Sissejuhatus

Programmeerimine on keeruline. Seda tunnistavad ka kõige kogenumad programmeerijad. Aga sellest keerulisem on programmeerimise õppimine. Kus kohast alusta? Mis keelt ma peaksin õppima? Need on küsimused millele iga algaja on pidanud vastuse leidma.

Selle juhendi eesmärk, on anda sinule, kui algajale, tõuge programmeerimise maailma. Järgides seda juhendit õpid sa programmeerimise põhiprintsiipe ja kui kõik edukalt on läbitud, on meil valmis saanud 2 mängu mis töötavad sinu veebibrauseris ning mida saad jagad oma sõprade või perega.

TypeScript

Selleks, et brauseris nähtavaid mänge kirjutada, meil suuri programmeerimiskeele valikuid ei ole. Aga see ei ole halb asi, sest see eest on meil just selleks spetsialiseeritud keel olemas nimega TypeScript.

Võibolla oled eelnevalt juba kuulnud terminit JavaScript või oled sellega eelnevalt kokku puutunud. TypeScript on selle edasi arendus ja modernses arendus maailmas kiirelt asendamas JavaScripti. TypeScript annab meile, arendajatele, kindlustunde, mis JavaScriptis puudub ning parandab "arendus kogemust" (ing. keeles *"developer experience"*) andes meile kasutusele andmetüübid, mis JavaScriptis puuduvad täielikult.

Erinevalt JavaScriptist, tuleb TypeScripti kompileerida. See tähendab, et kood mis me kirjutame, tuleb muuta brauserile loetavaks, sest modernsed brauserid ei toeta TypeScripti otseselt. TypeScript kompileerib aga JavaScriptiks, millest brauserid hästi aru saavad.

Node.js

TypeScripti kompileerimiseks on meil omakorda vaja tööriista nimega Node.js (siit edasi kutsume seda lihtsalt "Node"). Node võimaldab kasutada nii JavaScript kui TypeScripti enamates kohtades, kui ainult brauser ja on üks enim kasutatuid

paketihaldureid programmeerimis maailmas. Üks pakettidest mida läbi Node on võimalik saada on ka TypeScript.

Selleks, et kompileerida meie TypeScript koodi saame kirjutada Nodes väikesed skriptid, mida jooksutades meie kood muudetakse brauserile loetavaks. Ka saab need skriptid muuta automaatseks, ehk nii kui meie kood muutub kompileeritakse kood. Sellest, aga täpsemalt hiljem.

Käsurida

Käsurida on tööriist milleta mitte ükski programmeerija hakkama ei saa. See on absoluutselt vajalik tööriist mida kasutatakse igapäevaselt. See võib esmapilgul tunduda hirmutav ja võõras, aga kui harjud seda kasutama muudab see igapäevased tegevused kiiremaks ja suurendab sinu arvutikasutamise võimekust.

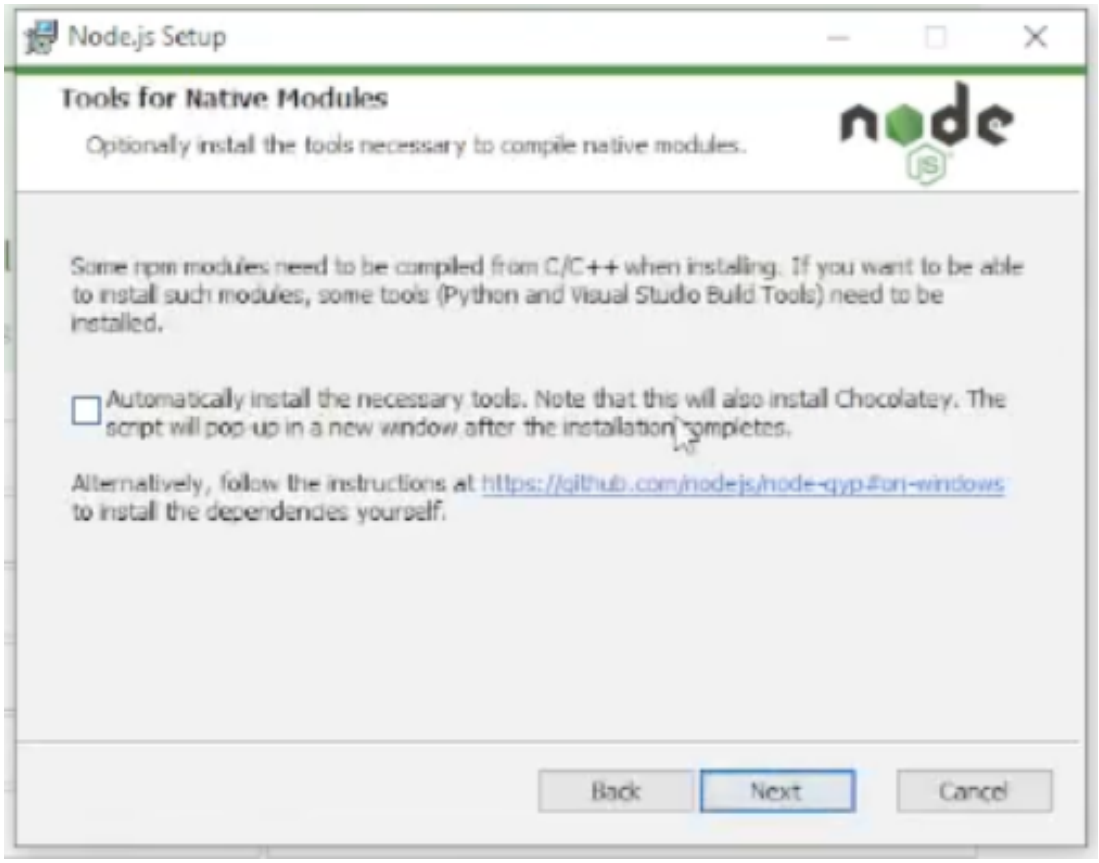
Paigaldus

Selleks, et saaksime asuda asja kallale, peame eelnevalt mõned asjad arvutisse installeerima. Juhend on olemas nii Windows kui ka MacOS masinatele.

Windows

Windowsil Node.JS installimiseks peame minema Node.JS kodulehele, mis asub siin: https://nodejs.org/en/. Kui lehe avame on meil valida kahe versiooni vahel: LTS versioon, mis on vanem, aga rohkem testitud, ning kõige viimasem versioon, millele on kõige uuemad funktsionaalsused. Kuna meil ei ole vaja kõige uuemaid funktsionaalsuseid, siis laeme alla LTS versiooni, mis juhendi koostamise hetkel on 16.14.2.

Käivitame alla laetud installeri ning läbime kõik selle sammud. Küll tuleb tähele panna, et me ei installiks ekstra tööriistu, mida meil vaja ei ole. Seda on kerge vahele jätta, aga võib installimisprotsessi pikendada märkimisväärselt.



Kui Node.JS on installitud, kontrollime igaks juhuks üle, et install oli edukas. Avame käsurea kasutades *Command Prompt* rakendust. Sinna kirjutame ühe lühikese käskluse, mis tagastab meile installitud Node.JS versiooni.

```
node -v
```

Peaksime nägema käsureal midagi sellist.

```
C:\Users\coppe>node -v
v16.14.2
```

Ka kontrollime, kas npm, millega me installime TypeScripti, sai edukalt installitud. Selle jaoks me jällegi küsime selle versiooni kasutades käsurea käsklust.

```
npm -v
```

Selle käskluse vastus peaks olema sarnane Node versiooni kontrollimisele.

Nüüd saame liikuda TypeScripti installimise juurde. Selleks kasutame npm-i. Samas käsureal sisestame käskluse, mis installib globaalselt meile nii TypeScripti kui ka *ts-*

node tööriista, mis lubab meil TypeScripti ilma kompileerimata jooksutada. Selle jaoks sisestame sellise käskluse:

```
npm install -g typescript ts-node
```

Kui oled siia kohta jõudnud, siis saame alustada koodi kirjutamisega!

Kui tekkis raskusi installimise juures, siis selle sammu jaoks on olemas ka video materjal: https://www.youtube.com/watch?v=P9QOeXwI5cQ

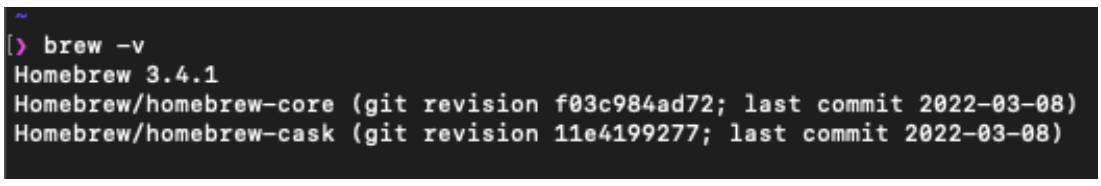
MacOS

Selleks, et MacOSi peal installida nii Node kui ka TypeScript peame eelnevalt installima Homebrew. Homebrew on MacOS-i paketihaldur, mille abil saab installida tuhandeid erinevaid pakette oma arvutisse ning programmeerimis maailmas tuleb see väga kasuks. Selleks, et Homebrewd installeerida peame minema Homebrew kodulehele: https://brew.sh/. Seal leiame suurelt kirjutatult “Install Homebrew“ ning selle all on kirjas üks käsurea käsklus.

Selleks, et seda koodi rida kasutada avame Terminali rakenduse. Kleebime selle rea siia ning käivitame selle. Ootame kuni installprotsess jõuab lõpuni ja siis kontrollime, kas homebrew on edukalt installitud kasutades brew versiooni kontrollimise käsklust.

```
brew -v
```

Peaksime terminalis nägema midagi sellist.



Nüüd kui Homebrew installitud installime Node. Ka see käib läbi terminali rakenduse, kasutades brew install käsklust.

```
brew install node
```

Kui installeerimine on lõppenud, saame kontrollida, kas Node sai installitud korrektselt kasutades käsklust, mis kontrollib Node versiooni.

```
node -v
```

Peaksime nägema midagi sellist.

Nodega tuleb ka kaasa Node Package Manager ehk NPM. Just läbi NPMi installime me TypeScripti ja hiljem mõned paketid. Kontrollime et ka see sai installitud edukalt.

```
npm -v
```

Kui Node ja NPM edukalt installitud, saame edasi liikuda TypeScripti installimise juurde. TypeScripti installimine käib sarnaselt Nodele, aga seekord kasutame Homebrew asemel NPMi.

```
npm install -g typescript ts-node
```

Install skriptis -g tähendab, et pakett mille installime, installitakse globaalselt ning see on kättesaadav igal pool. Peale seda on kaks paketti mis me soovime installida. TypeScript on meie TypeScripti keele pakett ja ts-node on lisatööriist, mis aitab meil TypeScript faile käivitada mugavamalt.

Kui oled siia kohta jõudnud, siis saame alustada koodi kirjutamisega!

Esimesed koodiread

Enne kui jõuame mängude tegemiseni peame õppima ära kõige elementaarsemad programmeerimise põhitõed. Kust parem alustada kui klassikalise Hello World failiga.

Hello world!

Teeme endale arvutis uue kausta (nimeks võiks olla nt. *basics*) ning avame selle kausta oma valitud tekstiredaktoris. Kui kaust on avatud, saame luua oma esimese TypeScript faili. TypeScripti failid kasutavad faililaiendit *.ts* seega loome faili *hello.ts* ning avame selle.

Hello world'i fail traditsiooniliselt teeb vaid ühte asja: prindib käsureale "Hello world!". Selleks, et seda kuvada, peame me vaid ühe rea kirjutama.

```
console.log("Hello world!")
```

Nüüd avame oma käsurea. Ole kindel, et oled oma käsureal navigeeritud õiges kaustas. Kui avasid käsurea tekstiredaktori sees, siis peaks kõik juba korras olema. Käsureas saame nüüd meie *hello.ts* faili käima panna ja kui kõik on korrektne, prinditakse meile välja "Hello world!".

```
ts-node hello.ts
```

Peaaegu iga programmeerija on alustanud just selle programmiga ja nüüd kui meie oleme ka oma *hello world* programmi valmis saanud, saame liikuda edasi huvitavamate asjade juurde.

Muutujad ja andmetüübid

Programmeerimise üks põhilisemaid elemente on muutujad. Muutuja on sümbol, mis hoiab mingit kindlalt väärtust mingil kindlal mäluaadressil. Näiteks, paneme valge karbi sisse punase palli ja musta karbi sisse sinise palli. Need pallid on nüüd nendes karpides nii kaua, kuni me otsustame nendega midagi teha. Kui võtame valgest karbist palli ära, siis on karp tühi. Sama moodi, saavad ka muutujad olla tühjad. Kui üritame nüüd punase palli panna musta karpi, kus juba üks pall on ees, näeme et seda me teha ei saa, sest karp on mõeldud vaid ühe palli jaoks. Muutujatel kutsume seda andmetüübiks. TypeScriptis saame määrata muutujatele andmetüüpe ning sellega tagada endale kindluse, et õiged andmed satuksid õigetesse kohtadesse.

Teeme uue faili *muutujad.ts* ja avame selle. Muutujaid saame luua kui kirjutame alguses kas *let* või *const* ja siis anname oma muutujale kirjeldava nimetuse. Võtit *let*

kasutame juhul, kui tahame oma muutujat veel muuta peale tema loomist, *const* aga siis, kui muutujal on ainult üks väärtus, mis kunagi ei muutu. Peale muutuja nimetust, paneme kirja meie muutuja andmetüübi. Andmetüüp eraldatakse kooloniga.

TypeScriptis andmetüüpe on palju ja kui soovime saame neid lausa juurde luua. Aga meile kõige tähtsamad on *number, string* ja *boolean*.

Number tüüp võib olla nii negatiivne kui positiivne number ja võib olla nii täisarv kui koma kohtadega.

String tüüp võib olla mõni sõna, täht või isegi number, aga ümber peavad olema jutumärgid.

Boolean on kõige lihtsam andmetüüp keerukuse poolest. Tema väärtused saavad olla *true* või *false*.

Need kolm on põhilised andmetüübid, mida meie kasutama hakkame. Loome siis oma failis uue muutuja nimega *count*, andmetüübiks number ja algväärtuseks 1.

```
let count: number = 1;
```

Selleks, et ikka kindel olla, kas meie muutujal on see väärtus, mis me talle lisasime, prindime ta välja käsureale kasutades sama käsklust, mida kasutasime oma “Hello world!“ näites. Console.log sisendiks, saame anda ükskõik mis muutuja (isegi mitu muutujat, kui nad komaga eraldada) ning nad prinditakse meile kas siis brauseri konsooli või käsureale nagu meil hetkel.

```
let count: number = 1;
console.log(count);
```

Kasutame jälle ts-node käsklust selleks, et käivitada oma programm ja näeme, et meie muutuja väärtus on tõesti 1.

Nüüd kui oleme kindlad, et meie muutuja hoiab väärtust võiks proovida seda muuta. Liidame meie muutujale *count* juurde kahe ja prindime selle välja.

```
let count: number = 1;
console.log(count);
```

```
count = count + 2;
console.log(count);
```

Siin näeme, et kui muutujale uue väärtuse anname, ei pea enam teistkorda *let* võtit kasutama. Käivitame jälle oma programmi ja teeme kindlaks, et muutus toimus reaalsuses ka. Näeme et meile prinditakse käsureale alguses 1 ja siis peale seda 3. Oleme edukalt loonud uue muutuja, andnud talle algväärtuse ja seda algväärtust muutnud.

Muutujaid saab meil olla mitu, ainus limiit nende arvule on riistvara võimekus. Ka on võimalik muutujaid ka omavahel kasutada. Selle demonstreerimiseks loome uue muutuja, nt *count2* ja anname talle väärtuseks kolme ning samamoodi nagu eelnevalt, lisame oma vanale muutujale count nüüd mitte numbri otse, vaid hoopis meie uue muutuja count2.

```
let count2: number = 3;
count = count + count2;
console.log(count)
```

Nagu näeme, lisati edukalt muutujale *count* kolm ning selle väärtus on nüüd viis. Kuid kui soovime palju selliseid tehteid teha oma programmis ei ole iga kord nende tehete välja kirjutamine just kõige parem tava. Koodi kordamisest hoidumine on üks tähtsamaid programmeerimise põhiprintsiipe ning sellest reeglist kinni pidamiseks saame kasutada me funktsioone.

ÜLESANDED:

- Luua ise mõned muutujad ning printige need käsureale
- Luua tehted kasutades enda loodud muutujaid ning nende tulemus käsureale printida

Funktsioonid

Funktsioonid on programmeerimises üks kasulikemaid tööriistu peale muutujate. Nad on iseseisvad koodi jupid, mis täidavad konkreetseid ülesandeid. Funktsioonidel on tavaliselt sisendiks mingid andmed, mida funktsiooni sees töödeldakse ja tulemus tagastatakse. Funktsioone on võimalik välja kutsuda mitmeid kordi, mis teevad nad eriti kasulikuks kohtades, kus esineb palju korduvat funktsionaalsust.

Proovime siis ka kohe järele. Meie eesmärk on kirjutada funktsioon mis võtab sisendiks kaks numbrit ja tagastab nende summa. TypeScriptis tähistatakse funktsioone *function* võtmega. Sellele järgneb funktsiooni nimetus ning siis sulgude sees funktsiooni parameetrid. Kui funktsioonil on rohkem kui üks parameeter, siis peavad nad olema komaga eraldatud. Parameetritel peavad olema ka andmetüübid tähistatud. Kõige lõpus on loogelised sulud, mille sees asub meie funktsiooni funktsionaalsus.

```
function someFunctionName(a: number) {
  // Some code
}
```

Funktsioonide nimed peaksid alati kirjeldama tema funktsionaalsust. Kui näiteks meie funktsioon käivitab auto, siis ta nimi võiks olla *startCar.* Seega ei pea me väga kaua nuputama, mida oma esimese funktsiooni nimeks peaksime panema. Meie funktsioon vajab ka kahte parameetrit, nende nimed võiksid olla *a* ja *b*, ning kuna soovime numbreid kokku liita, siis loomulikult nende andmetüüpideks on *number.*

```
function addTwoNumbers(a: number, b: number) {

}
```

Juba hakkab midagi looma. Me saame oma funktsiooni küll juba välja kutsuda koodis, aga hetkel see midagi ei tee. Peame lisama ka nn. funktsiooni keha, mis asub loogeliste sulgude vahel. Oleme juba kogenud muutujate loomisel, seega loome uue muutuja *c,* ning tema väärtuseks anname muutujate *a* ja *b* summa. Nüüd selleks et seda väärtust tagastada saame kasutada *return* käsklust. Selleks teeme funktsiooni sees uue rea ning kirjutame:

```
return c;
```

Ja funktsioon ongi valmis. Lisaks on hea lisada funktsioonidel nende tagastatav andmetüüp. See käib kooloniga peale funktsiooni parameetreid. Meie kasutusjuhul hetkel on tagastatav andmetüüp *number*. Kogu lahendus peaks välja nägema selline.

```
function addTwoNumbers(a: number, b: number): number {
  let c = a + b;
  return c;
}
```

Nüüd kui funktsioon on valmis kirjutatud saame seda ka kasutada oma koodis. Funktsioone kutsutakse koodis välja nende nime järgi. Sellele järgneb sulgude sees funktsiooni parameetrid, aga seekord mitte parameetrite kirjeldus vaid nende väärtused. Ehk kui meie funktsiooni parameetri andmetüübiks on number, siis anname funktsiooni parameetriks mõne numbri või mõne muutuja, mille väärtus on number.

