Sviluppare VideoGiochi con HTML5

HTML5 rappresenta un insieme di tecnologie introdotte negli ultimi anni per lo sviluppo di applicazioni Web. Tecnologie così potenti che consentono anche di creare applicazioni complesse come i videogame. Iniziamo col definire a grandi linee gli elementi di base che sfrutteremo per comporre un videogame:

- markup **HTML** (parte statica), che fornisce il contenitore del gioco e in cui definiamo quali API utilizzare e quali eventuali librerie linkare.
- codice **JavaScript** (parte dinamica), è il codice che utilizziamo per sfruttare le API messe a disposizione dal contenitore e creare il nostro Game Engine.
- A livello di contenitore introduciamo l'uso dell'elemento **canvas**, che porta con sé le API per gestire il disegno di immagini e primitive, per produrre applicazioni animate, effetti sonori ecc. Approfondimenti: https://www.w3schools.com/html/html5_canvas.asp

Software e strumenti da utilizzare

Per seguire gli esempi proposti nella guida avremo bisogno di poco software di base:

- Un browser munito di console per il Debug del nostro codice (consiglio **Google Chrome**)
- Un editor di testo per scrivere il codice (tra i migliori **Notepad++**, un editor gratuito e opensource, scaricabile anche in versione portable (senza installazione).

Concetti di Base

Sviluppare un videogioco significa avere a che fare con il **codice** per l'interazione con l'utente, per la gestione della grafica e gestione delle animazioni, le temporizzazioni etc. Oltre a questo abbiamo da gestire **assets** (o risorse) di vari generi: sprite, background, suoni, file esterni (nei quali memorizzare dati come salvataggi, mappe di scenari ecc.).

Canvas e Context 2d

Grazie all'elemento **canvas** di HTML5, è possibile sfruttare le API dedicate al disegno di forme e immagini, senza la necessità di utilizzare plugin esterni al browser (come Flash), o applet java. Creiamo quindi una semplice pagina HTML (index.html), che faccia da contenitore:

```
<!doctype html>
<html>
<head>
<title>Sviluppare Video Giochi con HTML5</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0, minimum-
scale=1.0, maximum-scale=1.0, user-scalable=no">
</head>
<body>
   <div id="GameDiv" style="margin:auto;">
        <canvas id="GameCanvas" >
        Il tuo browser non supporta l'elemento Canvas Html5
        </canvas>
   </div>
</body>
</html>
```

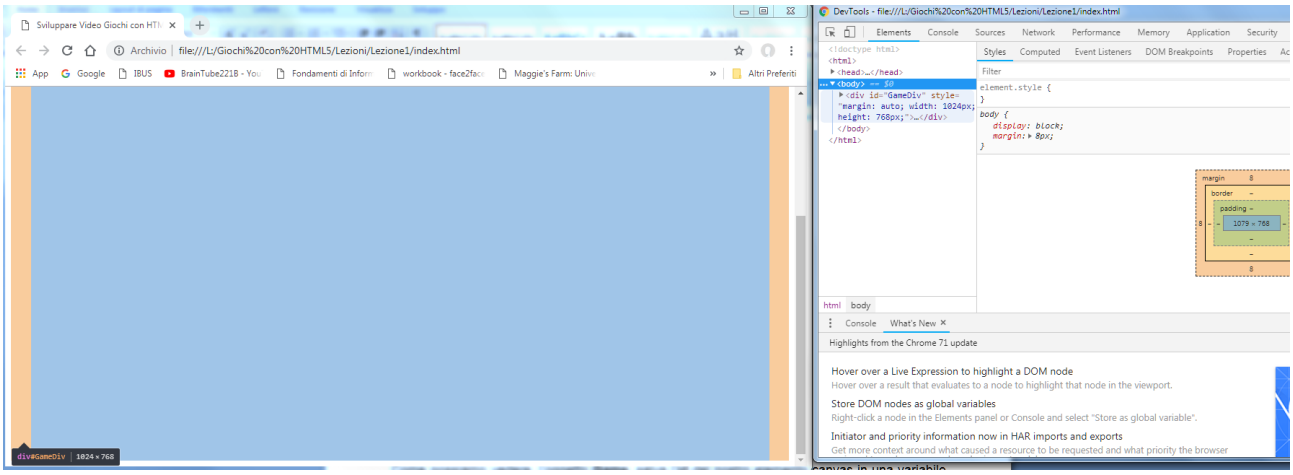
Creiamo quindi un file javascript (main.js), che si occuperà di gestire le diverse parti del framework, e lo agganciamo al nostro contenitore HTML come codice esterno richiamato all'interno del tag <head>:

```
<script language="javascript" src="main.js" ></script>
```

