6
Projektni obrasci

Projektni obrazac (engl. *design pattern*) je višekratno upotrebljivo rešenje za poznat problem; definicija izraza je zaista široka i njeno značenje može se odnositi na više domena iste aplikacije. Međutim, taj izraz se često povezuje i s dobro poznatim skupom objektno orijentisanih obrazaca koji je 90-ih godina načinila popularnim knjiga *Design Patterns: Elements of Reusable Objektno orijentisan Software*, *Pearson Education*, čiji su autori gotovo legendarna **četvoročlana banda** (engl. *Gang of Four*, *GoF*): Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides. Taj konkretan skup obrazaca često ćemo nazivati *tradicionalni* projektni modeli ili GoF projektni modeli.

Primena tog skupa objektno orijentisanih projektnih modela na JavaScript nije tako linearna, ni formalna kao što bi bila u nekom klasično objektno orijentisanom jeziku. Kao što znamo, JavaScript je jezik koji podržava više modela programiranja, objektno je orijentisan, zasniva se na prototipovima i dinamički određuje tip objekata; svoje funkcije tretira kao kao građane prvog reda i omogućava programiranje u stilu funkcionalnog programiranja. Te osobine čine JavaScript vrlo raznovrsnim jezikom, što autoru aplikacija pruža ogromnu moć, ali u isto vreme prouzrokuje i cepkanje stilova programiranja, konvencija, tehnika, a na kraju čak obrazaca svog ekosistema. Postoji toliko mnogo načina da se pomoću JavaScripta postigne isti rezultat, da svako ima svoje mišljenje o tome koji je najbolji pristup problemu. Jasan pokazatelj tog fenomena jeste obilje frejmvorkova i pristrasnih biblioteka u JavaScriptovom ekosistemu; verovatno ih nijedan drugi jezik nije video toliko, naročito sada pošto je Node.js doneo JavaScriptu toliko novih iznenađujućih mogućnosti i uveo toliko mnogo novih scenarija upotrebe.

U tom kontekstu, priroda JavaScripta je uticala i na tradicionalne projektne obrasce. Postoji toliko mnogo načina na koje se mogu implementirati, da njihova tradicionalna, „tvrda“ objektno orijentisana implementacija znači da oni prestaju da budu obrasci. U nekim slučajevima, nisu čak ni mogući, budući da JavaScript, kao što znamo, nema *prave* klase, ni apstraktne interfejse. Međutim, ono što ostaje nepromenjeno jeste izvorna ideja na kojoj se svaki obrazac zasniva, problem koji rešava i koncepti koji čine srž rešenja.

U ovom poglavlju, razmotrićemo kako se neki među najvažnijim GoF projektnim obrascima preslikavaju na Node.js i njegovu filozofiju, što znači da ćemo ponovo otkritui njihovu važnost, ali sa druge tačke gledišta. Među tim tradicionalnim projektnim obrascima, pogledaćemo i neke „manje tradicionalne“ projektne obrasce generisane unutar JavaScriptovog ekosistema.

Proučićemo sledeće projektne obrasce:

- Fabrika (Factory)
- Otkrivajući konstruktor (Revealing constructor)
- Posrednik (Proxy)
- Dekorater (Decorator)
- Adapter
- Strategija (Startegy)
- Stanje (Sate)
- Šablon (Template)
- Midlver (Middleware)
- Komanda (Command)

Ovo poglavlje pretpostavlja da čitalac barem donekle zna kako deluje nasleđivanje u JavaScriptu. Imajte u vidu i da ćemo u celom ovom poglavlju obrasce često opisivati pomoću generičkih i intuitivnijih dijagrama umesto standardnog UML-a, pošto mnogi dijagrami mogu imati implementaciju koja se zasniva ne samo na klasama, nego i na objektima, pa čak i na funkcijama.

Obrazac Fabrika

Naše putovanje započinjemo onim što je verovatno najjednostavniji i najuobičajeniji projektni obrazac na platformi Node.js: **factory** (fabrika).

