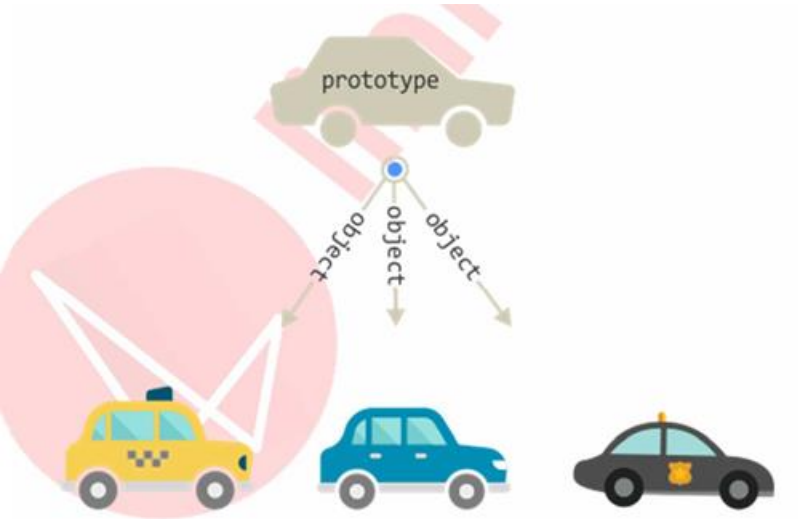
Prototipovi

Osnovni postulati objektno orijentisanog programiranja su apstrakcija, enkapsulacija, nasleđivanje i polimorfizam. Apstrakcija omogućava da se složeni entiteti iz realnog sveta modeluju sa nivoom detalja koji odgovara potrebama programske logike, dok se enkapsulacija odnosi na postupak objedinjavanja informacija i funkcionalnosti unutar jedne celine – objekta. Oba upravo definisana postulata u prethodnoj lekciji su praktično demonstrirana na primerima realnih JavaScript objekata. Ipak, postizanje nasleđivanja i polimorfizma još uvek nije demonstrirano na realnim primerima. Za njihovu realizaciju u JavaScriptu neophodno je prethodno razumeti jedan veoma značajan pojam jezika. Reč je o prototipovima.

Prototipovi su svakako najkompleksniji pojam u kompletnoj priči o objektima JavaScript jezika. Oni mogu biti veoma teški za razumevanje, pogotovu ukoliko već poznajete neki od objektno orijentisanih jezika koji ne poseduje prototipove. Stoga će lekcija koja je pred vama u potpunosti biti posvećena pojmu prototipova.

Šta su prototipovi?

Prototipovi su osnovni pojam koji u jeziku JavaScript obezbeđuje mogućnost nasleđivanja. Nasleđivanje omogućava da jedan objekat ili više njih određene osobine naslede od nekog drugog objekta (slika ispod).



Slika iznad ilustruje osnovnu ulogu prototipova u JavaScriptu. Na vrhu slike može se videti jedan prototip, odnosno objekat čija svojstva i metode nasleđuju tri različita objekta koja se nalaze ispod njega. Tako se prototip može doživeti kao određeni šablon, čije osobine mogu da naslede neki drugi objekti.

Zbog postojanja prototipova, JavaScript se veoma često naziva **prototipski baziran jezik**.

Prototipovi na delu

Sa različitim osobinama prototipova upoznavaćemo se postepeno u nastavku ove lekcije. Za početak će biti prikazano da su prototipovi prisutni čak i unutar objekata koje smo mi kreiraliu prethodnoj lekciji.

U prethodnoj lekciji, kao najefikasniji način za kreiranje objekata u JavaScriptu, prikazan je pristup koji podrazumeva prethodno definisanje konstruktorske funkcije:

```
function Car(make, model, weight, color){

      this.make = make;
       this.model = model;
       this.weight = weight;
       this.color = color;
        }
```

Prikazana funkcija je zapravo funkcija koja će nam omogućiti kreiranje većeg broja objekata različitih automobila, sa identičnim skupom svojstava (`make`, `model`, `weight` i `color`). Prikazana funkcija se ni po čemu ne razlikuje od bilo koje druge JavaScript funkcije. Ipak, kako bi se ona upotrebila kao konstruktorska funkcija i time dobio jedan objekat, dovoljno je ispred njenog poziva postaviti ključnu reč `new`:

```
var car1 = new Car("Subary", "Legacy", 1563, "black");
```

Korišćenjem ključne reči `new`, koja je navedena ispred poziva funkcije, JavaScript izvršnom okruženju je rečeno da funkciju `Car()` želimo da upotrebimo za kreiranje novog objekta. Stoga je njena povratna vrednost objekat koji poseduje svojstva definisana u telu funkcije. Kreirani objekat je smešten unutar promenljive `car1`.