Nel file main.js creiamo finalmente l'oggetto principale del game engine, grazie alla funzione costruttore **Game**:

```
function Game() {
    //il <div> che contiene l'elemento canvas
    this.div = document.getElementById("GameDiv");
    this.div.style.width = "1024px";
    this.div.style.height = "768px"
    //l'elemento <canvas>
    this.canvas = document.getElementById("GameCanvas");
    this.canvas.setAttribute("width", "1024");
    this.canvas.setAttribute("height", "768");
    this.canvas.defaultWidth = this.canvas.width;
    this.canvas.defaultHeight = this.canvas.height;
    //nasconde il cursore
    this.canvas.style.cursor = "pointer";
    //context 2d
    this.ctx = this.canvas.getContext("2d");
    if (!this.ctx) {
        alert("Il tuo browser non supporta HTML5, aggiornalo!");
    }
}
function StartGame() {
    //crea un istanza di Game
    game = new Game();
}
window.addEventListener('load', function () {
    StartGame();
}, true);
```

Come possiamo vedere, l'oggetto **Game**, salva l'**id** del nostro elemento canvas in una variabile pubblica locale "this.canvas": in questo modo potremo accedere all'elemento canvas da qualsiasi oggetto riferendoci a un instanza di **Game** (vedremo in seguito come fare).
Modifichiamone quindi le dimensioni secondo le nostre preferenze e salviamo queste informazioni in due variabili (**defaultWidth** e **defaultHeight**) nell'oggetto/elemento canvas, che utilizzeremo in seguito per implementare il fullscreen.
Per utilizzare le funzioni di rendering, dobbiamo poi ottenere l'oggetto Context 2d dell'elemento canvas.
Nota: *Possiamo controllare se HTML5 è supportato dal browser in uso, verificando la variabile this.ctx ed eventualmente mostrare un messaggio in cui diciamo all'utente di aggiornare il proprio browser, o reindirizzarlo in un'altra pagina.*
Infine creiamo una funzione StartGame eseguita al termine del loading del documento, che si occuperà di creare un istanza della nostra function **Game** (usiamo la prima lettera minuscola per distinguerla dalla funzione/oggetto).
possiamo verificare che tutto funzioni ispezionando con il browser.



Nozioni di programmazione ad oggetti in JavaScript

Un oggetto in javascript può essere creato semplicemente definendo una variabile ad esempio:

var persona = {};

in questo modo avremmo definito un oggetto vuoto che non avrebbe molto senso e che in ogni caso eredita tutte le caratteristiche di object che è il progenitore di tutti gli oggetti.

Proviamo a costruire un oggetto persona che abbia più senso:

```
var person = {
    name: ['Bob', 'Smith'],
    age: 32,
    gender: 'male',
    interests: ['music', 'skiing'],
    bio: function () {
        alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes
' + this.interests[0] + ' and ' + this.interests[1] + '.');
    },
    greeting: function () {
        alert('Hi! I\'m ' + this.name[0] + '.');
    }
};
```

un oggetto è composto da più membri, ognuno dei quali ha un nome (ad esempio name e age sopra) e un valore (es. ['Bob', 'Smith'] e 32). Ogni coppia nome / valore deve essere separata da una virgola e il nome e il valore in ciascun caso sono separati da due punti. La sintassi segue sempre questo schema:

```
var objectName = {
    member1Name: member1Value,
    member2Name: member2Value,
    member3Name: member3Value
};
```

Il valore di un membro oggetto può essere praticamente qualsiasi cosa: nel nostro oggetto persona abbiamo una stringa, un numero, due array e due funzioni. Le prime quattro voci sono elementi di dati e vengono definite **proprietà** dell'oggetto . Gli ultimi due elementi sono funzioni che consentono all'oggetto di fare qualcosa con quei dati e sono indicati come **metodi** dell'oggetto .