Kui meie funktsioon tagastab väärtuse, ja meie *addTwoNumbers* funktsioon seda ka teeb, siis tuleb see väärtus ka kuhugi salvestada. Selleks saame jällegi kasutada muutujaid. Me võime luua mõne täiesti uue muutuja või kasutada olemasolevaid muutujaid nende väärtuste salvestamiseks. Proovime seda järele oma *count* muutujaga. Me teame et selle väärtus on hetkel kolm, aga soovime sellele kaks juurde liita. Sarnaselt nagu eelnevalt anname talle uue väärtuse, aga seekord see väärtus tuleb meie uuest funktsioonist.

```
count = addTwoNumbers(count, 2);
console.log(count);
```

Nagu näeme, meie funktsioon töötab ja meil on nüüd tööriist, mida saame kergelt taaskasutada. Aga kui soovime ühte muutujat nt. kümme korda suurendada, oleks ühe ja sama rea kümme korda uuesti kirjutamine siiski kordamine, ning nagu eelnevalt sai öeldud soovime seda mitte teha. Siin tulevad kasutusele tsüklid.

ÜLESANDED:

- Luua ise uus funktsioon, mis korrutaks kaks numbrit kokku (korrutus tehte jaoks saab kasutada * märki)
- Luua funktsioon, mis jagaks kaks numbrit (jagamis tehte jaoks saab kasutada / märki)

Tsüklid

Tsüklid on programmeerimises tihedalt kasutusel olevad elemendid. Nad lubavad meil käivitada mingeid kindlaid koodiridasid mingi kindel arv kordi järjest. Tsükleid kasutades tuleb aga ka ettevaatlik olla, sest ka mõni väike viga võib tsükli igavesti töötama jätta ning programm jookseb kokku.

TypeScriptis on kaks põhilist tsüklit *for* ja *while* tsüklid. Võtame alguses käsile *while* tsükli. *While* tsükkel töötab niikaua, kuni talle antud tingimus vastab tõele. Tingimuse kontroll toimub alati enne tsükli algust, seega kui tingimus enam ei vasta tõele, siis tsükkel lõpetatakse ära ja programm saab üle jäänud koodiga jätkata.

Proovime ka ise järele. Teeme selle jaoks uue faili *loops.ts* ning avame selle oma tekstiredaktoris. Loome alguses muutuja *count* ning anname talle algväärtuseks nulli. *While* tsüklit kutsutakse välja nagu looks uut funktsiooni, aga neile ei anta nime, ega parameetreid. Parameetrite asemel on hoopis tingimus, mis hoiab funktsiooni töötamas. Loogeliste sulgude vahele läheb kood, mida igas tsüklis käivitatakse.

```
while (someCondition) {
  // Some code
}
```

Meie tahame oma *count* muutujat suurendada niikaua kuni ta väärtus on üle 20. Seega meie tsükli tingimus oleks järgmine: tsükkel töötab niikaua kuni muutuja *count* väärtus on alla 20. Numbrite väärtuseid saab kergelt võrrelda kasutades matemaatikast tuntuid sümboleid > ja <. Selleks, et kontrollida kas number on võrdne, kasutame mitte ühte, ega kahte, vaid kolme järjestikust võrdusmärki ===.

```
1 < count;
4 > count;
3 === count;
```

Seega tuleb meil tsükli tingimusena kontrollida kas *count* väärtus on väiksem kui 20 ning tsükli keha sees suurendame *count* väärtust ühe võrra. Selleks, et muutuja väärtust ühe võrra suurendada on olemas ka lühendatud süntaks, kus muutuja nime taha lisatakse kaks plussmärki.

```
while (count < 20) {
  count++;
}
```

Liigume *for* tsüklite juurde. *For* tsüklites oleneb tsükli tööaeg iteraatori väärtusest. Iteraator tähistatakse reeglina *i* tähega ning talle antakse tsüklit luues väärtus, limiit ja väärtus millega teda iga tsükli käigus suurendatakse. Süntaksi poolest sarnaneb *for* tsükkel *while* tsüklile. Lisaks on iteraatori väärtus kasutatav tsükli sees.

```
for (let i = 0; i < 10; i++) {
  // Some code
}
```

For tsüklis tingimuse asemel on meil tsükli töö juhendid. Me loome uue muutuja tsükli jaoks nimega i, millest saab meie iteraator. Järgmisena me märgime ära, et i väärtus peab jääma alla nulli ja viimasena on meil märgitud, et i suureneb ühe võrra iga tsükkel. Ehk kui soovime, et meie *count* muutujat suurendataks viis korda kümne võrra, siis anname iteraatorile limiidiks 5.

```
for (let i = 0; i < 5; i++) {
  count = count + 10;
}
```

Nüüd, kui oleme käsitlenud põhitõdesid, saame liikuda edasi meie mängude loomise juurde, kus õpime neid ka praktiliselt kasutama.

ÜLESANDED:

- Luua tsükkel, mis kutsub välja meie eelnevalt loodud liitmis tehte, ning annab parameetrite väärtusteks iteraatori ja iteraator + 5. Saadud vastus välja printida.
- Printida käsureale välja korrutus tabel, kasutades kahte tsüklit. Üks nendest tsüklitest pesitseb teise sees. Pesitseva tsükli puhul tuleb tähele panna, et kasutada tuleb iteraatori jaoks mingit sümbolit, mis ei ole i, sest i on juba kasutusel eelneva tsükli poolt. Soovitatav on liikuda iteraatorite nimedega tähestikulises järjekorras ehk antud juhul oleks teise tsükli iteraatoriks j.

Mängu programmeerimine

Nüüd kui oleme ennast tuttavaks teinud programmeerimise kõige tähtsamate põhiprintsiipidega, saame lõpuks liikuda edasi meie mängude arendamisega. Esimeseks mänguks teeme me klassikalise mängu Pong. Pong on üks esimesi video mänge ning on perfektne alguspunkt meie mänguarendus teekonnal oma lihtsuse poolest ning meil on võimalik kasutada kõike mida me eelnevas peatükis õppisime. Me saame ka eelneva peatüki kõrvale võta abimeheks juhul, kui on vaja meelde tuletada mõnda kontsepti või mõne elemendi funktsionaalsust. Aga enne, kui me saame liikuda oma mängu loomise juurde, on meil vaja luua asukoht, kus meie mängu hakatakse kuvatama. Selle jaoks toome me kasutusel tööriista nimega HTML.

HTML on interneti üks tähtsamaid osasid. Ilma selleta, ei oleks võimalik veebilehti visuaalselt kuvada ja ilma sellete ei ole ka meie mängu võimalik visuaalselt kuvada. HTML failid koosnevad erinevatest elementeidest, mille sees käivad omakorda elemendid jne. Igal HTML failil peab olema nii head kui ka *body* element selleks, et ta oleks korrektne.

Head element sisaldab üldist informatsiooni meie lehe kohta: lehe pealkiri, keel jne. Ka kasutatakse head elementi väliste failide sidumiseks HTML failiga. Seal on meil võimalus sisse importida oma mängu kood.

Body elemendis on kogu lehe sisu. Lehe sisu struktuur võib koosneda tuhandetest elementidest kuid võib ka piirduda vaid ühe elemendiga. Meie kasutusjuhu puhul piisab vaid ühest elemendist ning selleks on *canvas* element. *Canvas* lubab meil joonistada kasutaja ekraanile nii vektor- kui ka rastergraafika elemente ning neid ka animeerida. Tehniliselt ka meie mäng on vaid animatsioon, mis reageerib kasutaja sisendile.

Loome endale uue kausta, paneme sellele mingi kirjeldava nime (nt. pong, mäng1) ja avame selle oma tekstiredaktoris. Kui projekt avatud, loome uue faili ja anname talle nimeks index.html. Kui avame kausta operatsiooni süsteemi failibrauseris ja avame selle näeme, et see avaneb veebibrauseris, hetkel küll on näha vaid valge leht. Liigume tagasi oma tekstiredaktorisse ja avame oma loodud HTML faili. Selles failis meil palju

lisada vaja pole. Meil on vaja luua head ja *body* element ning lisada mõned elemendid ka nende sisse. Meie algse HTML faili struktuur on näha all olevas koodijupis.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Game 1</title>
</head>
<body>

</body>
</html>
```

Meil on olemas nüüd HTML faili skelett, aga nüüd oleks vaja ka lisada ülejäänud sisu, nimelt meie *canvas* element ja meie koodi sisse importimis rida. Lisame oma *body* elementi alguses *canvas* elemendi. Anname talle klassi nimega *canvas* ja ka samanimelise id. HTML elementidel on hea tava kui on klassinimi olemas juhuks, kui soovime neid hiljem stiilida. Id aga lubab meil viidata kindlale elemendile koodi sees. Nendele lisaks, määrame ka oma *canvase* suuruse. Seda on võimalik *canvas* elemendil määrata *width* ja *height* võtmesõnade abil. Meie paneme *canvase* laiuseks 1280 ja kõrguseks 720, ühikuks siin on pikslid.

```
<body>
  <canvas class="canvas" id="canvas" width="1280" height="720"></canvas>
</body>
```

Nüüd on meil koht olemas kuhu joonistada saame, aga nüüd oleks vaja lisada veel koht, kus meie kood imporditakse sisse ja kasutusele võetakse. Liigume tagasi üles head elementi ning lisame uuele reale peale *title* elementi uue elemendi *script*. *Script* elemendi sisse me ka midagi ei lisa, vaid lisame talle kaks võtme sõna. Esiteks lisame *src* võtmesõna, millega määrame ära, mis faili soovime kasutusele võtta ning teine võtmesõna, *defer*, ütleb meie veebibrauserile, et lae see fail alles siis, kui kogu *body* elemendi sisu on jõudnud ära laadida, muidu võib esineda, et meie skript laetakse enne kui *body* on valmis ja meie mäng ei tööta. Siin *script* elemendis tuleb ka tähele panna, et me impordime .js lõpuga faili, mitte .ts ja see tuleneb sellest, et TypeScript kompileeritakse ümber JavaScript failideks selleks, et seda veebibrauseris kasutada oleks võimalik.

```
<script src="./index.js" defer></script>
```

Kood

HTML fail valmis ja õigesti seadistatud saame liikuda edasi koodi kirjutamise poole. Loome oma projekti kaustas uue faili nimega index.ts. Selle faili sisse tuleb meie kogu mängu loogika. Suuremates projektides on hea loogika jagada eraldi failideks, mille sees olevad komponentidel on mingi kindel ülesanne, aga meie kasutusjuhul on see ebavajalik arvestades, et kogu meie mängu kood tuleb kokku umbes 170 koodirida.

Kõige esimese asjana võiksime saada ühenduse oma *canvas* elemendiga ning proovida sinna joonistada midagi. Selle jaoks saame kasutada TypeScriptis kaasas olevaid funktsioone, mis võimaldavad meil viidata HTML elementidele ning kasutada nende sisse ehitatud funktsionaalsust meie koodis.

Loome muutuja nimega canvas ja anname talle tüübiks *HTMLCanvasElement.* TypeScriptis on just HTML-iga töötamiseks kaasas sadu erinevaid andmetüüpe, mida saame kasutada arenduse käigus. Selle muutuja paneme võrduma oma *canvas* elemendiga. Selle jaoks saame kasutada *document.getElementById* funktsiooni, mis on ka üks TypeScripti standard funktsioone. Sulgude sisse kirjutame oma *canvas* elemendi id.

Kuid hetkel see lahendus veel ei kompileeri. Võid märgata, et su tekstiredaktoris on *canvas* muutuja all punane joon. See on selletõttu, et *document.getElementById* funktsioon tagastab meile viite tavalise HTMLElement tüübile. Kuna me teame, et element mille id on canvas on kindlalt *canvas* element, siis saame lisada rea lõppu lisaks ‘as HTMLCanvasElement’.

Selleks, et canvase peale joonistada, peame kasutama canvase konteksti. Meie kasutusjuhul, kasutame 2d konteksti, mis võimaldab meil joonistada kahe dimensioonilisi kujundeid. Selleks loome uue muutuja nime **ctx** ja tema andmetüübiks on *CanvasRenderingContext2D.* Canvase konteksti saame kätte läbi oma eelnevalt deklareeritud **canvas** muutuja. Meil on võimalik kasutada *canvas.getContext* funktsiooni, ning parameetrina võtab see funktsioon konteksti väärtuse, meie puhul on see ‘2d’. Kui kõik korrektselt tehtud peaks meie mängu kaks esimest koodi rida välja nägema järgmiselt.

```
const canvas: HTMLCanvasElement = document.getElementById("canvas") as HTMLCanvasElement;
const ctx: CanvasRenderingContext2D = canvas.getContext("2d");
```

Nüüd on meil ühendus meie HTML failiga ning ka meie *canvas* elemendiga ehk saame asuda joonistama. Canvasele joonistamine käib kasutades meie **ctx** muutujat ning tema sees olevaid funktsioone. Esimese asjana peame määrama värvi mida soovime kasutada. Canvase kontekstis on võimalik määrata nii joone, kui ka täitevärvi. Meie kasutusjuhul kasutame ainult täite värvi. Muidugi kui on soovi oma versiooni mängust erilisemaks teha, saab alati eksperimenteerida ka joone värviga.

Selleks, et täite värvi muuta saame muuta *ctx.fillStyle* väärtust. Pane tähele, et tegu ei ole funktsiooniga, vaid seekord väärtusega meie **ctx** muutuja sees. *Fillstyle* väärtus võib olla mõni värv sõnadega (black, red, green, blue) või värvikood. Värvikoodid annavad meil võimaluse kasutada miljoneid erinevaid värve, aga meile piisab kasutada värvinimesid. Anname siis *fillStyle* väärtuseks *black*. Siin tuleb ka meeles pidada, et *fillStyle* väärtus on globaalne, ehk kui kord sellele väärtuse anname, siis kõik kujundid mida joonistame on selle kindla värviga nii kaua kuni uuesti värvi muudame.

Kui värv kindel saame liikuda joonistamise juurde. Siin õnneks midagi keerulist ei ole, kõik käib ühe funktsiooniga. Värvime kogu *canvase* tausta mustaks kasutades *ctx.fillRect* funktsiooni. Sellel funktsioonil on neli parameetrit: x koordinaat, y koordinaat, laius ja kõrgus. Kuna soovime kogu *canvase* värvida, siis nii meie x ja y väärtusteks on 0 ning laius ja kõrgus on suurused, mis määrasime HTML failis. Kuid selle asemel, et need siia uuesti kirjutada, saame need väärtused kätte läbi meie **canvas** muutuja ning selle *height* ja *width* väärtuste. Kasutades neid väärtuseis tagame meile kindluse, et kui me peaksime *canvas* elemendi suurust muutma, siis meie kood töötab korrektselt edasi. Kui kõik tehtud peaks koodis olema järgmised kaks rida juures.

```
ctx.fillStyle = 'black';
ctx.fillRect(0, 0, canvas.width, canvas.height);
```

Nüüd on vaja kompileerida meie *index.ts* fail. Eelnevalt kasutasime käsureal *ts-node* käsklust. See töötas meie kasutusjuhul hästi, sest kogu meie kood eelnevalt töötas ainult käsureal. Nüüd aga on vaja, et meie kood töötaks veebibrauseris. Selleks kasutame *tsc index.ts* käsklust. See loob meile vajaliku *index.js* faili, mida meie HTML

fail loeb. Kui see tehtud, saame avada oma HTML faili ning meile peaks vastu paistma üks suur must ristkülik. Sellest saab meie mängu taust.

Mängu elementide joonistamine

Meil on olemas mängu taustavärv ning saame liikuda edasi mängu elementide joonistamise juurde. Siin ei ole mingit üllatust, et protsess on täpselt sama. Pong mängus on mängija kujutatud valge ristkülikuna, mis liigub ekraanil üles ja alla. Joonistame siis valge ristküliku, mis istub veidi äärtest eemal, umbes 20 pikslit, kasutades samu funktsioone, mida kasutasime tausta värvimiseks. Anname laiuse väärtuseks 30 ja kõrguseks 120. Ära unusta eelnevalt värv määrata! Kui nüüd kompileerime *index.ts* faili ning avame oma HTML faili, siis näeme, et mustale taustale on tekkinud valge ristkülik.

```
ctx.fillStyle = 'white';
ctx.fillRect(20, 20, 30, 120);
```

Muudame oma joonistamise protsessi veidi mugavamaks. Praegu peame iga kord kirjutama need kaks rida koodi, mis tekitab kordust. Nagu eelnevast mäletame, siis koodi kordus on asi, millest peaks eemale hoidma. Selle probleemi saame lahendada funktsiooni tegemisega. Deklareerime uue funktsiooni *drawRectangle*. Määrame sellele ka viis parameetrit: x, y, width ja height andmetüüpiga *number* ning color andmetüübiga *string.* Selle funktsiooni sisse kopeerime meie eelnevalt kirjutatud kaks koodirida, aga muudame neile antud väärtuseid. Meie funktsiooni color parameetri anname *fillStyle* väärtuseks, ja ülejäänud parameetrid lähevad *fillRect* funktsiooni parameetriteks.

```
function drawRectangle(x: number, y: number, width: number, height: number, color: string) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, width, height);
}
```

Nüüd kui meil on enda joonistamise funktsioon olemas, saame sellega asendada meie eelmised joonistamis käsklused.