Bitno je primetiti da konstruktorska funkcija poseduje nekoliko parametara čije vrednosti postavlja za vrednosti svojstava kreiranog objekta. Stoga je moguće napisati nešto ovako i dobiti vrednost jednog svojstva kreiranog objekta:

```
console.log(car1.make);
```

Ovakav kod unutar konzole ispisuje vrednost `make` svojstva, što je u primeru vrednost *Subary*.

Pored nekoliko svojstava, konstruktorska funkcija `Car()` ne poseduje nijednu metodu:

```
console.log(car1.update());
```

Sada je nad promenljivom koja čuva referencu na objekat automobila pozvana metoda `update()`. S obzirom na to da objekat `car1` ne poseduje takvu metodu, dolazi do emitovanja izuzetka:

```
Uncaught TypeError: car1.update is not a function
```

Ipak, pokušajmo da nad objektom `car1` pozovemo još neku funkciju koju nismo samostalno definisali:

```
console.log(car1.toString());
```

Ovoga puta, prilikom pozivanja funkcije `toString()` ne dolazi do pojave izuzetka. Štaviše, unutar konzole dobija se i povratna vrednost ove funkcije:

```
[object Object]
```

Sve ovo upućuje na to da objekat `car1` zaista poseduje metodu `toString()` iako mi nju nigde nismo samostalno definisali. *Krivac* za ovakvo ponašanje upravo je prototip. Naime, prototip omogućava našem objektu da koristi metode koje su definisane unutar prototipa `Object` JavaScript objekta.

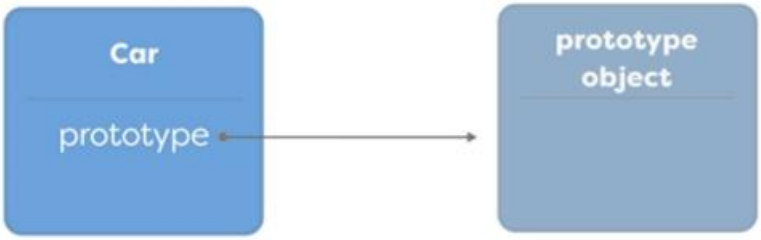
Lanac prototipova

Kako biste mogli da razumete na koji način naši objekti mogu da koriste svojstva i metode koji nisu direktno definisani unutar njih, neophodno je razumeti jedan veoma važan pojam JavaScript jezika – lanac prototipova (*engl. prototype chain*). Lanac prototipova u nastavku će biti objašnjen postepeno, a pritom ćemo se upoznati i sa brojnim propratnim pojmovima u vezi sa prototipovima i JavaScript nasleđivanjem.

Svaki objekat u JavaScriptu može naslediti osobine nekog drugog objekta, ali isto tako i njegove osobine mogu naslediti drugi objekti. Na taj način kreira se hijerarhija u kojoj svaki objekat nasleđuje osobine objekata koji se u takvoj hijerarhiji nalaze iznad. Na upravo prikazanom primeru takva hijerarhija izgleda kao na slici ispod.

Slika iznad ilustruje razlog zbog koga smo mi nešto ranije bili u mogućnosti da nad objektom `car1` pozovemo metode koje nisu bile definisane direktno unutar njega. Ipak, bitno je razumeti da slika uprošćeno prikazuje način na koji se u jeziku JavaScript obavlja nasleđivanje osobina između objekata. Naime, već je rečeno da se u JavaScriptu nasleđivanje postiže upotrebom prototipova. Drugim rečima, objekti ne nasleđuju direktno objekte, kao što to slika uprošćeno prikazuje, već se u procesu nasleđivanja isključivo koriste specijalni objekti – prototipovi.

Prilikom kreiranja funkcija u JavaScript jeziku, izvršno okruženje ovog jezika funkcijama automatski dodaje svojstvo **prototype**. Ovo svojstvo čuva referencu na jedan poseban objekat – prototipski objekat takve funkcije. Ovo praktično znači da se prilikom kreiranja funkcije `Car()` iz prethodnog primera automatski obavlja i kreiranje objekta prototipa, koji se za funkciju vezuje korišćenjem svojstva `prototype` (slika ispod).