Il valore di un membro oggetto può essere praticamente qualsiasi cosa: nel nostro oggetto persona abbiamo una stringa, un numero, due array e due funzioni. Le prime quattro voci sono elementi di dati e vengono definite **proprietà** dell'oggetto . Gli ultimi due elementi sono funzioni che consentono all'oggetto di fare qualcosa con quei dati e sono indicati come **metodi**dell'oggetto .

Un oggetto come questo è indicato come un **oggetto letterale** - abbiamo letteralmente scritto il contenuto dell'oggetto mentre lo creiamo.
si accede alle proprietà e ai metodi dell'oggetto utilizzando la **notazione a punti** . Il nome dell'oggetto (persona) funge da **spazio** dei **nomi** - deve essere inserito per primo per accedere a qualsiasi cosa **racchiusa** nell'oggetto, esempio:

```
person.age
person.interests[1]
person.bio()
```

È persino possibile rendere il valore di un membro dell'oggetto un altro oggetto. Ad esempio la proprietà nome anziché definirla come array potremmo definirla come oggetto come segue:

```
name : {first: 'Bob',last: 'Smith'}
```

Qui stiamo effettivamente creando un **sottospazio dei nomi** . Sembra complesso, ma in realtà non lo è: per accedere a questi elementi devi solo concatenare il passo in più alla fine con un altro punto.

```
person.name.first
person.name.last
```

Programmazione orientata agli oggetti: le basi

Gli oggetti possono contenere dati e codice correlati, che rappresentano le informazioni sulla cosa che si sta tentando di modellare, e la funzionalità o il comportamento che si desidera avere. I dati oggetto (e spesso anche le funzioni) possono essere memorizzati ordinatamente (la parola ufficiale è **incapsulata**) all'interno di un pacchetto di oggetti (a cui può essere assegnato un nome specifico a cui si fa riferimento, a volte chiamato **spazio dei nomi**), che semplifica la struttura e accesso; gli oggetti sono anche comunemente usati come archivi di dati che possono essere facilmente inviati attraverso la rete.
JavaScript utilizza funzioni speciali chiamate funzioni di **costruzione** per definire gli oggetti e le loro caratteristiche. Sono utili perché spesso incontri situazioni in cui non sai quanti oggetti creerai; I costruttori forniscono i mezzi per creare il numero di oggetti di cui hai bisogno in modo efficace, allegando dati e funzioni a loro come richiesto. Ad esempio:

```
function Person(first, last, age, gender, interests) {
    this.name = {
        'first': first,
        'last': last
    };
    this.age = age;
    this.gender = gender;
    this.interests = interests;
    this.bio = function () {
               alert(this.name.first + ' ' + this.name.last + ' is ' + this.age + ' years
       old. He   likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');
    };
    this.greeting = function () {
        alert('Hi! I\'m ' + this.name.first + '.');
    };
}
```

per creare un'istanza dell'oggetto:

```
var person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

la variabile person1 adesso contiene:

```
{name: {first:'Bob', last:'Smith'},
  age: 32,
  gender: 'male',
  interests: ['music', 'skiing'],
  bio: function() {
    alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes ' +
this.interests[0] + ' and ' + this.interests[1] + '.');
  },
  greeting: function() {
    alert('Hi! I\'m ' + this.name[0] + '.');
  }};
```

Altri modi per creare istanze di oggetti

Finora abbiamo visto due diversi modi per creare un'istanza di oggetto: dichiarare un oggetto letterale e utilizzare una funzione di costruzione, ci sono altri modi per costruire un oggetto che è bene conoscere.

usare il costruttore `Object()`per creare un nuovo oggetto:

```
var person1 = new Object();
```

in questo modo abbiamo creato un oggetto vuoto che poi può essere popolato di proprietà e metodi:

```
person1.name = 'Chris';
person1['age'] = 38;
person1.greeting = function () {
    alert('Hi! I\'m ' + this.name + '.');
};
```

È anche possibile passare un oggetto letterale al costruttore `Object()`come parametro:

```
var person1 = new Object({
    name: 'Chris',
    age: 38,
    greeting: function () {
        alert('Hi! I\'m ' + this.name + '.');
    }
});
```

in alternativa si può utilizzare il metodo create per creare un nuovo oggetto partendo da uno già esistende:

```
var person2 = Object.create(person1);
```

person2 ha gli stessi campi e gli stessi metodi di person1

Game Obect ed ereditarietà

Questi concetti saranno fondamentali nello sviluppo dei vari oggetti che andranno a comporre il nostro gioco infatti andremo a creare dei **GameObject**, cioè oggetti generici (function), che possono contenere tutte le informazioni e risorse necessarie agli elementi di gioco. A partire da questo andremo a specializzare : un personaggio, un nemico, una piattaforma, un’interfaccia grafica e ogni elemento del nostro videogame.

Prototipi di oggetti

Un lato negativo di questo approccio è l'efficienza: per ogni oggetto *person* creato, una nuova funzione `greeting` viene creata e collegata ad esso – creare 1000 oggetti significherebbe che l'interprete JavaScript deve allocare spazio in memoria per 1000 funzioni, sebbene esse abbiano lo stesso comportamento. Questo comporterà uno spreco non necessario di memoria nell'applicazione

Secondo, questo approccio ci priva di alcune interessanti possibilità. Questi oggetti *person* non condividono niente – sono state realizzate dalla stessa funzione, ma sono completamente indipendenti l'uno dall'altro.

Inoltre se volessimo cambiare o aggiungere una proprietà a questi oggetti dovremmo modificarli uno per uno.

Usare un costruttore per creare oggetti

In JavaScript, le entità che creano gli oggetti con comportamento condiviso sono funzioni chiamate in modo speciale. Queste funzioni speciali sono i *costruttori*. Facciamo un semplice esempio: creiamo una funzione che tira fuori oggetti indipendenti di tipo auto, usando un costruttore:

```
function Macchina() {
    this.suona = function () {
        console.log("tu tu");
    }
}
```

Quando questa funzione viene chiamata usando la parola chiave *new*, come in questo esempio:

```
var miaAuto = new Macchina();
```

restituisce implicitamente un nuovo oggetto con la funzione *suona* collegata.

Usando le parole chiavi *this* e *new*, la creazione esplicita e la restituzione del nuovo oggetto non sono più necessari – esso è creato e restituito "dietro le quinte" (la parola chiave *new* è quella che crea il nuovo, "invisibile" oggetto, e segretamente lo passa alla funzione *Car* come variabile *this*). Potete pensare a questo meccanismo come se fosse questo pseudo-codice:

```
//pseudocodice d'esempio
function Macchina(this) {
    this.suona = function () {
        console.log("tu tu");
    }
    return this
}
var nuovoOggetto={}
var miaAuto = new Macchina(nuovoOggetto);
```

questo è più o meno come la soluzione precedente – non dobbiamo creare ogni singolo oggetto auto manualmente, ma ancora non possiamo modificare il comportamento del metodo *honk* e avere la modifica automaticamente in tutte le auto create.

Ma abbiamo raggiunto il primo obiettivo. Usando un costruttore, tutti gli oggetti ricevono una proprietà speciale che li collega al proprio costruttore:

```
function Macchina() {
    this.suona = function () {
        console.log("tu tu");
    }
}

var miaAuto1 = new Macchina();
var miaAuto2 = new Macchina();