Generički interfejs za izradu objekata

Već smo podvukli činjenicu da je u JavaScriptu funkcionalni model programiranja često bolje rešenje od čisto objektno orijentisanog dizajna, zbog njegove jednostavnosti, upotrebljivosti i *malog otiska*. To je posebno tačno kada pravimo nove instance objekata. U stvari, izrada objekta pozivanjem fabrike, umesto direktne izrade novog objekta na osnovu prototipa, pomoću operatora `new` ili pozivanjem metode `Object.create()`, znatno je zgodnije i fleksiblnije rešenje u više pogleda.

Prvo i pre svega, fabrika nam omogućava da razdvojimo izradu objekta od njegove implementacije; fabrika u suštini „skriva“ izradu nove instance što nam pruža veću fleksibilnost i veći stepen kontrole nad načinom na koji ćemo to uraditi. Unutar koda fabrike, novu instancu možemo napraviti tako što ćemo iskoristiti ograde, na osnovu prototipa i operatora `new`, pozivanjem metode `Object.create()` ili čak vratiti različitu instancu u zavisnosti od određenog uslova. Korisnik fabrike ne mora ništa da zna o načinu na koji je nova instanca napravljena. Suština je u tome da, kada koristimo operator `new`, svoj kôd vezujemo za jedan konkretan način izrade objekta, dok u JavaScriptu mžemo imati znatno veću fleksibinost, a da nas ona ne košta gotovo ništa. Kao kratak primer, pogledajmo jedostavnu fabriku koja pravi objekte tipa `Image`:

```
function createImage(name) {
  return new Image(name);
}
const image = createImage('photo.jpeg');
```

Fabrika (ili fabrička funkcija) `createImage()` možda izgleda kao potpuno nepotrebna; zašto klasu `Image` ne bismo instancirali direktno, pomoću operatora `new`? Nešto nalik sledećem redu koda:

```
const image = new Image(name);
```

Kao što smo već pomenuli, upotreba operatora `new` vezuje naš kôd za jedan određen tip objekta; u prethodnom slučaju, to su objekti tipa `Image`. Umesto toga, fabrika nam pruža znatno veću fleksibilnost; zamislite da poželimo da refaktorišemo klasu `Image`, tako što je podelimo na manje klase, po jednu za svaki format slike koji podržavamo. Ako fabriku odredimo kao jedini način za izradu novih objekata tipa `Image`, možemo je napisati kao što sledi, bez narušavanja funkcionisanja bilo kog dela postojećeg koda:

```
function createImage(name) {
  if(name.match(/\.jpeg$/)) {
    return new JpegImage(name);
  } else if(name.match(/\.gif$/)) {
    return new GifImage(name);
  } else if(name.match(/\.png$/)) {
    return new PngImage(name);
  } else {
    throw new Exception('Nepodržan format');
  }
}
```

Naša fabrika nam omogućava i da ne izlažemo konstruktore objekata koje ona proizvodi i sprečava njihovo proširivanje ili menjanje (sećate se principa malog otiska?). Na prlatformi Node.js, to se može postići izvoženjem samo fabrike, a konstruktori objekata ostaju privatni.

Mehanizam koji nameće kapsuliranje

Fabrika se može iskoristiti i kao mehanizam za kapsuliranje, zahvaljujući ogradama.

Kapsuliranje (engl. *encapsulation*) je tehnika kontrolisanja pristupa internim detaljima objekta tako što se spoljašnjem kodu ne dozvoljava da direktno manipuliše njima. Interakcija s objektom se odvija isključivo kroz njegov javni interfejs, što izoluje spoljašnji kôd od izmena detalja implementacije objekta. Ta tehnika se zove i **skrivanje informacija**. Kapsuliranje je jedan od osnovnih principa objektno orijentisanog dizajna, zajedno s nasleđivanjem, polimorfizmom i apstrakcijom.

