

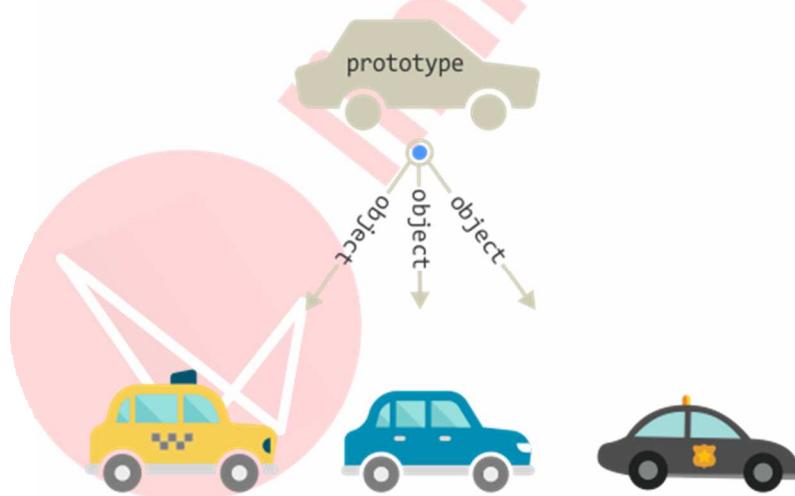
Prototipovi

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 prići o objektima JavaScript jezika. Oni mogu biti veoma teški za razumevanje, pogotovo 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 2.1).



Slika 2.1. Prototip

Slika 2.1. 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 kreirali u 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:

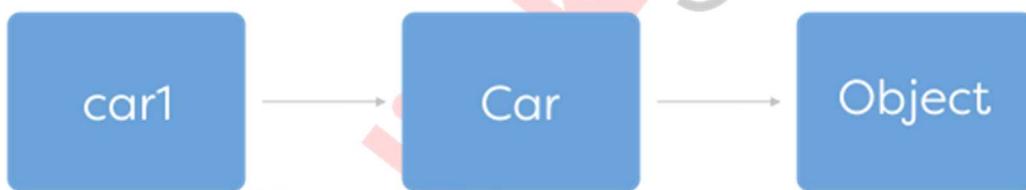
```
[object Object]
```

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.

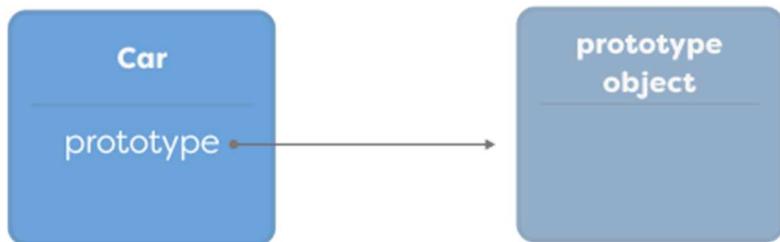
Svaki objekat u JavaScriptu može naslediti osobine nekog drugog objekta, ali isto tako i njegove osobine mogu naslediti drugi objekti. Na taj način kreira se hijerarhija u kojoj svaki objekat nasleđuje osobine objekata koji se u takvoj hijerarhiji nalaze iznad. Na上pravо prikazanom primeru takva hijerarhija izgleda kao na slici 2.2.



Slika 2.2. Primer hijerarhije nasleđivanja u JavaScriptu

Slika 2.2. 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 2.2. 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 2.3).



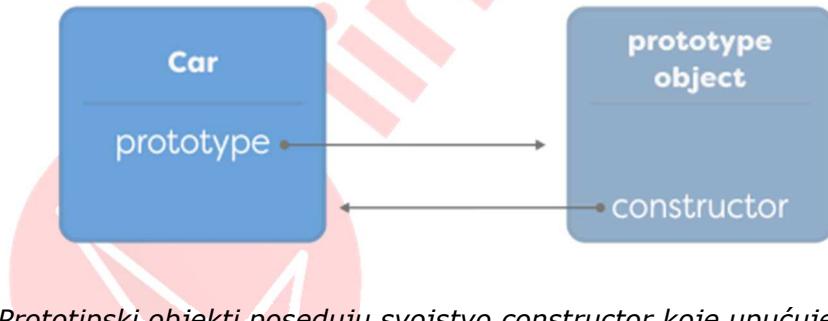
Slika 2.3. Funkcija Car i pripadajući prototip

Na slici 2.3. može se videti da funkcija `Car()` poseduje svojstvo `prototype` koje ukazuje na prototipski objekat.

Ne zaboravite da su funkcije objekti

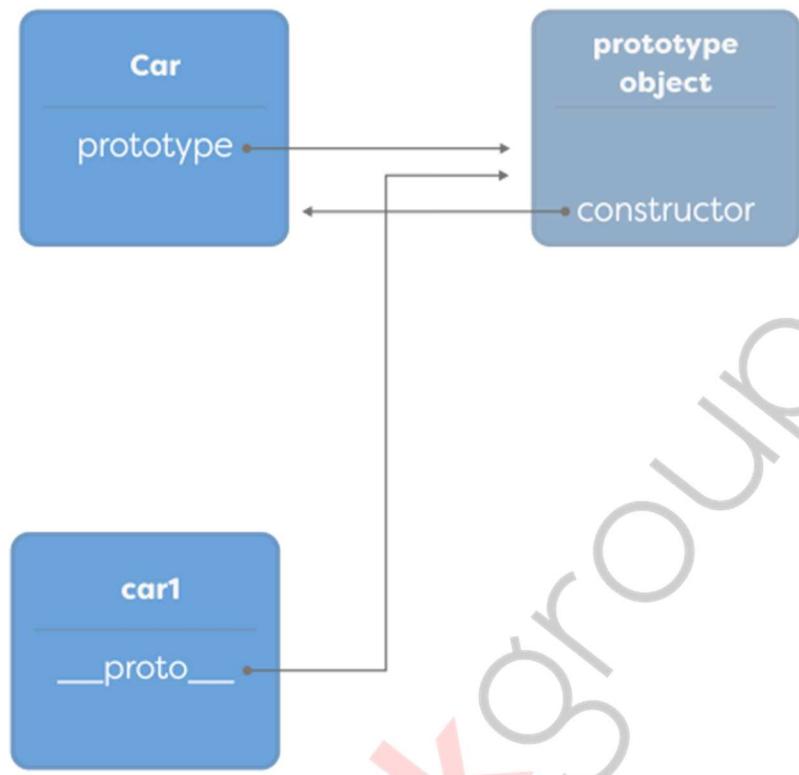
Ukoliko se pitate zbog čega se funkcija na slici 2.2. prikazuje kao da je reč o objektu, odgovor je vrlo jednostavan. U JavaScriptu funkcije su objekti. Zbog toga se može reći funkcija `Car()` ili objekat `Car`, u zavisnosti od konteksta u kome se posmatra.

