ES6 Klasser

Erik Olsson

FEU17

Innehållsförteckning

Sammanfattning

2015 släpptes det som kallas för ES6 eller ECMAScript 2015. Detta introducerade en hel rad nya funktioner till JavaScript, varav en utav dem var *Klasser*.
Klasser förenklade och förtydligade skapandet av *constructor functions* över det redan existerande tillvägagångssättet *prototyping* & *inheritance* genom att gruppera ihop större delar av koden under ett och samma scope. Detta betyder dock inte att det är något fel med att använda sig av det äldre sättet om man känner sig mer bekväm med det, speciellt inte eftersom prototyping, i vissa fall, har en något bättre prestanda än klasser.

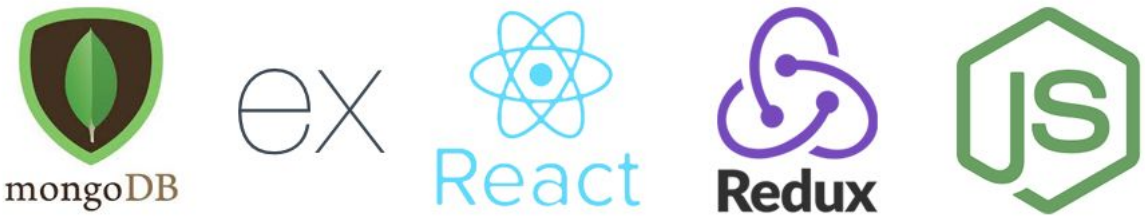
Introduktion

Med introduktionen av ES6 kom bland annat vad man kallar för Classes till Javascript. Ett så kallat "syntaktiskt socker" till Javascripts Prototyping och Inheritance (Tania Rascia, 2018, https://www.digitalocean.com/community/tutorials/understanding-classes-in-javascript) . Men finns det några fördelar (eller nackdelar) med klasser, och när bör man använda sig av Klasser gentemot t.ex. "rena" objekt?

Denna rapport syftar på att fördjupa sig i Javascripts relativt nyintroducerade teknik Class, något som i många andra programmeringsspråk såsom C++ eller Java länge varit en självklarhet.

Bakgrund

Dataspel har precis som programmering länge varit något jag haft ett väldigt stort intresse för och som haft en stor plats i mitt liv, och att göra ett examensarbete som grundar sig i två utav de största och för mig mest viktiga delarna i mitt liv var för mig självklart.
För att lära mig så mycket som möjligt under arbetet så har jag också valt att försöka använda mig av så många tekniker vi lärt oss hittills under utbildningen som möjligt - React och Redux för frontend, tillsammans med Node, Express och MongoDB för backend.



Figur 1: Min stack för detta projekt: MongoDB, Express, React, Redux, och NodeJS

Syfte och Frågeställning

Mitt syfte med detta arbete är att lära mig mer om klasser i Javascript och vad det finns för fördelar och nackdelar med det. Jag har länge läst mycket om klasser i både Javascript och andra programmeringsspråk men har i stort sett aldrig använt mig av det före detta arbete och därför ser jag

det som en bra chans till att både få egen erfarenhet av att använda det men också lära mig om det mer fördjupande.

Avgränsningar

Trots detta finns det en del avgränsningar jag behövt göra för att hinna klart med arbetet inom tidsramen. Detta har inneburit att jag har utelämnat delar av spelet som jag egentligen hade velat göra. T.ex. har jag fått hoppa över att göra ett ”survival mode” där man, till skillnad från det nuvarande gamemodet där man slåss mot 6 förbestämda motståndare i samma ordning, istället hade slagits mot en slumpmässigt genererad motståndare i sin egna nivå varje gång man börjar.

Metod

En stor del av arbetet med att svara på dessa frågor kommer för mig gå ut på att läsa och fördjupa mig i hur prototyper och klasser fungerar i JavaScript, men en del kommer även gå ut på att praktiskt testa vilket som i detta projekt passar bäst och erbjuder den enklaste och smidigaste vägen till det jag vill åstadkomma.