Kao što znamo, u JavaScriptu nemamo *modifikatore nivoa pristupa* (na primer, ne možemo deklarisati promenljivu kao privatnu), pa zato jedini način da obezbedimo kapsuliranje jeste pomoću opsega vidljivosti funkcija i ograda. Fabrika čini vrlo jednostavnim definisanje privatnih promenljivih; na primer, pogledajte naredni kôd:

```
function createPerson(name) {
  const privateProperties = {};

  const person = {
    setName: name => {
      if(!name) throw new Error(Osoba mora imati ime');
      privateProperties.name = name;
    },
    getName: () => {
      return privateProperties.name;
    }
  };

  person.setName(name);
  return person;
}
```

U prethodnom kodu, iskoristili smo ograde da bismo napravili dva objekta: objekat `person` (osoba) koji predstavlja javni interfejs koji fabrika vraća, i objekat `privateProperties` koji predstavlja grupu privatnih svojstava koja su nedostupna od spolja i kojima se može manipuslisati samo kroz interfejs koji izlaže objekat `person`. Na primer, u prethodnom kodu, obezbeđujemo da svojstvo `name` (ime osobe) uvek ima određenu vrednost; to ne bi bilo moguće kada bi `name` bilo samo svojstvo objekta `person`.

Fabrike su samo jedna od tehnika koje imamo za definisanje privatnih članova objekata; u stvari, drugi mogući pristupi su sledeći:
Definisanje privatnih promenljivih u konstruktoru (kao što preporučuje Douglas Crockford: `http://javascript.crockford.com/private.html`).
Upotreba konvencija, na primer, ime svojstva počinje znakom podvlaka "_" ili dolar "$" (međutim,to tehnički ne sprečava pristup članu objekta od spolja)
Upotreba ES2015 objekata tipa `WeakMap`: (`http://fitzgeraldnick.com/weblog/53/`).
Vrlo detaljan tekst na tu temu objavila je zajednica Mozilla: `https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Contribut or_s_Guide/Private_Properties`.

Izrada jednostavnog analizatora koda

Sada ćemo razmotriti kompletan pimer upotrebe fabrike. Napravićemo jednostavan *analizator koda*, što je objekat sa sledećim osobinama:

- Metoda `start()` započinje sesiju analize
- Metoda `end()` završava sesiju i prikazuje na konzoli koliko je trajalo izvršavanje koda

Prvo ćemo napraviti datoteku `profiler.js`, koja ima sledeći sadržaj:

```
class Profiler {
  constructor(label) {
    this.label = label;
    this.lastTime = null;
  }

  start() {
    this.lastTime = process.hrtime();
  }

  end() {
    const diff = process.hrtime(this.lastTime);
    console.log(
      `Izvršavanje "${this.label}" je trajalo ${diff[0]} sekundi i
        ${diff[1]} nanosekundi.`
    );
  }
}
```

U prethodnoj klasi nema ničeg posebnog; samo smo iskoristili standardni tajmer visoke rezolucije da bismo zabeležili vreme pozivanja metode `start()`, a zatim smo izračunali proteklo vreme u metodi `end()` i prikazali rezultat na konzoli.

Ako jedan takav analizator upotrebimo u stvarnoj aplikaciji da bismo saznali vreme izvršavanja pojedinih rutina, lako možemo zamisliti ogromnu količinu podataka koju bismo poslali na standardni izlaz, naročito u produkcijskom okruženju. Ono što bismo mogli da uradimo jeste da te analitičke podatke preusmerimo na drugo odredište, na primer, u bazu podataka, ili kao drugu mogućnost, da analizator potpuno isključimo ako aplikacija radi u produkcijskom režimu. Jasno je da ako objekat `Profiler` instanciramo direktno pomoću operatora `new`, u klijentskom kodu ili u samom objektu `Profiler` bila bi nam potrebna određena dodatna logika koja bi birala različite režime rada. Umesto toga, možemo upotrebiti fabriku da bismo apstrahovali izradu objekta `Profiler`, tako da, u zavisnosti od toga da li aplikacija radi u produkcijskom ili razvojnom režimu, vraća potpuno funkcionalan objekat `Profiler`, ili samo „simuliran“ objekat sa istim interfejsom, ali s praznim metodama koje ne rade ništa. To ćemo uraditi u modulu `profiler.js`, koji će umesto konstruktora objekta `Profiler`, izvoziti samo funkciju, našu fabriku. Kôd izgleda ovako:

```
module.exports = function(label) {
  if(process.env.NODE_ENV === 'development') {
    return new Profiler(label);                          //[1]
  } else if(process.env.NODE_ENV === 'production') {
    return {                                             //[2]
      start: function() {},
      end: function() {}
    }
  } else {
    throw new Error('Morate zadati NODE_ENV');
  }
};
```