Svaki prototipski objekat poseduje svojstvo **constructor** koje upućuje na originalnu konstruktorsku funkciju (slika 2.4).



Slika 2.4. Prototipski objekti poseduju svojstvo constructor koje upućuje na originalnu konstruktorsku funkciju

Pojam nasleđivanja u JavaScriptu realizuje se isključivo upotrebom prototipskih objekata. Jedan takav objekat možete videti na desnoj polovini slika 2.3. i 2.4. To praktično znači da objekti koji se kreiraju korišćenjem konstruktorske funkcije `Car()` nasleđuju sve ono što je definisano unutar prototipskog objekta funkcije `Car()` (slika 2.5).



Slika 2.5. car1 objekat nasleđuje sve ono što je definisano unutar prototipskog objekta Car() konstruktorske funkcije

U dijagramu na slici 2.5. sada je uvršćen i car1 objekat, odnosno objekat koji se kreira korišćenjem konstruktorske funkcije Car(). Svaki objekat u JavaScriptu čuva referencu na prototipski objekat od koga nasleđuje osobine. U primeru je takvo svojstvo imenovano sa `__proto__` i može se videti da ono upućuje na prototipski objekat konstruktorske funkcije Car().

Načini za pristup prototipu nekog JavaScript objekta

U JavaScript jeziku dugo vremena, zapravo sve do ECMAScript 2015 verzije, nije postojao zvanični način za pristup prototipu nekog objekta. Zbog toga su proizvođači web pregledača samostalno kreirali svojstvo `__proto__`. Obratite pažnju na to da se na početku i na kraju naziva ovoga svojstva nalaze po dva karaktera donja crta (*engl. underscore*). Vremenom su svi web pregledači, osim starijih verzija Internet Explorera, uvrstili u svoja izvršna okruženja podršku za ovakvo svojstvo.

Pojavom *ECMAScript 2015* specifikacije predstavljen je i zvaničan način za dolazak do reference na prototip nekog objekta. Reč je o pristupu koji podrazumeva korišćenje metode `getPrototypeOf()` objekta `Object`:

```
Object.getPrototypeOf(obj)
```

Kako bismo dokazali da `__proto__` svojstvo `car1` objekta upućuje na prototip `Car()` konstruktorske funkcije, možemo napisati nešto ovako:

```
console.log(car1.__proto__ === Car.prototype);
```

Prikazanom naredbom porede se reference i tipovi na koje upućuju svojstvo `__proto__` objekta `car1` i svojstvo `prototype` objekta `Car`. Unutar konzole se dobija vrednost:

```
true
```

Identično je moglo biti postignuto i upotrebatim metode `getPrototypeOf()` objekta `Object`, kako bi se došlo do prototipa nekog objekta:

```
console.log(Object.getPrototypeOf(car1) === Car.prototype);
```

Sada je umesto svojstva `__proto__` upotrebljena metoda `Object.getPrototypeOf()` kako bi se došlo do prototipa nekog objekta.