```
drawRectangle(0, 0, canvas.width, canvas.height, 'black');
drawRectangle(20, 20, 30, 120, 'white');
```

Mängu elementide liikuma panemine

Nüüd kui meil on korralik joonistamis funktsioon olemas, saame liikuda edasi oma mängu elementide liigutamise poole. Selleks saame kasutada TypeScriptis sisse ehitatud *requestAnimationFrame* funktsiooni. Enne selle kasutusele võtmist, peame tegema natukene ettevalmistust.

Algatuseks loome kaks muutujat, x ja y, faili alguses ning anname neile väärtuseks 0. Nendest saavad meie mängija koordinaadid, mis määravad meie ristküliku asukoha. Kuna muutujad on seda tüüpi, mida soovime hiljem modifitseerida, siis tuleb neid deklareerida *let* võtmega.

```
let x = 0;
let y = 0;
```

Teeme uue funktsiooni *draw* ning liigutame oma tausta ning ka mängija joonistamise funktsioonid sinna sisse. Peale seda suurendame nii x kui ka y muutuja väärtust viie võrra. Viimasena kutsume välja *requestAnimationFrame* funktsiooni, ning anname talle parameetriks meie *draw* funktsiooni. Funktsioon *requestAnimationFrame* ootab kuni meie *draw* funktsioon oma toimetused ära lõpetab ning seejärel kutsub ta uuesti välja. Seega, meie *draw* funktsioonist saab meie mängu põhiline loogika funktsioon, kus kõik mängu osad kindlas järjekorras tööle pannakse. Kutsume välja meie *draw* funktsiooni ning värskendame oma HTML lehte.

```
function draw() {
  drawRectangle(0, 0, canvas.width, canvas.height, 'black');
  drawRectangle(x, y, 30, 120, 'white');

  x = x + 5;
  y = y + 5;

  requestAnimationFrame(draw);
}

draw();
```

Meie mängija liigub diagonaalselt mööda ekraani, aga temast jääb järele saba. See juhtub seetõttu, et *canvas* ei puhasta ennast ise. Selle jaoks aga, saame kirjutada funktsiooni, mis seda meie jaoks teeb. Loome uue funktsiooni *clearCanvas* ja

liigutame oma tausta värvimise funktsiooni sinna sisse. Enne seda kutsume välja *ctx.clearRect* funktsiooni, mis puhastab *canvase* pealt kõik joonistatud ristkülikud. Seega meie poolt loodud funktsioon teeb alguses ekraani puhtaks ja see järel joonistab tausta tagasi. Kutsume selle välja meie *draw* funktsiooni alguses selleks, et seda iga kaadri alguses välja kutsutaks.

```
function clearCanvas() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawRectangle(0, 0, canvas.width, canvas.height, 'black');
}

function draw() {
  clearCanvas();
```

Mängija liigutamine

Saades mängija liikuma on meil nüüd võimalik ta liikuma panna ka kasutades klaviatuuri sisendit. Enne kui selleni jõuame loome paar uut konstanti endale faili algusesse. Alustuseks loome kaks konstanti CANVAS_WIDTH ja CANVAS_HEIGHT kasutades võtit *const*. Näeme, et nende nime stiil erineb meie eelnevalt deklareeritud muutujatest. Globaalsetel konstante on hea tava erineval viisil kirjutada kui tavalisi muutujaid selleks, et tunneks kiiresti ära millise muutujaga on tegu. Nende kahe konstandi väärusteks paneme vastavalt canvas.width ja canvas.height. Asendame ka koodis kõik kohad kus eelnevalt kasutasime neid kahte väärtust meie uute muutujatega.

Teiseks loome konstandi PADDLE_X_OFFSET millele anname väärtuseks 40. See väärtus määrab meie mängija kauguse ekraani äärest. Anname ka meie x muutuja väärtuseks meie värskelt loodud konstandi. Viimasena loome konstandi PADDLE_SPEED mille väärtuseks anname hetkel 20. See konstant määrab meie liikumis kiiruse. Nüüd kui need konstandid paigas, saame liikuda edasi klaviatuuri sisendi lugemise juurde.

```
const CANVAS_WIDTH = canvas.width;
const CANVAS_HEIGHT = canvas.height;

const PADDLE_X_OFFSET = 40;

const PADDLE_SPEED = 20;
```

```
let x = PADDLE_X_OFFSET;
```

Loome uue funktsiooni nimega *handleMovement* ning anname talle parameetriks event. Selle parameetri tüübiks anname *KeyboardEvent.* Läbi selle parameetri, saame kätte kasutaja poolt vajutatud nupud. Siin funktsiooni sees võtame kasutusele funktsionaalsuse nimega *switch* käsklus. Oma loomuses on *switch* sama mis *if/else*, aga on tunduvalt loetavam ja kergemini hooldatav tulevikus. *Switch statementi* kutsume välja nagu tavalist funktsiooni. Tema parameetriks antakse mingi muutuja mille väärtust võrreldakse *switchi* sees. Meie kasutusjuhul on selleks *event.key*.

```
switch (event.key) {
  }
```

Switch käskluse sisse teeme *case*-id. Igal *case*-il on mingi väärtus millega meie event.key peab võrduma selleks, et *case* käivituks. *Case*-i sees saame kirjutada funktsionaalsuse, mida soovime, et täidetaks ning iga *case*-i viimasel real on *break. Break* teatab lihtsalt, et siin funktsionaalsus lõppeb ja *switch* võib kinni panna.

```
    case 'w':
      y = y - PADDLE_SPEED;
      break;
    case 's':
      y = y + PADDLE_SPEED;
      break;
    default:
      break;
```

Nagu näeme, siis meie kasutusjuhul on meil kaks *case*-i: kui sisend on w ja kui sisend on s. Meil on ka vaikimis *case* olemas, aga selle jätame tühjaks. Kui meie sisend on w, siis vähendame muutuja y väärtust ning kui sisend on s, siis suurendame muutuja y väärtust. Kui kõik õigesti, siis peaks meie *handleMovement* funktsioon välja nägema selline.

```
function handleMovement(event: KeyboardEvent) {
  switch (event.key) {
    case 'w':
      y = y + PADDLE_SPEED;
      break;
    case 's':
      y = y - PADDLE_SPEED;
```

```
      break;
    default:
      break;
  }
}
```

Kuid me ei saa seda funktsiooni lihtsalt niisama käivitada nagu me eelnevalt oleme teinud. Selleks, et kasutaja klaviatuuri sisendit kätte saama, peame lisama nn. kuulaja klaviatuuri sisendile. Selleks saame kasutada TypeScriptis olemas olevat *document.addEventListener* funktsiooni. See funktsioon võtab kaks parameetrit: string, mis näitab mis käsklust kuulatakse ning funktsioon, mis käskluse peale käivitatakse. Meie käsklus, mida me kuulame, on *keydown.* Kutsume selle funktsiooni välja enne meie *draw* funktsiooni välja kutsumist.

```
document.addEventListener('keydown', handleMovement);
draw();
```

Liikumis piirded

Hetkel meie mängija saab üles ja alla liikuda, kuid selle asemel, et äärtes seisma jääda, liigub meie mängija ekraanilt lihtsalt välja. Õnneks saame kiirelt ja kergelt juurde lisada meie eelnevalt kirjutatud koodi.

Alustuseks loome paar uut konstanti faili alguses. Anname ühele konstandile nimeks PADDLE_HEIGHT ja väärtuseks 120 ning teisele anname nimeks PADDLE_WIDTH ja väärtuseks 30. Asendame meie mängija joonistamis funktsiooni parameetris seal oleva 30 ja 120 meie uute konstantidega.

```
  drawRectangle(x, y, PADDLE_WIDTH, PADDLE_HEIGHT, 'white');
```

Meil oli vaja see väärtus panna konstandi sisse, sest mängija kõrguse väärtus tuleb liikumis piirdete juures kasutusele. Selleks, et piirded panna peame muutma oma *handleMovement* funktsiooni sisu. Kui kasutaja vajutab w nuppu, siis peame kontrollima, et uus väärtus ei oleks väiksem kui 0 ja kui kasutaja vajutab s nuppu, siis peame kontrollima, et uus väärtus ei oleks suurem *canvase* kõrgusest. Selle implementeerimiseks saame kasutada *if* lauset.

```
case 'w':
    if (y - PADDLE_SPEED >= 0) {
```

```
        y = y - PADDLE_SPEED;
      }
      break;
case 's':
      if (y + PADDLE_SPEED + PADDLE_HEIGHT <= CANVAS_HEIGHT) {
        y = y + PADDLE_SPEED;
      }
default:
      break;
```

Nüüd kui HTML lehte värskendame, siis näeme, et meie mängija jääb *canvase* sisse ning ei liigu sealt enam välja. Saame liikuda teise mängija loomise kallale.

Teine mängija

Meil on üks funktsioneeriv mängija olemas, kuid Pongi ei saa mängida ainult ühe mängijaga. Selleks loome teisele poole väljakut ka teise mängija, kes on ka kasutaja poolt juhitav. Enne seda peame koodis mõned muudatused tegema. Kuna meil saab kohe olema kaks mängijat, siis lihtsalt muutujad x ja y enam ei kõlba. Seega muudame nende nimetused ümber player1x ja player1y. Vaatame kindlalt üle, et igal pool kus nad on koodis kasutatud saavad nad ümber nimetatud, muidu kood ei kompileeri.

Asume nüüd tööle teise mängija kallale. Õnneks on meil juba täpselt teada mis sammud peame läbima. Alustuseks loome kaks muutujat player2x ja player2y. Muutuja player2x väärtuseks anname CANVAS_WIDTH – PADDLE_X_OFFSET – PADDLE_WIDTH, sest hetkel veel asuvad meie koordinaadid ristküliku vasakul ülemises nurgas. Muutuja player2y väärtuseks anname 0.

Nüüd kui koordinaatide muutujad olemas, siis järgmisena võiksime joonistada teise mängija *canvase* peale. Kui me juba seda teeme, siis võiksime oma mängijate joonistamise loogika ühte tõsta. Selleks loome uue funktsiooni *drawPlayers* ning tõstame oma esimese mängija joonistus funktsiooni selle sisse. Seejärel kopeerime selle uuele reale ja muudame esimese mängija koordinaadid teise mängija koordinaatideks. Kutsume oma *drawPlayers* funktsiooni *draw* funktsiooni sees välja PEALE *clearCanvas* funktsiooni.

```
function drawPlayers() {
  drawRectangle(player1x, player1y, PADDLE_WIDTH, PADDLE_HEIGHT, 'white');
  drawRectangle(player2x, player2y, PADDLE_WIDTH, PADDLE_HEIGHT, 'white');
}
```

ÜLESANNE: Lisada funktsionaalsus ka teise mängija liigutamiseks. Teise mängija liigutamiseks kasutame nooleklahve, nimelt just üles ja alla. Funktsionaalsus tuleks lisada olemasoleva *switchi* sisse. Ülesannet lahendades pane kindlasti tähele, et kasutaksid just teise mängija koordinaate, mitte esimese mängija koordinaate.

LAHENDUS: Teise mängija liigutamise jaoks peame looma *switchi* sisse kaks uut case. Saame jällegi kopeerida eelmise mängija koodi ning seda taas kasutada: w asemel on *ArrowUp*, s asemel on *ArrowDown* ja esimese mängija koordinaadid tuleb asendada teise mängija koordinaatidega.

```
case 'ArrowUp':
      if (player2y - PADDLE_SPEED >= 0) {
        player2y = player2y - PADDLE_SPEED;
      }
      break;
case 'ArrowDown':
      if (player2y + PADDLE_SPEED + PADDLE_HEIGHT <= CANVAS_HEIGHT) {
        player2y = player2y + PADDLE_SPEED;
      }
```

Kui nüüd kompileerime koodi ja HTML lehe värskendame, siis peaksime nägema kahte mängijat ning mõlemat saab liigutada. Meil on vaid üks element puudu mängitavast mängust: pall.

Koodi muutused

Nagu eelnevalt mainisin, siis hetkel näitavad meie koordinaadid mängija ülemist vasakut nurka. See aga võib tekitada meile probleeme, seega peame veidike koodi ümber kirjutama. Programmeerimises tuleb tihti ette, et eelnevalt kirjutatud koodi on vaja ümber kirjutada kui vajaduses seda nõuavad.

Alustame meie *drawRectangle* funktsioonist. Peame muutma x ja y väärtuseid mis antakse *ctx.fillRect* funktsioonile. Hetkel alustatakse joonistamist otse nendest koordinaatidest, mis ette antakse. Muudame selle nii, et x ja y mis antakse on keskmine koordinaadid ja me arvutame välja joonistamis alguspunkti koordinaadid kasutades ristküliku laiust ja kõrgust, mis on meil kaasaantud funktsiooni parameetrites.

```
ctx.fillRect(x - width / 2, y - height / 2, width, height);
```

Kui see muudatus tehtud, peame muutma oma mängijate algkoordinaatide väärtusi. Anname mõlema mängija y koordinaadiks poole *canvase* kõrgusest ning konstandi *player2x* väärtuseks paneme CANVAS_WIDTH – PADDLE_X_OFFSET, sest enam ei ole meil vaja arvestada mängija laiusega.

```
let player1x = PADDLE_X_OFFSET;
let player1y = CANVAS_HEIGHT / 2;

let player2x = CANVAS_WIDTH - PADDLE_WIDTH;
let player2y = CANVAS_HEIGHT / 2;
```

Ka on muudatusega katki läinud meie *canvase* puhastus funktsioon. Seal peame andma x ja y koordinaatideks *canvase* kesk koordinaadid.

```
function clearCanvas() {
  ctx.clearRect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);
  drawRectangle(CANVAS_WIDTH / 2, CANVAS_HEIGHT / 2, CANVAS_WIDTH, CANVAS_HEIGHT,
'black');
}
```

Nüüd on jäänud meil veel parandada meie liikumis piirangud. Ka see osa meie koodist oli kirjutatud arvestades et mängijate x ja y koordinaadid asuvad nende vasakul üleval nurgas. Kuna meie koordinaadid viitavad nüüd mängija keskpunktile, peame lisama igasse kontrolli ka arvutuse kus lisatakse ka mängija kõrgus. Kui see tehtud, saame liikuda edasi järgmise mängu elemendi juurde.

```
case 'w':
      if (y - PADDLE_SPEED - PADDLE_HEIGHT / 2 >= 0) {
        y = y - PADDLE_SPEED;
      }
      break;
case 's':
      if (y + PADDLE_SPEED + PADDLE_HEIGHT / 2 <= CANVAS_HEIGHT) {
        y = y + PADDLE_SPEED;
      }
case 'ArrowUp':
      if (player2y - PADDLE_SPEED - PADDLE_HEIGHT / 2  >= 0) {
        player2y = player2y - PADDLE_SPEED;
      }
      break;
case 'ArrowDown':
      if (player2y + PADDLE_SPEED + PADDLE_HEIGHT + PADDLE_HEIGHT / 2  <=
CANVAS_HEIGHT) {
```

```
        player2y = player2y + PADDLE_SPEED;
      }
 default:
      break;
```

Võib tunduda tüütu ümber kirjutada koodi, mis juba eelnevalt on kirjutatud, aga tarkvaraarenduses tuleb tihti ette olukordi, kus eelnevalt kirjutatud kood vajab muudatusi selleks, et oleks lihtsam edasi arendada rakendust.

Pall

Liigume edasi meie mängu kõige tähtsama osa, palli, juurde. Loeme üles palli omadused mida meil vaja on. Meie pall on kindla suurusega ruudu kujuline, ta peaks liikuma kindla kiirusega, ta algne positsioon peab olema ekraani keskel, kui pall puudutab mängijat siis lüüakse ta vastas suunda, kui pall tabab kas alumist või ülemist ekraani serva põrgatatakse ta tagasi ning kui pall tabab kas vasakut või paremat ekraani serva, siis saab üks mängijatest punkti.

Alustame palli ekraanile joonistamisega. Protsess on meile nüüd juba tuttav. Loome uue konstandi faili alguses nimega BALL_SIZE ja anname talle väärtuseks 30. Sellest saab nii meie palli laius kui ka kõrgus. Loome ka kaks muutujat *ballX* ja *ballY* ja anname nendele väärtusteks *canvase* kekspunktid.

```
const BALL_SIZE = 30;

let ballX = CANVAS_WIDTH / 2;
let ballY = CANVAS_HEIGHT / 2;
```

Nimetame oma *drawPlayers* funktsiooni ümber *drawGameObjects* ja lisame sinna sisse palli joonistamise funktsiooni. Ära unusta vahetada ka meie *draw* funktsioonis välja kutsutud *drawPlayers* funktsioon uue nimega.

```
function drawGameObjects() {
  drawRectangle(player1x, player1y, PADDLE_WIDTH, PADDLE_HEIGHT, 'white');
  drawRectangle(player2x, player2y, PADDLE_WIDTH, PADDLE_HEIGHT, 'white');
  drawRectangle(ballX, ballY, BALL_SIZE, BALL_SIZE, 'white');
}
```

Kui kõik korrektselt tehtud, siis näeme, et keset ekraani on nüüd tekkinud valge ruut. Nüüd on vaja see ka liikuma panna.

Palli liikumine

Palli liikumine on veidike keerukam oma loogika poolelt, kui mängijate liigutamine. Mängijad liikusid vaid üles ja alla, aga pall peab liikuma nii üles-alla, vasakule-paremale ja ka diagonaalselt. Seda kõike peab ta tegema hoides sama kiirust mistahes suunas ta liigub. Õnneks tuleb meile siin matemaatika abiks.

Alguseks loome uue konstandi faili algusesse nimega BALL_SPEED ning väärtuseks võiks hetkel anda 15, hiljem saab alati muuta kui tuleb välja et pall on liiga kiire või aeglane. Loome ka seekord kolm muutujat: *ballXspeed, ballYspeed* ja *ballAngle.* Kuigi meil on ette määratud palli kiirus 15, siis nii x kui ka y kiirus on alati erinevad olenedes palli liikumise nurgast. Neid väärtuseid saame arvutada välja kasutades. Muutujas *ballAngle* hoiame palli liikumis suuna nurka.

```
const BALL_SPEED = 15;

let ballXspeed = 0;
let ballYspeed = 0;
let ballAngle = 0;
```

Loome uue funktsiooni *setBallMovement.* Selle funktsiooni sees hakatakse välja arvutama meie x ja y telgede liikumis kiirused. Selleks, et x telje kiirus kätte saada korrutame meie määratud palli kiiruse väärtuse (mis asub konstandis BALL_SPEED) palli nurga koosinusega ja y telje kiiruse saame kätte korrutades palli kiiruse väärtuse palli nurga siinusega.