Planen är att innan jag börjar arbetet på spelet kommer jag att lägga c.a en vecka på att leta och läsa artiklar, diskussioner o.s.v som handlar om just Klasser och Prototyper för att lära mig mer, och sedan kommer jag hoppa in i att skapa spelet efter detta.

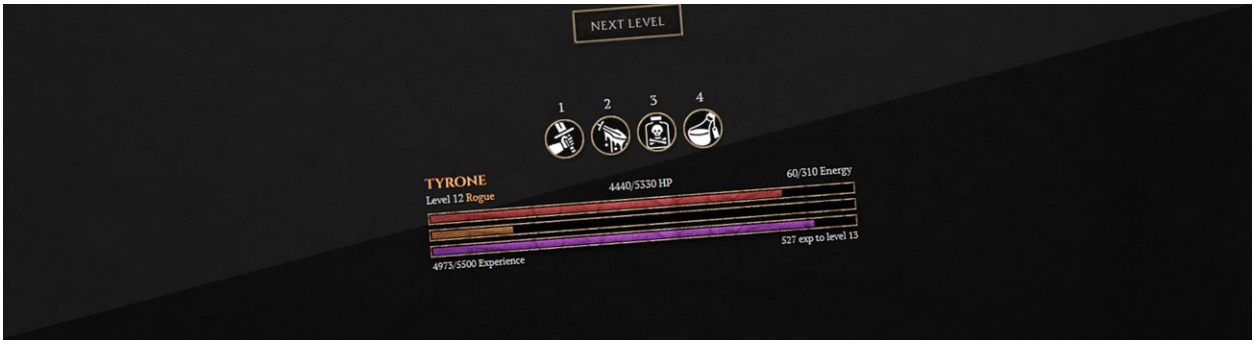
Design

Något som jag är relativt ovan vid att göra men som jag valde att göra för detta projektet är att skapa en wireframe för hur jag ville att min spel skulle se ut, jag valde att rita detta på papper för hand och sedan implementera det i sidan. Trots detta så fick sidan genomgå en hel del större designändringar, både för att implementera funktionalitet och för att bara göra det hela lite tydligare och snyggare. Jag gick totalt igenom 3 olika versioner både på spelar UI:t och på min logotyp för spelet innan jag blev nöjd.

Den första versionen av UI:t fick jag slopa då jag inte kunde komma på ett bra sätt att ändra på "orbsen" (se Figur 2) för att visa när man förlorar HP eller resursers, jag gick då över till att köra på bars istället (som den lila experience baren var redan från början).

Figur 2: UI version 1, svårhanterade HP/Resurs-frames

Det andra UI:t var jag från början ganska nöjd med, men allt eftersom upptäckte jag fler och fler saker jag inte var nöjd med, t.ex. planerade jag inte att få plats med experience-baren från början, vilket hade gjort det lite konstigt att trycka in den någonstans i designen. För det andra blev själva containern som innehåller HP/resurs-barsen lite liten och ihopklämd och jag kände att där inte fanns tillräckligt med white space.
Den tredje versionen blev precis som jag ville, det var enkelt och tydligt vad som är vad, det var inget uteblivet som behövde få plats utan allt är där som ska vara med.



Figur 3: Slutversionen

Klasser, Prototyper och Spells

Som jag skrivit tidigare är *Klasser* alltså endast "syntaktiskt socker" för *Prototyping* och *Inheritance,* och vad det betyder är att kollar man i "bakgrunden" så händer det i stort sett exakt samma sak oberoende på vilket av de två metoderna man använder. Klasser har tagit konceptet bakom prototyping, och förenklat eller förtydligat koden så att den är mer sammanhängande och enkel.
En klass är egentligen bara en ritning som säger hur det objekt man skapar ska se ut, vad det betyder är alltså att det man får ut när man skickar in data för att skapa en ny instans av en klass är ett objekt med den struktur som man förbestämt i klassen.
Samma sak gäller Prototyping, både och använder sig i grunden av vad man kallar en "constructor function", alltså en funktion som tar emot data och *constructar* ("bygger") ett objekt utifrån den specificerade datan.