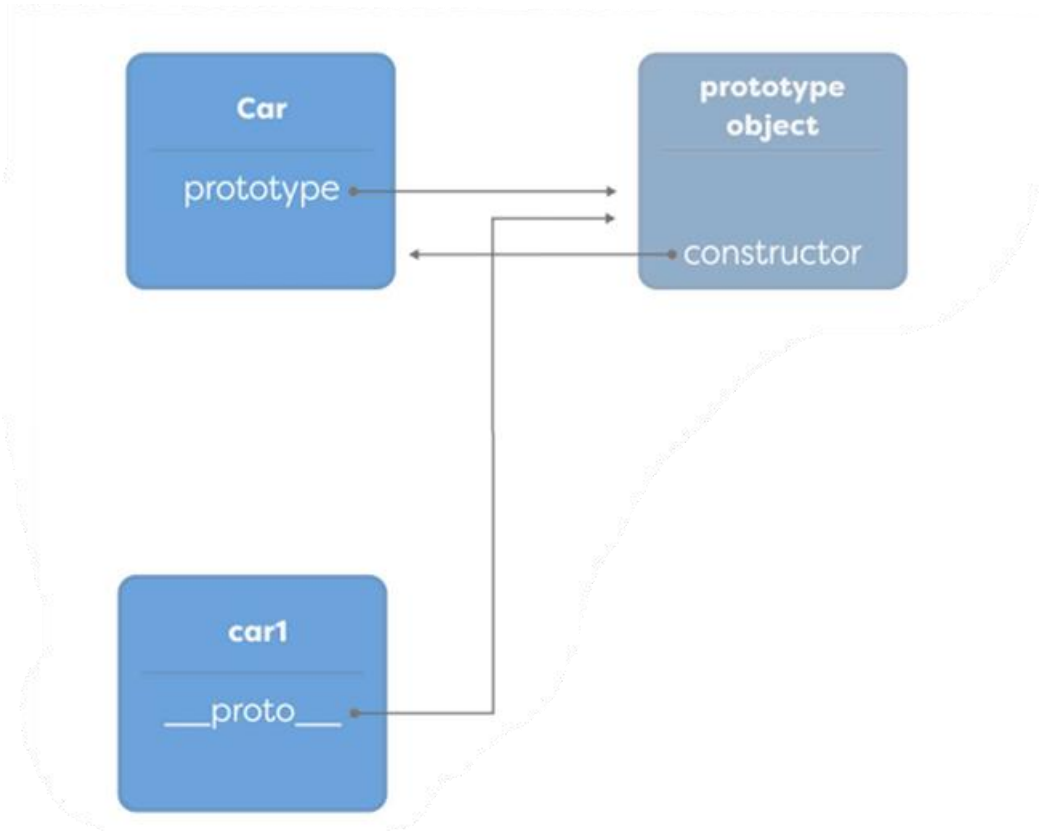
Funkcija Car i pripadajuci prototip

Na slici iznad može se videti da funkcija `Car()` poseduje svojstvo `prototype` koje ukazuje na prototipski objekat.

Napomena: U JavaScriptu funkcije su objekti. Zbog toga se može reći funkcija Car() ili objekat Car, u zavisnosti od konteksta u kome se posmatra.

Svaki prototipski objekat poseduje svojstvo **constructor** koje upućuje na originalnu konstruktorsku funkciju, slika ispod.

Pojam nasleđivanja u JavaScriptu realizuje se isključivo upotrebom prototipskih objekata. To praktično znači da objekti koji se kreiraju korišćenjem konstruktorske funkcije `Car()` nasleđuju sve ono što je definisano unutar prototipskog objekta funkcije `Car()` (slika ispod).



car1 objekat nasleđuje sve ono što je definisano unutar prototipskog objekta Car() konstruktorske funkcije.

Kreiranje Car() konstruktorske funkcije

Započećemo kreiranjem konstruktorske funkcije `Car()`:

```
function Car(make, model, weight,
    color) {this.make = make;
    this.model =
    model; this.weight
    = weight;
    this.color =
    color;

    this.getName = function() {
        return this.make + " " + this.model;
    }

    this.getInfo = function() {
        return "Make: " +  this.make + "\n"
        +"Model: " + this.model + "\n" +
        "Weight: " + this.weight + "\n" +
        "Color: " + this.color
    }
}
```

Unutar konstruktorske funkcije definisano je nekoliko svojstava i dve metode. Reč je o metodama za ispis imena vozila (`getName()`) i ispis kompletnih informacija o vozilu (`getInfo()`). Ovakva konstruktorska funkcija može se iskoristiti za kreiranje objekata automobila na sledeći način:

```
let car1 = new Car("Subary", "Legacy", 1563, "black");

let car1Name = car1.getName();
let car1Info = car1.getInfo();

console.log(car1Name);
console.log(car1Info);
```

Prikazanim naredbama obavljeno je kreiranje jednog objekta automobila. Zatim je obavljeno pozivanje metoda `getName()` i `getInfo()` i štampanje povratnih vrednosti unutar konzole:

```
Subary Legacy

Make: Subary
Model: Legacy
Weight: 1563
Color: black
```

Kodu za realizaciju prikazanog primera ne bi imalo šta da se zameri dok se u priču ne uključe i još neki objekti, koji predstavljaju specifične tipove automobila. Na primer za pikap automobil ili kabriolet. Mi želimo da takvi objekti osobine naslede od objekta `Car`.

