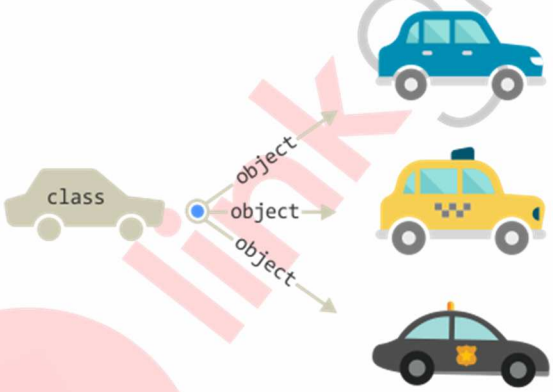
Klase

JavaScript je objektno orijentisan jezik baziran na prototipovima. To praktično znači da su centralne figure JavaScript jezika objekti, dok se prototipovi koriste za realizaciju nasleđivanja i polimorfizma. Tako je sve do juna 2015. godine osnovni način za kreiranje objektnih šablona podrazumevao kreiranje konstruktorskih funkcija, dok se nasleđivanje obavljalo upotrebom prototipova. Ipak, sa pojavom verzije ECMAScript 2015 (ES6), JavaScript je obogaćen brojnim novim elementima, među kojima su i klase.

Pojmu klasa u jeziku JavaScript biće posvećena kompletna lekcija koja je pred vama.

Šta su klase?

Većina modernih, popularnih, objektno orijentisanih jezika poznaje pojam klasa. Klase predstavljaju šablone za izgradnju objekata. Unutar njih objedinjuju se osnovne karakteristike i ponašanja koje će imati svi objekti koji se kreiraju korišćenjem konkretne klase.



Slika 3.1. Klasno programiranje

Slika 3.1. ilustruje osnovni postulat klasnog programiranja. Klasa je obrazac, odnosno šablon, na osnovu koga se kreiraju objekti. Tako je na levoj polovini slike 3.1. prikazana klasa, a na desnoj objekti koji se kreiraju na osnovu takve klase.

Klase definišu skup svojstava (*marka*, *model*, *težina*, *boja* i slično), ali i skup ponašanja (*startuj motor*, *promeni brzinu*, *ubrzaj*, *ukoči* i tako dalje). Objekti se kreiraju na osnovu klasa, a na osnovu jedne klase može se kreirati proizvoljan broj objekata. Svi objekti kreirani korišćenjem iste klase poseduju identičan skup svojstava. Ipak, svaki objekat kreiran na osnovu jedne klase može imati različite osobine. Tako jedan objekat može predstavljati automobil honda accord, težine 1.560 kg, crne boje. Drugi objekat može predstavljati lexus IS350, težine 1.680 kg, sive boje.

Proces kreiranja objekata na osnovu klasa u objektno orijentisanom programiranju naziva se **instanciranje**.

Klase u jeziku JavaScript

Godine 2015. u ECMAScript specifikaciju uvršćena je funkcionalnost klasa. Ipak, kao što ćete uskoro moći da vidite, klase su u JavaScriptu samo zaobilazni put za kreiranje objektnih šablona korišćenjem konstruktorskih funkcija i za nasleđivanje upotrebom prototipova. Tako je JavaScript i dalje ostao isključivo prototipski baziran jezik, dok je funkcionalnost klasa zamišljena samo kao olakšica za kreiranje šablona i postizanje nasleđivanja.