console.log(miaAuto1.constructor); //outputs [Function:Maccina]
console.log(miaAuto2.constructor); //outputs [Function:Maccina]
```

Tutte le *miaAutos* create sono collegate al costruttore *Macchina*. Questo è ciò che le rende una classe di oggetti collegati, e non solo un mucchio di oggetti che hanno nome simile e le stesse funzioni.

Usare il prototyping per condividere efficientemente comportamenti tra oggetti

mentre nella programmazione class-based la classe è il posto dove inserire le funzioni che gli oggetti condivideranno, nella programmazione prototype-based, il posto dove inserire queste funzioni è l'oggetto che fungerà da prototipo per gli altri oggetti.
Ma dove è l'oggetto che fa da prototipo per il nostro oggetto *miaAuto* – noi non l'abbiamo mai creato!
E' stato creato implicitamente per noi, e assegnato alla proprietà:
`miaAuto.prototype`

Ecco la chiave per condividere funzioni tra gli oggetti: ogni volta che invochiamo una funzione su un oggetto, l'interprete JavaScript cerca di trovare quella funzione nell'oggetto chiamato. Ma se non trova la funzione nell'oggetto stesso, chiede all'oggetto il puntatore al suo prototype, quindi va nel prototype, e chiede la funzione. Se la trova, la esegue.
Questo significa che possiamo creare oggetti *miaAuto* senza nessuna funzione, creare la funzione *suona* nel loro prototype, e finire con tutti gli oggetti *miaAuto* che sanno come suonare – perché ogni volta che l'interprete cerca di eseguire la funzione *suona* su un oggetto *miaAuto*, sarà ridirezionato al prototype, ed eseguirà la funzione *suona* li definita.

Ecco come impostare il tutto:

```
function Macchina() {}
    Macchina.prototype.suona = function () {
        console.log("tu tu");
    }
}

var miaAuto1 = new Macchina();
```

```
var miaAuto2 = new Macchina();

miaAuto1.suona(); //esegue Macchina.prototype.suona() e stampa tu tu
miaAuto2.suona(); //esegue Macchina.prototype.suona() e stampa tu tu
```

Il nostro costruttore è ora vuoto, perché per le nostre semplici auto, non è necessario altro.

Siccome entrambe le *miaAutos* sono state create tramite il costruttore, il loro prototype punta a *Macchina.prototype* – eseguire *miaAuto1.suona()* significherà eseguire *Macchina.prototype.suona()*.

Vediamo questo cosa ci permette di fare. In JavaScript, gli oggetti possono essere cambiati a runtime. Questo è possibile anche per i prototype. Che è il motivo per cui possiamo cambiare il comportamento di *suona* anche dopo che le nostro auto sono state create:

```
function Macchina() {}
    Macchina.prototype.suona = function () {
        console.log("tu tu");
    }
}

var miaAuto1 = new Macchina();
var miaAuto2 = new Macchina();

miaAuto1.suona(); //esegue Macchina.prototype.suona() e stampa tu tu
miaAuto2.suona(); //esegue Macchina.prototype.suona() e stampa tu tu

 Macchina.prototype.suona = function () {
        console.log("beep beep");
    }
miaAuto1.suona(); //esegue Macchina.prototype.suona() e stampa beep beep
miaAuto2.suona(); //esegue Macchina.prototype.suona() e stampa beep beep
```

Ovviamente, possiamo aggiungere altre funzioni a runtime:

```
function Macchina() {}
    Macchina.prototype.suona = function () {
        console.log("tu tu");
    }
}

var miaAuto1 = new Macchina();
var miaAuto2 = new Macchina();

 Macchina.prototype.guida = function () {
        console.log("vroom");
    }
miaAuto1.guida(); //esegue Macchina.prototype.suona() e stampa vroom
miaAuto2.guida(); //esegue Macchina.prototype.suona() e stampa vroom
```

Ma potremmo addirittura decidere di trattare una sola delle nostre auto in maniera differente:

```
function Macchina() {}
    Macchina.prototype.suona = function () {
        console.log("tu tu");
    }
}
```

```
var miaAuto1 = new Macchina();
var miaAuto2 = new Macchina();

miaAuto1.suona(); //esegue Macchina.prototype.suona() e stampa tu tu
miaAuto2.suona(); //esegue Macchina.prototype.suona() e stampa tu tu

