det som inte är metoder – som **firstName**, **lastName**, **age**. Metoder kallar vi de funktioner som är kopplade till klassen – t.ex. vår **constructor**.

Vi kan lägga till fler metoder:

```
class Person {

  constructor(settings){
    var defaults = {
      firstName: 'John',
      lastName: 'Doe',
      age: 25
    };
    Object.assign(this, defaults, settings);
  }

  sayHi(){
    return `Hi, my name is ${firstName} ${lastName}!`
  }

}

var aPerson = new Person({
   firstName: 'Anna',
   lastName: 'Andersson',
   age: 32
});
aPerson.sayHi(); // => Hi, my name is Anna Andersson!
```

Arv

Om vi vill kan vi skapa en klass som ÄR som en annan klass men mer specifik (och som ärver alla egenskaper och metoder från sin föräldraklass):

```
class Student extends Person {

   constructor(settings){

     // We must call super - first thing
     // we do in the the constructor
     // if this class extends another one
     // (super is the constructor of the parent class)
     super(settings);

     // The other defaults are already
     // defined in Person
     var defaults = {courses: []};
     Object.assign(this, defaults, settings);

   }

}

var aStudent = new Student();
aStudent.courses.push('Programming');
// {
//   firstName: 'John',
//   lastName: 'Doe',
//   age:25,
//   courses:['Programming']
// }
```

Komposition

Det kallas **komposition** när en klass HAR kopplingar till andra klasser (och ofta även skapar instanser av dessa).

Notera även i detta exempel att en klass kan ärva från inbyggda klasser i JavaScript – i detta fall **Array**.

```
class StudentList extends Array{

  constructor(students){
    super();

    // students is a raw array with
    // generic "settings"-objects each
    // representing a student
    for(let student in students){
      // let's create som real Students
      // and push them to this StudentList
      this.push(new Student(student));
    }
  }

}

var list = new StudentList([
   {
     firstName: 'Anna',
     lastName: 'Andersson',
     age:24,
     courses: ['Programming']
   },
   {
     firstName: 'Beata',
     lastName: 'Bengtsson',
     age:24,
     courses: ['SCRUM']
   },
]);
```

Getters och setters

Getters

Ibland vill vi räkna ut värdet på en egenskap varje gång den efterfrågas, men ändå presentera den som en **egenskap** istället för en **metod** för den som använder sig av klassen. Då skapar vi getters:

```javascript
class Person {

  constructor(settings){
    var defaults = {
      firstName: 'John',
      lastName: 'Doe',
      age: 25
    };
    Object.assign(this, defaults, settings);
  }

  get name(){
    return `${this.firstName} ${this.lastName}`
  }

  get nameInCaps(){
    return this.name.toUpperCase();
  }

}

var aPerson = new Person();
aPerson.name; // => "John Doe"
aPerson.nameInCaps; // => "JOHN DOE"
```

Setters

Vi kan även skapa setters så att vi kör en metod när de som använder sig av klassen tror att de bara sätter en egenskap. I settern kan vi göra kontroller av att värdet som sätts är rimligt:

```
class Person {

  constructor(settings){
    var defaults = {
      firstName: 'John',
      lastName: 'Doe',
      age: 25
    };
    Object.assign(this, defaults, settings);
  }

  get name(){
    return `${this.firstName} ${this.lastName}`
  }

  set name(val){
    val = val.split(' ');
    if(
      val.length != 2 ||
      val[0].length === 0 ||
      val[1].length === 0
    ) {
      throw('A person should have a ' +
       'first name and a last name');
    }
    this.firstName = val[0];
    this.lastName = val[1];
  }

}

var aPerson = new Person();
aPerson.name = 'Erik Arvidsson';
aPerson.firstName; // => "Erik"
aPerson.lastName; // => "Arvidsson"
aPerson.name; // => "Erik Arvidsson"
```

Statiska metoder och egenskaper

Statiska metoder och egenskaper är knutna till klassen, *inte* en enskild instans.

Vi kan t.ex. använda dem för att minnas alla instanser som har skapats av en klass.

```
class Person {

  constructor(settings){
    var defaults = {
      firstName: 'John',
      lastName: 'Doe',
      age: 25
    };
    Object.assign(this, defaults, settings);

    // Add a static property, if not set already
    Person.allInstances = Person.allInstances || [];
    // Push this to the static property allInstances
    Person.allInstances.push(this);
  }

  static get numberOfPersons(){
    return this.allInstances.length;
  }

  static findByFirstName(firstName){
    return this.allInstances.filter((instance)=>{
      return instance.firstName === firstName;
    });
  }

}

var person1 = new Person({firstName:'Anna', lastName: 'Alm'});
var person2 = new Person({firstName:'Beata', lastName: 'Alm'});
var person3 = new Person({firstName:'Anna', lastName: 'Ek'});

// This will return 3
console.log(Person.numberOfPersons);
// This will return an array with two items
// (person1 and person3)
console.log(Person.findByFirstName('Anna'));
```

Kolla vilken klass en instans är av?

Vi kan kontrollera vilken klass en instans är av på två olika sätt (ett som är snällt och tar hänsyn till arv, ett mera exakt):

```
var aStudent = new Student();

// instanceof

// this is true (yes, a Student is a kind of Person);
console.log(aStudent instanceof Person);
// this is of course also true
console.log(aStudent instanceof Student);

// constructor

// this is false (a Student wasn't created by calling Person)
console.log(aStudent.constructor === Person);
// but this is true (a Student was created by calling Student)
console.log(aStudent.constructor === Student);
```

Hur gör man det med klass i JavaScript?

Vissa saker man kan göra med inbyggda nyckelord i andra programmeringsspråk verkar "saknas" i JavaScripts implementation av klasser. Men det finns sätt att åstadkomma det mesta.

Abstrakta klasser

En **abstrakt klass** är en klass som inte kan instansieras i sig, men andra klasser kan ärva från dem. Det finns inget speciellt kommando för detta i JavaScript, men det är ändå enkelt att åstadkomma:

```
class Organism { // an abstract class

  constructor(){
    if(this.constructor === Organism){
      throw("Organism is an abstract class.");
    }
  }

}

class Animal extends Organism {
  //...
}

// this works
var anAnimal = new Animal();
// but this will throw an error
var anOrganism = new Organism();
```

Finala klasser

En **final klass** är en klass som andra klasser inta kan ärva från.

```
class SpecialFrog {

  constructor(){
    if(this.constructor !== SpecialFrog){
      throw("You can't be more special " +
        " than a SpecialFrog!");
    }
  }

}

class SuperSpecialFrog extends SpecialFrog {
  //...
}

// this works
var aSpecialFrog = new SpecialFrog();
// but this will throw an error
var aSuperSpecialFrog = new SuperSpecialFrog();
```

Singletons

En **singleton** är en klass vi bara kan skapa *en* instans av.

Variant 1

```
class God {

  constructor(){

    if(
      this.constructor === God &&
      this.constructor.once
    ){
      throw('There can only be one God!');
    }
    this.constructor.once = true;

  }

}

// this works
var aGod = new God();
// but this will throw an error
var anotherGod = new God();
```

Variant 2

Vi kan även tänka oss en **singleton** som en klass vi inte kan instansiera alls, utan som bara har statiska metoder.

```
class God {

  constructor(){
    God.name = "God";
    throw("You can't construct God!");
  }

  static sayHi(){
    return `Hi, my name is God!`;
  }

}

// this works
God.sayHi();  // => "Hi, my name is God!"
// this will throw an error
aGod = new God();
```

Multipelt arv

Det finns inget multipelt arv i JavaScript. (En klass kan inte ärva från två andra klasser.) Men vi kan simulera det med hjälp av en s.k. **proxy**.

```
class Wolf {

  constructor(){
    this.furry = true;
  }

  howl(){
    return "Auooooowww!";
  }

}

class Human {

  constructor(){
    this.twoLegged = true;
  }

  talk(){
    return "Hi there! I'm a human!";
  }

}

class Werewolf extends Human {

  constructor(){
    super();

    // Inherit that wolf blood
    var wolf = new Wolf();
    return new Proxy(this,{
      get: (obj,prop) => {
        obj = obj[prop] === undefined ? wolf : obj;
        return Reflect.get(obj,prop);
      }
    });

  }

}

var aWerewolf = new Werewolf();
aWerewolf.talk(); // => "Hi there! I am a human!"
aWerewolf.howl(); // => "Auooooowww!"
aWerewolf.furry; // => true
aWerewolf.twoLegged // => true
```

Privata egenskaper och metoder

Det finns inget *enkelt* sätt att göra egenskaper och metoder privata (dvs. ej läs/skriv/anropningsbara utanför klassens egna metoder) i JavaScript.

Ibland används konventionen att börja namnen på egenskaper och metoder man *skulle vilja* att andra programmerare inte rör när de använder klassen med *underscore* – "_".

Men vill man att de verkligen ska vara privata blir man tvungen att skriva ett litet bibliotek (ca 50-100 rader kod) som använder sig av s.k. **proxies**. Ett sådant som du ladda ner här (http://nodebite.se/classprivates.js) är **classprivates**.

Med ett sådant bibliotek kan man få **äkta** privata egenskaper och metoder.

```
class Person {

  constructor(){
    this.noSecret = "Ain't no secret";
    this._secret = "This is a big secret";
  }

  _internalOnly(){
    return this._secret;
  }

  forEveryone(){
    return this._internalOnly();
  }

}

// If we are using the library classprivates
Person = classprivates(Person);

// Test
var aPerson = new Person();
console.log(aPerson.noSecret); // => "Ain't no secret"
console.log(aPerson.secret); // => undefined
console.log(aPerson.forEveryone()); // => "This is a big secret";
console.log(aPerson._internalOnly); // => undefined
```