Fabrika koju smo napravili apstrahuje izradu objekta `Profiler` od njegove implementacije:

- Ako aplikacija radi u razvojnom režimu (development), vraćamo nov, potpuno funkcionalan objekat `Profiler`.
- Ako aplikacija radi u produkcijskom režimu (production), vraćamo simuliran objekat u kojem su metode `start()` i `stop()` prazne funkcije.

Zgodna osobina koju treba istaći jeste to da, zahvaljujući JavaScriptovom dinamičkom određivanju tipa, mogli smo da u jednom slučaju vratimo objekat instanciran pomoću operatora `new`, odnosno jednostavan objektni literal u drugom slučaju (to je takođe poznato kao **duck typing** /pačje određivanje tipa/ `https://en.wikipedia.org/wiki/Duck_typing`). Naša fabrika savršeno radi svoj posao; unutar naše fabričke funkcije, objekte možemo konstruisati na koji god način želimo, ili možemo izvršavati dodatne inicijalizacione korake ili vraćati različit tip objekta u zavisnosti od određenih uslova, kao i sve to zajedno, a da pri tome korisnika objekta izolujemo od svih tih detalja. Moć ovog obrasca je lako razumljiva.

Sada se možemo igrati s našim analizatorom; to je moguć slučaj upotrebe fabrike koju smo ranije napravili:

```
const profiler = require('./profiler');

function getRandomArray(len) {
  const p = profiler('Generisanje niza s ' + len + ' elemenata');
  p.start();
  const arr = [];
  for(let i = 0; i < len; i++) {
    arr.push(Math.random());
  }
  p.end();
}

getRandomArray(1e6);
console.log('Kraj');
```

Promenljiva `p` sadrži instancu našeg objekta `Profiler`, ali ne znamo kako je ona napravljena, ni šta je njegova implementacija na ovom mestu u kodu.

Ako prethodni kôd umetnemo u datoteku `profilerTest.js`, lako možemo ispitati ove pretpostavke. Da biste ispitali program s uključenom analizom, zadajte sledeću komandu:

```
export NODE_ENV=development; node profilerTest
```

Prethodna komanda pokreće naš analizator i ispisuje rezultate analize na konzoli. Ako želimo da ispitamo simulirani oblik analizatora, zadaćemo sledeću komandu:

```
export NODE_ENV=production; node profilerTest
```

Primer koji smo upravo predstavili samo je jednostavna primena obrasca fabričke funkcije, ali jasno prikazuje prednosti razdvajanja detalja izrade jednog objekta od njegove implementacije.

Sastavljive fabričke funkcije

Pošto sada znamo kako se na platformi Node.js implementiraju fabričke funkcije, spremni smo za uvođenje novog naprednog obrasca koji odnedavno privlači sve veću pažnju u zajednici JavaScript programera. Reč je o **sastavljivim** (engl. *composable*) **fabričkim funkcijama**, koje se mogu međusobno kombinovati da bi se dobile nove poboljšane fabričke funkcije. One su naročito korisne kad treba da konstruišemo objekte koji svoja ponašanja i svojstva „nasleđuju“ iz više različitih izvora, bez obaveze da gradimo složene hijerarhije klasa.

Taj koncept možemo razjasniti pomoću jednostavnog i korisnog primera. Recimo da želimo da napravimo videoigru s likovima na ekranu koji se mogu ponašati na različite načine: mogu se *kretati* po ekranu i mogu *zamahivati sabljom* i *pucati.* I da, da bi bili upotrebljivi likovi, moraju imati određena osnovna svojstva, kao što su broj životnih poena, položaj na ekranu i ime.