 miaAuto2.guida = function () {
        console.log("beep beep");
    }
miaAuto1.guida(); //esegue Macchina.prototype.suona() e stampa tu tu
miaAuto2.guida(); //esegue Macchina.prototype.suona() e stampa beep beep
```

E' importante capire cosa accade dietro le quinte di questo esempio. Come abbiamo visto, quando viene chiamata una funzione su un oggetto, l'interprete segue un certo percorso per trovare la vera posizione di quella funzione.

Mentre per *miaAuto1*, non c'è ancora una funzione *suona* nell'oggetto stesso, questo non è più vero per *miaAuto2*. Quando l'interprete chiama *miaAuto2.suona();*, c'è una funzione nell'oggetto *miaAuto2* stesso. Pertanto, l'interprete non seguirà più il percorso attraverso il prototype di *miaAuto2*, ma eseguirà invece la funzione dell'oggetto *miaAuto2 (polimorfismo)*.

Questa è una delle differenze principali rispetto alla programmazione class-based: mentre gli oggetti sono relativamente "rigidi" ad es. in Java, dove la struttura di un oggetto non può essere cambiata a runtime, in JavaScript, l'approccio prototype-based collega gli oggetti di una certa classe più genericamente tra di loro, il che permette di cambiare la struttura degli oggetti in qualsiasi momento.

Inoltre, notate come condividere funzioni tramite il prototype del costruttore sia molto più efficiente del creare oggetti che hanno le proprie funzioni, anche se identiche. Come detto in precedenza, l'engine non sa che queste funzioni sono identiche, e deve allocare memoria per ogni funzione di ogni oggetto. Questo non è più vero quando condividiamo funzioni tramite un prototype comune – la funzione in questione viene messa in memoria solo una volta, e non conta quanti oggetti *miaAuto* saranno creati, essi non hanno le funzioni al loro interno, fanno semplicemente riferimento al proprio costruttore, nel quale prototype viene trovata la funzione.

Per darvi un'idea di cosa comporta questa differenza, ecco un semplice confronto. Il primo esempio crea 1.000.000 di oggetti che hanno la funzione al loro interno:

```
var C= function(){
 this.f=function(cibo){
 console.log(cibo);
    }
}
var a=[];
for( var i=0;i<1000000;i++){
    a.push(new C();
}
```

In Google Chrome, questo comporta un heap snapshot di 328MB. Ecco lo stesso esempio, ma ora la funzione è condivisa attraverso il prototype del costruttore:

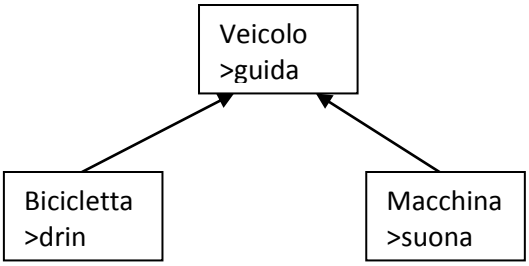
```
var C= function(){
 C.prototype.f=function(cibo){
 console.log(cibo);
```

```
    }
}
var a=[];
for( var i=0;i<1000000;i++){
    a.push(new C();
}
```

Questa volta, la dimensione dello heap snapshot è di soli 17MB, circa il 5% della soluzione non efficiente.

Object-orientation, prototyping ed ereditarietà

Vediamo ora come funziona l'ereditarietà, supponiamo di voler realizzare una gerarchia come la seguente:



Strutturare questa relazione in un linguaggio class-based come Java è lineare: dovremmo definire una classe *Veicolo* con un metodo *guida*, e due classi *Macchina* e *Biciclettae* che estendono entrambe la classe *Veicolo*, e implementano rispettivamente i metodi *suona* e d*rin*.

Questo farà si che sia gli oggetti macchina che bicicletta ereditino il comportamento drive dall'ereditarietà delle loro classi.

Come funziona questo in JavaScript, dove non abbiamo classi ma prototype?

Vediamo prima un esempio e poi analizziamolo. Per mantenere il codice breve, partiamo solo con un'auto che eredità da un veicolo:

```
function Veicolo() {}
    Veicolo.prototype.guida = function () {
        console.log("vrooon");
    }
}

function Macchina(){}
Macchina.prototype= new Veicolo()
Macchina.prototype.suona=function () {
        console.log("tu tu");
    }

