Autori: © Divna Krpan, Saša Mladenović, Goran Zaharija

OOP Vježbe 05
Nasljeđivanje

—

Bilješke

Nasljeđivanje

Nasljeđivanje je jedan od važnih koncepata objektno orijentiranog programiranja koji nam omogućuje definiranje klasa koje nasljeđuju, proširuju ili modificiraju ponašanje već postojeće klase. Klasa od koje se nasljeđuje naziva se **osnovna klasa** (eng. base class), a klasa koja nasljeđuje elemente osnovne klase naziva se **izvedena klasa** (eng. derived class). C# i .NET podržavaju isključivo **jednostruko nasljeđivanje**, a to u praksi znači da izvedena klasa može naslijediti elemente samo iz jedne osnovne klase. No, nasljeđivanje je tranzitivno, što znači da izvedena klasa B koja nasljeđuje od klase A, može biti osnovna klasa za novu klasu C (A ⇒ B ⇒ C).

Prilikom nasljeđivanja nije potrebno navoditi koji elementi će se naslijediti jer se autmatski nasljeđuju svi.

Obzirom na elemente jezika koje smo koristili do sad (definicija klase pomoću **class**), onda ćemo i dalje nastaviti u tom stilu. Prema tome, nasljeđivanje u JavaScriptu ćemo implementirati pomoću ključne riječi **extends**:

```
class Osoba {
  // klasa koja sadrži općenite podatke o osobi
}

class Student extends Osoba {
  // klasa koja pored informacija o osobi
  // sadrži i podatke specifične za studente
}
```

Uzmimo malo detaljnije primjer osobe:

```
class Osoba {
  constructor(im, pr, gr) {
    this.ime = im;
    this.prezime = pr;
```

```
    this.grad = gr;
  }

  ispis() {
    console.log("osoba");
    console.log(this.ime + " " + this.prezime);
  }
}
```

Osoba ima ime, prezime, grad u kojem stanuje te metodu kojim ispisujemo informaciju o kojem objektu se radi te ime i prezime. Definirat ćemo klasu *Student* koja će proširiti klasu *Osoba* s novim detaljima. Neka klasa *Student* ima dodatan podatak koji se odnosi na godinu studija te konstruktor u kojem se unosi samo ime i prezime, a grad će biti "Split".

Prilikom stvaranja klase *Student* koja nasljeđuje od *Osoba*, primijetit ćete da u trenutku definiranja konstruktora te izvedene klase, moramo omogućiti pozivanje konstruktora osnovne klase *Osoba*. Konstruktor osnovne klase pozivamo pomoću ključne riječi: **super**.

```
class Student extends Osoba {
  constructor(imeS, preS){
    super(imeS, preS, "Split");
    this.godina = 1;
  }
}
```

Definirali smo konstruktor koji prima samo ime i prezime studenta, dok će grad poslati kao parametar u konstruktor osnovne klase. Konstruktor osnovne klase se mora pozvati prije pisanja ostalih stvari u konstruktoru izvedene klase čak i u situaciji kad je konstruktor nove klase prazan, moramo pozvati konstruktor osnovne klase.

Ako pokušamo izvršiti sljedeći kôd:

```
let a = new Osoba("Pero", "Perić", "Split");
a.ispis();

let b = new Student("Ante", "Antić");
b.ispis();

console.log(a);
console.log(b);
```

Dobit ćemo rezultat:

```
osoba
Pero Perić
osoba
Ante Antić
▶ Osoba {ime: 'Pero', prezime: 'Perić', grad: 'Split'}
▶ Student {ime: 'Ante', prezime: 'Antić', grad: 'Split', godina: 1}
```

Primijetit ćemo da je objekt tipa *Student* naslijedio sve što ima *Osoba*, uključujući i metodu *ispis()* koja ispisuje ime i prezime na konzolu, dakle: atribute i ponašanje.