```
function setBallMovement() {
  ballXspeed = BALL_SPEED * Math.cos(ballAngle);
  ballYspeed = BALL_SPEED * Math.sin(ballAngle);
}
```

Lisaks *setBallMovement* funktsioonile, loome funktsiooni *setBallAngle* millel on üks parameeter *angle* andmetüübiga number. Selle funktsiooni sees anname meie *ballAngle* muutujale väärtuse mis antakse funktsioonile parameetris kaasa ning peale seda kutsume välja meie eelnevalt loodud *setBallMovement* funktsiooni, sest need kaks funktsiooni sõltuvad tugevalt üksteisest.

```
function setBallAngle(angle: number) {
  ballAngle = angle;
```

```
  setBallMovement();
}
```

Viimasena peame kutsuma välja *setBallAngle* funktsiooni ENNE *draw* funktsiooni selleks, et palli liikumis loogika oleks paigas enne, kui meie mängu loogika tööle hakkab.

```
setBallAngle(130);
draw();
```

Nüüd kui ettevalmistused on tehtud, saame palli liikuma panna. Loome uue funktsiooni *moveBall.* Hetkel loome me pallile väga algelise liikumis loogika ning lihtsalt suurendame palli x koordinaati palli x telje kiirusega ja palli y koordinaati palli y telje kiirusega.

```
function moveBall() {
  ballX += ballXspeed;
  ballY += ballYspeed;
}
```

Muidugi tuleb see meil ka *draw* funktsiooni sees ka välja kutsuda. Kui see tehtud kompileerime koodi ja värskendame oma HTML lehte ning meie pall peaks diagonaalselt ekraanil ringi liikuma, aga hetkel sõidab ta rahumeeli ekraanilt välja. Õnneks on seda kerge parandada.

Palli põrkamine

Palli põrkama panemiseks saame kasutada sarnast loogikat, mida me kasutasime mängija liikumise piiramiseks. Peame lisama *moveBall* funktsiooni sisse paar kontrolli, mis kontrollivad palli koordinaate vastu ekraani servasid. Kui peaks juhtuma, et pall läheb vastu ülemist või alumist seina, siis tuleb y telje liikumis suund ümber pöörata. Seda saame teha korrutades meie praegust kiiruse väärtust miinus ühega. Kasutame siin *or* operaatorit, mis lubab meil mitu võrdlust teha ühes *if* lauses. *Or* operaatoris peab üks pooltest võrduma tõese väitega selleks, et *if* lause sisu käivitataks.

```
function moveBall() {
  if (ballY + ballYspeed > CANVAS_HEIGHT || ballY + ballYspeed - BALL_SIZE
/ 2 < 0) {
```

```
    ballYspeed = ballYspeed * -1;
  } else {
    ballY += ballYspeed;
  }
  ballX += ballXspeed;
}
```

Palli x-telje käitumine

Kui pall x-telje ääri puudub, on ta käitumine teistsugune, kui y-teljega oli. Peame lisama *moveBall* funktsiooni sisse ka kontrollid, mis viivad palli tagasi ta algpositsioonile juhul kui pall x telje ääri puudub. Lisaks võiks pall ka uue suvalise nurga saada.

Alustame selle loogika implementeerimist luues uue funktsiooni *getRandomInt* millel on kaks parameetrit: min ja max ning mõlemate andmetüübid on number. Kuna me tahame kindlasti täis arvu saada, siis saame kasutada *Math.ceil* ja *Math.floor* funktsioone min ja max peal ning juhul kui nad ei ole täisarvud, saame neist täisarvud.

Viimasena tuleb meil lisada *return* lause. Selles *return* lauses loome meie uue suvalise numbri kasutades *Math.floor* ja *Math.random* funktsioone. Meie lahendus annab meile numbri miinimum ja maksimum numbrite vahel, kusjuures miinimum väärtus on kaasarvatud, aga maksimaalne väärtus, mille andsime max parameetriga, ei ole.

Kuna meie funktsioon tagastab väärtuse, saame märkida ära ka andmetüübi, mis tagastatakse. Praegusel juhul on ka selle andmetüüp number.

```
function getRandomInt(min: number, max: number): number {
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min) + min);
}
```

Loome uue funktsiooni *resetBall.* Siin me palju tegema ei pea, sest saame kasutada ära eelnevalt loodud funktsioone ja meie uut abi funktsiooni. Kutsume välja selle funktsiooni sees *setBallAngle* funktsiooni ning anname talle parameetri väärtuseks otse meie *getRandomInt* funktsiooni, mille parameetrite väärtused on 0 ja 360. Peale seda paneme ka palli x ja y koordinaadid võrduma *canvase* kekspunktiga.

```
function resetBall() {
  setBallAngle(getRandomInt(0, 360));
  setBallMovement();
  ballX = CANVAS_WIDTH / 2;
  ballY = CANVAS_HEIGHT / 2;
}
```

Viimaseks lähme jälle meile tuttavat *moveBall* funktsiooni mudima. Peame lisama sinna sisse kontrolli, et kui pall peaks puutuma x-telje ääri, siis kutsutakse välja meie vast loodud *resetBall* funktsioon.

```
  if (ballX + ballXspeed > CANVAS_WIDTH || ballX + ballXspeed < 0) {
    resetBall();
  }
```

Nüüd kui kompileerime koodi ja teeme lehele värskenduse näeme, et pall täitsa põrkab mööda ekraani ringi ning juhul kui ta mängija seljataha satub, lendab ta tagasi ekraani keskele ning hakkab uues suvalises suunas ringi põrkama.

Mängija ja pall

Oleme lähedal oma mängu valmis saamiseni. Meil on veel puudu üks tähtis osa meie mängu loogikast: mängija ja palli vahelise kokkupõrke loogika. Kui pall puudutab mängijat, peab palli liikumis suund vahetuma. See on sarnane loogikale, mille tegime y-telje jaoks, kuid seekord x teljega. Keerulisemaks teeb selle aga see, et meie mängijad saavad liikuda ning nende suurus on vaid 120 pikslit. Asume tööle!

ÜLESANNE: Luua funktsioon, mis kontrollib, kas palli koordinaadid on ristumas mängijate koordinaatidega. Seda on võimalik teha ühe *if* lausega, aga kui see tundub liiga keeruline, siis pole ka hullu kasutada kahte eraldi *if* lauset.

LAHENDUS: Loome uue funktsiooni *handleCollision* ning selle sees loome ühe *if* lause. Selles *if* lauses peame kontrollima mõlema mängija puhul, kas meie pall riivab kasvõi ühe piksliga mängijat. Siin peame kasutama nii meile tuttavat *or* operaatorit kui ka *and* operaatorit.

```
function handleCollision() {
  if (
    (ballX - BALL_SIZE / 2 <= player1x + PADDLE_WIDTH / 2 &&
      ballY + BALL_SIZE / 2 >= player1y - PADDLE_HEIGHT / 2 &&
```

```
      ballY - BALL_SIZE / 2 <= player1y + PADDLE_HEIGHT / 2) ||
    (ballX + BALL_SIZE / 2 >= player2x - PADDLE_WIDTH / 2 &&
      ballY + BALL_SIZE / 2 >= player2y - PADDLE_HEIGHT / 2 &&
      ballY - BALL_SIZE / 2 <= player2y + PADDLE_HEIGHT / 2)
  ) {
    ballXspeed = ballXspeed * -1;
  }
}
```

Pane siin kindlasti tähele, et mõlemad *or* operaatori pooled on sulgude sees, sest me soovime, et nende sisu käsitletaks kui ühe tingimusena. Kutsume selle funktsiooni välja meie *draw* funktsiooni sees peale palli liigutamist.

Kui nüüd kompileerime ning lehte värskendame saame proovida palli lüüa vastaspoolele. Märkame, et vahepeal pall jääb mängija sisse kinni või põrkab mitu korda kiiresti ühe koha peal. Õnneks on see üsna kerge parandada

Loome uue muutuja faili algusesse nimega *lastCollisionTime*. Sellele väärtuseks anname *Date.now*, mis annab meile praeguse aja. *Date.now* annab väärtuseks numbri mis on võrdeline ajaga mis on kulunud 1. jaanuarist 1970 millisekundites. Seda kutsutakse *Unix Epoch*-iks. Seda muutujat kasutades saame luua oma palli ja mängija kokkupõrke loogikale puhvri, mille sees ei ole võimalik kokkupõrget toimuda.

```
let lastCollisionTime = Date.now();
```

Enne *handleCollision* funktsiooni loome uue funktsiooni *checkCollisionCooldown.* Sellest funktsioonist hakkame tagastama boolean andmetüübiga väärtust ehk märgime ka selle ära. Parameetreid sellel ei ole. Funktsiooni sees loome uue muutuja *timeRightNow* ning selle väärtuseks anname ka *Date.now*. Järgmisel real peame kontrollima, kas *timeRightNow* ja *lastCollisionTime* vahe on suurem kui 300. Kui jah siis tagastame true, kui ei siis tagastame false.

```
function checkCollisonCooldown() {
  const timeRightNow = Date.now();

  if (timeRightNow - lastCollisionTime > 300) return true;
  else return false;
}
```

Nüüd tuleb seda funktsiooni kasutada meie *handleCollision* funktsioonis. Loome seal esimesele reale uue muutuja *isCollisionAllowed* ja panema ta võrduma meie *checkCollisionCooldown* funktsiooniga ehk tema väärtus on nüüd kas *true* või *false*. Rida peale seda loome uue *if* lause, kus kontrollima, kas *isCollisionAllowed* väärtus on *false.* Kui nii on, siis meie funktsioon edasi enam ei lähe. Seda saame teha kirjutades kohe peale sulge ***return.*** See viskab meid funktsioonist välja ning midagi edasi ei tehta.

```
const isCollisionAllowed = checkCollisonCooldown();
if (!isCollisionAllowed) return;
```

Viimasena peame andma muutujale *lastCollisionTime* uue ajalise väärtuse, juhul kui kokkupõrge toimub. Seda saame teha samas kohas kus muudame x-teljel liikumise kiirust vastupidiseks.

```
    lastCollisionTime = Date.now();
    ballXspeed = ballXspeed * -1;
```

Kui nüüd mängida proovime, siis näeme, et pall enam mängija sisse kinni ei ole ning põrkab rõõmsalt mööda ekraani edasi-tagasi.

Punktid

Oleme jõudnud oma mängu viimase funktsionaalsuse juurde: punktide salvestamine. Loome faili algusesse uued muutujad: *player1Score* ja *player2Score* ning nende väärtused on 0, sest muidu ei oleks ju aus kui mängijad alustaksid erinevate skooridega.

Selleks, et mängijad punkte juurde saaksid, peame jällegi minema muutma meie *moveBall* funktsiooni. Siin peame x-telje kontrolli lahti lööma kaheks, ehk üks *or* operaatori pool üheks *if* lauseks ja teine teiseks. Mõlema *if* lause sees kutsume välja *resetBall* funktsiooni, aga enne seda peame ka ühele mängijale punkte juurde lisama. Kui pall lendab paremalt poolt ekraanilt välja saab mängija 1 punkte juurde ja kui vasakult poolt, siis saab mängija 2 punkte juurde.

```
if (ballX + ballXspeed > CANVAS_WIDTH) {
  player1Score++;
  resetBall();
```

```
  }

  if (ballX + ballXspeed < 0) {
    player2Score++;
    resetBall();
  }
```

Kui see tehtud, peaksime punktid kusagil kuidagi välja kuvama ka. Traditsiooniliselt on seda kuvatud väljaku peal. Seega loome uue funktsiooni *drawScore.* Siin me kahjuks ei saa kasutada ära oma ristküliku joonistamise funktsiooni, aga oma iseloomult on teksti kuvamine väga sarnane.

Enne kui teksti ekraanile joonistame peame andma tekstile suuruse ja fondi. Font tähistab mis stiilis tekst välja näeb kui teda ekraanil kuvatakse. Selleks, paneme *ctx.font* võrduma '50px Arial'.

```
function drawScore() {
  ctx.font = "50px Arial";
}
```

Kui font paigas saame joonistada oma teksti. Selleks saame kasutada *ctx.fillText* funktsiooni. Selle esimeseks parameetriks on string, mida soovime välja printida, meie punktid aga on number andmetüübiga. Võib tunduda, et siin tekib probleem, aga õnneks on TypeScriptis sisse ehitatud mitmed funktsioonid, mis aitavad ühest andmetüübist teise väärtuseid muuta. Selleks, et meie numbrist string saada, saame kasutada *toString* funktsiooni, mis on kõikidel number andmetüüpi funktsioonidel sisse ehitatud. Seda saame välja kutsuda järgmiselt

```
player1Score.toString()
```

Teksti kuvamise funktsiooni teised parameetrid määravad teksti asukoha x ja y. Meie x peaks olema keskpunktist umbes 60 pikslit ühele või teisele poole ning y väärtus võiks olla 100 pikslit ekraani servast. Seda teha mõlema mängija punktide jaoks.

```
  ctx.fillText(player1Score.toString(), CANVAS_WIDTH / 2 - 65, 100);
  ctx.fillText(player2Score.toString(), CANVAS_WIDTH / 2 + 50, 100);
```

Punktide vahele võiks kuvada ka mingi vahemärgi (koolon või sidekriips). Ka selle saame väljakuvada kasutades sama funktsiooni. Selle y koordinaat oleks sama mis punktidel, aga x koordinaat võiks olla täpselt keskel.

```
  ctx.fillText(":", CANVAS_WIDTH / 2, 100);
```

Nüüd tuleb meil veel see funktsioon välja kutsuda *drawGameObjects* funktsiooni sees. Koodi kompileerides ja lehte värskendades näeme, et meil on kenasti kuvatud 0 : 0 keset ekraani. Kui mõni mängija punkti saab läheb tema punktisumma suuremaks ning kui keegi saab 10 punkti alustatakse mäng uuesti.

Lõpetuseks

Sellega sai meie esimene mäng valmis, aga nagu alati programmeerimises on, siis alati on võimalik midagi juurde lisada. Näiteks saaks lisada mängija ja palli kokkupõrkesse uue palli nurga arvutamise arvestades kuhu täpselt pall tabas. Ka saaks lisada mingi sõnumi mida näidatakse kui mängija võidab.

Klassid

Enne, kui saame liikuda oma järgmise mängu juurde on meil vaja endale tuttavaks teha üks programmeerimise levinumaid metoodikaid: objektorienteeritud programmeerimine. Objektorienteeritud programmeerimine struktureerib koodi klassideks, mille abil saame luua objekte, mis käituvad vastavalt klassi juhistele. Üks lihtne viis selgeks teha klassi ja objekti erinevus, on vaadata klassi kui juhendit mingi eseme ehitamiseks ja objekt on siis valmis tehtud objekt. Kasutades objektorienteeritud programmeerimist saame eraldada erinevad funktsionaalsused ning muuta kood palju kergemini muudetavaks.

Klasse märgitakse võtmega *class* millele järgneb klassi nimi ning see järel loogelised sulud. Sulgude sisse läheb klassi kogu loogika. Klassi sees asuvad klassi liikmed mis on muutujaid ja funktsioone, mida kutsutakse meetoditeks. Nii muutujatel kui ka funktsioonidel saame ka täpsustada kas nad on saadaval ka väljaspool klassi või mitte kasutades võtmeid *private***,** *public* ja *protected.* Kui neid võtmeid ei kasuta, siis on see sama hea kui kasutada *public* võtit. Reeglina on muutujatel alati ees *private,* sest me ei taha, et mõni väline klass neid saaks otse muuta ja kasutada. Nende kätte saamiseks tehakse *getter* ja *setter* funktsioonid.

Proovime ise ka järele ja loome uue TypeScript faili nimega *Person.ts* ja loome seal sees klassi nimega Person.

```
class Person {}
```

Loome kaks muutujat klassi sees. Muutujate deklareerimisel klassi sees me ei saa kasutada ei *const* ega *let* võtmeid. Loome muutuja *age,* andmetüübiga *Number* ja muutuja *name*, andmetüübiga *String*. Mõlemad on privaatsed ja mõlemale me algväärtuseid ei anna.

```
  private age: Number;
  private name: String;
```

Nende väärtused saame me kui me objekti loome, ning nende väärtuste kätte saamiseks peame looma peale meie klassi muutujaid meetodi nimega *constructor*. Selle meetodi parameetriteks paneme *age,* andmetüübiga *Number* ja *name,*

andmetüübiga *String* ning meetodi sees anname parameetri *age* väärtuse meie klassi liikmele *age* ja sama ka teise muutujaga. Peame siin tähele panema, et kui me tahame kasutada klassi liikmeid klassi sees, siis peame enne muutuja nime kirjutama *this* siis punkt ja peale seda muutuja nimi

```
constructor(age: Number, name: String) {
  this.age = age;
  this.name = name;
}
```

Nüüd on meie klassi muutujatel väärtused olemas, aga meil oleks vaja ka neid väljaspool klassi kätte saada. Seda saame teha luues funktsioonid, mille ainus otstarve on tagastada meile nende muutujate väärtused. Neid kutsutakse *getter* funktsioonid. Reeglina on nende nimeks lihtsalt *get* ja muutuja nimi.

```
getAge(): Number {
  return this.age;
}

getName(): String {
  return this.name;
}
```

Kui me soovime ka meie klassi liikmeid väljaspool klassi muuta, siis on meil vaja ka luua funktsioonid, mille ainsaks otstarbeks on võtta väärtus ja anda see vastava muutuja väärtuseks. Selliseid funktsioone kutsutakse *setter* funktsioonideks. Nimetamisel on neil sarnane reegel *getter* funktsioonidele ainult, et *get* asemel on *set*.

```
setAge(val: Number) {
  this.age = val;
}

setName(val: String) {
  this.name = val;
}
```

Nüüd kui meil on klass ehk juhend olemas saame luua objekte, mis töötavad nagu juhendis on kirjas. Objektide loomine on väga sarnane tavalise muutuja loomisega. Me kasutame võtit *const,* anname sellele nimetuse ning see järel paneme võrduma mingi väärtusega. Selleks väärtuseks on meie klass, aga selle ette peab käima *new,* sest

me loome uue objekti. Klassi nimetuse taha loome sulud, ning sulu sees anname need parameetrid, mis me defineerisime meie *constructor* funktsiooni sees.

```
const person = new Person(5, "Juku");
```

Meil on objekt loodud ning me saame seda kasutada. Selleks, et objekti siseseid funktsioone välja kutsuda peame alguses kirjutama meie objekti nime, siis punkti ja alles siis funktsiooni nime. Loome ühe muutuja mille väärtuseks anname meie uue loodud inimese nime ja kasutame *console.log()* funktsiooni selleks et seda välja kuvada. Nagu käsureas näeme, siis saime kätte nime "Juku".