Premeštanje metoda u objekat prototipa

Prvi korak za postizanje opisanog nasleđivanja biće modifikacija već kreirane konstruktorske funkcije `Car()` i njenog prototipa:

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

     Car.prototype.getName = function () {
     return this.make + " " + this.model;
     }

     Car.prototype.getInfo = function () {
         return "Make: " + this.make + "\n"
         +
              "Model: " + this.model + "\n" +
              "Weight: " + this.weight + "\n" +
              "Color: " + this.color
     }
```

Sada su dve funkcije koje su se nalazile unutar konstruktorske funkcije, odnosno unutar objekta `Car`, izmeštene unutar prototipskog objekta – `Car.prototype`. Kreiranje objekata automobila funkcionisaće identično kao i nešto ranije, što možete da proverite ukoliko pokušate da kreirate objekat automobila i pozovete metode `getName()` i `getInfo()`. Jednostavno, ove metode više nisu unutar objekta `Car`, već unutar objekta `Car.prototype`. Kao što ste nešto ranije mogli da pročitate u poglavlju o lancu prototipova, JavaScript će ovemetode prvo pokušati da pronađe unutar samog objekta, pa zatim unutar konstruktorske funkcije i, na kraju, unutar prototipskih objekata. Stoga, iz ugla funkcionisanja dosadašnjeg primera, ništa se zapravo neće promeniti.

Premeštanje metoda `getName()` i `getInfo()` unutar `Car.prototype` objekta omogućiće nam da takve metode naslede i objekti koje ćemo uskoro kreirati. Prvo ćemo se pozabaviti kreiranjem objekata kabrioleta.

Kreiranje konstruktorske funkcije Convertible()

Kreiranje konstruktorske funkcije `Convertible()` započećemo na sledeći način:

```
function Convertible(make, model, weight, color, roofType)
{

}
```

Konstruktorska funkcija `Convertible()` zasad poseduje samo potpis. Iz potpisa se može videti da ona poseduje identične parametre kao i funkcija `Car()` uz jedan novi parametar koji se odnosi na tip krova automobila. S obzirom na to da želimo da `Convertible` objekat nasledi objekat `Car`, telo konstruktorske funkcije biće definisano na sledeći način:

```
function Convertible(make, model, weight, color, roofType) {
        Car.call(this, make, model, weight, color);

           this.roofType = roofType;
           }
```

Umesto pojedinačnog dodeljivanja vrednosti parametara svojstvima objekta `Convertible`, sada je iskorišćen nešto drugačiji pristup, koji podrazumeva upotrebu jedne specijalne JavaScript funkcije `call()`.

JavaScript call() funkcija

JavaScript funkcija `call()` omogućava da se metoda koja pripada nekom objektu pozove unutar nekog drugog objekta, kao da je sastavni deo takvog objekta. Ova metoda kao prvi argument prihvata objekat nad kojim će funkcija biti pozvana, dok se ostali argumenti odnose na parametre koji se takvoj funkciji prosleđuju.

Metoda `call()` najčešće se koristi za povezivanje konstruktorskih funkcija prilikom nasleđivanja, baš kao u prethodnom primeru:

```
Car.call(this, make, model, weight, color);
```

Na ovaj način, konstruktorska funkcija `Car()` pozvana je unutar funkcije `Convertible()`, baš kao da je reč o metodi koja se nalazi unutar samog objekta. Njoj je prosleđen tekući objekat, korišćenjem ključne reči `this`, a zatim i svi parametri koje metoda `Car()` prihvata. Rezultat će biti postavljanje vrednosti svojstava objekta `Convertible`, bez potrebe za dupliranjem logike koja je već definisana unutar funkcije `Car()`.

Na kraju, može se rezimirati: svojstva koja objekti `Car` i `Convertible` dele inicijalizuju se unutar `Car()` funkcije. Svojstvo koje je karakteristično za `Convertible` objekat inicijalizuje se unutar konstruktorske funkcije `Convertible()`.