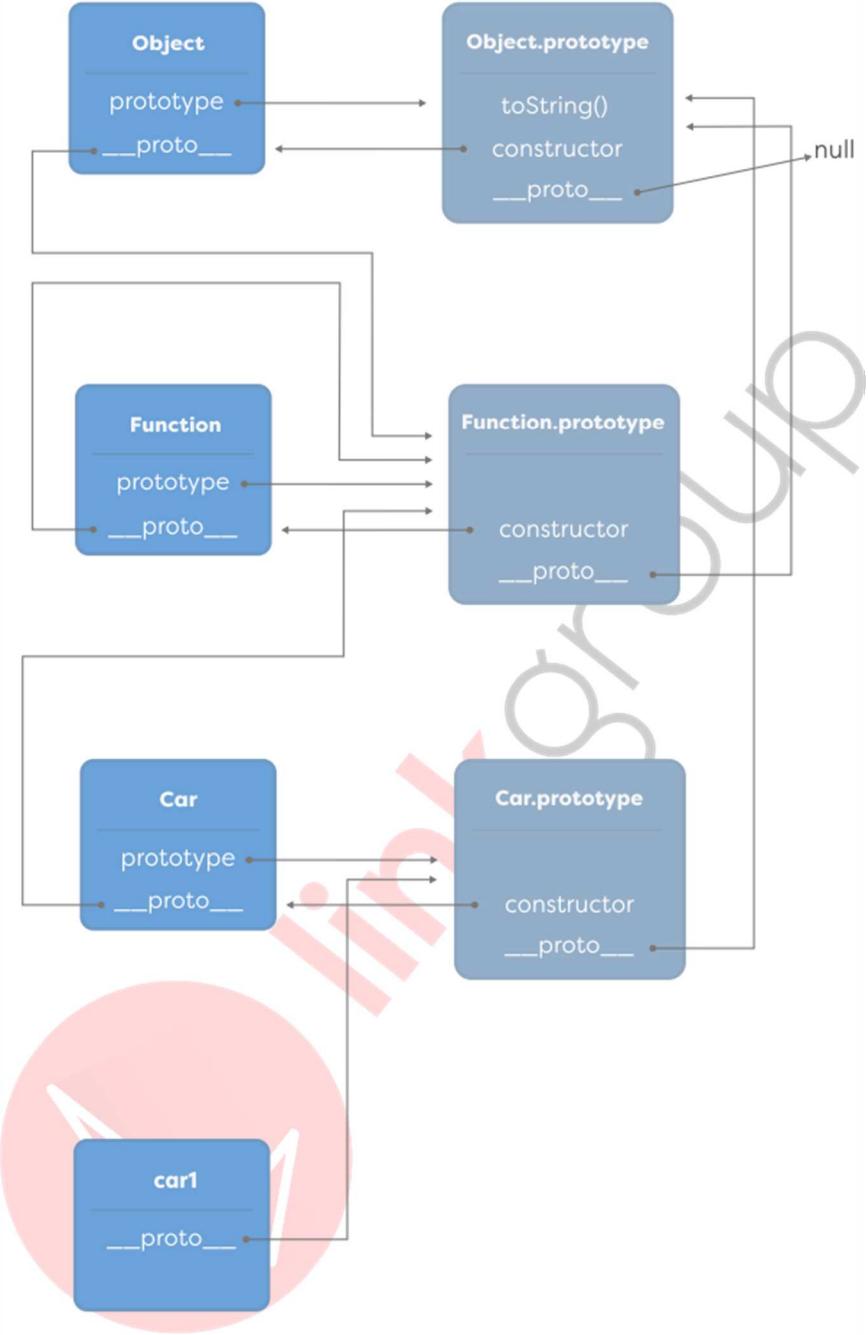
Malopre je rečeno da objekti u JavaScriptu svojstva i metode nasleđuju direktno iz prototipskih objekata. To praktično znači da je objektu `car1` na raspolaganju `constructor` svojstvo koje se nalazi unutar prototipa funkcije `Car()`. Stoga je moguće napisati nešto ovako:

```
var car2 = new car1.constructor("Ford", "Taurus", 1876, "blue");
```

Na ovaj način obavljen je kreiranje još jednog objekta automobila. Ipak, prilikom kreiranja objekta nije direktno korišćena konstruktorska funkcija `Car()`, već smo do nje došli zaobilaznim putem – korišćenjem svojstva `constructor`. Na ovaj način smo dokazali sledeće:

- objekat `car1` zaista poseduje svojstvo `constructor` iako ono nije direktno definisano unutar takvog objekta; svojstvo naravno dolazi od prototipskog objekta funkcije `Car()`
- svojstvo `constructor` čuva referencu na konstruktorskiju funkciju, pa je zapravo poziv ovoga svojstva jednak pozivu konstruktorske funkcije; u prikazanom primeru se pozivanjem svojstva `constructor` nad objektom `car1`, uz upotrebu ključne reči `new`, kreira `car2` objekat.

Ipak, do sada još uvek nije prikazan kompletan dijagram na kome se vidi na koji način naš `car1` objekat dobija metodu `toString()`. Stoga, sledi slika sa dijagramom koji oslikava kompletan lanac prototipova našeg `car1` objekta (slika 2.6).



Slika 2.6. Lanac prototipova `car1` objekta

Na slici 2.6. sada se može videti kompletan lanac prototipova `car1` objekta. Dijagramu su pored već prikazanih dodati i prototipski objekti i konstruktorske funkcije koje se u hijerarhiji nalaze iznad. Analizom dijagrama mogu se zaključiti brojne važne činjenice o lancu prototipova:

- **svi prototipski objekti direktno nasleđuju prototip objekta Object – Object.prototype**

Baš kao i svi drugi objekti, i prototipovi poseduju nezvanično svojstvo `__proto__` koje ukazuje na roditeljski prototip. Zbog toga sledeća poređenja rezultuju true vrednošću:

```
console.log(Car.prototype.__proto__ === Object.prototype);  
//true  
console.log(Function.prototype.__proto__ === Object.prototype);  
//true
```

- **prototip objekta Object poslednji je u lancu prototipova, te zbog toga on ne poseduje roditeljski prototip**

S obzirom na to da `Object.prototype` ne poseduje roditeljski prototip, vrednost njegovog `__proto__` svojstva je `null`, što se lako može proveriti:

```
console.log(Object.prototype.__proto__); //null
```

- **roditeljski prototip svih konstruktorskih funkcija jeste objekat `Function.prototype`**

Objekti koji predstavljaju konstruktorske funkcije (`Car`, `Function`, `Object`...) imaju zajednički roditeljski prototip – `Function.prototype`. To se vrlo lako može dokazati korišćenjem svojstva `__proto__` konstruktorskih funkcija:

```
console.log (Car.__proto__ === Function.prototype); //true  
console.log (Object.__proto__ === Function.prototype); //true  
console.log (Function.__proto__ === Function.prototype); //true
```

Sva prikazana poređenja rezultuju true vrednošću. To je tačno čak i za `Function()` konstruktorsku funkciju. Tako je ona ujedno i jedina funkcija čija svojstva `__proto__` i `prototype` imaju identičnu vrednost.