Potrebno je da definišemo nekoliko vrsta likova, po jedan za svaki oblik ponašanja:

- **Character**: osnovni lik koji ima životne poene, položaj na ekranu i ime
- **Mover**: lik koji može da se kreće po ekranu
- **Slasher**: lik koji može da zamahuje sabljom
- **Shooter**: lik koji može da puca (dok ima municije!)

Bilo bi dobro kad bismo mogli da definišemo nove vrste likova, koji kombinuju postojeće oblike ponašanja. Želimo potpunu slobodu i mogućnost da pored postojećih definišemo, na primer, još i sledeće likove:

- **Runner**: lik koji se pomera po ekranu
- **Samurai**: lik koji se pomera i zamahuje sabljom
- **Sniper**: lik koji puca (ali se ne pomera)
- **Gunslinger**: lik koji se i pomera i puca
- **Western Samurai**: lik koji se pomera, zamahuje sabljom i puca

Kao što vidite, pošto želimo potpunu slobodu za kombinovanje mogućnosti svakog osnovnog lika, trebalo bi da sada bude očigledno da taj problem ne možemo modelirati na jednostavan način pomoću klasa i nasleđivanja.

Zato ćemo umesto toga upotrebiti sastavljive fabričke funkcije, a to konkretno znači da ćemo iskoristiti **specifikaciju stamp** implementiranu u obliku modula `stampit`: (`https://www.npmjs.com/package/stampit`).

Taj modul nudi intuitivni interfejs za definisanje fabričkih funkcija koje se mogu međusobno kombinovati radi izrade novih fabričkih funkcija. To u suštini znači da ćemo definisati fabričke funkcije koje će generisati objekte sa zadatim skupom svojstava i metoda, a to ćemo raditi pomoću praktičnog i zgodnog interfejsa koji ih opisuje.

Pogledajmo koliko lako možemo definisati osnovne tipove za našu igricu. Počećemo od osnovnog tipa lika:

```
const stampit = require('stampit');

const character = stampit().
  props({
    name: 'anonymous',
    lifePoints: 100,
    x: 0,
    y: 0
});
```

U prethodnom odlomku koda, definisali smo fabričku funkciju `character`, koja omogućava izradu novih instanci osnovnih likova. Svaki lik će imati sledeća svojstva: `name`, `lifePoints`, `x` i `y`, a njihove podrazumevane vrednosti biće `anonymous`, `100`, `0` i `0`. Metoda `props` modula `stampit` omogućava definisanje tih svojstava. Da bismo iskoristili tu fabričku funkciju, možemo uraditi nešto nalik sledećem:

```
const c = character();
c.name = 'John';
c.lifePoints = 10;
console.log(c); // { name: 'John', lifePoints: 10, x:0, y:0 }
```

Definišimo sada našu fabričku funkciju `mover`:

```
const mover = stampit()
  .methods({
    move(xIncr, yIncr) {
      this.x += xIncr;
      this.y += yIncr;
      console.log(`${this.name} se pomerio na [${this.x}, ${this.y}]`);
    }
  });
```

U ovom slučaju, pomoću funkcije `methods` modula `stampit` deklarišemo sve metode koje će imati objekti napravljeni pomoću fabričke funkcije `mover`. U definiciji našeg lika `Mover`, imamo funkciju `move` koja povećava koordinate `x` i `y` instance. Obratite pažnju na to kako svojstvima instance pristupamo unutar metode pomoću rezervisane reči `this`.

Pošto sada razumemo osnovne koncepte, lako možemo dodati definicije fabričkih funkcija za tipove `Slasher` i `Shooter`:

```
const slasher = stampit()
  .methods({
    slash(direction) {
      console.log(`${this.name} zamahuje ${direction}`);
    }
  });

  const shooter = stampit()
    .props({
      bullets: 6
    })
    .methods({
      shoot(direction) {
        if (this.bullets > 0) {
          --this.bullets;
            console.log(`${this.name} puca na ${direction}`);
        }
      }
    });
```

Obratite pažnju na to kako smo za definisanje naše fabričke funkcije `shooter` upotrebili i funkciju `props` i funkciju `methods`.