Det är först när man vill ge sitt objekt olika getters eller setters som de två verkligen skiljer sig åt.
Använder man sig av klasser lägger man helt enkelt bara in funktionen i klassens kropp precis som vi är vana vid i t.ex en React-component.
När man använder sig av prototyping för att lägga till funktionerna så måste man istället lägga till den i constructorns prototype, ändrar man i prototypen ändrar man den för alla instanser man skapat via funktionen.

Från början valde jag att strukturera min kod på ett sånt sätt att varje spell var ett eget objekt med metoder och properties, strukturen såg förenklat ut som i figur 4.

```
let spell = {
    _baseDamage: 15,
    _rank: 1,
    get rank: function() {
        return this._rank
    },
    set rank: function(n) {
        set this._rank = n;
    },
    rankModifiers: [1, 2.25, 3.75],
    get damage: function() {
        return this._baseDamage * this.rankModifiers[this._rank]
    }
};
```

Figur 4: En förenklad återskapning av första versionen av spell strukturen

Kollar man bara på koden som den är så är det inga problem med den, den funkar som är tänkt. Man kan hämta attackens skada genom att kalla på getter funktionen spell.damage och man kan ändra på spellranken genom att kalla på setter funktionen spell.rank.

Jag körde alltså på denna sortens struktur de första veckorna eller så, men när jag väl skulle börja arbeta på att kunna spara en karaktär till databasen eller till localStorage (webbläsarens lagringsminne där man kan spara data på användarens dator mellan användningar) så uppstod problem.
När jag sparade karaktärerna så sparade jag alltså användarens alla spells i en array ungefär som i figur 5.

```
character.spells = [
    spell1 = {
        rank: 1,
        get damage: function() {
            return this._baseDamage * this.rankModifiers[this._rank]
        }
    },
    spell2 = {
        rank: 1,
        get damage: function() {
            return this._baseDamage * this.rankModifiers[this._rank]
        }
    },
    spell3 = {
        rank: 1,
        get damage: function() {
            return this._baseDamage * this.rankModifiers[this._rank]
        }
    }
]
```

Figur 5: Användarens spells som de ser ut när de sparas till databasen

Även här finns det inga tydliga problem med att varje spell är ett objekt med getters och setters, utan det är först när man ska hämta datan från databasen igen som problemen börjar uppstå, nämligen det faktum att det är väldigt svårt att spara en getter/setter-funktion i JSON-format.
Konverterar man en getter-funktion likt de ovan till JSON-format så tappar man getter-funktionaliteten och funktionen kommer att konverteras till ett fast värde, nämligen det värdet som funktionen har när man "kallar" på det vid konverteringen, detta betyder att när man väl laddar in spellsen igen från databasen och "tolkar" (eller parse-ar) dem från JSON till objekt, kommer det istället att se ut som i figur 6.

```
character.spells = [
    spell1 = {
        rank: 1,
        damage: 250
    },
    spell2 = {
        rank: 1,
        damage: 325
    },
    spell3 = {
        rank: 1,
        damage: 95
    }
]
```

Figur 6: Användarens spells som de ser ut efter man hämtat dem från databasen igen

Efter att vi hämtar det från databasen igen har vi alltså tappat all funktionalitet som vi får från getter-funktionen då den istället har konverterats till en string med det aktuella värdet vid konverteringen.

Klasser: En lösning på problemet

För att lösa detta använde jag mig av klasser. Istället för att varje spell var ett objekt med egna getter och setter-funktioner gjorde jag om varje spell till ett "platt" objekt med endast properties i form av strings eller numbers, och skapade sedan en Spell-klass som blev min ritning för skapandet av spells. Nu såg min struktur ut som på figur 7 nedan.

```
let spell = {
    baseDamage: 15,
    rank: 1,
    rankModifiers: [1, 2.25, 3.75]
}

Class Spell {
    constructor(data) {
        this._baseDamage = data.baseDamage,
        this._rank = data.rank,
        this.rankModifiers = data.rankModifiers;
    }

    get damage: function() {
        return this._baseDamage * this.rankModifiers[this._rank];
    }
}
```

