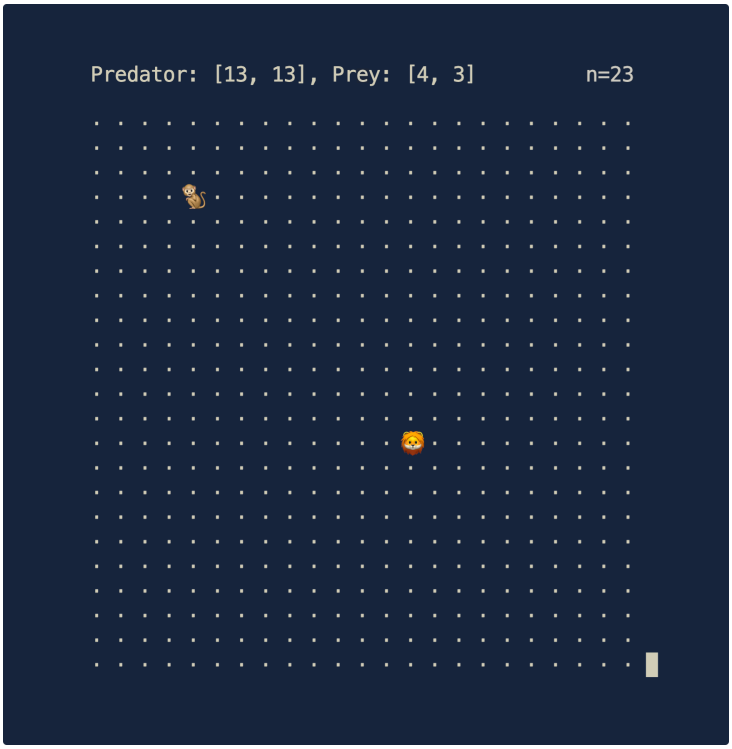
02101 Indledende Programmering
Hjemmeopgave 1

s203775 s194549 s204747

8. oktober, 2021



Arbejdsfordeling

Vi samarbejdede om alle opgaver, men uddelte løse ansvarsområder til forskellige gruppemedlemmer.

s203775 var ansvarlig for: problem 1 og 3.a.
s194549 var ansvarlig for: problem 2 og hjælp på 3.b.
s204747 var ansvarlig for: 3.b. og hjælp på problem 3.a

1. Problem 1: Luhn algoritme / mod-10

Programmet deles op i 3 de-kompositionerede metoder, som alle bruger lokale variable, således at virkemåderne er separeret på tværs af metoderne (fremfor uforudsigeligt sammenkoblet). Metoden `check` samler de 2 metoder `transform` og `sum`. Først transformeres den talrække der er givet som input i `check` vha. `transform`, hvor resultatet herfra skubbes videre til `sum`. Til sidst udfører `check` en simpel sammenligning på betingelsen: at summen fra `sum` skal være delelig med 10. Det er klart metoden `transform` som er nøglen til at løse problemet korrekt. Se Bilag: Luhn / mod-10 implementering.

2. Problem 2: Interval Søgning

Opgave 2 blev løst ved først at sørge for at heltallet man fik givet var opløftet i et tal der gjorde at den var lig med eller større den nedre grænse af intervallet man fik givet. Dette opløftede tal blev så tjekket for at se om det var større end den øvre grænse af intervallet. Hvis den var større end den øvre grænse ville et 'false' svar komme frem, og hvis ikke ville et 'true' svar komme frem.

Der var også en mulighed til hvis heltallet var 1 og den nedre grænse var større end 1, hvor den så vil skrive 'false'.

3. Problem 3: Prey-Predator simulering

Da både delopgave a) og b) deler næsten alt kode, så vil der først gøres en overordnet redegørelse for koden der er fælles, og til sidst de specifikke implementeringsvarianter til at løse delopgaverne.

Til løsning af problem 3 gøres i stor udstrækning brug af de objekt-orienterede muligheder som Java stiller til rådighed. Situationen fortolkes således, at der eksistere nogle "vektor-dyr" med en hastighed s og position, som bliver sat på et $n \times n$ kvadratisk og kartesisk heltals-gitter.

Dyrenes klasse bliver `PredatorPray` klassen i sig selv, og de deler egenskaberne at de har en hastighed, en position, og et gitter tilknyttet til sine instanser. Man kunne argumentere for, at gitteret der er tilknyttet hvert dyr bør være tilknyttet som en reference til et objekt, men da gitterets eneste egenskab er dens kvadratiske dimension n, så tilknyttes den (for simplicitetens skyld) som en reference til den primitive `int n`, der bliver givet til `runSimulation()` metoden.

I `runSimulation()` bliver der altså instantieret 2 dyr, `prey` og `predator`, med klassen `PredatorPray`, hvor begge dyr får tildelt tilfældige positioner på gitteret i klassens constructor metode `PredatorPray()`.

Flere steder både i delopgave a) og b) gøres der brug af tilfældige koordinater indenfor gitteret, så der oprettes en statisk klasse tildelt `randomCoordinates()` som ikke ved noget om gitteret, men blot finder tilfældige koordinator givet et maksimum og minimum. Funktionen kan derfor *bl.a.* bruges til ovenstående problemstilling med tilfældige startpositioner af et `PredatorPray`, hvis altså gitterets dimensioner gives som `min` og `max` parametre til den statiske metode.

Da der også arbejdes meget med koordinater og vektorer, udarbejdes der en statisk underklasse `Vector`, som gør det muligt at behandle koordinaterne som vektorer, med tilhørende matematiske vektor operationer defineret som statiske- (eller instans; begge udstilles) metoder på `Vector` klassen. Det er en fordel af lave sådan en utility klasse selv i denne situation, da det kun er få matematiske operationer der er behov for at løse opgaven, og desuden giver det mulighed for at `Vector` typen underliggende arbejder med primitive `int` arrays. Mange biblioteker som eksistere til at arbejde med vektorer er tiltænkt mere tunge opgaver, så der arbejdes oftest med `double`, og afarter af mere komplicerede collection typer som eks. `ArrayList`. Til denne opgave er der ikke behov for disse mere sofistikerede datastrukturer, og det giver en *lille* optimering mht. evalueringstid at der istedet blot arbejdes med simple primitive `int[]`. `Vector` typen er dog udarbejdet til at virke på n-ordens vektorer, så det ville være let at udvide den eksisterende kode til at fungere i højere dimensioner.

I simulationen skal dyrene også kunne skiftes til at tage ture. Dette håndteres indternt i `runSimulation()` vha. af en `while`-løkke. Ideen er at bruge `runSimulation()` som et samlingspunkt for de mange andre

metoder der er skabt vha. *"procedural decomposition"*. Altså er det en slags implementeringsspecifik kode, og her er løkken valgt at blive betragtet som en del af implementeringen. Indenfor løkken skal dyrene træffe beslutninger baseret på deres positioner, og bevæge sig derefter.

Fælles for begge dyr er nemlig at de kan bevæge sig, og derfor udstiller `PredatorPray` en instans metode `move()`, der tager en `Vector` som parameter, og ligger denne til dyre-instansens eksisterende position *men* indenfor gitteret. `Vector` typen udstiller en statisk metode `clamp()`, som netop gør det muligt at holde vektorens komposanter indenfor gitteret ved at begrænse dem begge indenfor `min` og `max`, som er parametrene metoden tager.

Hvordan dyrene bevæger sig er dog underordnet, og dette håndteres istedet via statiske metoder som kan implementeres på kryds af tværs af begge dyr efter behov. I opgaven bliver de selvfølgelig implementeret passende, så der opstår den beskrevede "predator-prey" dynamik. Ideen er at implementere meningsfuld **komposition** af metoderne, **fremfor nedarvning** og inter-afhængige klasser der beskriver en bestemt type dyr. Et dyr er et dyr, og de kan principielt det samme!

a)

(Fortsat fra ovenstående afsnit)

... Eksempelvis bevæger `prey` sig med samme funktionalitet, som der bruges til at vælge tilfældige startpositioner til dyrene - nemlig `randomCoordinates()`. Her gives istedet grænser via `prey.speed`, altså dyrets hastighed.

For at udlede hvordan `predator` skal bevæge sig, gøres der primært brug af de statiske utility metoder på `Vector` typen. Algoritmen er ekstremt simpel, og kan beskrives som følgende: det ultimative mål for `predator` er at have samme koordinater som `prey`. Derfor findes den vektor som ligger imellem dyrenes positioner. Denne vektor skal begrænses af hastigheden for `predator`, eller mere præcist; den kvadrat som angiver grænsen for et område hvor `predator` maksimalt må bevæge sig indenfor. Begrænsningen til området kan gøres simpelt vha. `clamp()`, som netop begrænser hver komposant inden for en angivet `min` og `max`.

Generelt viste det sig at der var mange fordele ved at definere en selvstændig `Vector` type. Resten af implementeringen handler mest om formatering, og en af de fede ting i Java er, at man kan overskride den `toString()` instans metode, som er den der underliggende bliver kaldt, hvis man printer instansen til standard output. Derfor printer `Vector` typen sig selv nøjagtigt som beskrevet i opgaven, `[x;y]`, fordi `toString()` er blevet defineret efter kravene, og kræver derfor ikke en wrapper metode i selve implementering hvor instansen bliver printet.

b)

På grund af den de-kompositionerede struktur af koden, var det kun et spørgsmål om at sætte metoderne anderledes sammen i implementeringen til b). "Teleport" er egentligt bare det som sker når et dyr bliver instantieret på gitteret, nemlig igen `randomCoordinates()`, konkret givet i intervallet $0...n-1$. Her sættes metoden undner et `if` statement, som tjekker om komposanterne fra positionen af `prey` er delelig med s. Hvis betingelsen ikke er opfyldt, så bliver programmet helt identisk med a), med undtagelse af kravet $s \geq 2$.

Se Bilag: Java klasser.

(bonus)

For sjovs skyld er der også blevet udformet en lille "vektor-savanne". Denne kan afprøves ved at køre `PredatorVisualizer.java`, og desuden ses den på forsiden af dokumentet. Her visualiseres simulationen i terminalen, hvor hvert gitterpunkt renderes med en prik, og dyrene som passende emojis. Selvom eksperimentet blot var for fornøjelsens skyld, så hjalp det faktisk til at løse problemerne mere nøjagtigt, da simulationens opførsel blev øjeblikkeligt tydelig og visuel.

Bemærkninger

`while`-løkken som styrer dyrenes ture i `runSimulation()` brydes bl.a. når `t` bliver større end antallet af ture / iterationer som er kørt. I opgaven står der dog:

> ... $t \geq 0$ is the number of *moves* to be performed.

Så denne implementering holder faktisk styr på antallet af moves hvert dyr tager, selvom det her blot vil betyde at det totale antal *moves* er dobbelt så stort som antallet af ture (da hvert dyr begge tager 1 move hver tur = 2 moves i alt). I den konkrete kode implementering holdes der derfor øje med hvor mange *moves* som `predator` har taget, givet ved `predator.moves`. Dette er et arbitrært valg, og dette kunne ligeså godt være `prey.moves`.

Hver gang et dyr tager et *move* forøges dens instans-variabel `int moves` med `1`. Dette er givet ved instans metoden *move*, som også er ansvarlig for at ligge nye koordinator til den eksisterende position.

Et mere præcist udtryk for `t` havde nok været *turns* eller *rounds*, men igennem test på CodeJudge fandt vi frem til, at det i hvertfald ikke var det faktiske totale antal *moves* som er foretaget der skulle styre slagets gang.

Branchless programming: eksperiment

`Vector` typen har en statisk `clamp()` metode, som bruges forskellige steder. Denne er faktisk skrevet med en *brancheless* tilgang, dvs. uden *betingede erklæringer* (if, switch osv.). Dette er sandsynligvis en del *langsommere* end hvis det var blevet implementeret med et if statement. Faktisk kunne metoden være blevet implementeret meget kort og elegant med en ternary operator, som der også er givet et udkommenteret alternativ til i selve koden. Det var et interessant ekseperiment for at undersøge hvordan `boolean`'s konverteres til `int`'s i Java under en aritmisk operation. Ideen er nemlig at udnytte den numeriske værdi af en boolsk værdi (`1 || 0`), og så bruge det som en slags betinget faktor i en sum, som enten inkludere eller ekskludere led baseret på "betingelsen".

I nogle programmeringssprog ville dette måske give en hurtigere evalueringstid, men selv uden dybere kendskab til hvordan Java compiler til maskinkode, er det sikkert at sige, at det er meget *usandsynligt* at compileren genkender dette obskure mønster, og derfor ender det formentligt med noget overhead i og med at typerne skal konverteres frem og tilbage. Ikke desto mindre var det sjovt og lærerigt!

4. Bilag: Java klasser

Problem 1, Java implementering

```java
public class NumberCheck {

  public static void main(String[] args) {
    System.out.println(check("3475"));
    // -> true
    System.out.println(check("41032"));
    // -> false
  }

  public static boolean check(String number) {
    int[] transformedDigits = transform(number);
    int checksum = sum(transformedDigits);
    return checksum % 10 == 0 ? true : false;
  }

  // Transform number, e.g. 3475 -> 6455
  private static int[] transform(String number) {
    char[] digitChars = number.toCharArray();
    int[] out = new int[number.length()];

    for (int index = 0; index < digitChars.length; index++) {
      // Traverse array backwards but retain
      // positive index incrementation.
      // Calculation of the index in the array, i:
      int i = (digitChars.length - 1) - index;

      // Apply algorithm to digit
      int digit = Character.getNumericValue(digitChars[i]);
      int twice = digit * 2;
      boolean isOddIndex = index % 2 == 0 ? false : true;
      if (isOddIndex) {
        if (twice < 10) {
          out[i] = twice;
        } else {
          out[i] = (twice % 10) + 1;
        }
      } else {
        out[i] = digit;
      }
    }
    return out;
  }

  private static int sum(int[] nums) {
    int out = 0;
    for (int i = 0; i < nums.length; i++) {
      out += nums[i];
    }
    return out;
  }
}
```

Problem 2, Java implementering

```java
public class IntervalSearch {

  public static void main(String args[]) {
    System.out.println(intervalContains(5, 11, 1));

  }

  public static boolean intervalContains(int g1, int g2, int b) {
    // Turning the integers into doubles, as it is needed later
    double i = 0;
    double number = inttodouble(b);
    double power;
    double g1new;
    double g2new;

    // Putting the smallest number as the start of the interval
    if (g2 < g1) {
      g1new = inttodouble(g2);
      g2new = inttodouble(g1);
    } else {
      g1new = inttodouble(g1);
      g2new = inttodouble(g2);
    }

    // If the smallest number in the interval is not 1, but b is, then it will have
    // to return false or else it will create a never ending loop
    if (g1new != 1.0 && number == 1) {
      return false;
    } else {
      // If b is smaller than the lower limit, the power has to be increased
      do {
        i += 1;
        power = Math.pow(number, i);
      } while (power < g1new);

      // Now if the power is smaller than the top limit it returns true, if not, false
      if (g1new <= power && power <= g2new) {
        return true;
      } else {
        return false;
      }

    }

  }

  // Just a quick system to turn integers to doubles
  public static double inttodouble(int i) {
    double d = 1.0 * i;
    return d;
  }
```

```
52
53 }
```

Problem 3.a, Java implementering

```
1  import java.util.Random;
2
3  public class PredatorPray {
4    private int grid;
5    Vector position;
6    int speed;
7    int moves;
8
9    public PredatorPray(int grid_, int speed_) {
10     this.speed = speed_;
11     this.grid = grid_;
12     this.position = randomCoordinates(0, this.grid);
13     this.moves = 0;
14   }
15
16   public void move(Vector here) {
17     Vector nextPosition = this.position.add(here);
18     this.position = nextPosition.clamp(0, this.grid);
19     this.moves++;
20   }
21
22   private static Vector randomCoordinates(int min, int max) {
23     Vector out = new Vector(2);
24     for (int i = 0; i < 2; i++)
25       out.set(i, new Random().nextInt(max - min) + min);
26     return out;
27   }
28
29   public static void main(String[] args) {
30     runSimulation(23, 2, 20);
31   }
32
33   public static void runSimulation(int n, int s, int t) {
34     System.out.println("n=" + n + " s=" + s + " t=" + t);
35     if (n > 0 && s > 0 && t >= 0) {
36       PredatorPray predator = new PredatorPray(n - 1, s);
37       PredatorPray prey = new PredatorPray(n - 1, s);
38       System.out.println(prey.position + " " + predator.position);
39       while (!Vector.equal(predator.position, prey.position) && t >
             predator.moves) {
40         prey.move(randomCoordinates(-prey.speed, prey.speed));
41
42         Vector between = Vector.subtraction(prey.position, predator.
             position);
43         Vector maximalDistance = Vector.clamp(between, -predator.speed,
             predator.speed);
44         predator.move(maximalDistance);
45
46         System.out.println(prey.position + " " + predator.position);
47       }
```

```
48        if (Vector.equal(predator.position, prey.position)) {
49          System.out.println("Catch!");
50        }
51      } else {
52        System.out.println("Illegal Parameters!");
53      }
54    }
55
56    // Custom utility class for working with int vectors
57    private static class Vector {
58      private final int[] array;
59
60      public Vector(int d) {
61        this.array = new int[d];
62      }
63
64      public Vector(int[] a) {
65        this.array = a;
66      }
67
68      public int get(int index) {
69        return this.array[index];
70      }
71
72      public void set(int index, int value) {
73        this.array[index] = value;
74      }
75
76      public int dimension() {
77        return this.array.length;
78      }
79
80      // overwrite toString() method to print formatted vector
81      // instead of primitive / native array.toString()
82      public String toString() {
83        String[] collect = new String[this.array.length];
84        for (int i = 0; i < this.array.length; i++) {
85          collect[i] = String.valueOf(this.array[i]);
86        }
87        return "[" + String.join(";", collect) + "]";
88      }
89
90      public Vector add(Vector V) {
91        return addition(new Vector(this.array), V);
92      }
93
94      public Vector clamp(int min, int max) {
95        return clamp(new Vector(this.array), min, max);
96      }
97
98      // an operation consists of: adding two vectors with weights w1 and
          w2
99      // finally scaling by s
100     // used to build addition, subtraction and scaling
101     private static Vector operation(Vector A, Vector B, int w1, int w2,
          int s) {
```

```
102       int[] result = new int[A.dimension()];
103       for (int i = 0; i < result.length; i++) {
104         result[i] = (A.get(i) * w1 + B.get(i) * w2) * s;
105       }
106       return new Vector(result);
107     }
108
109     public static Vector addition(Vector A, Vector B) {
110       return operation(A, B, 1, 1, 1);
111     }
112
113     public static Vector subtraction(Vector A, Vector B) {
114       return operation(A, B, 1, -1, 1);
115     }
116
117     // branchless approach just because i can
118     //
119     // converting boolean values to int using
120     // bit shifting
121     public static Vector clamp(Vector V, int min, int max) {
122       int[] clamped = new int[V.dimension()];
123       for (int i = 0; i < clamped.length; i++) {
124         clamped[i] = (V.get(i) * (1 & Boolean.hashCode(min < V.get(i) &&
              V.get(i) < max) >> 1))
125             + (min * (1 & Boolean.hashCode(V.get(i) <= min) >> 1))
126             + (max * (1 & Boolean.hashCode(V.get(i) >= max) >> 1));
127         // simpler branched version with ternary operator:
128         // clamped[i] = V.get(i) > max ? max : V.get(i) < min ? min : V.
              get(i);
129       }
130       return new Vector(clamped);
131     }
132
133     public static boolean equal(Vector A, Vector B) {
134       return A.get(0) == B.get(0) && A.get(1) == B.get(1);
135     }
136   }
137 }
```

Problem 3.b, Java implementering

```
1  import java.util.Random;
2
3  public class PredatorPrayTeleport {
4    private int grid;
5    Vector position;
6    int speed;
7    int moves;
8
9    public PredatorPrayTeleport(int grid_, int speed_) {
10     this.speed = speed_;
11     this.grid = grid_;
12     this.position = randomCoordinates(0, this.grid);
13     this.moves = 0;
14   }
```

```
15
16    public void move(Vector here) {
17      Vector nextPosition = this.position.add(here);
18      this.position = nextPosition.clamp(0, this.grid);
19      this.moves++;
20    }
21
22    private static Vector randomCoordinates(int min, int max) {
23      Vector out = new Vector(2);
24      for (int i = 0; i < 2; i++)
25        out.set(i, new Random().nextInt(max - min) + min);
26      return out;
27    }
28
29    public static void main(String[] args) {
30      runSimulation(23, 2, 20);
31    }
32
33    public static void runSimulation(int n, int s, int t) {
34      System.out.println("n=" + n + " s=" + s + " t=" + t);
35      if (n > 0 && s >= 2 && t >= 0) {
36        PredatorPrayTeleport predator = new PredatorPrayTeleport(n - 1, s)
              ;
37        PredatorPrayTeleport prey = new PredatorPrayTeleport(n - 1, s);
38        System.out.println(prey.position + " " + predator.position);
39        while (!Vector.equal(predator.position, prey.position) && t >
              predator.moves) {
40          boolean xDivisibleBy_s = prey.position.get(0) % s == 0;
41          boolean yDivisibleBy_s = prey.position.get(1) % s == 0;
42          if (xDivisibleBy_s && yDivisibleBy_s) {
43            // Teleport!
44            prey.move(randomCoordinates(0, n - 1));
45          } else {
46            prey.move(randomCoordinates(-prey.speed, prey.speed));
47          }
48
49          Vector between = Vector.subtraction(prey.position, predator.
                position);
50          Vector maximalDistance = Vector.clamp(between, -predator.speed,
                predator.speed);
51          predator.move(maximalDistance);
52
53          System.out.println(prey.position + " " + predator.position);
54        }
55        if (Vector.equal(predator.position, prey.position)) {
56          System.out.println("Catch!");
57        }
58      } else {
59        System.out.println("Illegal Parameters!");
60      }
61    }
62
63    // Custom utility class for working with int vectors
64    private static class Vector {
65      private final int[] array;
66
```

```
67      public Vector(int d) {
68        this.array = new int[d];
69      }
70
71      public Vector(int[] a) {
72        this.array = a;
73      }
74
75      public int get(int index) {
76        return this.array[index];
77      }
78
79      public void set(int index, int value) {
80        this.array[index] = value;
81      }
82
83      public int dimension() {
84        return this.array.length;
85      }
86
87      // overwrite toString() method to print formatted vector
88      // instead of primitive / native array.toString()
89      public String toString() {
90        String[] collect = new String[this.array.length];
91        for (int i = 0; i < this.array.length; i++) {
92          collect[i] = String.valueOf(this.array[i]);
93        }
94        return "[" + String.join(";", collect) + "]";
95      }
96
97      public Vector add(Vector V) {
98        return addition(new Vector(this.array), V);
99      }
100
101     public Vector clamp(int min, int max) {
102       return clamp(new Vector(this.array), min, max);
103     }
104
105     // an operation consists of: adding two vectors with weights w1 and
          w2
106     // finally scaling by s
107     // used to build addition, subtraction and scaling
108     private static Vector operation(Vector A, Vector B, int w1, int w2,
          int s) {
109       int[] result = new int[A.dimension()];
110       for (int i = 0; i < result.length; i++) {
111         result[i] = (A.get(i) * w1 + B.get(i) * w2) * s;
112       }
113       return new Vector(result);
114     }
115
116     public static Vector addition(Vector A, Vector B) {
117       return operation(A, B, 1, 1, 1);
118     }
119
120     public static Vector subtraction(Vector A, Vector B) {
```

```
121       return operation(A, B, 1, -1, 1);
122     }
123
124     // branchless approach just because i can
125     //
126     // converting boolean values to int using
127     // bit shifting
128     public static Vector clamp(Vector V, int min, int max) {
129       int[] clamped = new int[V.dimension()];
130       for (int i = 0; i < clamped.length; i++) {
131         clamped[i] = (V.get(i) * (1 & Boolean.hashCode(min < V.get(i) &&
              V.get(i) < max) >> 1))
132             + (min * (1 & Boolean.hashCode(V.get(i) <= min) >> 1))
133             + (max * (1 & Boolean.hashCode(V.get(i) >= max) >> 1));
134         // simpler branched version with ternary operator:
135         // clamped[i] = V.get(i) > max ? max : V.get(i) < min ? min : V.
              get(i);
136       }
137       return new Vector(clamped);
138     }
139
140     public static boolean equal(Vector A, Vector B) {
141       return A.get(0) == B.get(0) && A.get(1) == B.get(1);
142     }
143   }
144 }
```

Problem 3: bonus, Java implementering

```
1  import java.util.Random;
2
3  public class PredatorVisualizer {
4    private int grid;
5    Vector position;
6    int speed;
7    int moves;
8
9    public PredatorVisualizer(int grid_, int speed_) {
10     this.speed = speed_;
11     this.grid = grid_;
12     this.position = randomCoordinates(0, this.grid);
13     this.moves = 0;
14   }
15
16   public void move(Vector here) {
17     Vector nextPosition = this.position.add(here);
18     this.position = nextPosition.clamp(0, this.grid);
19     this.moves++;
20   }
21
22   private static Vector randomCoordinates(int min, int max) {
23     Vector out = new Vector(2);
24     for (int i = 0; i < 2; i++)
25       out.set(i, new Random().nextInt(max - min) + min);
26     return out;
```

```
27    }
28
29    public static void main(String[] args) {
30      runSimulation(23, 1, 200);
31    }
32
33    public static void runSimulation(int n, int s, int t) {
34      System.out.println("n=" + n + " s=" + s + " t=" + t);
35      if (n > 0 && s > 0 && t >= 0) {
36        PredatorVisualizer predator = new PredatorVisualizer(n - 1, s);
37        PredatorVisualizer prey = new PredatorVisualizer(n - 1, s * 3);
38        while (!Vector.equal(predator.position, prey.position) && t > (
              predator.moves)) {
39          clearScreen();
40          prey.move(randomCoordinates(-prey.speed, prey.speed));
41
42          Vector between = Vector.subtraction(prey.position, predator.
                position);
43          Vector maximalDistance = Vector.clamp(between, -predator.speed,
                predator.speed);
44          predator.move(maximalDistance);
45
46          String stats = "Predator: " + predator.position + ", Prey: " +
                prey.position;
47          System.out.print(stats);
48          for (int i = 0; i < (n * 2) - stats.length() - 5; i++)
49            System.out.print(" ");
50          System.out.print("n=" + n + "\n\n");
51          String roundStats = "Rounds: " + (predator.moves);
52          System.out.print(roundStats);
53          for (int i = 0; i < (n * 2) - roundStats.length() - 35; i++)
54            System.out.print(" ");
55          System.out.print("/ t=" + t + '\n');
56
57          drawScreen(n, predator.position, prey.position);
58        }
59      } else {
60        System.out.println("Illegal Parameters!");
61      }
62    }
63
64    private static void clearScreen() {
65      System.out.print("\033[H\033[2J");
66      System.out.flush();
67    }
68
69    private static void drawScreen(int n, Vector p1, Vector p2) {
70      try {
71        for (int i = 0; i < n; i++) {
72          System.out.println();
73          for (int j = 0; j < n; j++) {
74            if (j == p1.get(0) && i == p1.get(1)) {
75              // System.out.print(" ");
76              System.out.print("");
77            } else if (j == p2.get(0) && i == p2.get(1)) {
78              // System.out.print(" ");
```

```
            System.out.print("");
          } else {
            System.out.print("ů ");
          }
        }
      }
      Thread.sleep(150);
    } catch (InterruptedException e) {
      System.out.println("got interrupted!");
    }
  }

  // Custom utility class for working with int vectors
  private static class Vector {
    private final int[] array;

    public Vector(int d) {
      this.array = new int[d];
    }

    public Vector(int[] a) {
      this.array = a;
    }

    public int get(int index) {
      return this.array[index];
    }

    public void set(int index, int value) {
      this.array[index] = value;
    }

    public int dimension() {
      return this.array.length;
    }

    // overwrite toString() method to print formatted vector
    // instead of primitive / native array.toString()
    public String toString() {
      String[] collect = new String[this.array.length];
      for (int i = 0; i < this.array.length; i++) {
        collect[i] = String.valueOf(this.array[i]);
      }
      return "[" + String.join(";", collect) + "]";
    }

    public Vector add(Vector V) {
      return addition(new Vector(this.array), V);
    }

    public Vector clamp(int min, int max) {
      return clamp(new Vector(this.array), min, max);
    }

    // an operation consists of: adding two vectors with weights w1 and
       w2
```

```
134     // finally scaling by s
135     // used to build addition, subtraction and scaling
136     private static Vector operation(Vector A, Vector B, int w1, int w2,
            int s) {
137       int[] result = new int[A.dimension()];
138       for (int i = 0; i < result.length; i++) {
139         result[i] = (A.get(i) * w1 + B.get(i) * w2) * s;
140       }
141       return new Vector(result);
142     }
143
144     public static Vector addition(Vector A, Vector B) {
145       return operation(A, B, 1, 1, 1);
146     }
147
148     public static Vector subtraction(Vector A, Vector B) {
149       return operation(A, B, 1, -1, 1);
150     }
151
152     // branchless approach just because i can
153     //
154     // converting boolean values to int using
155     // bit shifting
156     public static Vector clamp(Vector V, int min, int max) {
157       int[] clamped = new int[V.dimension()];
158       for (int i = 0; i < clamped.length; i++) {
159         clamped[i] = (V.get(i) * (1 & Boolean.hashCode(min < V.get(i) &&
                V.get(i) < max) >> 1))
160             + (min * (1 & Boolean.hashCode(V.get(i) <= min) >> 1))
161             + (max * (1 & Boolean.hashCode(V.get(i) >= max) >> 1));
162         // clamped[i] = V.get(i) > max ? max : V.get(i) < min ? min : V.
              get(i);
163       }
164       return new Vector(clamped);
165     }
166
167     public static boolean equal(Vector A, Vector B) {
168       return A.get(0) == B.get(0) && A.get(1) == B.get(1);
169     }
170   }
171 }
```

5. Bilag: Luhn / mod-10 implementering

Der findes mange måder at implementere en Luhn checksum algoritme på, herunder især hvordan talrækken bliver transformeret. I denne opgave er implementeringen givet ved en talrække af n cifre, som kan itereres igennem baglæns med et index der går fra 0 til n. Hvorvidt indexets numeriske værdi er et lige eller ulige tal, afgør hvorvidt cifferet skal fordobles. Hvis det fordoblede tal er højere end 9, så vil det resultere i et 2-cifret, hvorfor det fordoblede tal ikke længere kan bruges.

For at løse denne sidste problematik, tages det fordoblede tal i modulus 10, hvorefter 1 lægges til. En alternativ måde at tænke om det på, kunne være at man tager det første ciffer i det fordoblede tal, og lægger til det andet ciffer. Ligeledes findes der også en alternativ, og mere imperativ (mindre matematisk rigid), formulering til hvordan talrækken af cifre kan itereres, hvor der ikke tages højde for indexets paritet: start med det næstsidste ciffer i rækken, og påfør transformationen på hver anden næste kommende ciffer,

gående fra højre mod venstre.

Uanset implementeringen, bliver resultatet dog ens. Her er selvfølgelig forsøgt at skabe nøjagtigt den implementering som er beskrevet i opgaven. Den store udfordring ved netop denne implementering i kode er, at der er to forskellige indexer, som går i hver deres retning: optælling og nedtælling. Talrækken skal itereres baglæns, mens indexet i hver iteration stiger postivt med optælling.

En løsning kunne være at sortere talrækken før og efter transformationen, men det ville ikke være særligt effektivt mht. evalueringstid. Derfor itereres der i en løkke over et positivt stigende index, hvorfra det negativt aftagende index til talrækken kan udledes og beregnes, da længen af talrækken / arrayet kendes.
Bilag: Java klasser