```
const personName = person.getName();
console.log(personName);
```

Sellega on meil kõik vajalik teada, et hakata objekte kasutama ka mängude tegemisel. Kindlasti kui nüüd mõelda meie eelmise mängu peale võime näha, et seal oleksid objektid vägagi kasuks tulnud koodi lihtsustamiseks.

ÜLESANNE: Proovige ise luua klass auto millel on klassi liikmeteks mõned muutujad ja meetodid. Kutsuge neid meetodeid välja ja proovige ka objekti siseste muutujate väärtuseid muuta.

Teine mäng: snake

Meie teiseks mänguks, mis me koostame tuleb klassikaline ussimäng. Mängu eesmärk on lihtne: mängija on uss, uss kasvab süües toitu, mis tekib suvaliselt väljakule ja uss ei tohi puutuda ei enda keha ega seinasid.

Loome uue kausta ja avame selle oma tekstiredaktoris ja ka terminalis. Meil tuleb enne arendamise alustamist installida mõned lisa paketid, mida me kasutame selleks, et abistada meid arenduse juures. Veenduge, et olete terminalis õige kausta juures. Terminalis sisestame ühe käskluse, millega installime kolm paketti. Need kolm paketti on meil vajalikud selleks, et me saaksime oma koodi mitme faili peale jaotada, aga kuna veebibrauserid ei ole võimelised veel kasutama süntaksi, mida me selle jaoks kasutame, siis on vaja need failid üheks failiks kokku kompileerida.

```
npm i ts-loader webpack webpack-cli
```

Kui käsk on oma töö lõpetanud näeme, et meile on kausta tekkinud mitu uut faili ja ka üks kaust. Alustame kaustast, mis meile loodi. *Node_modules* kaust luuakse *npmi* poolt ning seal hoitakse meie installitud paketid ja nende poolt vaja minevad paketid. Meie seda kausta ei puudu. *Package-lock.json* fail hoiab andmeid kõikide installitud pakettide kohta. See on eriti kasulik, kui projekti kallal töötab mitu inimest, sest see tagab, et kõigil on kasutusel sama versiooniga paketid. Ka seda meie ei puudu.

Package.json hoiab informatsiooni, meie projekti kohta. Hetkel on seal näha vaid meie installitud pakette *dependencies* all. Meil on vaja lisada ka *scripts* osa, kuhu me saame lisada meie enda skriptid, mis käivitavad meie poolt installitud paketid. Kopeerime all oleva versiooni *package.json* failist ja asendame meie praeguse faili sisu sellega.

```
{
  "scripts": {
    "build": "webpack",
    "dev": "webpack -w"
  },
  "dependencies": {
    "ts-loader": "^9.2.6",
    "webpack": "^5.68.0",
    "webpack-cli": "^4.9.2"
  }
}
```

Lisasime *scripts* alla *dev* skripti. Selle skripti käivitame enne kui hakkame arendama, ning *webpack* hakkab jälgima meie projekti kausta ning igakord kui see näeb muudatusi, siis kompileerib see meie koodi. Seega ei pea me enam ise koodi kompileerima.

Selleks, et *webpack* töötaks nii nagu meil on tarvis, on meil vaja luua konfiguratsiooni fail. Selleks loome uue faili nimega *webpack.config.js* ja avame selle. Kopeerime all oleva näite sisu, ning lisame selle meie värskelt loodud faili sisse.

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

module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: {
    main: "./src/main.ts",
  },
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: "script.js"
  },
  resolve: {
    extensions: [".ts", ".js"],
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        loader: "ts-loader"
      }
    ]
  }
};
```

Lisaks *webpacki* konfiguratsiooni failile on meil seekord vaja ka TypeScripti enda konfiguratsiooni faili, mille nimi on *tsconfig.json*. Loome selle faili ja kopeerime all oleva koodilõigu sisu selle faili sisse. See fail ütleb TypeScriptile kuidas koodi peaks kompileerima ja mis reegleid kompilaator järgima peaks. Meie eelmises mängus meil seda vaja ei läinud, sest kasutasime vaikimisi seatud reegleid, aga seekord vajab seda just *webpack*.

```
{
```

```
  "compilerOptions": {
    "target": "es2017",
    "module": "commonjs",
    "lib": ["dom", "es6", "es2017", "esnext.asynciterable"],
    "skipLibCheck": true,
    "sourceMap": true,
    "outDir": "./dist",
    "moduleResolution": "node",
    "removeComments": true,
    "noImplicitAny": true,
    "strictNullChecks": false,
    "strictFunctionTypes": true,
    "noImplicitThis": true,
    "noUnusedLocals": false,
    "noUnusedParameters": false,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "allowSyntheticDefaultImports": true,
    "esModuleInterop": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "resolveJsonModule": true,
    "baseUrl": "."
  },
  "exclude": ["node_modules"],
  "include": ["./**/*.ts"]
}
```

Kõige viimasena peame tegema projekti kaustas uue kausta nimega *src.* Selles kaustas hakkab olema kogu meie mängu kood. Hoida kood eraldi lähtekoodi kaustas on hea tava, eriti kui projektil on palju erinevaid konfiguratsiooni faile. See hoiab meie projekti puhtamana ja hiljem on kergem tagasi tulla ja leida õiged failid õigetes kohtades.

Loome *src* kausta sees faili *main.ts* ja kirjutame sinna klassikalise *Hello World* näite. Just seda faili kasutatakse *webpacki* poolt meie sisend punktina ehk nagu ka ta nimi meile ütleb on tegu meie projekti põhi failiga. Proovime oma projekti nüüd ehitada. Me eelnevalt lisasime oma projekti info faili kaks skripti: build ja dev. Proovime kasutada *build* skripti ning vaatame mis juhtub. Me näeme, et tekkis uus kaust nimega *dist,* mille sees on üks fail nimega *script.js*. See on meie fail, mida veebibrauserid lugema hakkavad.

```
npm run build
```

HTML

Nagu meie eelmine mäng, hakkab ka meie uus mäng töötama veebibrauserites kasutades HTML-i. Õnneks saame kasutada ära meie eelmise mängu HTML-i saja protsendiliselt. Seega kopeerime oma eelmise mängu kaustast meie uue mängu kausta *index.html* faili. Avame selle oma tekstiredaktoris, sest meil on vaja teha mõned muudatused. Eelnevalt tehti meile kompileeritud fail otse projekti kausta, kuid seekord tekib meie kompileeritud fail projekti kaustas olevasse *dist* kausta. Seega peame muutma täpset teed skripti failini, mis on kirjeldatud meie HTML failis. Asendame *"./index.js" "./dist/script.js"-ga.* Rohkem me ei pea HTML faili puutuma. Igaks juhuks kontrollige üle, et teie HTML fail vastab all olevale kooditükile.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Game 1</title>
  <script src="./dist/index.js" defer></script>
</head>
<body>
  <canvas class="canvas" id="canvas" width="1280" height="720"></canvas>
</body>
</html>
```

Põhi klass

Asume nüüd koodi kirjutamise juurde. Esimese asjana avame *main.ts* faili ja kustutame meie *Hello World* näite ära. Seejärel loome siia uue klassi nimega *Main*. Loome talle tühja *constructor* meetodi ja ka *run* meetodi, mille sees prindime konsooli välja *"running"*. Sellest klassist saab meie mängu keha. Siin klassi sees loome kõik vajalikud objektid, mida on vaja selleks, et mäng funktsioneeriks. Samuti läbi selle klassi pannakse mäng tööle ja hallatakse selle tööd.

```
class Main {
  constructor() {}

  run() {
    console.log("running");
  }
}
```

Kutsume ka selle klassi siin samas failis kohe välja ka. Loome uue muutuja nime *main* (siin tähelepanna, et see muutuja peab olema klassist väljas ja peale klassi, muidu võib tekkida veateateid) ja anname talle väärtuseks uue *Main* objekti. Peale seda uuel real kutsume välja *Main* objekti *run* meetodi.

Nüüd selleks, et koodi kompileerida ei kasuta me seekord *build* skripti, vaid kasutame *dev* skripti. Näeme, et meie kood kompileeritakse ja käsk jääb ootama ning jälgima muudatusi meie projekti kaustas. Seega ei pea me rohkem seda näppima. Avame meie HTML faili oma veebibrauseris ning vaatame konsooli. Näeme, et seal on prinditud meie kompileeritud faili poolt *"running"*.

Canvas

Loome uue faili meie *src* kaustas nimega *Canvas.ts* ja loome sinna sisse uue klassi nimega *Canvas*. Selle klassi sisu tuleb väga sarnane meie eelmise mängu joonistamise funktsioonidele. Loome klassile kaks muutujat: *canvas*, mille andmetüübiks on HTMLCanvasElement ja *ctx,* mille andmetüübiks on CanvasRenderingContext2D. Tundub ju tuttav. Need on täpselt samad muutujad, mis me lõime oma eelmises mängus, ainult et nüüd anname neile väärtused alles *constructor* funktsiooni sees. Muutujatele *constructor* funktsiooni sees väärtuste andmine on hea tava, sest siis on kindlus, et kõik toimub objekti loomise ajal. Muidugi on sellel erandeid nagu näiteks mõned kindlad väärtused lihtsate andmetüüpidega (number, string, boolean).

```
  private canvas: HTMLCanvasElement;
  private ctx: CanvasRenderingContext2D;
```

Loome siis ka *constructor* meetodi millel on üks parameeter: id, mille andmetüübiks on string. See id on meie *canvas* elemendi id, mis peab vastama sellega, mis meie HTML failis on. Meetodi sees anname mõlemale muutujale väärtused samamoodi nagu me andsime meie esimeses mängus, seega kui ei ole meeles saame korra visata pilgu sinna peale. Ainus erinevus on siin see, et id tuleb *constructor* meetodi parameetrist.

```
  constructor(id: string) {
    this.canvas = document.getElementById(id) as HTMLCanvasElement;
    this.ctx = this.canvas.getContext('2d');
  }
```

Järgmisena tuleks meil siia ka üle tuua meie *drawRectangle* ja *clearCanvas* funktsioonid, sest meil läheb ka neid uuesti tarvis. Siin kohas tuleb tähele panna, et nende sisu tuleb muuta vastavaks klassi kirjapildile.

```
drawRectangle(x: number, y: number, width: number, height: number, color: string) {
  this.ctx.fillStyle = color;
  this.ctx.fillRect(x - width / 2, y - height / 2, width, height);
}

clearCanvas() {
  this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
  this.drawRectangle(
    this.canvas.width / 2,
    this.canvas.height / 2,
    this.canvas.width,
    this.canvas.height,
    'black'
  );
}
```

Viimasena tuleks meil ka lisada kaks meetodit, mis tagastaksid *canvas*e pikkuse ja laiuse. Meil küll ei ole eraldi klassi liikmeid selle jaoks, aga meil on võimalus nad kätte saada läbi meie *canvas* elemendi viite. Sellega on meil *Canvas* klassi põhi valmis ning me saame seda kohe kasutusse võtta.

```
getWidth(): number {
  return this.canvas.width;
}

getHeight(): number {
  return this.canvas.height;
}
```

Klassi sisu on meil nüüd olemas, aga meil on vaja ka teha see klass saadavaks teistest failidest. Seda saab teha eksportides. Faili lõppu loome uue rea ja seal määrame selle faili vaikimisi ekspordi, milleks on meie loodud klass. Seda saame teha kasutades võtit *export default* millele järgneb klassi nimi. Ühes failis saab olla vaid üks vaikimisi eksporditav liige ning see võib olla nii muutuja, funktsioon või klass.

```
export default Canvas;
```

Pildi joonistamine

Avame jälle *main.ts* faili ning teeme siin mõned lisad. Enne kui saame alustada oma loodud klassi kasutamist, peame importima selle siia faili. Seda saame teha faili kõige alguses. Loome uue rea kohe faili alguses enne *Main* klassi. Selleks, et importida midagi teisest failist saame kasutada *import* võtit, millele järgneb selle asja nimetus, mida me impordime (meil oleks see *Canvas),* siis võti *from,* millele järgneb tee failini, millest soovime midagi importida.

```
import Canvas from './Canvas';
```

Loome *Main* klassis uue muutuja nimega *canvas* ning anname sellele andmetüübiks meie *Canvas* klassi. Seejärel konstruktori sees anname sellele muutujale väärtuseks uue *Canvas* objekti, mille parameetriks on meie *canvas* elemendi id. Viimasena kustutame *run* meetodis konsooli printimise käsu ära ning selle asemele kutsume välja meie *canvas* objekti *clearCanvas* meetodi. Kui avame nüüd HTML faili, siis näeme, et seal vaatab meile vastu tuttav must kast.

```
class Main {
  private canvas: Canvas;

  constructor() {
    this.canvas = new Canvas('canvas');
  }

  run() {
    this.canvas.clearCanvas();
  }
}
```

Loome *Main* klassi uue meetodi nimega *draw*. See meetod hakkab töötama sama moodi nagu töötas meie eelmise mängu *draw* funktsioon: kutsutakse välja vajalikud funktsioonid ja see järel kasutatakse *requestAnimationFrame* funktsiooni. Tuttav ja lihtne.

ÜLESANNE: Enne kui lahendust vaatate, proovige ise luua *Main* klassi eelnevalt kirjeldatud meetod ning uuendage *run* meetodit nii, et ainus asi mis seal toimub on *draw* meetodi välja kutsumine.

```
  draw() {
```

```
    this.canvas.clearCanvas();
    requestAnimationFrame(() => this.draw());
  }
```

Ruudustik

Erinevalt meie eelmisest mängust, kus pall sai liikuda pea igal koordinaadil, hakkab meie mänguväljak seekord koosnema ruudustikust, mis määrab kus mänguobjektid saavad olla. Selleks, et seda ruudustikku endale kergemini visualiseerida teeme ühe meetodi *Canvas* klassi, mis joonistab välja meie mängu ruudustiku.

Enne kui saame ruudustiku joonistamise meetodi luua, loome endale ühe abi funktsiooni joonte joonistamise jaoks *canvase* peal. Joone joonistamine töötab sarnaselt ristküliku joonistamisele, aga sellel on mõned erinevused. Joone joonistamiseks peab ära märkima punktid mida joon läbib. Punkte saab olla rohkem kui kaks seega on võimalik sellega päris keerukaid jooni tekitada. Meie kasutusjuhuks läheb tarvis vaid kahte paari koordinaate: alg- ja lõppkoordinaadid.

Loome meetodi nimega *drawLine*. Sellel meetodil on viis parameetrit: *startX,* andmetüübiga number, *startY,* andmetüübiga number, *endX,* andmetüübiga number, *endY* andmetüübiga number ning *color,* andmetüübiga string ning vaikeväärtusega *"#ffffff",* mis on valge. Värvi saame joone jaoks sättida sarnaselt ristkülikule, aga seekord peame andma väärtuse *ctx.strokeStyle* atribuudile. Selleks, et alustada joone joonistamist, peame välja kutsuma *ctx.beginPath* funktsiooni ja seejärel panema paika meie joone algpunkti kasutades *ctx.moveTo* funktsiooni, mille parameetriteks on *startX* ja *startY.* Selleks, et lõpp koordinaadid sättida saame kasutada *ctx.lineTo* funktsiooni, mille parameetriteks on *endX* ja *endY.* Nüüd kui punktid paigas on veel jäänud kutsuda välja *ctx.stroke()* funktsioon, mis joonistab läbi meie antud punktide joone *canvas*e peale.

```
drawLine(startX: number, startY: number, endX: number, endY: number, color: string = '#ffffff') {
  this.ctx.strokeStyle = color;
  this.ctx.beginPath();
  this.ctx.moveTo(startX, startY);
  this.ctx.lineTo(endX, endY);
  this.ctx.stroke();
}
```

Me peame ka ära otsustama kui suured ruudud meie ruudustikus on. Lähme kiirelt tagasi *Main* klassi juurde ning lisame sinna uue muutuja nimega *cellSize* andmetüübiga number ja anname talle kohe väärtuseks 20. Siin on tegu ühe sellise muutujaga, mille puhul ei ole vahet kas väärtus antakse talle kohe või objekti loomise ajal. Loome ka sama muutuja *Canvas* klassis, aga sellele väärtust kohe ei anna. Selle väärtuse saame *constructor* meetodi läbi. Lisame *constructor* meetodile uue parameetri mille nimeks on ka *cellSize* ning anname selle väärtuse *Canvas* klassis olevale *cellSize* muutujale.

```
private cellSize: number;

constructor(id: string, cellSize: number) {
  this.canvas = document.getElementById(id) as HTMLCanvasElement;
  this.ctx = this.canvas.getContext('2d');
  this.cellSize = cellSize;
}
```

Loome ka *cellSize* muutuja jaoks *getter* funktsiooni, aga mitte *setter* funktsiooni, sest me ei taha, et see väärtus muutuks.

```
getCellSize(): number {
  return this.cellSize;
}
```

Saame nüüd liikuda oma abijoonte joonistava meetodi loomise juurde. Anname meetodile nimeks *drawHelperLines* ning loome selle funktsiooni sees kaks for-tsüklit. Esimesega käime mööda *canvase* laiust ning *cellSize* suuruste vahedega joonistame ülevalt alla jooned. Teise tsükliga käime mööda *canvase* kõrgust ning ka siin *cellSize* suuruste vahedega joonistame seekord horisontaalselt jooned. Joonte joonistamiseks kasutame loomulikult meie eelnevalt loodud *drawLine* meetodit.

ÜLESANNE: Enne kui lahendust vaatate, proovige ise implementeerida see meetod nii et see vastaks kirjeldusele.

```
drawHelperLines() {
  for (let i = this.cellSize; i <= this.canvas.width; i += this.cellSize) {
    this.drawLine(i - 1, 0, i - 1, this.canvas.height);
  }
  for (let i = this.cellSize; i <= this.canvas.height; i += this.cellSize) {
    this.drawLine(0, i - 1, this.canvas.width, i - 1);
```

```
    }
  }
```

Viimasena kutsume selle meetodi välja *Main* klassi *draw* meetodis.