Figur 7: Nya spell-strukturen

Vad detta gjorde för mig var alltså att varje gång jag skulle använda mig av spells i själva spelet kunde jag konvertera alla spell-objekt till "riktiga" spells med hjälp av Spell-klassen (se figur 8).

```
character.spells = [
    spell1 = {
    baseDamage: 15,
    rank: 1,
    rankModifiers: [1, 2.25, 3.75]
    },
    spell2 = {
        baseDamage: 15,
        rank: 1,
        rankModifiers: [1, 2.25, 3.75]
    },
    spell3 = {
        baseDamage: 15,
        rank: 1,
        rankModifiers: [1, 2.25, 3.75]
    }
]

character.spells.map(spell => new Spell(spell));
```

Figur 8: Hur jag skapade Spells

På detta sätt kunde jag på ett enkelt sätt spara användarens spells till både databasen och localStorage i JSON-format utan att tappa någon funktionalitet eller data, och sedan enkelt konvertera spellsen till en Klass som hade alla metoder jag behövde.

Detta va såklart möjligt även innan Klasser introducerades till JavaScript, men tillvägagångssättet i min åsikt är en hel del krångligare och mindre tydligt. Koden för en "Spell" kan med det som kallas för prototyping, alltså att man använder sig av funktioner och deras prototyper istället, se ut som på figur 9.

```
function Spell (data) {
    this._baseDamage = data.baseDamage;
    this._rank = data.rank;
    this.rankModifiers = data.rankModifiers;
}

Object.defineProperty(Spell.prototype, "damage", {
    get: function() {
        return this._baseDamage * this.rankModifiers[this._rank];
    }
})
```

Figur 9: En likadan Spell som tidigare fast som använder sig av Prototyping istället för Klasser

Som man kan se är det ingen större skillnad i mängden kod för att göra samma sak, men enligt mig är det mycket tydligare vad som händer och vad som hänger ihop när man använder sig av klasser gentemot att använda sig av prototyping, dels då allting håller ihop mer eftersom allt ligger i samma scope, och dels för att det är mer "straight-forward".

Prestanda: Klasser mot Prototyper

Något som inte är så viktigt för en relativt simpel sida men som vid större projekt spelar en väldigt stor roll är prestanda. Gregory Soloshchenko har i sin artikel *ES6 classes vs Prototypes performance overview* (https://medium.com/@soloschenko/es6-classes-vs-prototypes-performance-overview-dcab1e2fca9b) testat prestandan för både klasser och prototyper under 3 olika fall:

1. Skapa ett nytt objekt som en instans av en tom klass mot en tom funktion
2. Skapa ett nytt objekt som en instans av en tom klass som ärver från (extends) en annan tom klass mot en tom funktion som ärver från en annan tom funktion
3. Skapa ett nytt objekt som en instans av en tom klass som ärver från Date() mot en tom funktion som ärver från Date()

Alla testfall kördes både i Chrome och en lokal Node miljö där han körde varje testfall 1000 gånger, och för varje testfall skapande han 100 000 instanser av vardera typ.
Det slutgiltiga resultatet är att i det första testfallet är prototyper någon millisekund långsammare än klasser, men i båda de två andra testfallen där dem ärver från en annan klass/funktion eller från Date() så har prototyper ett litet övertag över klasser.

	ES6 Classes, ms	Prototypes, ms
Min	1	1
Max	11	5
Med	2	2
Avg	2.35	2.37

Figur 10: Soloshchenko, Gregory. Medium.com, "ES6 classes vs Prototypes performance overview", Testfall 1

	ES6 Classes, ms	Prototypes, ms
Min	4	1
Max	9	5
Med	6	3
Avg	5.99	2.95

Figur 11: Soloshchenko, Gregory. Medium.com, "ES6 classes vs Prototypes performance overview", Testfall 2