U redu, pošto smo sada definisali sve naše osnovne tipove, spremni smo za njihovo sastavljanje da bismo dobili nove moćne i izražajne fabričke funkcije:

```
const runner = stampit.compose(character, mover);
const samurai = stampit.compose(character, mover, slasher);
const sniper = stampit.compose(character, shooter);
const gunslinger = stampit.compose(character, mover, shooter);
const westernSamurai = stampit.compose(gunslinger, samurai);
```

Metoda `stampit.compose` definiše novu sastavljivu fabričku funkciju čiji rezultat je objekat s metodama i svojstvima sastavljivih fabričkih funkcija. Kao što i sami vidite, to je moćan mehanizam koji nam pruža veliku slobodu i omogućava da razmišljamo o ponašanjima, a ne o klasama.

Da bismo zokružili naš primer, sada ćemo instancirati i koristiti nov lik tipa `westernSamurai`.

```
const gojiro = westernSamurai();
gojiro.name = 'Yojimbo';
gojiro.move(1,0);
gojiro.slash('nalevo');
gojiro.shoot('desno');
```

Ovo će proizvesti sledeće:

```
Yojimbo se pomerio na [1, 0]
Yojimbo zamahuje ulevo
Yojimbo puca desno
```

Više informacija o specifikaciji `stamp` i ideja na kojima se ona zasniva naći ćete u prilogu čiji je autor Eric Elliot, koji je i izvorni autor te specifikacije: `https://medium.com/javascript-scene/introducing-the-stamp-specification-77f8911c2fee`.

U praksi

Kao što smo već rekli, fabrike su vrlo popularne na platformi Node.js. Mnogi paketi omogućavaju izradu novih instanci isključivo pomoću fabrika; kao primere možemo navesti sledeće:

- `Dnode` (`https://npmjs.org/package/dnode`): Ovo je **RPC** (**Remote Procedure Call** – daljinsko pozivanje procedura) sistem za Node.js. Ako pogledamo njegov izvorni kôd, videćemo da je celokupna logika implementirana u obliku klase čije je ime `D`; međutim, ona nije dostupna od spolja jer jedini izloženi interfejs je fabrika, koja nam omogućava da pravimo nove instance te klase. Izvorni kôd možete pogledati na: `https://github.com/substack/dnode/blob/34d1c9aa9696f13bdf8fb99d9d039367ad873f90/index.js#L7-9`.
- `Restify` (`https://npmjs.org/package/restify`): Ovo je frejmvork za izradu REST API-ja koji omogućava izradu novih instanci servera pomoću fabrike `restify.createServer()`, koja interno pravi novu instancu klase `Server` (koja nije dostupna). Izvorni kôd možete pogledati na: `https://github.com/mcavage/node-restify/blob/5f31e2334b38361ac7ac1a5e5d852b7206ef7d94/lib/index.js#L91-116`.

Drugi moduli izlažu i klasu i fabriku, ali u dokumentaciji preporučuju fabriku kao glavni—ili kao najpogodniji— način za izradu novih instanci; deo primera su sledeći:

- `http-proxy` (`https://npmjs.org/package/http-proxy`): Ovo je programabilna biblioteka za izradu posredničkih servera, gde se nove instance prave pomoću funkcije `httpProxy.createProxyServer(options)`
- Osnovni Node.js HTTP server: Ovde se nove instance prave uglavnom pomoću metode `http.createServer()`, mada je to zapravo samo prečica za `new http.Server()`
- `bunyan` (`https://npmjs.org/package/bunyan`): Ovo je popularna biblioteka za logovanje podataka; i njenoj datoteci `readme` file autori predlažu fabričku funkciju, `bunyan.createLogger()`, kao glavni način za izradu novih instanci, uprkos tome što bi to bilo ekvivalentno izvršavanju naredbe `new bunyan()`

Neki drugi moduli stavljaju na raspolaganje fabriku koja omogućava izradu drugih vrsta komponenata. Popularni primeri su paketi `through2` i `from2` (videli smo ih u poglavlju 5), koji nam omogućavaju da pojednostavimo izradu novih tokova pomoću fabričkih funkcija, što nas oslobađa od obaveze eksplicitne upotrebe nasleđivanja i operatora `new`.