Međutim, ako nam je to potrebno, u klasi *Student* možemo definirati drugačiju metodu *ispis()* kao i nove metode koje *Osoba* nema.

```
class Student extends Osoba {
  constructor(imeS, preS){
    super(imeS, preS, "Split");
    this.godina = 1;
  }

  ispis(){
    console.log("Student na godini " + this.godina);
  }
}
```

Ako sad ispišemo podatke dobit ćemo:

```
osoba
Pero Perić
Student na godini 1
▶ Osoba {ime: 'Pero', prezime: 'Perić', grad: 'Split'}
▶ Student {ime: 'Ante', prezime: 'Antić', grad: 'Split', godina: 1}
```

Ovim postupkom napravili smo novu implementaciju postojeće metode *ispis()* koju smo inače naslijedili iz osnovne klase. Takav pristup naziva se ***premošćivanje*** (eng. override), a to je primjer **polimorfizma**. Metode se jednako zovu, ali im je implementacija različita. Ako bi ipak htjeli pozvati implementaciju koja je u osnovnoj klasi, potrebno je koristiti ključnu riječ **super**.

```
  ispis(){
    super.ispis();
    console.log("Student na godini " + this.godina);
```

```
  }
```

U gornjem primjeru, najprije se izvrši metoda *ispis()* osnovne klase pa onda ostatak metode izvedene klase.

Pristupanje svojstvima i metodama klase

Svojstvima i metodama u klasi pristupamo pomoću ključne riječi **this**, točke i naziva svojstva, odnosno metode. Rezervirana ključna riječ **super** predstavlja "roditeljski" konstruktor tako da metodama i svojstvima osnovne klase ipak možemo pristupiti.

Konstruktor klase je također metoda. Poziva se pomoću ključne riječi **new**. Nakon pozivanja konstruktora pomoću **new**, onda ***this*** unutar konstruktora ukazuje na instancu objekta. Treba biti oprezan s korištenjem ključne riječi **this** unutar metoda za upravljanje događajima jer se može dogoditi da ne znači ono što mislimo. Postoji više načina kako riješiti taj problem, no o tome ćemo prema potrebi projekta.

Primjena nasljeđivanja u okviru za izradu igre

Okvir za izradu igre sastoji se od više klasa koje su raspoređene u više datoteka. Pri tome se nastoji da sve datoteke budu što više neovisne, odnosno da se što manje poziva kôd iz ostalih datoteka kako bi se po potrebi mogle zamijeniti novim verzijama.

Važno!

Za izradu projekta koristite najnoviji okvir (sadržaj foldera ***otter***), a svoj kôd pišite u folderu ***jsKod***. Nemojte mijenjati postojeće klase već koristite koncepte OOP kao što su npr. nasljeđivanje i premošćivanje kako bi proširili postojeće funkcionalnosti.

U ***main.js*** datoteci stvaraju se globalni objekti:

- SENSING - interakcija s korisnikom (pritisnute tipke na tipkovnici, mišu, i sl.)
- DISPLAY - prikaz igre (crtanje, promjena veličine prostora za crtanje i sl.)
- GAME - mapa i likovi u igri (učitavanje, brisanje, i sl.)
- ENGINE - rad s petljom igre.

Također, u ***main.js*** definiramo osnovne elemente sučelja:

- *btnMaps* - učitavanje mapa koje su dodane u *index.html*,
- *btnSetupGame* - postavljanje likova i parametara igre,
- *btnStart* - pokretanje igre,
- *btnStop* - zaustavljanje igre.

U igru možemo učitati više mapa, ako postoji više razina igre.

```
<!-- mape -->
```

```
<script src="primjer.js"></script>
<script src="catDog.js"></script>
```

Najprije se moraju učitati mape, pa onda datoteke koje pripadaju okviru (numerirane su tako da znamo redoslijed). Nakon toga učitavamo vlastiti kôd kojeg smo smjestili u datoteke:

```
 <!-- vlastiti kod -->
  <script src="jsKod/kod_01-likovi.js"></script>
  <script src="jsKod/kod_02-postavke.js"></script>
  <script src="jsKod/kod_03-logikaIgre.js"></script>
```

Naravno da možemo učitati još datoteka gdje je smješten kôd koji ne pripada okviru, ali pokušajmo ga organizirati na ovaj način: razdvojimo definicije klasa (*likovi.js*), postavljanje likova i ostalih parametara igre (*postavke.js*) te logike ili interakcija svih likova u igri (*logikaIgre.js*).