Kreiranje klasa u JavaScriptu

U JavaScriptu klase su zapravo specijalna vrsta funkcija. Ipak, umesto ključne reči `function`, za njihovo kreiranje se koristi specijalna ključna reč `class`:

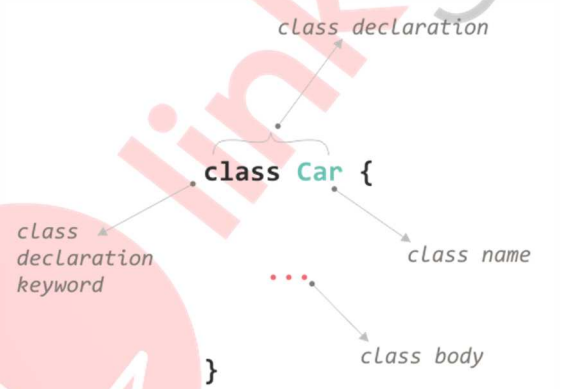
```
class Car {

  ...

}
```

Nakon ključne reči `class`, navodi se naziv klase. U primeru je to `Car`. Ključna reč `class` i naziv klase drugačije se nazivaju deklaracija klase. Pored deklaracije, klase poseduju i telo, koje se ograničava otvorenom i zatvorenom vitičastom zagradom. Trenutno je telo naše klase `Car` prazno.

Unutar tela klase mogu se naći klasni članovi – konstruktor i metode.



Slika 3.2. Osnovna struktura JavaScript klasa

Pitanje

Kreiranje klasa započinje definisanjem ključne reči:

- **class**
- this
- object
- klass

Objašnjenje:

Kreiranje klase započinje navođenjem specijalne ključne reči – class.

Konstruktor klase

Svaka klasa u JavaScriptu unutar svog tela može da poseduje i jednu specijalnu metodu koja se drugačije naziva **konstruktor**. Konstruktor se kreira korišćenjem ključne reči `constructor`:

```
class Car {

    constructor(make, model, weight, color) {
        this.make = make;
        this.model = model;
        this.weight = weight;
        this.color = color;
    }

}
```

Konstruktor je specijalna metoda unutar klase, koja se poziva prilikom kreiranja objekata, a najčešće se koristi da postavi vrednosti svojstava objekta koji se kreira. U prikazanom primeru mi smo klasi `Car` dodelili konstruktor koji prihvata četiri parametra. Vrednosti takvih parametara koriste se za postavljanje vrednosti svojstava *make*, *model*, *weight* i *color*, objekta koji se kreira.

Bitno je znati da unutar jedne klase može postojati samo jedna metoda obeležena ključnom rečju `constructor`. Drugim rečima, JavaScript klase mogu imati samo jedan konstruktor.

Konstruktor se unutar JavaScript klasa ne mora definisati, odnosno može se izostaviti. U takvoj situaciji, JavaScript izvršno okruženje samostalno definiše **podrazumevani konstruktor**:

```
constructor() {}
```

Podrazumevani konstruktor nema parametre, niti bilo kakvu logiku unutar svog tela. Onog trenutka kada se konstruktor samostalno definiše podrazumevani konstruktor prestaje da postoji.

Napomena

Bitno je reći da u ovom trenutku i dalje ne postoji način da se unutar klasa posebno definišu javna, privatna ili statička svojstva. Jedino što se može uraditi jeste definisanje svojstava koja će imati svaki objekat koji se kreira korišćenjem klase. To se najčešće obavlja baš kao u prikazanom primeru – unutar konstruktora nad ključnom rečju `this`. U toku je razvoj na implementaciji javnih i privatnih svojstava i očekuje se da se u budućnosti takva funkcionalnost pojavi kao sastavni deo jezika.

Kreiranje objekta korišćenjem klase

Osnovna svrha klasa u bilo kom programskom jeziku jeste definisanje šablona na čijoj osnovi će se kreirati objekti sa identičnim skupom svojstava. Tako se objekat koji predstavlja jedan automobil, korišćenjem upravo prikazane klase, može kreirati na sledeći način:

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

Postupak za kreiranje objekata na osnovu klase identičan je kreiranju objekata na osnovu konstruktorskih funkcija. Ispred naziva klase navodi se ključna reč **new**, a klasi se prosleđuju parametri koji su definisani konstruktorom.