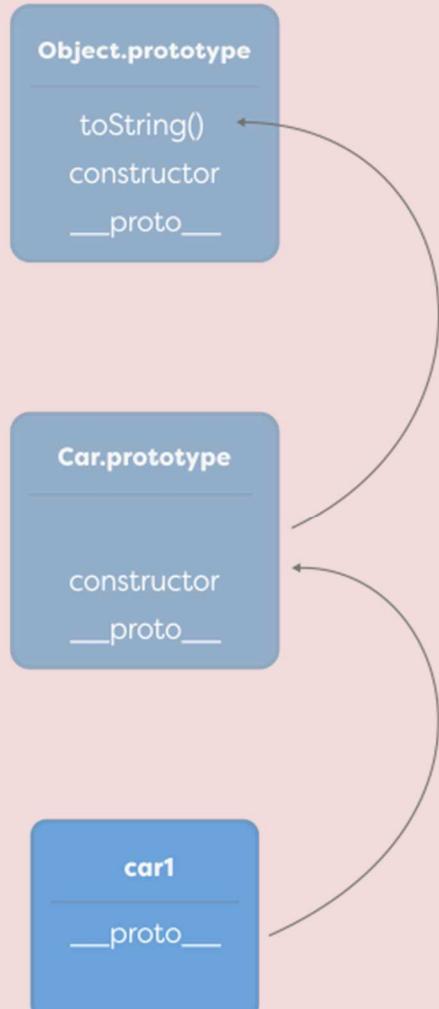
Kako JavaScript pronalazi svojstvo ili metodu u lancu prototipova?

Nakon upoznavanja sa strukturom lanca prototipova, možemo razumeti kako je metoda `toString()` dostupna i našem `car1` objektu. Ona je definisana unutar `Object.prototype` objekta, pa je upravo zbog toga dostupna i našem `car1` objektu.

Kada se metoda `toString()` pozove, nad objektom `car1` obavlja se sledeće:

- web pregledač inicijalno proverava da li `car1` poseduje metodu `toString()`
- ukoliko `car1` objekat, odnosno njegova konstruktorska funkcija `Car()`, ne poseduje metodu `toString()`, pretraga se nastavlja unutar prototipskog objekta `Car.prototype`
- ukoliko ni `Car.prototype` ne poseduje metodu `toString()`, prelazi se na prototip `Object` objekta, odnosno na objekat `Object.prototype`
- s obzirom na to da se metoda `toString()` nalazi unutar `Object.prototype` objekta, pretraga se završava.

Sve opisane korake ilustruje slika 2.7.



Slika 2.7. Način na koji se u lancu prototipova pronalazi metoda `toString()`

Metode dostupne svim objektima

Bitno je reći da metoda `toString()` nije jedina koja je posredstvom `Object.prototype` objekta dostupna svim ostalim objektima u jeziku JavaScript. Objekat `Object.prototype` poseduje i razne druge metode:

- `toLocaleString()`
- `toSource()`
- `hasOwnProperty()`
- ...

Sa različitim metodama `Object.prototype` objekta upoznavaćemo se kada se za tako nešto javi potreba.

Pitanje

Nezvanično svojstvo kojim je moguće pristupiti prototipu nekog objekta je:

- proto
- prototype
- constructor
- proto

Objašnjenje:

S obzirom na to da dugo vremena nije postojao zvaničan način za pristup prototipu nekog objekta, proizvođači web pregledača su samostalno kreirali svojstvo proto. Tako je ovo svojstvo moguće koristiti kako bi se došlo do prototipa nekog objekta.

Nasleđivanje korišćenjem prototipova

Lanci prototipova, o kojima je bilo reči u prethodnim redovima, osnova su na kojoj se temelji nasleđivanje u JavaScriptu. Prethodni primer ilustrovaо je nasleđivanje koje se u potpunosti odvija bez naše intervencije. Naime, svi objekti koje samostalno kreiramo automatski dobijaju metodu `toString()` koja se nalazi unutar prototipa objekta `Object`. Ovde je bitno razumeti da se u JavaScriptu isključivo nasleđuju osobine koje su definisane unutar objekata prototipova, a ne one koje se nalaze unutar konkretnih objekata, odnosno konstruktorskih funkcija. Na primer, metoda `toString()` nalazi se unutar `Object.prototype` objekta i upravo zbog toga naš `car1` objekat može da koristi takvu metodu. Ipak, sve metode i svojstva koja se nalaze unutar objekta `Object` dostupni su isključivo objektima tipa `Object`, odnosno samo objektima koji se direktno kreiraju korišćenjem konstruktorske funkcije `Object()`. Na primer, nešto ranije je prikazana metoda `Object.getPrototypeOf()`. Reč je o metodi koja je definisana unutar `Object` objekta. Upravo zbog toga jedna ovakva naredba proizvešće izuzetak:

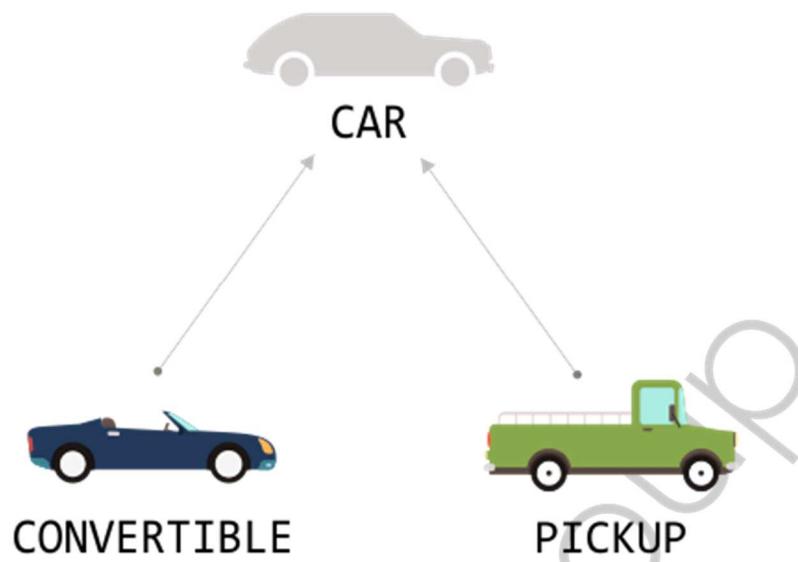
```
Car.getPrototypeOf(car1);
```

Naredba proizvodi:

```
Uncaught TypeError: Car.getPrototypeOf is not a function
```

Rezultat potvrđuje rečeno: **u JavaScriptu nasleđivanje se obavlja isključivo preko prototipskih objekata.**

S obzirom na to da sada znamo kako se u JavaScriptu postiže nasleđivanje, spremni smo da kreiramo jedan primer nasleđivanja između objekata koje ćemo samostalno definisati. Već kreirani objekat `Car` sada će biti nasleđen od strane dva specijalizovana objekta za kreiranje nešto konkretnijih varijanti automobila (slika 2.8).



Slika 2.8. Primer nasleđivanja po kome objekti tipa Convertible i Pickup nasleđuju objekat Car

Na slici 2.8. prikazano je ono što želimo da postignemo. Objekti Convertible i Pickup naslediće određene osnovne osobine od objekta Car. Pritom, objekti Convertible i Pickup u određenoj meri nadogradiće i izmeniti nasleđeni skup osobina kako bi se izvršilo modelovanje nekih specifičnosti ovakvih tipova automobila. Na primer, Convertible objekti imajuće svojstvo koje će čuvati informaciju o tipu krova (*platneni, metalni* i slično), dok će objekti Pickup posedovati svojstvo za čuvanje informacija o zapremini tovarnog prostora.

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

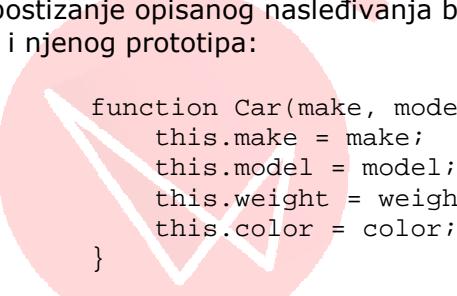
```
Subary Legacy
```

```
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. Naravno, misli se na nešto ranije spomenute objekte `Convertible` i `Pickup`. 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 funkcioniše 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 ove metode 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().

Postavljanje prototipa i reference na konstruktor

U dosadašnjem toku kreiranja Convertible objekta obavljeno je kreiranje konstruktorske funkcije koja koristi logiku već definisane Car() funkcije. Ipak, mi do sada ni na koji način nismo definisali logiku pomoću koje će Convertible objekat naslediti metode getName() i getInfo() koje se nalaze unutar Car.prototype objekta. Tako nešto biće obavljeno u narednim redovima.

Nešto ranije rečeno je da se prilikom kreiranja JavaScript funkcija automatski kreira i prototipski objekat. To znači da je u našoj situaciji kreiran objekat – Convertible.prototype. Ipak, kako bi Convertible nasledio metode iz Car prototipa, na neki način je potrebno ručno postaviti njegov prototip. To se može obaviti na sledeći način:

```
Convertible.prototype = Object.create(Car.prototype);
```

Prikazanom naredbom za prototip Convertible() konstruktorske funkcije postavlja se prototip konstruktorske funkcije Car(). To se obavlja uz asistenciju jedne posebne JavaScript metode – create().

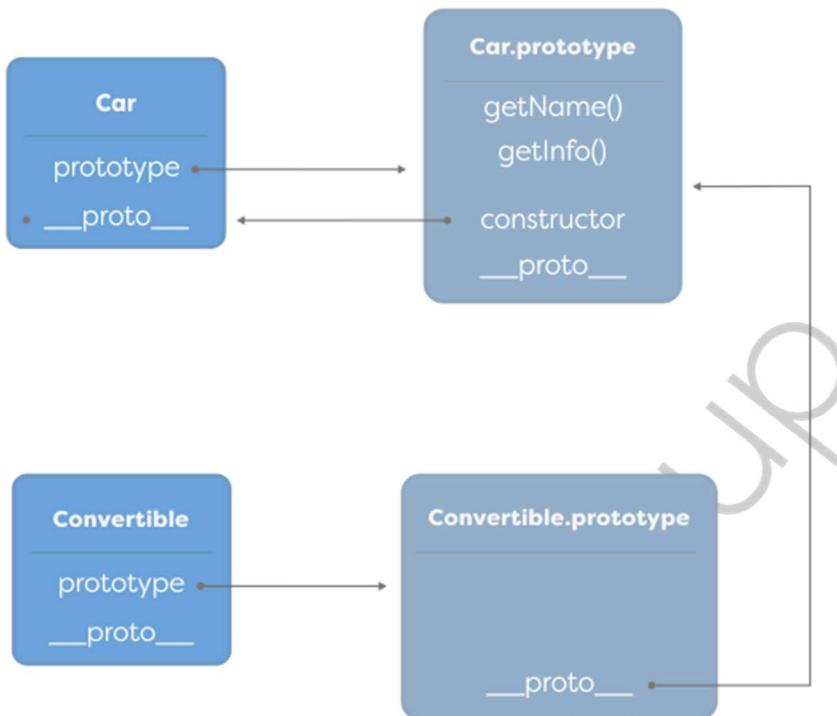
JavaScript create() funkcija

JavaScript funkcija create() koristi se za kreiranje novih objekata, čiji će prototip biti objekat koji se ovakvoj funkciji prosledi kao parametar. Stoga, ova funkcija prihvata jedan parametar:

```
Object.create(proto)
```

Parametar proto se odnosi na prototip koji će biti dodeljen novokreiranom objektu.

Korišćenjem upravo prikazane naredbe redefiniše se podrazumevano kreiran Convertible.prototype objekat i umesto njega se postavlja novi objekat, korišćenjem funkcije create(). Novopostavljeni prototip Convertible() funkcije ima svoj prototip – to je Car.prototype koji se funkciji create() prosleđuje kao parametar. Ono što je na ovaj način postignuto ilustrovano je slikom 2.9.



Slika 2.9. Redefinisanje Convertible.prototype objekta

Sa slike 2.9. sada se može videti da je konstruktorska funkcija Convertible preko svog prototipa povezana sa Car.prototype objektom, te će zbog toga ona, ali i svi objekti koji se pomoću nje kreiraju, imati pristup metodama getName() i getInfo(). Drugim rečima, oni će takve metode naslediti.

Na slici 2.9. može se videti još jedna zanimljiva situacija. S obzirom na to da smo mi samostalno postavili prototipski objekat konstruktorske funkcije Convertible (Convertible.prototype), takav prototipski objekat ne poseduje sopstveno constructor svojstvo koje upućuje na konstruktorskiju funkciju. Upravo zbog toga će ovakva situacija stvoriti jednu nelogičnost. Kao svoju konstruktorskiju funkciju, Convertible.prototype imaće funkciju Car(), zato što će svojstvo constructor naslediti od svog roditelja – Car.prototype objekta. Ovo se veoma lako može proveriti:

```
console.log(Convertible.prototype.constructor);
```

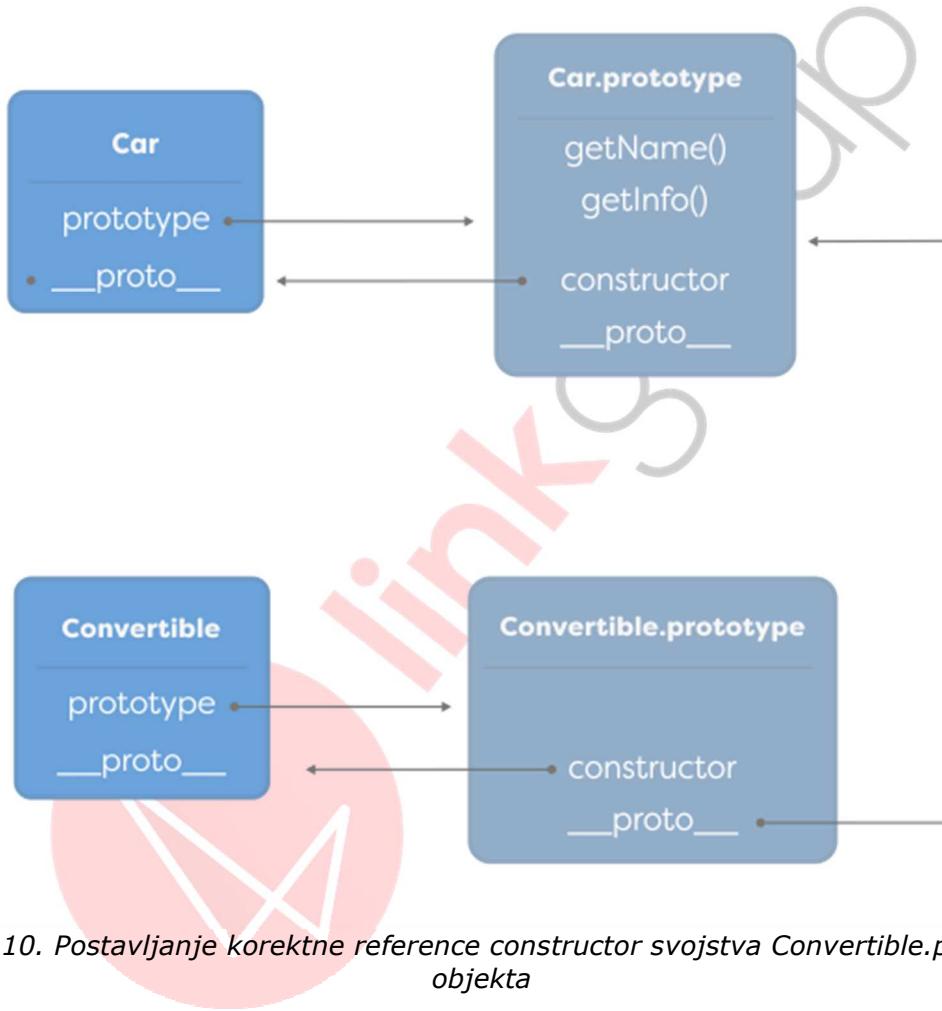
Rezultat ovakve naredbe biće sledeći ispis unutar konzole:

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

Ispis unutar konzole jasno pokazuje da je konstruktorska funkcija `Convertible.prototype` objekta zapravo funkcija `Car()`, iako to treba da bude funkcija `Convertible()`. Ovakvu nedoslednost lako ćemo ispraviti na sledeći način:

```
Convertible.prototype.constructor = Convertible;
```

Sada je za vrednost `constructor` svojstva `Convertible.prototype` objekta postavljena referenca na konstruktorskiju funkciju `Convertible()`. Na ovaj način se dobija objektna struktura kao na slici 2.10.



Slika 2.10. Postavljanje korektnje reference constructor svojstva `Convertible.prototype` objekta

Polimorfizam – redefinisanje metoda

U prethodnom poglavlju bavili smo se nasleđivanjem kao jednim od najznačajnijih postulata objektno orijentisanog programiranja. Uz nasleđivanje ide priča i o polimorfizmu, postulatu koji još nije demonstriran na realnom primeru. Zapravo, polimorfizam je postulat objektno orijentisanog programiranja koji nam je neophodan kako bismo dovršili primer nasleđivanja iz ove lekcije.

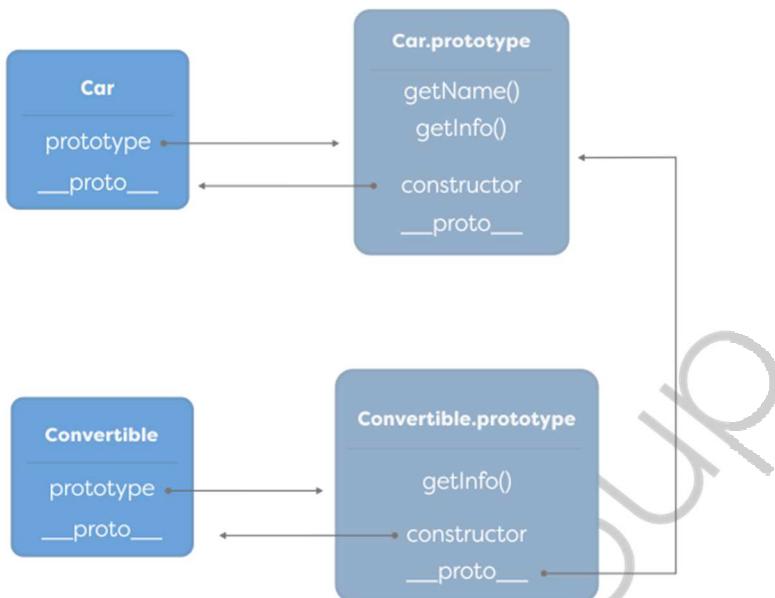
Nešto ranije ste mogli videti da se unutar `Car.prototype` objekta nalaze dve metode – `getName()` i `getInfo()`. Metoda `getName()` vraća objedinjene vrednosti make i model svojstava, dok metoda `getInfo()` obezbeđuje sveobuhvatne informacije o jednom automobilu. S obzirom na to da obe metode poseduju logiku koja je definisana unutar `Car.prototype` objekta, one će se ponašati identično i za objekte kreirane korišćenjem konstruktorske funkcije `Car()`, ali i za objekte koji se kreiraju korišćenjem funkcije `Convertible()`. Ipak, objekti koji se budu kreirali korišćenjem `Convertible()` funkcije poseduju jedno svojstvo više (`roofType`). Šta ukoliko želimo da i njega uvrstimo u ispis koji se dobija od strane metode `getInfo()`? Odnosno, šta ukoliko želimo da se za `Car` objekte od metode `getInfo()` dobija dosadašnja povratna vrednost, a za `Convertible` objekte povratna vrednost u koju će se uvrstiti i `roofType` svojstvo?

Da bi se tako nešto postiglo, neophodno je pribeći jednoj veoma često korišćenoj praksi u objektno orijentisanom programiranju. Reč je o redefinisanju metoda. Redefinisanje metoda omogućava da više objekata poseduje metode sa identičnim nazivom, ali da svaki objekat poseduje sopstvenu logiku jedne iste metode. Upravo takav pristup oslikava postulat polimorfizma, s obzirom na to da se polimorfizam odnosi na višezačност jednog istog pojma. U ovom slučaju takav pojam je metoda `getInfo()`, koja će, u zavisnosti od objekta nad kojim se pozove, imati različitu logiku.

Kako bi se postiglo opisano, odnosno kako bismo konstruktorskoj funkciji `Convertible()` i svim objektima koji se pomoću nje kreiraju dodelili sopstvenu verziju metode `getInfo()` dovoljno je uraditi sledeće:

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

Na ovaj način je unutar `Convertible.prototype` objekta dodata metoda `getInfo()`, koja sadrži specifičnu logiku koja će se koristiti samo za objekte koji se kreiraju korišćenjem konstruktorske funkcije `Convertible()`.



2.11. Redefinisanje `getInfo()` metode unutar `Convertible.prototype` objekta

Slika 2.11. ilustruje lanac prototipova nakon dodavanja specifične metode unutar prototipa `Convertible.prototype`. Zapravo, sada oba prototipska objekta poseduju metodu identičnog naziva (`getInfo()`), ali specifične logike. Reč je o klasičnom primeru polimorfizma.

Redefinisanje metode `getInfo()` omogućava nam da sada uradimo nešto ovako:

```

let car1 = new Car("Subary", "Legacy", 1563, "black");
let convertible1 = new Convertible("Honda", "S2000", 1274,
"silver", "Vinyl, soft-top");

console.log(car1.getInfo());
console.log(convertible1.getInfo());
    
```

Sada su kreirana dva objekta automobila korišćenjem različitih konstruktorskih funkcija – `Car()` i `Convertible()`. Zatim je nad oba takva objekta obavljeno pozivanje metode identičnog naziva – `getInfo()`. Unutar konzole dobija se:

```

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

Make: Honda
Model: S2000
Weight: 1274
Color: silver
Roof type: Vinyl, soft-top
    
```

Analizom ispisa unutar konzole lako je zaključiti da kreirani objekti poseduju metodu `getInfo()` specifičnih osobina.

Prototype dizajn šablon

U ovoj lekciji ilustrovana je implementacija još jednog softverskog dizajn šablona na primeru jezika JavaScript. Reč je o Prototype dizajn šablonu. Mogli ste da vidite da se postulati ovog dizajn šablona u JavaScriptu realizuju na samom jezičkom nivou. Drugim rečima, pojma prototipova je utkan u samu srž JavaScripta.

Sve ono što ste mogli da pročitate o JavaScript prototipovima važi i za sam Prototype dizajn šablon. Reč je o šablonu koji definiše pojam prototipa kao obrazac, odnosno šablon po kome se kreiraju drugi objekti. Ipak, Prototype dizajn šablon najkorisniji je za realizaciju nasleđivanja u nekom jeziku – svi oni elementi koji se definišu unutar prototipskog objekta zajednički su za sve objekte koji se u takvom lancu objekata nalaze ispod.

U JavaScriptu se veoma često kombinuju postulati Prototype i Constructor dizajn šablon. Ono što često može da zbuni početnike jeste mogućnost definisanja elemenata i unutar konstruktorskih funkcija, ali i unutar prototipskih objekata. Drugim rečima, svojstva i metode se mogu definisati na dva mesta – unutar konstruktora i unutar prototipova. Kao logično, nameće se pitanje gde definisati svojstva i metode? Odgovor na ovo pitanje zavisi od efekta koji se želi postići. Konstruktorske funkcije se koriste za definisanje svih onih karakteristika koje će razlikovati objekte koji se kreiraju. Drugim rečima, svojstva i metode koji se definišu unutar konstruktorskih funkcija jedinstveni su za svaki objekat koji se kreira korišćenjem takve funkcije:

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

Na primeru ovakve konstruktorske funkcije, definisana svojstva će se kreirati kao sopstveni članovi svakog od objekata:

```
let car1 = new Car("Subary", "Legacy", 1563, "silver");  
let car2 = new Car("Honda", "Accord", 1480, "black");
```

Svaki objekat kreiran korišćenjem konstruktorske funkcije `Car()` imaće sopstvena svojstva `make`, `model`, `weight` i `color`. Upravo zbog toga se svakom od objekata (`car1`, `car2...`) mogu postaviti jedinstvene vrednosti takvih svojstava (jedan za vrednost svojstva `make` može imati *Subary*, drugi *Honda*, treći *Toyota* i tako dalje).

Identično važi i za metode koje se definišu unutar konstruktorskih funkcija:

```
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;  
    }  
}
```

Kreiranjem objekata na osnovu ove konstruktorske funkcije, svaki od njih će imati sopstvenu kopiju funkcije `getName()`. S obzirom na to da tako nešto realno nije potrebno, prototipovi omogućavaju da se u ovakvima situacijama postigne i poboljšanje performansi. Premeštanjem ovakve funkcije unutar prototipa, svi objekti kreirani na osnovu `Car()` funkcije imaju samo referencu na jednu istu funkciju `getName()`, koja će se nalaziti unutar prototipskog objekta.

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

Zbog upravo navedenih osobina konstruktora i prototipova, kaže se da se u JavaScriptu često obavlja kombinovanje Prototype i Constructor dizajn šablonu.

Primer – kreiranje konstruktorske funkcije Pickup()

U dosadašnjem toku lekcije detaljno je objašnjen jedan primer nasleđivanja i polimorfizma u jeziku JavaScript, kreiranjem objekata `Car` i `Convertible`. Sada, kao primer za kraj ove lekcije, biće prikazano isto, ali na primeru još jedne konstruktorske funkcije – `Pickup()`. Ova konstruktorska funkcija služiće za kreiranje specijalizovanih objekata automobila – kamioneta, koji će deo svojih osobina naslediti od `Car` objekta.

```
function Pickup(make, model, weight, color, cargoVolume) {  
    Car.call(this, make, model, weight, color);  
    this.cargoVolume = cargoVolume;  
}  
Pickup.prototype = Object.create(Car.prototype);  
Pickup.prototype.constructor = Pickup;  
Pickup.prototype.getInfo = function () {  
    return "Make: " + this.make + "\n" +  
        "Model: " + this.model + "\n" +  
        "Weight: " + this.weight + "\n" +  
        "Color: " + this.color + "\n" +  
        "Cargo volume: " + this.cargoVolume;  
}  
let pickup1 = new Pickup("Ford", "F-150", 1846, "silver", 63);  
let pickup1Name = pickup1.getName();  
let pickup1Info = pickup1.getInfo();  
console.log(pickup1Name);  
console.log(pickup1Info);
```

Po identičnom principu koji je prikazan u ovoj lekciji sada je obavljen kreiranje konstruktorske funkcije `Pickup()`, koja veći deo svojih osobina nasleđuje od `Car` i `Car.prototype` objekata. Osobenost `Pickup()` konstruktorske funkcije odnosi se na svojstvo `cargoVolume`, sa obzirom na to da kamioneti poseduju otvoreni tovarni prostor, koji ne postoji kao sastavni deo standardnih automobila.

Nakon kreiranja konstruktorske funkcije `Pickup()` i definisanja njenog prototipa, u primeru se obavlja kreiranje i jednog objekta – `pickup1`. Nakon kreiranja objekta, pozivaju se metode `getName()` i `getInfo()`, a povratne vrednosti se ispisuju unutar konzole:

```
Ford F-150
```

```
Make: Ford
Model: F-150
Weight: 1846
Color: silver
Cargo volume: 63
```

Rezime

- prototip je obrazac, odnosno šablon po kome se kreiraju drugi objekti;
- prototipovi su osnovni pojam koji u JavaScriptu obezbeđuje mogućnost nasleđivanja;
- u JavaScriptu objekti ne nasleđuju jedni druge direktno, već se to čini posredstvom prototipova;
- hijerarhija u kojoj svaki prototipski objekat nasleđuje osobine objekata koji se u takvoj hijerarhiji nalaze iznad naziva se lanac prototipova;
- prilikom kreiranja funkcija u jeziku JavaScript, izvršno okruženje ovog jezika funkcijama automatski dodaje svojstvo `prototype`, koje čuva referencu na objekat koji će biti prototip svim objektima koji se budu kreirali korišćenjem takve funkcije;
- svaki prototipski objekat poseduje svojstvo `constructor`, koje upućuje na originalnu konstruktorskiju funkciju;
- pristup prototipskom objektu može se obaviti korišćenjem nezvaničnog svojstva `__proto__` ili korišćenjem metode `getPrototypeOf()` objekta `Object`;
- svi prototipski objekti direktno nasleđuju prototip objekta `Object` – `Object.prototype`;
- prototip objekta `Object` poslednji je u lancu prototipova, te zbog toga on ne poseduje roditeljski prototip;
- roditeljski prototip svih konstruktorskih funkcija jeste objekat `Function.prototype`
- JavaScript funkcija `call()` omogućava da se metoda koja pripada nekom objektu pozove unutar nekog drugog objekta, kao da je sastavni deo takvog objekta;
- JavaScript funkcija `create()` koristi se za kreiranje novih objekata, čiji će prototip biti objekat koji se ovakvoj funkciji prosledi kao parametar;
- polimorfizam se praktično realizuje redefinisanjem metoda u nasleđenim objektima.