I najzad, da biste videli nekoliko paketa koji interno koriste specifikaciju `stamp` i sastavljive fabričke funkcije, pogledajte na `react-stampit`(`https://www.npmjs.com/package/react-stampit`), gde je moć sastavljivih fabričkih funkcija prebačena u korisnički deo, što nam omogućava da lako sastavljamo ponašanja osnovnih komponenata; drugi primer je `remitter` (`https://www.npmjs.com/package/remitter`), pub/sub modul čija osnova je Redis.

Otkrivajući konstruktor

Obrazac otkrivajućeg konstruktora (engl. *revelaing construktor*) je relativno nov obrazac koji privlači sve više pažnje u zajednici Node.js i JavaScript programera, posebno zato što se koristi u nekim osnovnim bibliotekama kao što je `Promise`.

Taj obrazac smo implicitno videli u poglavlju 4, kada smo razmatrali obećanja, ali vratićemo se ponovo na njega i analizirati konstruktor objekta `Promise` da bismo ga detaljnije razmotrili:

```
const promise = new Promise(function (resolve, reject) {
  // ...
});
```

Kao što vidite, `Promise` prihvata funkciju kao argument svog konstruktora, a ta funkcija se zove **izvršna funkcija** (engl. *executor function*). Tu funkciju poziva interna implementacija konstruktora objekta `Promise` a svrha joj je da kodu konstruktora omogući manipulisanje samo ograničenim delom internog stanja obećanja u toku konstruisanja. Drugim rečima, ona služi kao mehanizam za izlaganje funkcija `resolve` i `reject` tako da se one mogu pozivati za menjanje internog stanja objekta.

Prednost je u tome što samo kôd konstruktora ima pristup funkcijama `resolve` i `reject`, a nakon konstruisanja objekta `promise`, on se može bezbedno prosleđivati dalje; nijedan drugi kôd neće moći da pozove metodu `reject` ili `resolve` i tako izmeni interno stanje obećanja.

To je razlog zbog kojeg je Domenic Denicola u jednom od svojig blogova nazavao ovaj obrazac „revealing constructor“.

Ceo Domenicov tekst je izuzetno zanimljiv jer analizira još i istorijske korene tog obrasca i poredi neke njegove aspekte s obrascem Šablon koji koriste node tokovi, kao i s drugim konstriukcijskim obrascima koji se koriste u starijim implementacijama biblioteka obećanja. Tekst možete pročitati na: `https://blog.domenic.me/the-revealing-constructor-pattern/`.

Odašiljač događaja samo za čitanje

U ovom odeljku, upotrebićemo obrazac otkrivajućeg konstruktora da bismo napravili *odašiljač događaja samo za čitanje*, što je specijalan odašiljač događaja čija metoda `emit` se ne može pozivati (osim iz funkcije koju prosledite konstruktoru).

Kôd klase `Roee` (*read-only event emitter*) smestićemo u datoteku `roee.js`:

```
const EventEmitter = require('events');

module.exports = class Roee extends EventEmitter {
  constructor (executor) {
    super();
    const emit = this.emit.bind(this);
    this.emit = undefined;
    executor(emit);
  }
};
```

U ovoj jednostavnoj klasi, proširujemo osnovnu klasu `EventEmitter` i prihvatamo funkciju `executor` kao jedini argument konstruktora.

Unutar koda konstruktora, pozivamo funkciju `super` da bismo pravilno inicijalizovali odašiljač događaja pozivanjem roditeljskog konstruktora, a zatim pravimo kopiju funkcije `emit` u lokalnu promenljivu i uklanjamo samu funkciju tako što joj dodeljujemo `undefined`.