Analys av resultat

Det är, enligt det jag kommit fram till, ingen större skillnad på klasser eller prototyping. Klasser är, precis som kdex skriver på MDN om klasser (kdex, 2017, Classes (MDN) https://developer.mozilla.org/sv-SE/docs/Web/JavaScript/Reference/Classes), endast syntetisk socker till prototyping, eller *prototype-based inheritance.* Prototyper har, i vissa fall, något bättre prestanda än klasser när det kommer till att skapa flera hundratusentals instanser, och det kan därför vara bättre att använda sig av prototyper om man t.ex. ska skapa ett spel där man använder sig av extrema mängder instanser för saker såsom partiklar. Men i de flesta fallen finns det finns det ingen anledning att välja det ena över det andra om man endast ser till prestandan.
När det kommer till det teoretiska så är klasser, trots ungefär samma mängd kod för att utföra samma uppgift som via prototyping, något enklare att förstå då all kod ligger sammanhängande i samma scope, istället för att vara uppdelad så som den ofta blir om man använder sig av prototyping.

Avslutande diskussion

Vad detta betyder är alltså att det inte spelar någon roll vilken av de två sätten man använder, utan att det bästa är att man använder sig av det man är van vid och har enklast för att använda.
Om man kommer från ett programmeringsspråk där man använt sig av klasser förut, kanske det är enklare att fortsätta med det även i JavaScript, men om JavaScript var ens första programmeringsspråk och man lärde sig använda prototyper innan klasser ens introducerades, då kan det vara smartare att fortsätta använda prototyper.
Det är helt upp till var och en att bestämma vilket som passar bäst (eller kanske ens arbetsgivare/team om det är så) för det man vill åstadkomma.

När det kommer till vart man ska använda sig av klasser gentemot objekt så beror det väldigt mycket på situationen och vad man skall göra. Om man bara behöver ett objekt så kan man hålla sig till att inte använda klasser, men om man vet att man skall göra flertal objekt med liknande eller samma struktur, kan det vara bra att använda sig av någon sorts constructor function.

Litteraturlista

Grey B, 2018, Should You Use Classes in Javascript?
https://medium.com/@vapurrmaid/should-you-use-classes-in-javascript-82f3b3df6195
Tania Rascia, 2018, Understanding Classes in Javascript
https://www.digitalocean.com/community/tutorials/understanding-classes-in-javascript

Dan Abramov, 2015, How to Use Class and Sleep at Night
https://medium.com/@dan_abramov/how-to-use-classes-and-sleep-at-night-9af8de78ccb4
Valentin PARSY, 2018, Javascript: Prototype vs Class
https://medium.com/@dan_abramov/how-to-use-classes-and-sleep-at-night-9af8de78ccb4
Rajaraodv, 2016, Is ”Class” In ES6 The New ”Bad” Part?
https://medium.com/@rajaraodv/is-class-in-es6-the-new-bad-part-6c4e6fe1ee65
kdex, 2017, Classes (MDN)
https://developer.mozilla.org/sv-SE/docs/Web/JavaScript/Reference/Classes
indiesquidge, 2018, Objects Over Classes
https://gist.github.com/indiesquidge/f8c486795d7dd455c0327ce7e0aa8c16
Jeff-Mot-Or, 2017, JavasScript/C++ Rosetta Stone
https://github.com/Jeff-Mott-OR/javascript-cpp-rosetta-stone
MoTTs_, 2018
https://www.reddit.com/r/javascript/comments/8q6267/why_are_js_classes_not_real_classes/e0gyorf/
Celsiusnotes.com, 2019, Prototypes in JavaScript
http://celsiusnotes.com/_proto_-prototype-prototype-etc-in-javascript/
Eric Elliott, 2016, Master the JavaScript Interview: What’s the Difference between Classes & Prototypal Inheritance
https://medium.com/javascript-scene/master-the-javascript-interview-what-s-the-difference-between-class-prototypal-inheritance-e4cd0a7562e9
Gregory Soloshchenko, 2017, ES6 Classes vs Prototypes performance overview
https://medium.com/@soloschenko/es6-classes-vs-prototypes-performance-overview-dcab1e2fca9b