Prikazanom naredbom obavlja se instanciranje klase i time se kreira objekat čija referenca se dodeljuje promenljivoj `car1`. Prilikom instanciranja klase, automatski se obavlja pozivanje konstruktorske metode, pa se samim tim i izvršava logika definisana konstruktorom, odnosno postavljanje vrednosti svojstava objekta koji se kreira.

Na identičan način je sada moguće obaviti kreiranje proizvoljnog broja objekata automobila različitih karakteristika:

```
var car2 = new Car("Ford", "Taurus", 1876, "blue");
var car3 = new Car("Porsche", "Panamera", 1963, "grey");
```

Hoisting se ne primenjuje nad deklaracijama klasa

Jedna od značajnijih razlika između funkcija i klasa jeste činjenica da se deklaracije klasa ne podižu na početak dokumenta, kao što je to slučaj sa funkcijama. Stoga, nije moguće obaviti instanciranje klase pre njene deklaracije:

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


class Car {

    constructor(make, model, weight, color) {
        this.make = make;
        this.model = model;
        this.weight = weight;
        this.color = color;
    }

}
```

Ovakva situacija proizvodi izuzetak:

```
Uncaught ReferenceError: Cannot access 'Car' before initialization
```

Metode klasa

Pored konstruktora, klasni članovi mogu biti i regularne metode:

```
class Car {

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

```
        }


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

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

    }
```

Sada su unutar klase `Car` smeštena još dva klasna člana – metode `getName()` i `getInfo()`. Metode su definisane korišćenjem naziva, nakon koga se navode zagrade sa opcionim parametrima i telo metode sa logikom između vitičastih zagrada.

Kreirane metode se sada mogu upotrebiti nad svakom instancom, odnosno objektom koji je kreiran korišćenjem ovakve klase:

```
car1.getInfo();
```

Ili:

```
car1.getName();
```

Poređenje konstruktorskih funkcija i prototipova sa klasama

Verovatno ste dosad primetili da se u ovoj lekciji kreiraju objekti identični onima u prethodnoj. To i jeste cilj, kako biste na najbolji način mogli da razumete svrhu klasa u JavaScriptu. Dosad kreirana logika klase `Car` može se uporediti sa konstruktorskom funkcijom kreiranom u prethodnoj lekciji (slika 3.3).

constructor function + prototype

```
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
}
```

class

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

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

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

Slika 3.3. Uporedni prikaz kreiranja šablona za proizvodnju objekata; na levoj polovini upotrebom konstruktorske funkcije i prototipa, a na desnoj korišćenjem klasa

Kod na levoj i onaj na desnoj polovini slike 3.3. su ekvivalentni, odnosno njima se postiže identičan efekat. Na slici se može videti i jedna veoma važna osobina metoda definisanih unutar JavaScript klasa – one automatski postaju sastavni deo prototipskog objekta, te su stoga na raspolaganju svim objektima koji se kreiraju korišćenjem konkretne klase.

get i set metode

Novina unutar ECMAScript 2015 specifikacije jesu i specijalne get i set metode, koje se mogu naći unutar JavaScript klasa, ali i unutar bilo kog drugog JavaScript objekta. Reč je o metodama koje je moguće pozivati kao da su svojstva. Iako ovo može zvučati zbunjujuće, u biti je vrlo jednostavno:

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

    get name() {
        return this.make + " " + this.model;
    }
```

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

}
```

U prikazanom kodu tradicionalna metoda koja se dosad zvala `getName()` pretvorena je u specijalnu get metodu. Ona započinje ključnom rečju `get`, nakon čega se navodi naziv get metode – `name()`. Logika je ostala identična – ova metoda vraća naziv automobila, što je zapravo objedinjena vrednost `make` i `model` svojstava. Povratna vrednost ovakve get metode sada se može dobiti kao da je reč o objektnom svojstvu:

```
var car1 = new Car("Subary", "Legacy", 1563, "black");
console.log(car1.name);
```