I najzad, pozivamo funkciju `executor` kojoj kao argument prosleđujemo snimljenu kopiju metode `emit`. Ono što je važno razumeti ovde jeste da pošto metodi `emit` dodelimo `undefined`, ona se više neće moći pozivati iz drugih delova koda. Naša kopirana verzija metode `emit` je definisana kao lokalna promenljiva koja će biti prosleđena samo funkciji `executor`. Taj mehanizam nam dozvoljava da metodu `emit` pozivamo samo iz funkcije `executor`.

Sada ćemo tu našu novu klasu iskoristiti da napravimo jednostavan generator impulsa (engl. *ticker*), što je klasa koja svake sekunde generiše nov *impuls* i održava brojač prethodno generisanih *impulsa*. Sadržaj našeg novog modula `ticker.js` je sledeći:

```
const Roee = require('./roee');

const ticker = new Roee((emit) => {
  let tickCount = 0;
  setInterval(() => emit('tick', tickCount++), 1000);
});

module.exports = ticker;
```

Kao što vidite, kôd je sasvim trivijalan. Instanciramo nov `Roee` objekat kojem logiku generisanja događaja prosleđujemo u obliku izvršne funkcije. Naša izvršna funkcija dobija metodu `emit` kao argument, da bismo mogli da je pozovemo za generisanje novog impulsa svake sekunde.

Pogledajmo sada kako se se koristi modul `ticker`:

```
const ticker = require('./ticker');

ticker.on('tick', (tickCount) => console.log(tickCount, 'TICK'));
// ticker.emit('nešto', {}); <-- Ovo neće raditi
```

Objekat `ticker` koristimo na isti način kao i svaki drugi koji se zasniva na klasi `EventEmitter` (i generiše događaje) i možemo mu pridružiti proizvoljan broj osluškivača pomoću metode `on`, ali u ovom slučaju, ako pokušamo da pozovemo metodu `emit`, naš kôd će izazvati grešku `TypeError: ticker.emit is not a function`.

Mada ovaj primer lepo igra svoju ulogu ilustracije kako se može iskoristiti obrazac otkrivajućeg konstruktora, vredi napomenuti da ta funkcionalnost samo za čitanje odašiljača događaja nije potpuno „neprobojna“ i da se ona i dalje može zaobići na više načina. Na primer, u našoj `ticker` instanci i dalje možemo generisati događaj ako to uradimo direktno pomoću izvornog prototipa `emit`, kao što sledi: `require('events').prototype.emit.call(ticker, 'nekiDogađaj', {});`

U praksi

Uprkos tome što je ovaj obrazac priličlno zanimljiv i vešto smišljen, zapravo je teško naći česte primere upotrebe, osim u konstruktoru objekta `Promise`.

Vredi napomenuti da je u toku razvoja nova specifikacija za tokove koja pokušava da ovaj obrazac prihvati kao bolju alternativu tekućem obrascu Šablon za opisivanje raznih objekata koji predstavljaju tokove: `https://streams.spec.whatwg.org`.

Važno je istaći i to da smo ovaj obrazac već koristili u poglavlju 5, kada smo implementirali klasu `ParallelStream`. Ta klasa prihvata kao argument konstruktora funkciju `userTransform` function (što je izvršpna funkcija).

Mada se u tom slučaju izvršna funkcija ne poziva u vreme konstruisanja objekta, nego iz interne metode `_transform` toka, opšti koncept obrasca i dalje važi. U stvari, to rešenja nam omogućava da određene interne detalje toka (na primer, funkciju `push`) učinimo dostupnim samo konkretnoj logici za transformaciju koju želimo da zadamo u vreme konstruisanja kada pravimo novu instancu toka `ParallelStream`.

Obrazac Posrednik

Posrednik (engl. *proxy*) jeste objekat koji kontroliše pristup drugom objektu, koji se zove **subjekat** (engl. *subject*). Posrednik i subjekat imaju identičan interfejs, što nam omogućava da transparentno jedan zamenjujemo drugim; u stvari, alternativno ime za ovaj obrazac je **surogat**, zamenik (engl. *surrogate*). Posrednik presreće sve ili samo neke operacije koje treba da se izvrše nad subjektom, čime proširuje ili dopunjuje njegovo ponašanje. Naredna slika prikazuje to predstavlja u obliku dijagrama: