ITUmulator (ITU Simulator – GRPRO projekt 2023)

En beskrivelse og start-guide

Dette dokument beskriver hvordan du kan komme i gang med at bygge din egen mini-verden ved brug af vores hjemmestrikkede bibliotek. **Det anbefales at alle gennemfører "kom i gang guiden",** eventuelt alene, så alle har en grundforståelse for hvordan man får systemet op at køre. Der kan være stor forskel på om I vil have mest gavn af at starte med afsnit 2 eller 3, man kan således læse de to afsnit i vilkårlig rækkefølge. **Bid mærke i at guiden også slutter med nogle opgaver til at komme i gang.**

0 Indholdsfortegnelse

1 Formål

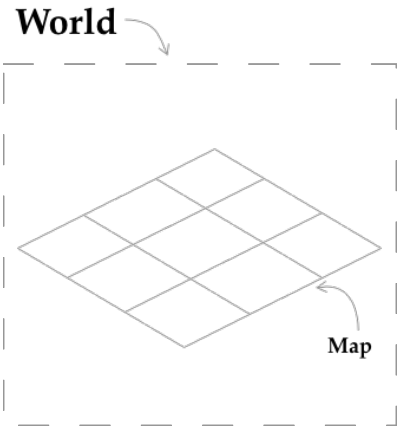
Der er flere grunde til at I får stillet dette bibliotek til rådighed. Først og fremmest, så I kan træne at bruge (og lære at forstå) biblioteker, I ikke har brugt før. Således har denne guide kun til formål at få jer i gang. Det er derfor jeres opgave at bl.a. læse den tilhørende Java-dokumentation for at bruge biblioteket til fulde. Denne guide vil derfor heller ikke gennemgå al funktionalitet gemt i biblioteket. Java-dokumentationen er at finde i selve projektet, og opbevares også her.

2 Bibliotekets opbygning

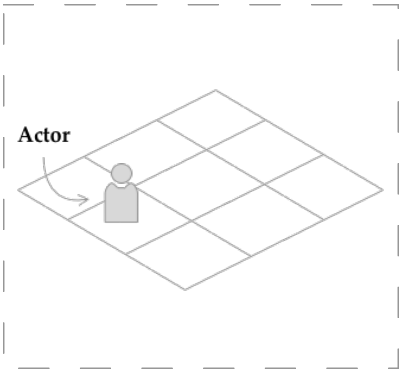
ITUmulator består af fire pakker; ***display***, ***executable***, ***simulator***, samt ***world***. Det er ikke alle fire pakker, I vil komme til at interagere med, men samlet tillader de os at køre vores simulation. Nedenunder kommer en kort introduktion til deres formål, men der henvises igen til Java dokumentationen for yderligere detaljer.

World

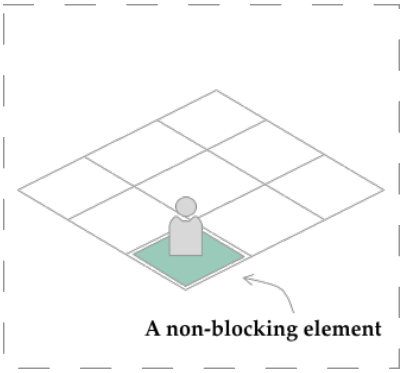
World vil være den pakke, vi vil arbejde med mest i løbet af projektet. Ved instantiering af en ***World*** (ud fra en given størrelse) skaber vi en verden bestående af et todimensionelt 'kort' og yderligere information omkring verden, såsom om det er nat eller dag. Dette illustreres i Figur 1. ***World*** opererer med klassen ***Location*** som er en abstraktion over et koordinatsæt. Ved brug af disse to ting kan man tilføje diverse aktører (f.eks. dyr, planter og græs) til kortet og verdenen. Dette illustreres i Figur 2. Selvom en ***World*** er todimensionel, kan vi faktisk have såkaldt *ikke-blokerende* elementer (dvs elementer, som andre objekter kan placeres ovenpå). Dette er illustreret i Figur 3, hvor det *ikke-blokerende* element eventuelt kan være græs. Man kan ikke have flere ikke-blokerende elementer oven på hinanden. Alt hvad vi placerer på kortet vil normalt være blokerende og dermed fylde en hel plads.



Figur 1

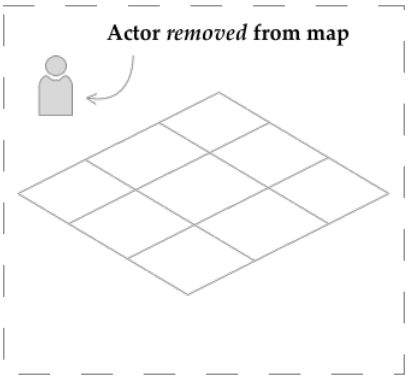


Figur 2

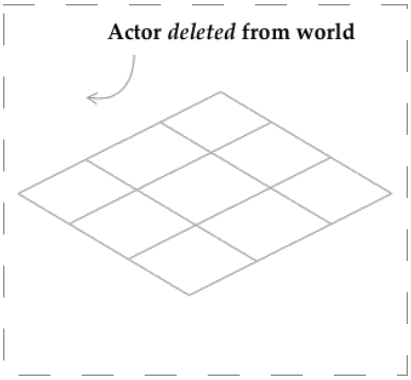


Figur 3

Sidst er det værd at nævne, at der i ***World*** er stor forskel på at *fjerne* noget og *slette* det. At *fjerne* noget betyder blot at det ikke længere er på kortet (altså vist i verdenen). Det eksisterer dog stadig i den. Dette vises på Figur 4. Dette betyder bl.a. at aktører som ikke er slettet, stadig *agerer* i verdenen når simulationen kører. Dette kan f.eks. udnyttes hvis et dyr gemmer sig i en hule, og på et tidspunkt skal beslutte sig for at hoppe ud. At *slette* noget betyder derimod at det både forsvinder fra kortet og verdenen (objektet holder op med at eksistere), som det forsøges at illustrere på Figur 5.



Figur 4



Figur 5

Simulator

Simulator-pakken sørger for at køre og styre vores simulation (og opdatere den grafiske brugergrænseflade ved behov). Denne pakke vil I for det meste ikke skulle anvende. Dog er det vigtigt at forstå ***Actor*** interfacet i pakken. For at systemet simulerer objekter, vi tilføjer (altså tillader at de kan *agere*), skal de implementere ***Actor*** interfacet. Interfacet består af en enkelt metode *act* som tager en ***World*** som parameter. Vi dykker ned i dette interface i startguiden om lidt.

Executable

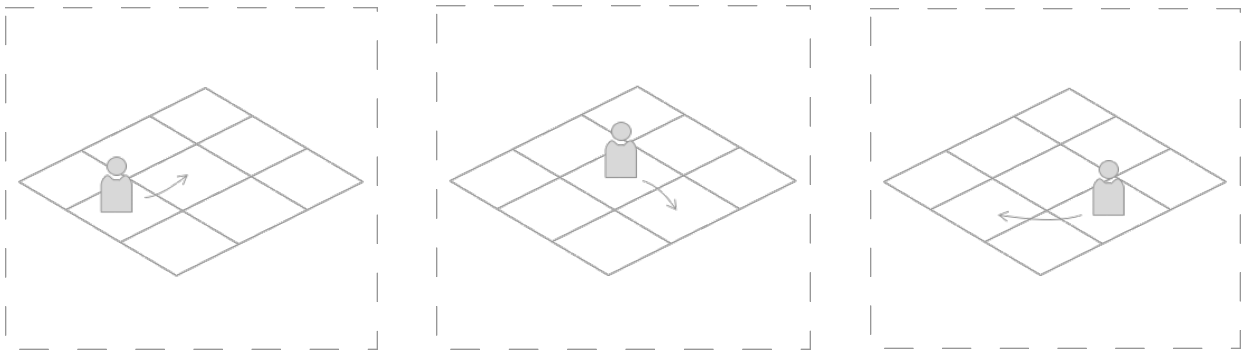
Executable-pakken indeholder tre abstraktioner, der skal gøre det lettere for os at bruge systemet som samlet enhed. Guiden vil introducere disse klasser. Hvis du ønsker dynamisk at ændre udseendet på et objekt undervejs i simulationen (hvilket guidet ikke viser), bør du kigge mod klassen **DynamicDisplayInformationProvider**.

Display

Display-pakken står for alt det grafiske. Denne vil vi heller ikke dykke ned i, da vi gennem ***Executable*** pakken (**Program**) kan styre de ting vi har brug for.

3 'Kom i gang'-guide

'Kom i gang'-guiden hjælper jer i gang med at anvende biblioteket. Det anbefales således, at alle i gruppen forsøger at følge denne selv. I selve projektet vil I skulle indlæse fra filer. Dog har I set dette flere gange før, og guiden her vil derfor ikke fokusere på dette. I stedet vil guiden fokusere på at tilføje en 'menneskelig' aktør, der tilfældigt kan bevæge sig rundt i vores verden indtil det bliver nat, hvor vores person simpelthen går hen og 'dør'. Bevægelsen illustreres på den følgende billedserie. **Bid mærke i at guiden også slutter med nogle opgaver til at komme i gang.**



Første skridt i denne proces, er at instantiere et nyt program i stil med følgende:

```
int size = 3; // størrelsen af vores 'map' (dette er altid kvadratisk)
int delay = 1000; // forsinkelsen mellem hver skridt af simulationen (i ms)
int display_size = 800; // skærm opløsningen (i px)

Program p = new Program(size, display_size, delay); // opret et nyt program
World world = p.getWorld(); // hiv verdenen ud, som er der hvor vi skal tilføje ting!
```

Hvor du præcist skriver denne kode er op til dig. Du bør dog løbende gøre dig overvejelser (i udviklingsprocessen) om det burde ændres! Det er vigtigt at du her husker at importere de rigtige klasser fra hhv. ***Executable*** og ***World*** pakken. Herefter kan vi oprette vores ***Person***. Lav derfor en ny klasse med det navn, og sørg for at klassen implementerer ***Actor*** interfacet fra pakken ***Simulator***:

```
public class Person implements Actor {
    @Override
    public void act(World world) {
        System.out.println("I ain't doin' nothin'!");
    }

}
```

Som du kan se, gør klassen ikke andet end at printe *" I ain't doin' nothin'!"* når metoden kaldes.

Vi kan nu gå tilbage til stedet hvor vi har instantieret vores program og tilføje en person (husk igen at importere de relevante klasser):

```
Person person = new Person();
Location place = new Location(0,1);
world.setTile(place, person);
```

Inden vi kører vores simulation ønsker vi at indstille udseendet på ***Person***. Der er her flere muligheder, og vi anbefaler at I undersøger klassen **DisplayInformation** og interfacet **DynamicDisplayInformationProvider** som kan bruges på flere måder til at styre hvordan objekter vises. Du kan her kigge i afsnit 4 hvis du ønsker mere information. Hvis vi ønsker at anvende en rød farve til vores personer kan vi f.eks. registrere det således i programmet:

```
DisplayInformation di = new DisplayInformation(Color.red);
p.setDisplayInformation(Person.class, di);
```

Her anvender vi standardbiblioteket ***awt*** til at generere en farve. Man kan naturligvis også angive farver på anden vis (søg her efter awt.Color). Derudover understøtter ***DisplayInformation*** bl.a. også billeder. Den første parameter til metoden *setDisplayInformation* kan se lidt obskur ud. Dette er blot for at fortælle at det er alle instanser af klassen ***Person*** denne visuelle indstilling gælder for. Udover billeder, kan vi også dynamisk (når programmet køre) opdatere billeder. Her bør du kigge mod ***DynamicDisplayInformationProvider*** hvis dette er noget du er interesseret i.

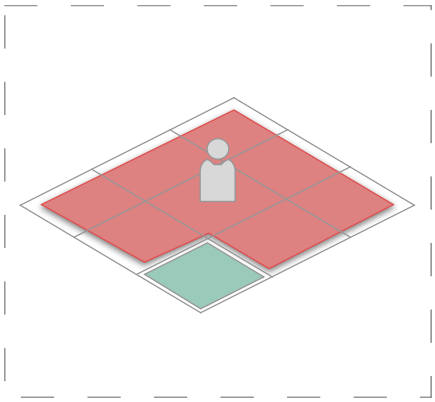
Herefter mangler vi blot at vise vores simulation, og køre den:

```
p.show(); // viser selve simulationen
for (int i = 0; i < 200; i++) {
    p.simulate();
} // kører 200 runder, altså kaldes 'act' 200 gange for alle placerede aktører
```

Når vi når til dette punkt, vil du bide mærke i at der blot printes ”*I ain’t doing nothing!*” tohundrede gange. Dette er jo i virkeligheden ikke det vi ønsker. Derfor kan vi således vende tilbage til klassen ***Person*** og implementere den egentlige logik i *act*. Her kan man f.eks. skrive følgende:

```
Set<Location> neighbours = world.getEmptySurroundingTiles();
List<Location> list = new ArrayList<>(neighbours);
Location l = list.get(0); // Linje 2 og 3 kan erstattes af neighbours.toArray()[0]
world.move(this,l);
```

Ved at kalde metoden *getEmptySurroundingTiles* returnérer ***World*** alle de omkringliggende pladser, som er tomme. Se eksempelvis figuren nedenunder (pladsen vi står på afleveres ikke tilbage):



Figur 6: De felter der returneres når *getEmptySurroundingTiles()* kaldes på den midterste plads.

Vi er på nuværende tidspunkt ikke interesseret i de pladser som allerede har noget på sig (det bliver relevant senere). Herefter ønsker vi egentligt blot at tage den første plads der er fri og bevæge os derover. Dette gør vi ved at kalde *move* med den nye placering. Bid mærke I at vi ikke fortæller ***World*** noget som helst om hvad der er den nuværende placering, men at vi anvender ***this*** som parameter, refererer vi til det nuværende objekt, som skal flyttes. (Læs op på diverse metoder der er tilgængelige i klassen ***World*** der findes mange forskellige brugbare metoder). Grunden til at vi laver sættet om til en liste, er for at kunne udtrække et element ud.

Når du har overskrevet *act* metoden med den ovenstående metode og kører simulationen, kan du nu se at vores person bevæger sig rundt på kortet. Dog er dette ikke helt tilfældigt, og personer dør ikke endnu, når det bliver nat. Derfor er dine opgaver nu at implementere følgende:

1) Anvend ***World***s *delete* metode til at fjerne personer, når det bliver nat.
2) Sørg for at lokationen personen bevæger sig over til, er helt tilfældig. Her kan klassen ***Random*** fra standardbiblioteket hjælpe dig med at give tilfældige tal. Dette kan bruges sammen med listens størrelse til at give en tilfældig lokation.
3) Hvad sker der, når der ikke er nogle ledige pladser? Fungerer dit program som forventet. Test dette ved at implementere en JUnit-test.

For at løse disse fire opgaver (specielt de tre første), bør du læse mere om klassen ***World***. Her anbefaler vi at du læser den tilhørende Java dokumentation og sætter dig ind i diverse muligheder klassen giver. Det kræver at du har sat dig ind i disse for at arbejde videre. Det er generelt en god idé at sætte dig ind i ***World*** klassen, og en stor del af arbejdet (inden I kan starte) er netop at læse og forstå de muligheder du har med klassen.

Når du har gjort dette, bør du implementere følgende: Sørg for at du tilfældigt placerer 10 personer på kortet. Du skal her også overveje hvad du bør gøre, hvis der allerede er en ting på pladsen. Nedenunder giver vi et bud på hvordan dette kan gøres. Du bør have forsøgt at gøre det selv inden.

Der er flere måder hvorpå vi tilfældigt kan placere 10 personer i verdenen (specifikt tænker her specifikt på to måder, kan du forklare begge?). Koden skriver vi samme sted, som simuleringen instantieres – faktisk lige før vi kalder *show*-metoden. Versionen vi nedfælder herunder, vælger tilfældige placeringer indtil vi har placeret 10 personer. Dette vil naturligvis ikke kunne lade sig gøre hvis der ikke er 10 ledige pladser. Derfor bør du starte med at ændre variablen ***size*** til at være 15 (dette giver os en verden at størrelsen 15x15, altså med 225 pladser). Det første vi gør, er at instantiere to variable, og lave skabelondelen af vores løkke:

```
int amount = 10;
Random r = new Random();

for(int i = 0; i < amount; i++){
    // Mere kode kommer her senere.
}
```

Vi erklærer mængden vi gerne vil tilføje (som variabel), og opretter en ny instans af klassen ***Random***. Det gør vi, da processen vi nu vil gøre følgende for hver iteration af vores løkke: Tilføj til en tilfældig plads (tilfældigt x og y koordinat):

```
int amount = 10;
Random r = new Random();

for(int i = 0; i < amount; i++){
    int x = r.nextInt(size);
    int y = r.nextInt(size);
    Location l = new Location(x,y);
    w.setTile(l, new Person());
}
```

Så langt så godt. Vi støder dog her ind i et problem, når vi forsøger, at køre programmet. Det kan nemlig være at vi kommer til at tilføje en person et sted hvor der allerede er en person! I dette tilfælde vil ***World*** rejse en undtagelse, da dette ikke er tilladt. Derfor bliver vi nødt til at undersøge, om den placering vi vil tilføje en person til, har plads. Hvis ikke der er plads, vil vi prøve med nogle nye tilfældige koordinater. Sådan fortsætter vi indtil vi finder en ledig plads:

```
for(int i = 0; i < amount; i++){
    int x = r.nextInt(size);
    int y = r.nextInt(size);
    Location l = new Location(x,y);
    // Så længe pladsen ikke er tom, forsøger vi med en ny tilfældig plads:
    while(!w.isTileEmpty(l)) {
        x = r.nextInt(size);
        y = r.nextInt(size);
        l = new Location(x,y);
    }
    // og herefter kan vi så anvende den:
    world.setTile(l, new Person());
}
```

[Bonusspørgsmål: *Hvad ville der ske, hvis vi "glemte" at øge size fra 3 (mens amount stadig var 10)?*]

Når du har gjort dette, er du mere eller mindre klar til at starte med først tema. Du bør dog stadig læse Java dokumentation og sætter dig ind i diverse muligheder World klassen giver. Det er en stor del af det begyndende arbejde.

4 Anvendelse af DynamicDisplayProvider

Hvis man vil have mere frihed i de grafiske visningsmuligheder, kan man tage et kig på **DisplayInformation** og **DynamicDisplayProvider**. En aktør kan vises med et billede (**BufferedImage**/**Image**) eller en farve (**Color**).

Programmet indlæser automatisk alle **.png**, **.jpeg** og **.jpg** filer som ligger i **resources** mappen som er sidestillet med **src** mappen i projektet. Et billede skal have et unikt navn da de automatisk bliver indlæst med deres filnavn (ergo kan man ikke have "**person.png**" og "**ny-mappe/person.png**").

Hvis man har tilføjet et billede i **resources** mappen kan det anvendes ved at lave en ny **DisplayInformation** med den mere avancerede konstruktør således.

```
DisplayInformation di = new DisplayInformation(Color.red, "person");
p.setDisplayInformation(Person.class, di);
```

Dermed anvendes "**person.png**" de steder hvor billeder kan anvendes. Den vil automatisk blive skaleret så godt som muligt. Det er underordnet hvilken sub-mappe "**person.png**" ligger i indenfor **resources** mappen. Kan billedet ikke bruges, anvendes farven.

DynamicDisplayProvider er en udvidelse du kan anvende hvis du på *runtime* (mens programmet kører) vil skifte billedet (f.eks. for at indikere forskellig opførsel). **Denne del er ikke nødvendig for at gennemføre projektet, men kan give mere kreativ frihed. Man kan også med fordel bruge egne ikoner.** Du skal her blot implementere interfacet **DynamicDisplayProvider** i din aktør, og dermed implementere **getInformation()**. Denne metode bliver kaldt hver gang din aktør kan tegnes, så du kan altså her returnere en ny **DisplayInformation** som dermed kan ændre farven/billedet du viser. Et eksempel kunne være at man (når en person sover om natten) vil anvende en farve og når personen er vågen en anden farve. Der er flere måder at gøre dette på, og eksemplet her er blot en metode.

For at sørge for at vores metode **getInformation()** returnerer **DisplayInformation** den rigtige farve, afhængigt af om det er dag eller nat, bliver vi nødt til at opbevare information om dag/nat et sted, da metoden ikke har adgang til **World**! Derfor tilføjer vi et felt i klassen Person:

```
boolean isNight = false;
```

Så kan vi nemlig opdatere denne hver gang vores **Act(…)** metode bliver kaldt:

```
@Override
public void act(World world) {
```

```
...
        isNight = world.isNight();
...
}
```

Nu ved vi, at hver gang **getInformation()** bliver kaldt, vil vi kunne anvende værdien af feltet *isNight* til at ændre hvilken farve en Person har (f.eks. for at indikere at personen sover). Det gør vi ved at returnere en DisplayInformation med forskellige farver:

```
@Override
public DisplayInformation getInformation() {
    if (isNight){
       return new DisplayInformation(Color.BLUE);
    } else {
       return new DisplayInformation(Color.GRAY);
    }
}
```

Ved brug af dette interface kan du således styre billedet / farven afhængigt af forskellige felter i klassen, f.eks. hvor sulten, stor, eller stærk en aktør er. De nævnte ting er blot eksempler. **Husk at anvendelsen af DynamicDisplayInformationProvider ikke er et krav.** Implementerer et objekt i World dette interface overskrider det alle andre regler for farve / billede man må have indstillet andet steds.