U kodu se prvo kreira jedan objekat (`car1`) na osnovu klase `Car`, a zatim se unutar konzole ispisuje vrednost svojstva `name`:

```
Subary Legacy
```

Zapravo, mi znamo da `name` nije svojstvo, već jedna `get` metoda, ali se to ne može naslutiti prilikom rukovanja samim objektima `Car` klase. Ovo je još jedan primer apstrakcije i enkapsulacije, odnosno skrivanja unutrašnjeg funkcionisanja jednog tipa unutar klase ili konstruktorske funkcije. Onaj ko koristi našu klasu `Car` uopšte ne mora da zna da unutar nje ne postoji svojstvo `name` i da se vrednost takvog svojstva formira spajanjem svojstava `make` i `model`.

Upravo prikazani primer ilustrovao je korišćenje jedne get metode kako bi se naizgled korišćenjem svojstva došlo do objedinjene vrednosti dva svojstva. Metode get i set koriste se veoma često kako bi se omogućio pristup nekom svojstvu objekta, uz obavljanje neke propratne logike. Takvu situaciju ilustruje ovakav primer:

```
class Car {
    constructor(make, model, weight, color) {
        this.make = make;
        this.model = model;
        this._weight = weight;
        this.color = color;
    }

    get weight() {
        return this._weight + "kg";
    }

    set weight(weight) {
        if (!isNaN(weight)) {
            this._weight = weight;
        } else {
            throw new TypeError('Value of weight must be a
number.');
        }
    }
}
```

Klasa `Car` sada poseduje jednu get i jednu set metodu. Metode su namenjene dobijanju i postavljanju vrednosti težine vozila. Ipak, dosadašnje svojstvo `weight`, čija vrednost se inicijalno postavlja unutar konstruktora, preimenovano je u `_weight`. Nazivu je dodat prefiks donja crta (*engl. underscore*). Naime, ovo je veoma česta praksa, po kojoj nazivi konkretnih svojstava započinju donjom crtom, a get i set metode se koriste za čitanje i upisivanje vrednosti. Naziv konkretnog svojstva je promenjen zato što u JavaScriptu metode get i set ne mogu imati nazive nekog već definisanog svojstva.

Kreirane get i set metode ilustruju osnovne osobine takvog tipa metoda. Naime, one se veoma retko koriste samo za postavljanje i čitanje vrednosti, već i za izvršavanje neke dodatne logike. Tako se get metoda `weight()` koristi za čitanje vrednosti svojstva `_weight` na koju se dodaje i tekst *kg*. Zatim, set metoda `weight()` koristi se i za proveravanje vrednosti koje se dodeljuje svojstvu `_weight`. Ukoliko se prosleđena vrednost ne može konvertovati u broj, emituje se izuzetak, s obzirom na to da se težina vozila mora izraziti u numeričkom obliku.

Ovakve get i set metode mogu se koristiti na sledeći način:

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

car1.weight = '1356';
console.log(car1.weight);
```

Nakon kreiranja objekta `car1`, prvo se obavlja redefinisanje vrednosti `_weight` svojstva. To se naizgled obavlja svojstvom `weight`, ali je zapravo reč o set metodi `weight()` koja se poziva kao da je svojstvo. Nakon postavljanja vrednosti, obavlja se i njen ispis. U konzoli se dobija:

```
1356kg
```

Može se videti da je na vrednost težine dodata i odrednica *kg*, što je proizvod logike `get` metode `weight()`.

Još jedan primer korišćenja ovakvih get i set metoda može da izgleda ovako:

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

car1.weight = 'some weight';
console.log(car1.weight);
```

Sada se za težinu vozila pokušava postaviti tekstualna vrednost *some weight*. S obzirom na to da se takva vrednost ne može konvertovati u broj, dolazi do emitovanja izuzetka, zahvaljujući logici definisanoj unutar set metode `weight()`:

```
Uncaught TypeError: Value of weight must be a number.
```

Statičke metode

Statičke metode su još jedan jezički element koji oduvek postoji unutar većine modernih programskih jezika, a u JavaScript je uvršćen zajedno sa klasama. Statičke metode su specijalna vrsta metoda koje postoje samo unutar klasa, te zbog toga nisu dostupne konkretnim instancama, odnosno objektima.