```
this.canvas.drawHelperLines();
```

Mänguobjektid

Liigume nüüd mänguobjektide implementeerimise juurde. Loome uue faili nimega *GameObject* ja selle sees loome ka sama nimelise klassi. Sellest klassist saab meie mänguobjektide alusklass, mis toob nad kõik ühele standardile. Seda kutsutakse pärilikuseks.

Loome *GameObject* klassis kolm muutujat: x, y ja *canvas.* Erinevalt tavalistest klassidest ei käi nende ette *private,* vaid hoopid *protected. Protected* võti töötab sarnaselt *private* võtmele, aga lubab kasutada märgitud klassi liikmeid klassidel, kes pärivad teda. *Private* seda ei võimalda. Loome ka siin *constructor* meetodi, mille parameetriteks on *canvas,* andmetüübiga *Canvas, x,* andmetüübiga number ja *y,* andmetüübiga number. Anname need väärtused edasi meie klassi liikmetele. Siin ei tohi ära unustada lisada *Canvas* klassi import lause faili algusesse.

Kui muutujad olemas, loome ka mõned meetodid, mida kõik *GameObject* tüüpi objektid hakkavad jagama. Loome x ja y muutujate jaoks *getter* ja *setter* meetodid ning loome ühe tühja *draw* meetodi. Draw funktsioon jääb tühjaks, sest eri mängu objektidel võivad olla erinevad nõuded nende välja kuvamise jaoks.

Kõige viimasena lisame eksport lause selleks, et saaksime seda klassi välistes failides ka kasutada.

```
import Canvas from './Canvas';

class GameObject {
  protected x: number;
  protected y: number;
  protected canvas: Canvas;

  constructor(canvas: Canvas, x: number, y: number) {
    this.canvas = canvas;
    this.x = x;
```

```
    this.y = y;
  }

  getX() {
    return this.x;
  }

  getY() {
    return this.y;
  }

  setX(val: number) {
    this.x = val;
  }

  setY(val: number) {
    this.y = val;
  }

  draw() {}
}

export default GameObject;
```

Mängija

Loome nüüd meie esimese mängu objekti ja selleks võiks olla mängija objekt. Loome uue faili nimega *SnakeHead.ts* ning loome seal sees sama nimelise klassi. Selles klassis tuleb meil üks lisa samm veel teha. Me peame pärima *SnakeHead* klassiga *GameObject* klassi. Seda saame teha kui peale klassi nimetust kirjutame võtme *extends* ja seejärel *GameObject.* Kui see tehtud, siis meie *SnakeHead* klass pärib nüüd *GameObject* klassi liikmeid ning vajadusel saame me neid ümberkirjutada.

```
class SnakeHead extends GameObject {

}
```

Nüüd loome *constructor* meetodi *SnakeHead* klassi jaoks. Algus on sama mis teiste klassidega. Meil on hetkel *SnakeHead* klassis kolm muutujat, mis me pärime *GameObject* klassilt. Lisame nende kolme muutuja jaoks parameetrid *constructor* meetodis, aga kuna meil on *GameObject* klassi *constructor* meetodis juba muutujate väärtuste omastamine tehtud, siis saame kasutada sellist funktsiooni nagu *super,* mille parameetriteks anname need kolm parameetrit, mida *GameObject* klassi *constructor*

meetod vajab. Selle funktsiooniga kutsume me välja ka alamklassi *constructor* meetodi.

```
constructor(canvas: Canvas, x: number, y: number) {
  super(canvas, x, y);
}
```

Enne kui *draw* funktsiooni funktsionaalsuse implementeerime lisame *Canvas* klassi ühe lisafunktsionaalsuse. Meie mäng koosneb ruutudest, seega enamus joonistus käsklusi hakkavad ka joonistama ruute. Loome siin uue meetodi nimega *drawCell* ning anname sellele kolm parameetrit: *x, y* ja *color.* Siin funktsiooni sees me ei tee midagi rohkemat, kui et me kutsume välja *drawRectangle* meetodi ning anname oma parameetrid edasi, aga selle meetodi *width* ja *height* parameetrite väärtused on *Canvas* klassi *cellSize* muutuja väärtus. Selle meetodiga säästame endale aega sellega, et me ei pea igakord küsima *cellSize* suurust, kui soovime kedagi joonistada.

```
drawCell(x: number, y: number, color: string) {
  this.drawRectangle(x, y, this.cellSize, this.cellSize, color);
}
```

Lähme tagasi *SnakeHead* klassi juurde ja loome *draw* meetodi. Meie alamklassis, *GameObject,* on juba sellenimeline meetod olemas, aga see on tühi ehk see ei tee mitte midagi. Kui me pärime mingit klassi teise klassiga, siis selle klassi sees, kes pärib alamklassi on võimalik alamklassis olevad meetodid üle kirjutada. Meetod millega me üle kirjutame peab olema sama nimeline ja ta parameetrid peavad olema vastavad alamklassis olevale meetodile. Meie *SnakeHead* klassi *draw* meetodi sisse tuleb ainult üks rida. Me kutsume siin välja *Canvas* klassi *drawCell* meetodi, mille me just tegime, ning anname selle meetodi parameetrite väärtuseks *SnakeHead* klassi x ja y väärtused koordinaatideks ning värviks anname valge, ehk "#ffffff".

```
draw() {
  this.canvas.drawCell(this.x, this.y, '#ffffff');
}
```

Kui see tehtud lisame ka faili lõppu eksport lause ning lähme täiendame *Main* klassi. Loome *Main* klassis uue muutuja nimega *gameObjects* mille andmetüübiks on *GameObject* ahel. Anname talle ka kohe väärtuseks tühja ahela. Me anname sellele ahelale tüübiks *GameObject,* sest me tahame kõik mängu objektid kiirelt läbi käija

ning nende kõigi *draw* meetodid välja kutsuda. Seda on kerge teha kui nad on kõik kusagil ühes kohas koos.

```
private gameObjects: GameObject[] = [];
```

Loome uue meetodi nimega *initGameObjects* ja selle sees lükkame meie *gameObjects* ahela sisse uue *SnakeHead* objekti. Anname talle parameetrites edasi *Main* klassi *canvas* muutuja ja koordinaatideks mõlemad 10. Ärge unustage *SnakeHead* klass importida.

```
  initGameObjects() {
    this.gameObjects.push(new SnakeHead(this.canvas, 10, 10));
  }
```

Viimasena on meil vaja *gameObjects* ahel läbi käija *draw* meetodis. Selleks saame kasutada meile tuntud for-tsüklit, aga see kord kasutame midagi targemat. TypeScriptis on peale tavalise for-tsükli olemas ka mitu mugavtatud versiooni. Üks nendest on *forEach* funktsioon, mis on sisse ehitatud ahelate sisse. Kutsume peale *drawHelperLines* meetodit välja *gameObjects.forEach* funktsiooni *draw* funktsiooni sees. Selle funktsiooni parameetriks on noolfunktsioon. Nool funktsioonid on kompaktsed alternatiivid klassikalistele funktsioonidele. Meie *forEach* funktsioon tagastab iga tsükliga ühe elemendi meie ahelast. Noolfunktsioon selle sees võtab selle elemendi kui parameetrina ning seejärel funktsiooni sees on võimalik seda elementi kasutada. Meie puhul, me kutsume noolfunktsiooni sees välja *GameObject* klassi *draw* meetodi.

```
    this.gameObjects.forEach(gameObject => {
      gameObject.draw();
    });
```

Selleks, et mängijat lõpuks näha on meil vaja *run* meetodi sees enne *draw* meetodi välja kutsumist välja kutsuda *initGameObjects* meetod, sest me tahame, et kõik mängu objektid oleksid valmis kasutamiseks enne, kui mäng tööle läheb. Kui nüüd avame HTML lehe näeme üleval vasakul nurgas valget ruutu. See on meie mängija.

Enne kui siin lõpetame, liigutame mängija eraldi muutuja sisse *Main* klassi sees. Loome muutuja *player* andmetüübiga *SnakeHead* ning anname talle *constructor* meetodi sees väärtuseks selle väärtuse, mille me *initGameObjects* meetodi sees

gameObjects ahela sisse lükkasime. Lisaks *initGameObjects* meetodis vahetame uue objekti ahelasse lükkamise ära *player* muutuja ahelasse lükkamisega.

```
private player: SnakeHead;

constructor() {
  this.canvas = new Canvas('canvas', this.cellSize);
  this.player = new SnakeHead(this.canvas, 10, 10);
}
```

Liikumine

Paneme nüüd selle ruudu ka liikuma. Alustame nagu ikka uue faili loomisega nimega *KeyboardHandler.ts* ja loome seal sama nimelise klassi. Siin klassi sees hakkab toimima kogu meie klaviatuuri loogika. Üks põhilisi objektorienteeritud programmeerimise põhimõtteid on hoida erinev loogika eraldi klassides. See hoiab klasside suurused väiksed ning muudab koodi kergesti muudetavaks, kui muutus on vaja teha vaid ühes kindlas kohas.

Loome *KeyboardHandler* klassi sees uue muutuja *activeKey* ja anname talle andmetüübiks string või *null.* Me tahame et meie uss mängu alguses ei liiguks, seega peab *activeKey* vaikeväärtus olema *null.* Loome ka *constructor* meetodi ning selle sees anname *activeKey* muutujale väärtuseks *null.*

ÜLESANNE: Seda kuidas kasutaja klaviatuuri vajutusi kätte saada oleme juba eelmises mängus teinud. Vaata eelmise mängu koodi ning ürita valmis teha see klass nii, et siin kuulataks mängija nupuvajutusi ning viimasena vajutatu nupuvajutus salvestataks *activeKey* muutujas. Lisaks oleks meil vaja ka meetodit mis tagastaks *activeKey* väärtuse. Siin tuleb tähele panna, et klassi meetodeid ei saa otse kuulajate sisse panna, nad tuleb panna nool funktsioonide sisse.

Vaatame kuidas see lahendus võiks välja näha. Loome meetodi nimega *keyDownListener* millel on parameetriks *event KeyboardEvent* andmetüübiga. Siin funktsiooni sees me ei tee midagi rohkemat, kui et me paneme *activeKey* muutuja võrduma *event.key* väärtusega. Loome ka meetodi *getActiveKey* ja selle meetodi sees me tagastame *activeKey* muutuja.

```
keyDownListener(event: KeyboardEvent) {
  this.activeKey = event.key;
}

getActiveKey(): string | null {
  console.log(this.activeKey);
  return this.activeKey;
}
```

Selleks, et kuulata kasutaja klaviatuuri sisendit kasutasime eelnevalt *document.addEventListener* funktsiooni. Seda saame kasutada ka seekord. Seekord me kutsume selle funktsiooni välja *constructor* meetodi sees. Nagu eelnevalt mainisin, ei ole võimalik klassi meetodeid otse kuulajate sees kasutada. Seega peame kasutama noolfunktsiooni, mille argumendiks on *e* ja selle anname edasi *keyDownListener* meetodile. Veidi teistmoodi kui eelmine kord, aga midagi uut siin ei ole. Lisame faili lõppu ka eksport lause.

```
constructor() {
  this.activeKey = null;
  document.addEventListener('keydown', e => this.keyDownListener(e));
}
```

Selleks, et see klass kasutusele võtta, loome *Main* klassis uue muutuja nimega *keyboardHandler* andmetüübiga *KeyboardHandler (ärge unustage seda importida)* ja *Main* klassi *constructor* meetodis enne mängija loomist paneme *keyboardHandler* muutuja võrduma uue *KeyboardHandler* objektiga. Kasutaja klaviatuuri sisendit hakkab hetkel kasutama ainult *SnakeHead* klass, aga on hea olla valmis tuleviku jaoks, kus võibolla ka mõni teine klass hakkab seda kasutama.

```
private keyboardHandler: KeyboardHandler;
```

SnakeHead klassi sees on meil ka vaja mõned muudatused teha. Impordime ka siin sisse *KeyboardHandler* klassi. Loome ka *SnakeHead* klassi sees muutuja nimega

keyboardHandler ja ka sellele anname andmetüübiks *KeyboardHandler*. Lisame *constructor* meetodi parameetrite hulka peale *canvas* parameetrit uue parameetri *keyboardHandler* ja anname selle väärtuse edasi klassi muutujale nimega *keyboardHandler.* Nüüd saab *SnakeHead* klass kasutada kasutaja sisendit.

```
  private keyboardHandler: KeyboardHandler;

  constructor(canvas: Canvas, keyboardHandler: KeyboardHandler, x: number, y: number)
{
    super(canvas, x, y);
    this.keyboardHandler = keyboardHandler;
  }
```

Peale *SnakeHead* klassi *constructori* muutmist peame muutma *Main* klassis ka *SnakeHead* objekti loomist. Hetkel on meil antud kolm parameetrit, aga nüüd vajab *SnakeHead* nelja parameetrit. Anname peale *canvas* parameetrit edasi ka *keyboardHandler* muutuja.

```
    this.player = new SnakeHead(this.canvas, this.keyboardHandler, 10, 10);
```

Saame nüüd lõpuks ussi liikuma panna. Loome *SnakeHead* klassis uue meetodi nimega *move* ning seal sees paneme uue muutuja nimega *activeKey* sisse *KeyboardHandler* klassi *getActiveKey* meetodi poolt tagastatud väärtuse.

ÜLESANNE: *move* meetodi sees tuleb meil kasutada jälle *switch* lauset täpselt sama moodi nagu meie eelmises mängus. Seekord on liikumis kiirus määratud mängu ruutude suuruse poolt ehk tuleb kasutada *Canvas* klassi *getCellSize* meetodit.

LAHENDUS: Vaatame milline võiks välja näha selle lahendus. Meil on *switch* lause, mille parameetriks on *activeKey* muutuja. Vastavalt aktiivsele nupule muudame ussi suunda. Seetähendab, et me peame muutma kas x või y väärtust.

```
move() {
  const activeKey = this.keyboardHandler.getActiveKey();

  switch (activeKey) {
    case 'w':
      this.y -= this.canvas.getCellSize();
      break;
    case 's':
      this.y += this.canvas.getCellSize();
      break;
    case 'd':
      this.x += this.canvas.getCellSize();
      break;
    case 'a':
      this.x -= this.canvas.getCellSize();
      break;
  }
}
```

Nüüd on jäänud meil veel lisada liikumine *Main* klassi *run* funktsiooni. Õnneks siin midagi keerulist ei ole. *Main* klassi *run* funktsioonis kutsume välja *SnakeHead* klassi *move* meetodi enne mängu objektide joonistamist. Me tahame, et kogu loogika tehakse enne joonistamist ära. Kui nüüd avame oma HTML faili, siis näeme et kui vajutame meie defineeritud nuppe, siis meie ruut tõesti liigub mööda ekraani ringi.

Hetkel liigub meie ruut väga kiirelt ringi. Iga kord kui *requestAnimationFrame* väljakutsutakse, siis tehakse uus joonistus. See oli hea meie eelmises mängus, aga traditsiooniliselt on ussimängus väikesed pausid peale igat liigutust. Õnneks midagi sellist implementeerida ei ole meil raske. Loome *Main* klassis uue muutuja *timeSinceLastFrame* ja anname sellele andmetüübiks *number.* Sellele muutujale me väärtust ei anna, isegi mitte *constructor* meetodis. Nüüd peame muutma *Main* klassi *draw* meetodit. Me peame kogu funktsiooni sisu peale *requestAnimationFrame* funktsiooni ümbritsema ühe *if* lausega, mis laseb mängu loogikal töötada ainult juhul, kui viimasest kaadrist on möödas rohkem kui 100 millisekundit või kui *timeSinceLastFrame* muutujal ei ole väärtust. Me peame ka lisama *if* lause sees viimasele reale *timeSinceLastFrame* muutujale praeguse ajahetke väärtuse andmise

kasutades *Date.now* funktsiooni. Kasutasime ka sarnast loogikat meie eelmises mängus ehk see peaks meile juba tuttav olema.

ÜLESANNE: Enne kui vaatate lahendust, proovige ise implementeerida eelnevalt kirjeldatud funktsionaalsus.

LAHENDUS: Vaatame milline võiks välja näha lahendus sellele probleemile. Me saame lahendada seda ühe *if* lausega kasutades *or* operaatorit. Selleks, et saada möödunud aega, peame *Date.now* funktsiooni poolt tagastatud väärtusest lahutama *timeSinceLastFrame* muutuja väärtuse. Siin tuleb tähele panna, et TypeScriptis saab kontrollida, kas muutujal on väärtus või mitte pannes muutuja ette hüüumärgi. Kui muutujal väärtust ei ole või väärtus on *false,* siis tagastatakse meile *true.*

```
if (!this.timeSinceLastFrame || Date.now() - this.timeSinceLastFrame > 100) {
  this.canvas.clearCanvas();
  this.canvas.drawHelperLines();
  this.player.move();
  this.gameObjects.forEach(gameObject => {
    gameObject.draw();
  });
  this.timeSinceLastFrame = Date.now();
}
```

Toit

Ussimängus uss kasvab toitu süües. Reeglina toit tekib suvalisse kohta väljaku peal ja kui uss sellest üle sõidab, siis süüakse see ära ja tekib uus saba tükk pea taha. Liigume edasi samm-sammu haaval ning alustame toidu ekraanile tekitamisega.

Enne kui loome toidu jaoks klassi, loome faili nimega *utils.ts.* Siin sees hakkavad olema funktsioonid, mis ei sobi ühegi klassi sisse ja mida võib mitmes erinevas kohas vaja kasutada. Selliseid funktsioone kutsutakse objektorienteeritud programmeerimises staatilisteks funktsioonideks.

Loome siin faili sees uue funktsiooni nimega *getRandomInt.* Tuttav nimi, sest see on täpselt see sama funktsiooni, mida me kasutasime oma eelmises mängus. Seega võib täiesti vabalt selle siia kopeerida. Meil tuleb teha üks väike muudatus. Me peame enne *function* võtit lisama veel ühe võtme nimega *export.* Kuna meie *utils.ts* faili võib tulla mitu erinevat funktsiooni, siis ei oleks siin õige kasutada vaikimiseksporti, vaid

eksportida kõik funktsioonid eraldi. Nende importimisel on teine kirjapilt, aga sellest räägime siis, kui meil seda funktsiooni vaja on.

Liigume nüüd toidu loomise juurde. Teeme uue faili nimega *Food.ts* ja loome seal sees samanimelise klassi, mis pärib *GameObject* klassi. Loome koheselt *constructor* meetodi ning selle sisu on hetkel väga sarnane *SnakeHead* klassi *constructor* meetodile. *Food* klassil on parameetriteks canvas, x ja y ning me anname need edasi kasutades *super* funktsiooni.

```
constructor(canvas: Canvas, x: number, y: number) {
  super(canvas, x, y);
}
```

Ka *Food* klassi *draw* meetod on väga sarnane *SnakeHead* klassi omale. Me kasutame ka siin *drawCell* meetodit *Canvas* klassist, aga seekord on värviks punane. Värve ei pea alati andma edasi värvikoodidena ning seekord *color* parameetriks anname “red“.

```
draw() {
  this.canvas.drawCell(this.x, this.y, 'red');
}
```

Viimasena on meil vaja luua meetod, mis tekitaks uued x ja y koordinaadid. Siin saame kasutada *utils* failis olevat *getRandomInt* funktsiooni. Enne seda peame tegema mõned muudatused *Canvas* klassis. Meil on vaja teada saada kui palju ruute meie ekraanil x teljel ja y teljel on. Selleks loome *Canvas* klassi kaks meetodit: *getXCellAmount* ja *getYCellAmount* mis tagastavad numbri.

ÜLESANNE: Proovige ise implementeerida *Canvas* klassis eelnevalt kirjeldatud meetodid.

LAHENDUS: Vaatame milline võiks olla lahendus sellele probleemile. Siin õnneks midagi rasket ei ole. Meil on kätte saadav *Canvas* klassi nii *canvas*e laius kui ka kõrgus ning meil on olemas ruudu suurus. Seega tuleb meil lihtsalt jagada laius ja kõrgus ruudu suurusega ja tagastada saadud väärtus.

```
  getXCellAmounts(): number {
    return this.canvas.width / this.cellSize;
  }

  getYCellAmounts(): number {
    return this.canvas.height / this.cellSize;
  }
```

Lähme tagasi *Food* klassi juurde ja impordime *utils* klassist *getRandomInt* funktsiooni. Kui proovime kasutada sama kirjapilti impordi jaoks *s* mida kasutame klassidega, siis meie kood ei kompileeri, sest *utils* failis puudub vaikimisi eksport, aga meie funktsioonil on eraldi eksport märgitud. Siin õnneks midagi keerulist ei ole. Me saame lisada import lauses *getRandomInt* ümber loogelised sulud ning meie import töötab nüüd. Juhul, kui soovime importida rohkem funktsioone *utils* failist, lähevad nende funktsioonide nimetused ka nende samade loogeliste sulgude sisse. Funktsiooni nimed tuleb eraldada komadega.

```
import { getRandomInt } from './utils';
```

Loome *Food* klassi sees meetodi *spawn* ning genereerime selle funktsiooni sees muutujate sisse uued x ja y ruudu koordinaadid. Me anname *getRandomInt* funktsiooni parameetriteks ühe ja vastavalt siis kas *getXCellAmount* või *getYCellAmount* meetodi *Canvas* klassist.

```
    const newXCellCoord = getRandomInt(1, this.canvas.getXCellAmounts());
    const newYCellCoord = getRandomInt(1, this.canvas.getYCellAmounts());
```

Neid numbreid me koordinaatidena kasutada ei saa veel, sest need numbrid annavad meile teada millise ruudu peale toit peab tekkima. Selleks, et saada täpsed koordinaadid peame korrutama saadud numbrid ruudu suurusega, mille saame kätte kasutades *Canvas* klassi *getCellSize* meetodit. Sellest arvust tuleb veel lahutada pool ruudu suurust.

```
  spawn() {
```

```
    const newXCellCoord = getRandomInt(1, this.canvas.getXCellAmounts());
    const newYCellCoord = getRandomInt(1, this.canvas.getYCellAmounts());
    this.x = this.canvas.getCellSize() * newXCellCoord - this.canvas.getCellSize() / 2;
    this.y = this.canvas.getCellSize() * newYCellCoord - this.canvas.getCellSize() / 2;
}
```

Lisame *constructor* meetodisse peale *super* funktsiooni ka nüüd *spawn* meetodi välja kutsumise. See tähendab, et igakord kui *Food* objekt luuakse, on tal uus suvaline asukoht.

Ja meie *Food* klass on valmis. Lisame nüüd selle ka *Main* klassi. Loome *Main* klassi uue muutuja nimega *food* ning anname sellele andmetüübiks *Food. Main* klassi *constructor* meetodis anname *food* muutujale väärtuseks uue *Food* objekti, mille parameetriteks anname *canvase* ja algseteks koordinaatideks anname 0, nii x kui y, sest meie x ja y koordinaadid on seatud *spawn* meetodi poolt.

Viimase asjana on meil vaja lisada see objekt ka *gameObjects* ahelasse. Seda teeme *initGameObjects* meetodi sees samamoodi nagu me tegime mängijaga. Kui see tehtud peaksime HTML faili avades nägema ekraanil lisaks meie mängija ruudule on tekkinud ka üks punane ruut, mis on siis ussi toit. Kui lehte värskendame näeme, et igakord on punane ruut uues kohas.

Toidu söömine

Hetkel meie ussil on toit olemas, aga süüa ta seda veel ei saa. Õnneks saame kergelt lisada ussile funktsionaalsuse. Lisame *SnakeHead* klassi uue muutuja nimega *foodReference* mille andmetüübiks on *Food* klass. Täiendame ka *constructor* meetodit uue parameetriga *food,* millel on ka andmetüübiks *Food.* Selle parameetri väärtuse anname edasi *foodReference* muutujale. Me peame ka uuendama *Main* klassis *SnakeHead* objekti loomist ning andma seal *food* parameetriks *Main* klassi *food* muutuja. Nüüd on meil ka *SnakeHead* klassi sees toidu objekt kättesaadav.

Loome *SnakeHead* klassis uue meetodi nimega *checkIfCollisonWithFood* parameetritega x ja y ning mõlemale andmetüübiks number. Siin funktsiooni sees me tagastame ühe reaga, kas x ja y, mis funktsioonile edasi anti on võrdsed toidu x ja y väärtusega.

ÜLESANNE: Implementeerida eelnevalt kirjeldatud meetodi sisu.

LAHENDUS: Vaatame milline võiks olla selle ülesande lahendus. Meil on võimalik saada *Food* klassist toidu x ja y väärtused kasutades nende muutujate vastavaid *getter* funktsioone. Kasutades *and* operaatorit, saame ühe reaga võrrelda nii x kui y koordinaate ning tagastada võrdluse tulemus.

```
checkIfCollisionWithFood(x: number, y: number) {
  return x === this.foodReference.getX() && y === this.foodReference.getY();
}
```

Nüüd tuleb meil teha täiendusi *move* meetodis. Me peame igakord kui me liigume kontrollima kas uuele positsioonile liikudes toimub toiduga kokkupõrge. Kui toimub, siis peame kutsuma välja *Food* klassi *spawn* meetodi selleks, et toit uude kohta liiguks. Lisaks on meil hetkel mitu korda välja kutsutud *Canvas* klassi *getCellSize* funktsioon. Meil tuleb kasuks panna see väärtus funktsiooni sees oleva muutuja sisse sama moodi nagu me oleme *activeKey* pannud.

```
move() {
  const activeKey = this.keyboardHandler.getActiveKey();
  const cellSize = this.canvas.getCellSize();

  switch (activeKey) {
    case 'w':
      if (this.checkIfCollisionWithFood(this.x, this.y - cellSize)) this.foodReference.spawn();
      this.y -= cellSize;
      break;
    case 's':
      if (this.checkIfCollisionWithFood(this.x, this.y + cellSize)) this.foodReference.spawn();
      this.y += cellSize;
      break;
    case 'd':
      if (this.checkIfCollisionWithFood(this.x + cellSize, this.y)) this.foodReference.spawn();
      this.x += cellSize;
      break;
    case 'a':
      if (this.checkIfCollisionWithFood(this.x - cellSize, this.y)) this.foodReference.spawn();
      this.x -= cellSize;
      break;
  }
}
```

Proovime nüüd järele, kas toit liigub kuhugi uude kohta kui me selle peale liigume. Värskendame HTML faili ning liigume ussiga toidu peale. Kui kõik on õigesti, siis hüppab toit kokkupõrke korral uude asukohta.

Ussi saba

Toidu tööle saamisega saame edasi liikuda saba loomise loogika juurde. Peame täiendama enne seda meie *GameObject* klassi selleks, et saaksime kergelt implementeerida ussi saba liikumise ja uute juppide loomise. Loome kaks muutujat *GameObject* klassis nimega *previousX* ja *previousY*, mille andmetüübid on numbrid. Anname neile kohe väärtuseks nullid. Loome ka neile *getter* funktsioonid.

Loome uue faili nimega *SnakeBody.ts* ja impordime siia nii *Canvas* kui ka *GameObject* klassi. Siin failis teeme uue klassi nimega *SnakeBody* ning pärime *GameObject* klassi. Loome *constructor* meetodi, mille parameetriteks on *canvas, x* ja *y* ning anname need kasutades *super* funktsiooni edasi meie alamklassi muutujatele.

Meie *SnakeBody* klassi sisu tuleb õnneks väga lühike. Meil on vaja lisada *draw* meetod, mis joonistaks samasuguse valge ruudu nagu *SnakeHead* klassil on. Lisaks on meil vaja luua *move* meetod, mille parameetrid on x ja y. Enne kui need parameetrid antakse klassi muutujate väärtusteks anname selle funktsiooni sees praegused x ja y väärtused *previousX* ja *previousY* väärtusteks. Viimasena ei tohi ära unustada klassi eksport lause.

ÜLESANNE: Implementeerida eelnevalt kirjeldatud klass ja selle sees olevad meetodid.

LAHENDUS: Vaatame milline võiks olla selle ülesande lahendus.

```
import Canvas from './Canvas';
import GameObject from './GameObject';

class SnakeBody extends GameObject {
  constructor(canvas: Canvas, x: number, y: number) {
    super(canvas, x, y);
  }

  draw() {
    this.canvas.drawCell(this.x, this.y, '#ffffff');
  }

  move(x: number, y: number) {
    this.previousX = this.x;
    this.previousY = this.y;
    this.x = x;
```

```
    this.y = y;
  }
}

export default SnakeBody;
```

Nüüd tuleb meil implementeerida *SnakeBody* klass *SnakeHead* klassis. Impordime *SnakeHead* klassis *SnakeBody* klassi ja loome uue muutuja *snakeBodyList,* mille andmetüübiks on *SnakeBody* objektide ahel. Siin hakkame hoidma kõiki ussi saba tükke, mis lisatakse järjest ahelasse kui uss sööb toidu ära.

```
  private snakeBodyList: SnakeBody[] = [];
```

Uuendame järgmisena *draw* meetodit. Lisame siia *forEach* tsükli, mis käib läbi kogu *snakeBodyList* ahela ning kutsub välja iga ussi saba tüki *draw* meetodi. Kui ei ole meeles, kuidas *forEach* tsükkel töötab, siis on võimalik meelde tuletada visates pilk peale *Main* klassile, kus me oleme juba korra seda kasutanud.

```
  draw() {
    this.canvas.drawCell(this.x, this.y, '#ffffff');
    this.snakeBodyList.forEach(body => body.draw());
  }
```

Peame looma uue meetodi, mille sees hoiame funktsionaalsust, mis toimub kui uss sööb toidu ära. Anname sellele meetodile nimeks *handleFoodCollision* ning liigutame sinna funktsiooni sisse *Food* klassi *spawn* meetodi välja kutsumise. Peale toidu uude kohta liigutamist, kontrollime kasutades *if* lauset *snakeBodyList* ahela pikkust, kui ahela pikkus on null, siis anname edasi *SnakeHead* klassi *previousX* ja *previousY* muutujad. Kui ahelas on liikmeid juba olemas, siis anname edasi viimase ahela liikme *previousX* ja *previousY* muutujad. See tagab, et uued saba tükid luuakse ussi viimase liikme eelmisele asukohale.

```
  handleFoodCollision() {
    this.foodReference.spawn();
    if (this.snakeBodyList.length === 0) {
      this.snakeBodyList.push(new SnakeBody(this.canvas, this.previousX, this.previousY));
    } else {
      const lastBodyIndex = this.snakeBodyList.length - 1;
      this.snakeBodyList.push(
        new SnakeBody(
          this.canvas,
          this.snakeBodyList[lastBodyIndex].getPreviousX(),
```

```
          this.snakeBodyList[lastBodyIndex].getPreviousY()
        )
      );
    }
  }
```

Loome veel ühe meetodi nimega *handleBodyMovement.* Selle meetodi sees peame käima läbi *snakeBodyList* ahela ning liigutama igat ussi keha tükki. Kasutame siin tavalist *for* tsüklit, sest meil on vaja kontrollida kasutades indekseid. Kui indeks on null, ehk tegu on esimese liikmega, siis kutsume välja sellel indeksil oleva *SnakeBody* objekti *move* meetodi ning anname selle parameetriteks *SnakeHead* objekti *previousX* ja *previousY* muutujad. Kui indeks on üle nulli, siis peame võtma *previousX* ja *previousY* väärtused eelnevalt saba elemendilt. Eelneva elemendi saame kätte lahutades praeguse tsükli indeksi väärtusest ühe.

ÜLESANNE: Implementeerida eelnevalt kirjeldatud meetod ning võtta see kasutusele *move* meetodi lõpus.

LAHNEDUS:

```
  handleBodyMovement() {
    for (let i = 0; i < this.snakeBodyList.length; i++) {
      if (i === 0) {
        this.snakeBodyList[i].move(this.previousX, this.previousY);
      } else {
        const previousBody = this.snakeBodyList[i - 1];
        this.snakeBodyList[i].move(previousBody.getPreviousX(),
previousBody.getPreviousY());
      }
    }
  }
```

Kui me nüüd meie HTML lehte värskendame ja ussiga ühe toidu ära sööme, siis näeme, et juurde tekib keha, mis liigub ussi pea järel ning kui sööme veel ühe toidu tekib selle keha järele veel üks keha. Meie uss on nüüd funktsionaalne, kuid meie mängu ülejäänud reeglid on veel meil implementeerimata.

CollisionHandler

Hetkel teostatakse kokkupõrke kontroll *SnakeHead* klassis. See veel ei ole probleem, sest me hetkel kontrollime ainult ühte sorti kokkupõrget, aga kui me lisame veel

kokkupõrke kontrolle, siis muutub meie klass väga suureks. Selleks, et seda ära hoida saame luua uue klassi, mille kogu eesmärk on kontrollida kokkupõrkeid.

Loome uue faili nimega *CollisionHandler.ts* ning teeme sinna sama nimelise klassi. Impordime siia faili *Canvas, Food* ja *SnakeBody* klassid. Loome *CollisionHandler* klassis kaks muutujat: *canvas* ja *food*, nende andmetüübid saame juba nende nimedest tuletada, ja loome ka *constructor* meetodi, kus küsime neile kahele muutujale väärtused.

Tõstame *SnakeHead* klassist *checkIfCollisionWithFood* ja *handleFoodCollision* meetodid ümber meie *CollisionHandler* klassi. Me peame *handleFoodCollision* meetodit kohandama selleks, et see töötaks siin klassis nii nagu see töötas eelnevalt *SnakeHead* klassi sees. Me kasutasime selles meetodis *SnakeHead* klassist kolme klassi liiget: *snakeBodyList, previousX* ja *perviousY*. Nüüd me muudame seda meetodit nii, et need väärtused tuleksid parameetrist. Proovige seda ise teha enne kui vaatate lahendust.

```
handleFoodCollision(snakeBodyList: SnakeBody[], previousX: number, previousY: number) {
  this.food.spawn();
  if (snakeBodyList.length === 0) {
    snakeBodyList.push(new SnakeBody(this.canvas, previousX, previousY));
  } else {
    const lastBodyIndex = snakeBodyList.length - 1;
    snakeBodyList.push(
      new SnakeBody(
        this.canvas,
        snakeBodyList[lastBodyIndex].getPreviousX(),
        snakeBodyList[lastBodyIndex].getPreviousY()
      )
    );
  }
}
```

Nüüd kui meie kokkupõrke kontroll ja töötlus on viidud eraldi klassi loome *Main* klassis uue muutuja nimega *collisionHandler* andmetüübiga *CollisionHandler* (klass tuleb importida, tee kindlaks, et eksport lause on olemas). *Main* klassi *constructor* meetodis tuleb meil luua peale toidu objekti loomist uus *CollisionHandler* objekt ja anda talle edasi *Main* klassi *canvas* ja *food* muutujad.

Nüüd tuleb see klass kasutusele võtta *SnakeHead* klassis. Kustutame ära *SnakeHead* klassis *checkIfCollisionWithFood* ja *handleFoodCollision* meetodid, kui seda enne ei teinud ja kustutame ära ka *foodReference* muutuja ja asendame selle *collisionHandler* muutujaga. Asendame *constructor* meetodis *food* parameetri *collisionHandler* parameetriga ja anname selle väärtuse edasi *collisionHandler* muutujale.

```
constructor(
  canvas: Canvas,
  keyboardHandler: KeyboardHandler,
  collisionHandler: CollisionHandler,
  x: number,
  y: number
) {
  super(canvas, x, y);
  this.keyboardHandler = keyboardHandler;
  this.collisionHandler = collisionHandler;
}
```

Nüüd tuleb meil ära vahetada *move* meetodis kasutusel olevad meetodid. Hetkel on meil veel kasutusel *SnakeHead* klassi meetodid, mis on nüüd kustutatud. Siin tuleb tähele panna, et *handleFoodCollision* vajab nüüd kolme parameetrit selleks, et see meetod töötaks nii nagu see peaks. Näitena on üks *switch* lause *case*, aga muuta tuleb kõik.

```
    case 'w':
      if (this.collisionHandler.checkIfCollisionWithFood(this.x, this.y - cellSize))
        this.collisionHandler.handleFoodCollision(
          this.snakeBodyList,
          this.previousX,
          this.previousY
        );
      this.y -= cellSize;
      break;
```

Kontrollime nüüd oma HTML lehel kas kõik töötab nii nagu enne. Kui kõik on õigesti tehtud, siis peaks meie mäng töötama samamoodi nagu enne.

Seinad

Hetkel meie uss liigub vabalt ekraanil ringi, aga meil on vaja anda talle ka piirangud. Nimelt ei tohiks saada uss ekraanilt välja liikuda ning kokkupõrkel seinaga saab mäng läbi. Alustame esimesest nõudes: ekraani piirded.

Loome *CollisionHandler* klassi uue meetodi nimega *checkIfCollisionWithWall,* mille parameetrid on x ja y ning nende mõlema andmetüübid on number. Selle meetodi sees me peame kontrollima, kas x või y väärtus on alla nulli või suurem kui *canvase* suurus, mille saame vastavate *getter* meetoditega kätte *canvas* muutujast.

ÜLESANNE: Implementeerida eelnevalt kirjeldatud meetod.

LAHENDUS:

```
  checkIfCollisionWithWall(x: number, y: number) {
    return x < 0 || y < 0 || x > this.canvas.getWidth() || y >
this.canvas.getHeight();
  }
```

Nüüd tuleb see meetod kasutusele võtta *SnakeHead* klassis. Peame jällegi muutma *move* meetodit. Igas *switch* lause *case* sees peame kontrollima, et liigutus, mida teeme ei puudutaks seina. Seega peame lisama veel ühe *if* lause, mis kutsub välja meie uue meetodi ning kui selle tagastatud väärtus on tõene, siis kirjutame käsureale "dead". Kui väärtus ei ole tõene, siis liigutame ussi edasi.

```
        if (this.collisionHandler.checkIfCollisionWithWall(this.x, this.y - cellSize)) {
          console.log('dead');
        } else {
          this.y -= cellSize;
        }
```

Nüüd on meil kood kenasti eraldatud ja meil on kergem leida meetodeid juhul, kui meil on vaja muuta nende funktsionaalsust.

Surm

Eelnevalt sai lisatud kokkupõrke kontroll seinaga, aga hetkel kirjutatakse meie käsureale "dead" ja uss sõidab rahulikult ekraanist välja. Meil tuleb nüüd implementeerida surma funktsionaalsus. Traditsiooniliselt sureb uss juhul, kui ta läheb enda kehaosale või seinale vastu. Meil hetkel veel kehaosa kokkupõrke kontrolli ei ole, aga me saame juba lisada surma funktsionaalsuse seinaga kokkupõrkele ning lähitulevikus, kui lisame ka ussi kehaosaga kokkupõrke, siis on meil sinna surma funktsionaalsuse lisamine väga kerge. Uss traditsiooniliselt vilgub kui on surnud ehk implementeerime selle ka.

Loome *SnakeHead* klassis kaks muutujat: *isAlive* ja *renderBody*. Mõlemad need on *boolean* andmetüübiga ning mõlemale anname kohe väärtuse *true.* Loome uue meetodi nimega *handleDeath* ja selle sees muudame *isAlive* muutuja väärtuse tõeseks. Nüüd tuleb see meetod *move* meetodis kasutusele võtta. Vahetame ära igas kohas kus on kasutusel *console.log('dead')* meie uue meetodiga. Lisaks loome *move* meetodis kõige alguses uue rea ja seal kontrollime, kas uss on surnud, kui uss on surnud, siis edasi me ei liigu. Seda saame teha kasutades tühja *return* lauset.

Selleks, et uss vilkuma panna, tuleb meil muuta *draw* meetodit. Paneme selle praeguse sisu ümber *if* lause, mis kontrollib kas *renderBody* muutuja on tõene. Peale seda loome veel ühe *if* lause, kus kontrollime kas uss on surnud. Kui uss on surnud, siis vahetame *renderBody* muutuja väärtust tema vastupidise vastu. Seega ainult iga teine kaader on ussi võimalik näha.

```
draw() {
  if (this.renderBody) {
    this.canvas.drawCell(this.x, this.y, '#ffffff');
    this.snakeBodyList.forEach(body => body.draw());
  }
  if (!this.isAlive) this.renderBody = !this.renderBody;
}
```

Värskendame HTML lehte ja proovime kohe järele. Sööme paar toidu tükki ära ja siis sõidame vastu seina. Uss peaks nüüd vilkuma ja me ei saa teda liigutada. Nüüd on vaja lisada ka kokkupõrke kontroll kehaga ning ka lisada sellele kokkupõrkele mängu lõpp.

Vigade parandus

Meie mäng on peaaegu valmis saamas, aga enne kui me viimased funktsionaalsused lisame peame parandama mõned vead, mis on meie ussi liikumises hetkel. Nimelt on meil liikumises kaks viga sees: kui vajutada mõnda muud nuppu peale w, a, s või d jääb uss seisma ning kõik ta keha osad liiguvad ta alla ja teine viga lubab ussil muuta liikumis suunda vastassuunda. Õnneks nende parandamine on kerge. Alustame esimesest veast.

Võtame lahti *KeyboardHandler* klassi ning lisame siin *keyDownListener* meetodisse ühe *if* lause, mis kontrollib kas vajutada nupp on w, a, s või d ning annab *activeKey* muutujale uue väärtuse ainult sellele juhul. Ja ongi parandatud. Proovime kohe järele

pannes ussi liikuma ning vajutades mõnda muud nuppu, mis ussi ei liiguta. Nagu näeme, uss liigub rahulikult edasi ning kui me teda mujale ei suuna liigub ta vastu seina. Üks viga parandatud, üks veel parandada.

Ka meie teise vea jaoks teeme muudatusi *KeyboardHandler* klassis. Loome uue meetodi nimega *checkIfOppositeKey* millel on üks parameeter *newKey,* andmetüübiga *string* ja see meetod tagastab *boolean* väärtuse. Selle funktsiooni sees me kontrollime kas uus sisend on vastupidine hetkel aktiivsele. Ehk kui aktiivne on w, siis kontrollime kas sisend on s jne. Seda tuleb teha kõigi nelja suunaga ning kui ükski kontroll vastab tõele, siis tagastame *true.* Kui ükski kontroll ei lähe läbi, siis tagastame *false.*

ÜLESANNE: Implementeerida eelnevalt kirjeldatud meetod ja võtta see kasutusele *keyDownListener* meetodis.

LAHENDUS: Vaatame milline võib olla lahendus sellele ülesandele. Selleks et kõiki nelja kontrolli elegantselt teha peame kasutama nii *or* kui ka *and* operaatoreid. Lisaks tuleb meil erinevad kontrollid panna ka sulgude sisse selleks, et neid ühtsetena koheldaks.

```
checkIfOppositeKey(newKey: string): boolean {
  if (
    (this.activeKey === 'w' && newKey === 's') ||
    (this.activeKey === 's' && newKey === 'w') ||
    (this.activeKey === 'a' && newKey === 'd') ||
    (this.activeKey === 'd' && newKey === 'a')
  ) {
    return true;
  }

  return false;
}
```

Viimasena tuleb meil *keyDownListener* meetodis lisada olemasoleva *if* lause sisse veel üks *if* lause, mis kontrollib kas selle *meetodi* tagastatud väärtus on *false.* Ainult sellel juhul antakse *activeKey* muutujale uus väärtus. Kui see tehtud, saame testida värskendades oma HTML lehte, kas meie ussi liikumine peab reeglitest kinni. Näeme et kui liigume paremale ei saa me koheselt vasakule liikuma hakata, vaid peame ümber pöörame ennast alguses kas üles või alla liigutades. Saame nüüd liikuda oma mängu viimase funktsionaalsuse juurde.

Ussi hammustamine

Oleme jõudnud oma mängu viimase funktsionaalsuse juurde. Meil on vaja implementeerida ussi surm juhul, kui uss hammustab enda keha. Õnneks on meil juba eelnevalt ees suur osa tööst tehtud ning kõik mis meil on vaja teha, on lisada *CollisionHandler* klassi uus meetod, mis kontrollib kas tuleb kokkupõrge ussi kehaga.

Loome *CollisionHandler* klassi uue meetodi nimega *checkIfCollisionWIthSnake,* millel on kolm parameetrit: x ja y, mille andmetüübid on number ning *snakeBodyList,* mille andmetüüb on *SnakeBody* objektide ahel. See meetod tagastab *boolean* andmetüübi. Selle meetodi sees kontrollime, et x ja y koordinaadid ei oleks samad ühegi ussi kere tükiga. Seega on meil vaja läbi käia kogu *snakeBodyList* ahel ning iga keretükiga teha kontroll.

ÜLESANNE: Implementeerida eelnevalt kirjeldatud funktsioon

LAHENDUS: Vaatame milline võib välja näha selle ülesande lahendus. Alguses kohe kontrollime, kas *snakeBodyList* pikkus on üle kahe, sest alla kolmese keha pikkusega ussil on võimatu endale pihta minna. Selle *if* lause sees kasutame *for* tsüklit selleks, et läbi käia kõik ussi kere tükid alustades kolmandast tükist. Ehk kuna ahelad algavad nulliga, siis see tähendab, et meie *for* tsükli indeksi väärtuseks on kaks. *For* tsükli keha sees kontrollime, kas x ja y koordinaadid on samad, mis on ussi kere tüki omad, kui see vastab tõele siis tagastame koheselt *true*. See peatab tsükli ning meetod tagastab väärtuse mille me määrasime. Meetodi viimasele reale lisame *return false*. Siia jõutakse juhul kui *snakeBodyList* ahel on liiga väike või tsükli sees ei leitud ühtegi ussi kere tükki, mis olid samade koordinaatidega kui meie antud koordinaadid.

```
checkIfCollisionWithSnake(x: number, y: number, snakeBodyList: SnakeBody[]): boolean {
  if (snakeBodyList.length > 2) {
    for (let i = 2; i < snakeBodyList.length; i++) {
      if (snakeBodyList[i].getX() === x && snakeBodyList[i].getY() === y) {
        return true;
      }
    }
  }
  return false;
}
```

See meetod tuleb meil nüüd kasutusele võtta *SnakeHead* klassi. Meie *move* meetod on väga suureks paisunud ja siin on väga palju kordavat koodi. Loome uue meetodi nimega *handleMovement,* millel on kaks parameetrit: *newX* ja *newY* ning mõlema andmetüübid on number. Kopeerime *move* meetodis *switch'i* seest esimese *case'i* sisu *handleMove* meetodi sisse ja kohandame seda nii et seda saaks universaalselt kasutada. Peame asendame kõik koordinaadi väärtused *newX* ja *newY* väärtustega ning selle asemel, et ainult *y* koordinaadile uut väärtust anda, anname mõlemale koordinaadile uue väärtuse. Lisaks võtame kasutusele *CollisionHandler* klassi *checkIfCollisionWithSnake* meetodi, mille me just tegime, samas kohas, kus me kontrollime kokkupõrget seinaga. Siin tuleb kasutada *or* operaatorit, selleks et kontrollida kas toimub üks või teine.

```
handleMovement(newX: number, newY: number) {
  if (this.collisionHandler.checkIfCollisionWithFood(newX, newY)) {
    this.collisionHandler.handleFoodCollision(this.snakeBodyList, this.previousX, this.previousY);
  }
  if (
    this.collisionHandler.checkIfCollisionWithWall(newX, newY) ||
```

```
      this.collisionHandler.checkIfCollisionWithSnake(newX, newY, this.snakeBodyList)
    ) {
      this.handleDeath();
    } else {
      this.x = newX;
      this.y = newY;
    }
  }
```

Nüüd asendame ära *move* meetodis igas *switch'i case* sees sisu meie uue meetodiga. Kui see tehtud, siis näeme, et kood näeb palju puhtam välja ning muudatusi on tulevikus palju kergem teha. Kontrollime ka kas meie uss sureb, kui vastu ussi kere liigume. Värskendame HTML lehte, ning mängime nii kaua kuni uss on umbes 7-8 ruudu pikkune ning siis üritame ussi keha süüa. Kui kõik on õigesti, siis uss jääb seisma ning hakkab vilkuma. Meie mängu viimane funktsionaalsus on valmis ning meil on jäänud veel vaid üks väike viga ära parandada.

Viimased sammud

Meie mäng on nüüd valmis, aga võib olla oled vahepeal testides märganud, et harvadel juhtudel, kui uss sööb toidu ära, tekib uus toit ussi peale. Seda me ei taha ning me peame lisama toidu loomisele kontrollid, mis vaatavad, et uued koordinaadid ei oleks juba okupeeritud ussi pea ega keha juppide poolt. Kuigi võimalus, et see juhtub on küll väga väike, aga mida suuremaks uss muutub seda suuremaks muutub ka võimalus, et toit luuakse ussi peale.

Avame kõige alguses *Canvas* klassi ja loome sinna sisse uue meetodi nimega *getCellCoordinates,* millel on ainult üks parameeter nimega *cellNumber*. See meetod võtab sisendiks ruudu numbri ning tagastab vastava koordinaadi. Näiteks kui anname *cellNumber* väärtuseks kolme, saame tagasi vastuseks 50, sest tehe, mis tehakse taustal korrutab *cellSize* väärtuse *cellNumber* väärtusega ning siis lahutab pool *cellSize* väärtusest selleks, et saada keskkoht. Oleme eelnevalt ka kasutanud sellist tehet, aga seekord paneme selle omaenda meetodi sisse.

```
  getCellCoordinates(cellNumber: number): number {
    return this.cellSize * cellNumber - this.cellSize / 2;
  }
```

Liigume nüüd *Food* klassi juurde ja hakkame kohe muudatusi tegema. Hetkel

genereeritakse meile uued koordinaadid *spawn* meetodi sees. Muudame *spawn* meetodit niimoodi, et ta võtaks endale parameetritest sisse uued x ja y koordinaadi väärtused ja annab need väärtused klassiliikmetele x ja y. Me peame ka muutma *constructor* meetodit, kus *spawn* meetod välja kutsutakse ja genereerima seal enne *spawn* meetodi välja kutsumist x ja y koordinaadid. Selleks kasutame ära meie just loodud *getCellCoordinates* meetodit ja *getRandomInt* meetodit nii nagu me eelnevalt kasutasime seda *spawn* meetodis. Siin aga muudame suvalise numbri genereerimist nii, et ei oleks võimalik genereerida koordinaadid, mis asuksid kõige üleval vasakul nurgas ehk asukohas, kus meie uss oma elu alustab.

```
const newXCellCoord = this.canvas.getCellCoordinates(
  getRandomInt(2, this.canvas.getXCellAmounts())
);
const newYCellCoord = this.canvas.getCellCoordinates(
  getRandomInt(2, this.canvas.getYCellAmounts())
);
this.spawn(newXCellCoord, newYCellCoord);
```

Loome nüüd uue meetodi nimega *handleSpawn,* mis hakkab põhi tööd tegema toidu asukoha genereerimisel. Anname talle kolm parameetrit: *snakeX ja snakeY,* mille andmetüübiteks on number ja *snakeBodyList* mille andmetüübiks on *SnakeBody* ahel. Esimese asjana loome *handleSpawn* meetodis kolm muutujat: *isEmptyCell*, andmetüübiga *boolean,* algväärtusega *false, newXCellCoord* ja *newYCellCoord* andmetüüpidega number ja mõlemale algväärtuseks 0. Siin tuleb tähele panna, et meil tuleb kasutada nende kolme muutujaga võtit *let* mitte *const,* sest me tahame nende muutujate väärtuseid muuta.

Nüüd tuleb meil kasutada *while* tsüklit, mis töötab niikaua kuni *isEmptyCell* väärtus on *false.* Esimese asjana selle tsükli sees genereerime *newXCellCoord* ja *newYCellCoord* muutujatele uued väärtused sama moodi nagu me *constructor* meetodis genereerisime, aga siin lubame me toidu genereerida ka üles vasakule nurka. Need numbrid peame genereerima tsükli sees, sest me tahame et iga kord kui tsükkel jookseb oleksid uued numbrid.

Kui numbrid olemas, siis tuleb need üle kontrollida. Kontrollime esimesena ussi pea koordinaadid üle, mis me saame läbi *snakeX* ja *snakeY* parameetrite. Kui meie genereeritud koordinaadid on samad *snakeX* ja *snakeY* väärtustega, siis anname

isEmptyCell väärtuseks *false,* kui ei ole, siis anname väärtuseks *true*. Peale koordinaatide kontrolli kontrollime kas *isEmptyCell* väärtus on tõene. Kui see ei ole tõene kasutame *continue* käsklust, mis lõpetab praeguse tsükli iteratsiooni ja liigub järgmise iteratsiooni juurde.

Selleks, et kontrollida ussi kehade koordinaate saame kasutada *for* tsüklit. Käime kõik ussikere tükid läbi ning teeme täpselt sama kontrolli, mida tegime ussi pea koordinaatidega. Siin aga, kui leitakse, et koordinaadid on tõesti samad, peale *isEmptyCell* muutujale *false* väärtuse andmist kutsutakse *break* käsklus välja, mis peatab *for* tsükli ning liigub sellest edasi, sest meil ei ole vaja üle jäänud ussi kere tükki kontrollida, sest me teame, et üks kere tükk on juba samadel koordinaatidel.

Kui kõik läks hästi ning kõik kontrollid läbitakse edukalt, siis meie mäng liigub *while* tsüklist välja. Meil on nüüd veel jäänud välja kutsuda *spawn* meetod ning teeme seda *handleSpawn* meetodi viimasel real peale *while* tsüklit.

```
handleSpawn(snakeX: number, snakeY: number, snakeBodyList: SnakeBody[]) {
  let isEmptyCell: boolean = false;
  let newXCellCoord: number = 0;
  let newYCellCoord: number = 0;

  while (!isEmptyCell) {
    newXCellCoord = this.canvas.getCellCoordinates(
      getRandomInt(1, this.canvas.getXCellAmounts())
    );
    newYCellCoord = this.canvas.getCellCoordinates(
      getRandomInt(1, this.canvas.getYCellAmounts())
    );

    if (snakeX === newXCellCoord && snakeY === newYCellCoord) {
      isEmptyCell = false;
    } else {
      isEmptyCell = true;
    }

    if (!isEmptyCell) continue;

    for (let i = 0; i < snakeBodyList.length; i++) {
      if (
        snakeBodyList[i].getX() !== newXCellCoord &&
        snakeBodyList[i].getY() !== newYCellCoord
      ) {
        isEmptyCell = true;
      } else {
        isEmptyCell = false;
```

```
          break;
        }
      }
    }

    this.spawn(newXCellCoord, newYCellCoord);
  }
```

Meil on jäänud nüüd veel viia sisse muudatused kahes kohas. Alustame *CollisionHandler* klassiga ja muutma *handleFoodCollision* meetodit. Peame lisama sellele kaks parameetrit: x ja y, mis on ussi pea x ja y koordinaadid. Lisaks on siin veel kasutusel *spawn* meetod, mille peame ära vahetama meie vast loodud *handleSpawn* meetodiga. Siin tuleb kindlasti tähele panna, et *handleSpawn* meetod kutsutakse kindlasti välja kõige viimase asjana. Viimasena tuleb meil *SnakeHead* klassis *handleMovement* meetodis lisada puudu olevad parameetrid *handleFoodCollision* välja kutsumisele.

```
  handleFoodCollision(
    snakeBodyList: SnakeBody[],
    x: number,
    y: number,
    previousX: number,
    previousY: number
  ) {
    if (snakeBodyList.length === 0) {
      snakeBodyList.push(new SnakeBody(this.canvas, previousX, previousY));
    } else {
      const lastBodyIndex = snakeBodyList.length - 1;
      snakeBodyList.push(
        new SnakeBody(
          this.canvas,
          snakeBodyList[lastBodyIndex].getPreviousX(),
          snakeBodyList[lastBodyIndex].getPreviousY()
        )
      );
    }
    this.food.handleSpawn(x, y, snakeBodyList);
  }
```

Kui see ka tehtud vaatame kas kõik töötab värskendades HTML lehte. Kui toit ilmub nii nagu enne, siis kõik töötab ja meie töö selle mänguga on tehtud.