var miaAuto = new Macchina();
miaAuto.guida(); //stampa vrooon
miaAuto.suona(); //stampa tu tu
```

In JavaScript, l'ereditarietà funziona tramite una catena di prototype.

Il prototype del costruttore *Macchina* è impostato a un nuovo oggetto *Veicolo*, che stabilisce il collegamento strutturale che consente all'interprete di cercare metodi negli oggetti genitore.

Il prototype del costruttore *Veicolo* ha la funzione *guida*. Ecco cosa accade quando all'oggetto *miaAuto* chiama il metodo *guida()*:

- l'interprete controlla se nell'oggetto *miaAuto* c'è un metodo guida, e non lo trova;
- l'interprete quindi chiede all'oggetto *miaAuto* il suo prototype, che è il prototype del suo costruttore *Macchina*;
- quando controlla *Macchina.prototype*, l'interprete vede un oggetto *Veicolo* che ha la funzione *suona* ma non la funzione *guida*;
- allora, l'interprete chiede all'oggetto *veicolo* il suo prototype, che è il prototype del suo costruttore *Veicolo*;
- mentre controlla *Veicolo.prototype*, l'interprete vede un oggetto che ha una funzione *guida* – l'interprete ora sa quale codice implementa il comportamento *miaAuto.guida()* e lo esegue.

Una società senza classi, rivisitata

Abbiamo appena imparato come emulare l'ereditarietà tradizionale (o classica) in JavaScript. Questo concetto era fondamentale per essere poi dimenticato e lasciato alle spalle, per accettare l'idea che in realtà in JavaScript non c'è un vero bisogno delle classi, e non c'è nemmeno bisogno di emularle – inoltre, ci vuole davvero parecchio codice per esprimere l'idea di "*vai a guardare in quell'oggetto, lui ha il codice di cui ha bisogno*", no?

E' stato Douglas Crockford a pensare ad una soluzione intelligente, che permette agli oggetti di ereditare direttamente l'uno dall'altro, senza il bisogno dell'inutile codice presentato nell'esempio precedente. La soluzione è un componente nativo di JavaScript : la funzione *Object.create()*, che funziona così:

```
Object.create = function (O) {
    function F() { }
    F.prototype = O;
    return new F();
};
```

Ora sappiamo abbastanza da capire cosa sta succedendo. Analizziamo un esempio:

```
var veicolo={};
veicolo.guida=function () {
        console.log("vrooor");
    }

var miaAuto = Object.create(veicolo) {
    miaAuto.suona=function () {
        console.log("tu tu");
    }
};
miaAuto.guida(); //stampa vrooon
miaAuto.suona(); //stampa tu tu
```

Anche se molto più conciso ed espressivo, questo codice esegue esattamente lo stesso comportamento senza la necessità di scrivere costruttori dedicati e collegare funzioni ai loro prototype. Come potete vedere, *Object.create()* si occupa di tutto, al volo, dietro le quinte. Viene creato un costruttore al volo, nel suo prototype viene settato l'oggetto che farà da modello per il nostro nuovo oggetto, e da questo setup viene infine creato un nuovo oggetto.

Concettualmente, questo è esattamente lo stesso caso dell'esempio precedente dove abbiamo definito che *Car.prototype* era un *new Veicolo().*

Ma aspetta! Abbiamo creato le funzioni *guida* e *suona* nei nostri oggetti, e non nei loro prototype, questo è poco efficiente!

Beh, in questo caso, non lo è. Vediamo perché:

```
var veicolo={};
veicolo.guida=function () {
        console.log("vrooor");
    }

var miaAuto = Object.create(veicolo) {
    miaAuto.suona=function () {
        console.log("tu tu");
    }
};


var mioVeicolo = Object.create(veicolo);
var miaAuto1 = Object.create(miaAuto);
var miaAuto1 = Object.create(miaAuto);

miaAuto1.suona() // produce tu tu
miaAuto2.suona() // produce tu tu

mioVeicolo.guida(); //  produce vroor
miaAuto1.guida(); //  produce vroor
miaAuto2.guida(); //  produce vroor
```

Abbiamo ora creato un totale di 5 oggetti, ma quante volte i metodi *guida* e suona esistono in memoria? Beh, quante volte sono stati definiti? Solo una, e quindi, questa soluzione è tanto efficiente quanto quella nella quale abbiamo definito l'ereditarietà a mano. Ma guardiamo i numeri:

```
var C= function(){
 C.f=function(cibo){
 console.log(cibo);
    }
}
var a=[];
for( var i=0;i<1000000;i++){
    a.push(Object.create(C));
}
```

E' venuto fuori che non è esattamente identico è risultato un head snapshot di 40MB, quindi c'è un po' di overhead. In ogni caso, in cambio di codice migliore, ne vale la pena.