Statičke metode u JavaScriptu se kreiraju korišćenjem ključne reči **static**:

```
class Car {
    constructor(make, model, weight, color) {
        this.make = make;
        this.model = model;
        this._weight = weight;
        this.color = color;
    }

    static kwToHp(kw){
        return (kw * 1.34102).toFixed(2);
    }
}
```

Primer ilustruje jednu statičku metodu definisanu unutar klase `Car` (ostale metode su izostavljene zbog bolje preglednosti). Statička metoda je kreirana korišćenjem ključne reči `static` koja je navedena ispred njenog naziva. Zapravo, upotreba ključne reči `static` jedina je osobenost kreiranja statičkih metoda. Svi ostali elementi su identični klasičnim metodama.

Kreirana statička metoda `kwToHp()` namenjena je konvertovanju snage motora automobila iz kilovata u konjske snage. Reč je o funkcionalnosti koja je univerzalna i karakteristična za sve vrste automobila. Stoga ima puno smisla omogućiti njeno korišćenje bez potrebe za prethodnim instanciranjem objekata, što je osnovna osobina statičkih metoda. Stoga se metoda `kwToHp()` upotrebljava na sledeći način:

```
let hpValue = Car.kwToHp(148);
```

Bitno je primetiti da se statička metoda poziva nad klasom. Statičke metode nije moguće pozvati nad instancama:

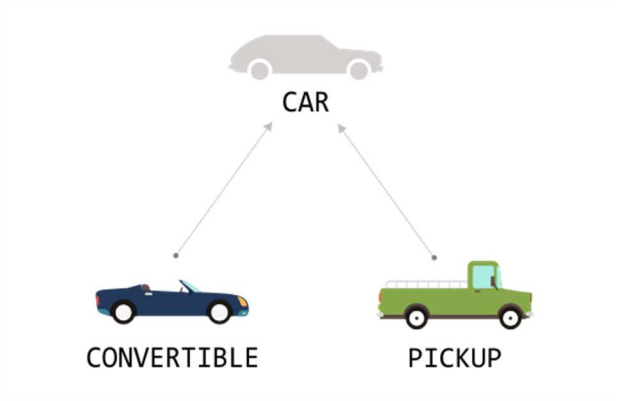
```
var car1 = new Car("Subary", "Legacy", 1563, "black");
let hpValue = car1.kwToHp(148);
```

Ovakav kod proizvodi izuzetak zato što statičke metode postoje isključivo unutar klasa:

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

Nasleđivanje korišćenjem klasa

Prethodna lekcija ilustrovala je osnovne postulate nasleđivanja i polimorfizma, koji se u JavaScriptu postižu upotrebom prototipova. Klase obezbeđuju da se proces nasleđivanja postigne na nešto jednostavniji i udobniji način. Stoga će u nastavku biti prikazan identičan primer nasleđivanja kao i onaj u prethodnoj lekciji, ali će ovoga puta realizacija biti obavljena korišćenjem klasa (slika 3.4).



Slika 3.4. Primer nasleđivanja

Slika 3.4. ilustruje strukturu nasleđivanja koje će biti postignuto između tri klase. `Car` će biti roditeljska klasa, koju će naslediti dve nove klase – `Convertible` i `Pickup`. Za početak, biće prikazana klasa `Car`:

```
class Car {
    constructor(make, model, weight, color) {
        this.make = make;
        this.model = model;
        this._weight = weight;
        this.color = color;
    }

    get name() {
        return this.make + " " + this.model;
    }

    get weight() {
        return this._weight + "kg";
    }

    set weight(weight) {
        if (!isNaN(weight)) {
            this._weight = weight;
        } else {
            throw new TypeError('Value of weight must be a
number.');
        }
    }

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

    static kwToHp(kw){
        return (kw * 1.34102).toFixed(2);
    }

}
```