Obzirom da ne želimo mijenjati postojeći okvir, koristit ćemo se konceptima OOP-a kako bi proširili ili prilagodili postojeće klase. Nastavljajući na prethodne vježbe, definiramo klasu ***Animal*** koja će naslijediti sve od klase ***Sprite***, ali će imati i neke nove stvari.

```
class Animal extends Sprite {

}
```

U klasu ćemo dodati konstruktor koji će postaviti *x* i *y* koordinate te *layer* koji pripada liku. To smo radili u metodi *postavi()* prošli put, ali nam je jasno da će te stvari trebati postaviti za svaku vrstu likova koje budemo koristili u igri.

Čim smo napisali konstruktor, moramo pozvati i konstruktor osnovne klase koji prima koordinate i dimenzije lika. Obzirom da već znamo kako su svi likovi koje smo koristili jednakih dimenzija kao i pločice (*tiles*), a to je 64 px onda ćemo to odmah postaviti u klasi *Animal*. Također se čini dobro malo smanjiti lika kako bi bio malo manji od ostalih pločica (lakše upada u rupe).

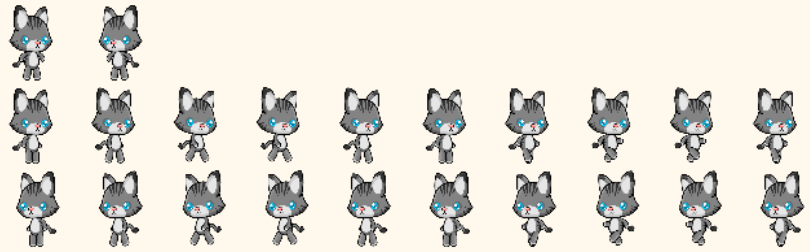
```
class Animal extends Sprite {
  constructor(x, y, layer) {

    super(x + 4, y + 4, 64 - 8, 64 - 8);

  }

}
```

Nadalje, potrebno je smjestiti animacije (*frame_sets*) u konstruktor. Teško je naći sve likove s jednakim brojem i rasporedom sličica u *tileset-u* tako da se ne može napraviti općenito za bilo kojeg lika. Npr. neki nemaju uopće animacija dok neki imaju 4, neki 10 za određene položaje. Okvir ima predviđene animacije hodanja (eng. walk) i mirovanja (eng. idle) za 4

smjera. Morate ih sve popuniti, a ako nemate za sve moguće smjerove i kretanja onda se svuda može staviti isti redni broj.

Za mačku i psa je to ovako:

```
this.frame_sets = {
     "up": [1],
     "walk-up": [1],
     "right": [1],
     "walk-right": [11, 12, 13, 14, 15, 16, 17, 18, 19],
     "down": [1],
     "walk-down": [1],
     "left": [2],
     "walk-left": [21, 22, 23, 24, 25, 26, 27, 28, 29]
   };
```

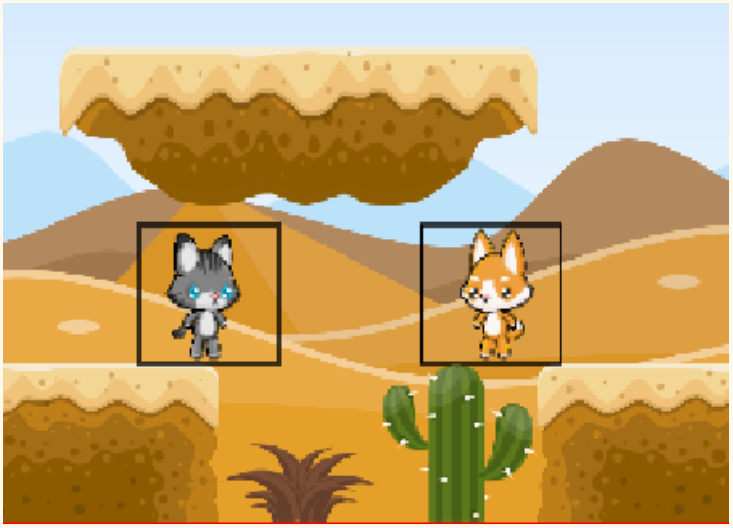


Sličica rednog broja 1 koristi se za desni pogled u stanju mirovanja, a sličica rednog broja 2 za lijevi. Ostali redni brojevi ovise o dimenzijama same mape i načinu na koji ste ih postavili na mapu.

Dodatno, u konstruktoru klase *Animal* možemo navesti neke druge stvari kao što je primjerice *okvir*.

```
    this.layer = layer;
    this.visible = true;
    this.okvir = true;
```

Prema tome možemo vidjeti kako stvarno izgleda sličica našeg lika:



Ponekad se čini da bi lik trebao "pasti" dolje jer više ne stoji na platformi, ali ima dosta praznog prostora okolo. To nam se ovdje događa jer su likovi visoki, a uski pa ne popunjavaju prostor s lijeve i desne strane. Međutim, zgodnije nam je kad su njihove sličice kvadratne kao i sve ostalo. Sad u postavljanju igre možemo samo dohvatiti *layer* koji pripada mački te ga poslati u konstruktor koji će obaviti ostalo.

```
function setupLevel() {

  let catLayer = GAME.getSpriteLayer("macka");
  let cat = new Animal(0, 0, catLayer);

}
```

Nakon toga, dodajemo *cat* u popis svih likova igre (koji se crtaju):

```
GAME.addSprite(cat);
```

Kod pokretanja igre, potrebno je:

1. Učitati sve mape klikom na *Maps*
2. Odabrati mapu koju želimo iz popisa
3. Kliknuti na *Setup game* za učitavanje postavki igre i postavljanje likova
4. Kliknuti na *Start* za pokretanje petlje igre.

Prilikom prvog pokretanja primijetit ćemo kako likovi padaju na dno prostora za crtanje. Platforme često funkcioniraju tako da likove privlači gravitacija na dno i padaju dok ne naiđu ili na platformu ili na dno ekrana. Okvir je napravljen tako da likovi ne mogu propasti van prostora za crtanje (ono što smo prošli put morali sami raditi).

Kolizije

Kad se radi o kolizijama (eng. collision), imamo dvije stvari na koje moramo paziti kod izrade platformske igre:

- Platforme po kojima se lik kreće,
- Drugi likovi s kojima je u interakciji.

Platforme mogu predstavljati čvrsto tlo po kojem može hodati ili neko opasno područje gdje može izgubiti bodove i sl. U okviru su predviđene platforme po kojima može hodati, a ne može ih proći s ni jedne strane. Osnovna ideja je: kad se lik pokuša postaviti na novu lokaciju, potrebno je provjeriti je li na toj lokaciji platforma. To možemo napraviti tako da platforme smatramo likovima te provjeravamo dodiruju li glavnog lika i s koje strane (ako je to važno). Drugi način je da pozicije platformi spremimo u posebnu mapu kolizija.

U okviru je već gotov drugi pristup s mapom kolizija. To možemo napraviti korištenjem podataka iz *layer-a* koje smo vidjeli na prvim vježbama.

Dovoljno je u *setup()* funkciji napisati naredbu:

```
GAME.activeWorldMap.setCollisions("platforme");
```

Ovdje je "platforme" naziv *layer-a* koji sadrži platforme. Polje *collision_map* sadrži niz brojeva koje generira Tiled, a u tom nizu nule predstavljaju prazan prostor, dok ostali brojevi predstavljaju točan indeks sličice (*tile*). Ako je prazno, ignorira se i lik prolazi, a ako je neki drugi broj onda neće dozvoliti prolaz.

Interakciju s drugim likovima ili objektima u igri omogućit ćemo definiranjem metode *touching()*.

Kretanje likova

Likovi u platformama se kreću pritiskanjem odgovarajuće tipke koju smo odredili ili kombinacije tipki (ako skače). Okvir je napravljen tako da kod svakog pritiska tipke, liku raste brzina u odgovarajućem smjeru. Primjer kretanja u lijevo:

```
moveLeft() {
   this.direction = 270;
   this.velocity_x -= 0.5;
 }
```

Metoda ***updatePosition()*** svaki ciklus osvježavanja igre brine o promjeni položaja lika. Zapravo se lik ponaša slično kao kad pritisnemo papučicu gasa u automobilu: ide dok držimo pritisnuto, a usporava i zaustavlja se kad pustimo tipku. Za usporavanje je zaduženo *trenje* (eng. friction) koje je uključeno u okvir.

Lik se pomiče po ***x*** odgovarajućoj brzinom:

```
this.x += this.velocity_x;
```

Ako je ta brzina negativna ići će lijevo, a u protivnom, ako je veća od nule ide desno.

```
this.velocity_x *= friction;
```

Trenje je vrijednost koja je manja od 1 tako da će ga s vremenom zaustaviti ako se više ne pritiska tipka za kretanje.

Obzirom na tip igre, nije potrebno pomicati lika prema dolje već on sam pada (ovisno o gravitaciji) dok ne naiđe na čvrsto tlo (platforma ili dno ekrana). Kad dotakne čvrsto tlo (padne) onda brzina po y postaje 0.

- ➢ Zadatak 1.
  - Napisati klasu ***Cat*** koja nasljeđuje od klase ***Animal***.
  - Neka klasa *Cat* ima konstruktor koji prima samo *layer* koji pripada mački.
  - Početne koordinate mačke su uvijek (0, 0) pa ćemo ih postaviti u konstruktoru klase *Cat*.
  - Iskoristimo konstruktor nove klase i za postavljanje zadanog smjera mačke: desno tj. 90 stupnjeva.
- ➢ Zadatak 2.
  - Napisati metodu ***jump()*** koja omogućuje skakanje mačke.
  - Metoda prima broj točkica koliko mačka može skočiti.
  - Ako se ne unese bit će 20.
  - Kad se pritisne tipka **w** mačka će skočiti prema gore.

Napomena: često se zna koristiti razmak (eng. *space*) za skakanje. Tu morate paziti jer ponekad na početku neki od zadnjih kliknutih "botuna" ostane u fokusu pa zapravo pristinuti razmak zna pokrenuti botun umjesto skoka. Potrebno je samo kliknuti negdje na stranici da se makne "fokus". Možda je bolje izbjeći razmak.

Globalni objekt **SENSING** ima informaciju o mišu: *SENSING.mouse*. Definiran je na isti način kao što smo to napravili s tipkama. Kako ne bi stalno morali voditi računa o nekim događajima, ono što nam je potrebno nastojimo grupirati u klasi *Sensing*, a podatke čuvamo u globalnom objektu *SENSING* kako bi nam bili dostupni tijekom cijele igre.

- ➢ Zadatak 3.
  - Klasa ***Sprite*** ima metodu ***clicked()*** koja provjerava je li lik kliknut.
  - Metoda *clicked()* mora dobiti informacije o mišu kako bi mogla provjeriti jesu li koordinate strelice miša u trenutku klikanja unutar lika.
  - Ako je pas kliknut, treba ga sakriti. Klasa *Sprite* ima i svojstvo *visible* koje se može postaviti na **true**/**false**.
- ➢ Zadatak 4.
  - Napisati metodu ***touching()*** u klasi ***Animal***.
  - Metoda *touching()* mora primati parametar tipa *Sprite*, a vraća **true** ako se trenutni lik i lik koji dolazi kao parametar dodiruju.
  - Poseban slučaj je kad lik kojeg prima nije vidljiv. U tom slučaju neće registrirati dodir već vraća **false**.
  - Treba li provjeriti i za lika koji koristi metodu *touching()*?

- Ako mačka tijekom igre dotakne psa, treba ispisati poruku na konzolu: “Diram psa!”.

Napomena: dobro je uključiti okvire za likove kako bi vidjeli dodiruju li se ili ne. Također, sakrijte psa klikom miša i provjerite hoće li ga mačka detektirati.

Pokušajte sakriti samo mačku pa vidite što će se dogoditi.

_