Klasa `Car` sadrži sve one elemente koji su unutar nje kreirani u dosadašnjem toku lekcije. Sada ćemo preći na realizaciju klase `Convertible`, koja će naslediti klasu `Car`:

```
class Convertible extends Car {

      ...

}
```

Ključna reč kojom se obavlja nasleđivanje između klasa u jeziku JavaScript jeste **extends**. Ova ključna reč navodi se između naziva klase koja nasleđuje i nasleđene klase. Sledeći korak jeste definisanje konstruktorske metode:

```
class Convertible extends Car {

    constructor(make, model, weight, color, roofType) {
        super(make, model, weight, color);
        this.roofType = roofType;
    }

}
```

Unutar konstruktorske metode koristi se još jedna specifičnost jezika JavaScript, koja je predstavljena u ECMAScript 2015 specifikaciji – ključna reč **super**. Dakle, `super` je ključna reč koja omogućava da se obavi pozivanje konstruktora ili pristup svojstvima i metodama roditeljske klase. Ključna reč `super` najčešće se koristi za pozivanje konstruktora roditeljske klase, što je učinjeno i u primeru. Prilikom takvog poziva prosleđuju se parametri koje prihvata konstruktor roditeljske klase. Stoga se upotreba ključne reči `super` u ovom primeru može uporediti sa korišćenjem metode `call()` u prethodnoj lekciji.

Klasa `Convertible` kreirana u prethodnim redovima, nasledila je sve osobine klase `Car`. Stoga je sada moguće napisati nešto ovako:

```
      let convertible1 = new Convertible("Honda", "S2000", 1274, "silver",
"Vinyl, soft-top");

      let convertible1Name = convertible1.name;  //Honda S2000
      let convertible1Weight = convertible1.weight;  //1274kg
```

Redefinisanje metoda u JavaScript klasama

Redefinisanje metoda unutar klasa koje nasleđuju neku drugu klasu takođe je pojednostavljeno u odnosu na direktno korišćenje prototipova. Metoda koju je potrebno redefinisati jednostavno se definiše unutar klase koja nasleđuje:

```
class Convertible extends Car {

    constructor(make, model, weight, color, roofType) {
        super(make, model, weight, color);
        this.roofType = roofType;
```

```
    }


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

}
```

Unutar klase `Convertible` redefiniše se metoda `getInfo()`. Reč je o metodi kojom se prikazuju kompletne informacije o jednom automobilu. S obzirom na to da kabrioleti imaju jednu osobenost, koja se odnosi na tip krova, metoda `getInfo()` je unutar klase `Convertible` redefinisana tako da u povratnu vrednost uključi i podatak o tipu krova:

```
let convertible1 = new Convertible("Honda", "S2000", 1274, "silver",
"Vinyl, soft-top");

let convertible1Info = convertible1.getInfo();
console.log(convertible1Info);
```

Nešto ranije je prikazana ključna reč `super`, koja omogućava pristup roditeljskoj klasi. Ključna reč `super` je dosad korišćena za pozivanje konstruktora roditeljske klase, ali se ona može koristiti i za pristup svojstvima i metodama iz roditeljske klase. Tako se redefinisana metoda `getInfo()` sada može preurediti, tako da već definisanu logiku unutar roditeljske `getInfo()` metode iskoristi, a doda samo ono što nedostaje:

```
class Convertible extends Car {

    constructor(make, model, weight, color, roofType) {
        super(make, model, weight, color);
        this.roofType = roofType;
    }


    getInfo() {
        return super.getInfo() + "\n" +
        "Roof type: " + this.roofType;
    }

}
```

Sada je proces nasleđivanja funkcionalnosti podignut za lestvicu više, pa je unutar metode `getInfo()` iskorišćena logika koja je navedena u roditeljskoj klasi. Metoda `getInfo()` roditeljske klase pozvana je navođenjem ključne reči `super`. Na vrednost dobijenu od `getInfo()` metode roditeljske klase dodat je i tekst koji je specifičan za objekte koji će biti kreirani klasom `Convertible`. Na ovaj način efikasno je rešen problem nepotrebnog ponavljanja jednog istog koda. Time su na kraju ispoštovana načela jednog od najznačajnijih programerskih principa – DRY (*engl. Don't Repeat Yourself*).

Primer – kreiranje klase Pickup

Za kraj ove lekcije biće prikazan i primer kreiranja klase `Pickup`, koja će naslediti klasu `Car`:

```
class Pickup extends Car {
    constructor(make, model, weight, color, cargoVolume) {
        super(make, model, weight, color);
        this.cargoVolume = cargoVolume;
    }
    getInfo() {
        return super.getInfo() + "\n" +
            "Cargo volume: " + this.cargoVolume;
    }
}
```

Nasleđivanje je obavljeno identično već viđenom pristupu, koji je prikazan u prethodnim redovima ove lekcije. Ovako kreirana klasa se na sledeći način može iskoristiti za kreiranje objekata:

```
let pickup1 = new Pickup("Ford", "F-150", 1846, "silver", 63);

let pickup1Name = pickup1.name;
let pickup1Info = pickup1.getInfo();

console.log(pickup1Name);
console.log(pickup1Info);
```

Objekti tipa `Pickup` poseduju sve osobine `Car` objekata, uz dodatak svojstva `cargoVolume`. `Pickup` objekti poseduju i sve metode koje su definisane unutar klase `Car`. Redefinisana je metoda `getInfo()` koja sada povratnu vrednost roditeljske metode nadograđuje ispisom vrednosti specifičnog svojstva `cargoVolume`.

Unutar konzole dobija se sledeći ispis:

```
Ford F-150

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

Rezime

- specifikacijom ECMAScript 2015 (ES6) u JavaScript dodat je pojam klasa;
- klase su u JavaScriptu samo zaobilazni put za kreiranje objektnih šablona korišćenjem konstruktorskih funkcija i za nasleđivanje upotrebom prototipova;
- klase predstavljaju šablone za izgradnju objekata;
- na osnovu jedne klase može se kreirati proizvoljan broj objekata sa identičnim skupom svojstava i metoda;

- proces kreiranja objekata na osnovu klasa se u objektno orijentisanom programiranju naziva instanciranje;
- JavaScript klase su specijalna vrsta funkcija;
- kreiranje klasa započinje navođenjem ključne reči `class`;
- JavaScript klase kao svoje članove mogu imati konstruktor i metode;
- konstruktor je specijalna metoda unutar klase, koja se poziva prilikom kreiranja objekata; deklariše se ključnom rečju `constructor`
- JavaScript klase mogu imati samo jedan konstruktor;
- ukoliko se konstruktor ne definiše, JavaScript izvršno okruženje samostalno kreira podrazumevani konstruktor;
- podrazumevani konstruktor nema parametara, niti bilo kakvu logiku unutar svog tela;
- za kreiranje objekta, ispred naziva klase navodi se ključna reč `new`
- deklaracije klasa se ne podižu (hoisting) na početak dokumenta, kao što je to slučaj sa funkcijama;
- get i set metode su one metode koje se mogu koristiti kao da su svojstva;
- statičke metode su specijalna vrsta metoda koje postoje samo unutar klasa, odnosno mogu se pozivati samo nad klasama; kreiraju se upotrebom ključne reči `static`
- ključna reč kojom se obavlja nasleđivanje između klasa u jeziku JavaScript jeste `extends`
- `super` je ključna reč koja omogućava da se obavi pozivanje konstruktora ili pristup svojstvima i metodama roditeljske klase.