1 Enkele kleine opgaven **totaal 20 punten**

4 punten **1.a**) Wat is het verschil tussen een *instantievariabele* en een *lokale variabele*? Geef van beide een voorbeeld

4 punten **1.b**) Bekijk de volgende *methode-header* (ook wel *signatuur* genoemd):

```
private int countBigBalls( List<Ball> balls, int minSize )
```

Benoem **alle** onderdelen in deze header kort en krachtig. Probeer niet te bedenken wat de methode mogelijk doet; geef enkel aan wat de betekenis van elke component is (gebruik termen als *type*, *parameter*, etc.).

4 punten **1.c**) Bekijk de volgende methode:

```
private int calc( int x ){
   return x * x * x;
}

private int calculateSomething ( int n ) {
    int j = 0;
    int i = 1;
    while ( i <= n ) {
        j = j + calc( i );
        i++;
    }
    return j;
}
```

Wat is het resultaat van de aanroep `calculateSomething( 4 )`? Licht je antwoord toe.

4 punten **1.d**) Wat is het doel van een constructor? In wat voor opzicht(en) onderscheidt een constructor zich van andere methodes? Noem 2 verschillen.

4 punten **1.e**) Definieer een methode

```
public int nrOfCollisions( List<Ball> balls, Ball otherBall )
```

die voor de lijst `balls` van ballen bepaalt hoeveel ervan botsen met de eveneens meegegeven bal `otherBall`. Het middelpunt van een bal bepaal je met de methodes `getX` en `getY`; de straal met `getRadius`. Enkele `List`-methodes die je misschien kunt gebruiken zijn: **int** `size()`, `Object get( int index )`, **void** `remove( Object o )`, **boolean** `isEmtpy()`.

2 Complexiteit **totaal 10 punten**

5 punten **2.a**) De NS wil in de (verre) toekomst de reistijd op het nationale spoorwegnet verkorten door sommige trajecten te verbeteren zodanig dat er een hogesnelheidstrein overheen kan rijden. Gegeven alle plaatsen waar zich een treinstation bevindt en alle verbindingen hiertussen. Geef een algoritme om te bepalen welke verbindingen hiervan moeten worden aangepast zodanig dat

i) alle steden verbonden zijn via zo'n hogesnelheids-verbinding en

ii) de totale lengte van deze hogesnelheids-verbindingen minimaal is.

Specificeer dit algoritme door middel van een 'hoog niveau' flowchart (dat wil zeggen, een flowchart zonder veel details).

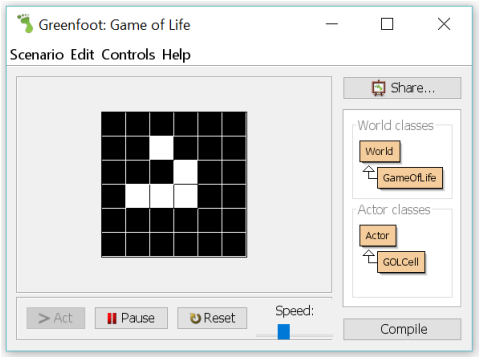
5 punten

2.b) Tijdens het college is een artikel genoemd met de titel "An efficient algorithm for solving nonograms". *Nonograms* kennen wij ook wel onder de naam 'Japanse puzzels'. In de introductie van het artikel schreven de auteurs: "Solving nonogram is a NP-complete problem.". Probeer zo nauwkeurig mogelijk te beschrijven wat hiermee bedoeld wordt? Dit artikel heeft niet voor al te grote opschudding gezorgd. Wat denk je wat dit zegt over de complexiteit van het gepresenteerde "efficient algorithm"?

3 The game of Life

totaal 25 punten

De *Game of Life* is een computersimulatie waarbij gebruik wordt gemaakt van een tweedimensionaal raster met vierkante cellen die levend of dood kunnen zijn. De Game of Life werkt met *generaties*. Om te bepalen of een cel in de volgende generatie dood of levend is wordt er een aantal regels toegepast. De nieuwe toestand van een cel hangt af van de *huidige* toestand van z'n 8 buurcellen (de cellen op de hoekpunten worden als buurcellen gerekend). Er geldt nu het volgende:



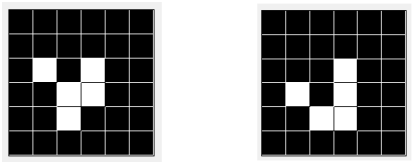
Figuur 1: game of life

i) Een levende cel gaat dood als hij minder dan 2 of meer dan 3 levende buurcellen heeft.

ii) Een dode cel komt tot leven als hij precies 3 levende buurcellen heeft.

iii) In alle andere gevallen verandert de toestand van de cel niet.

In onderstaand plaatje worden twee opeenvolgende generaties gegeven. Hierin zijn de witte cellen levend en de zwarte cellen dood.



Voor het linkerplaatje geldt het volgende. De witte cel op positie $(1, 2)$ heeft slechts één levende buur en zal dus op basis van regel i) dood gaan. De witte cel op $(2, 3)$ heeft juist te veel levende buren en zal op grond van dezelde regel eveneens sterven. De dode cel op $(1, 3)$ heeft precies 3 levende buren en zal (regel ii)) tot leven komen. Het rechterplaatje ontstaat uit het linker door voor alle cellen de Game of Life-regels toe te passen.

In deze opdracht ga je deze simultatie in Greenfoot implementeren. De cellen uit deze simulatie worden objecten van de klasse `GOLCell` (G(ame) O(f) L(ife) Cell). Het idee is om ieder veld in de Greenfoot-wereld te vullen met een `GOLCell`-object. Verder zal iedere keer dat de `act`-methode wordt aangeroepen elke cel zélf zijn volgende toestand bepalen.

De klasse `GOLCell`

We geven we een klein deel van de klasse `GOLCell` voor:

```
public class GOLCell extends Actor
{
    private boolean         myCurrentState;
    private GreenfootImage  myLivingBody;
    private GreenfootImage  myDeadBody;

    // Initialize the state and create the cell body's images
    public GOLCell ( boolean initialState, int cellSize ) {
        myCurrentState = initialState;
        myLivingBody = makeCellImage( cellSize, Color.WHITE, Color.BLACK );
        myDeadBody   = makeCellImage( cellSize, Color.BLACK, Color.WHITE );
        setImageOfCell();
    }

    public void act() {
        applyRules();
    }

    private List<GOLCell> getNeighborCells() {
        return getNeighbours( 1, true, GOLCell.class );
    }

    private GreenfootImage makeCellImage( int cellSize, Color fillColor, Color lineColor )
    { ... }

}
```

De toestand van de cel is opgeslagen in de **boolean** instantievariabele `myCurrentState`, waarbij **true** aangeeft *levend* en **false** *dood*. Verder hebben we twee instantievariabelen van het type `GreenfootImage` voor het opslaan van de plaatjes waarmee iedere cel wordt weergeven. Iedere toestand heeft z'n eigen plaatje. Het plaatje wordt gemaakt in de methode `makeCellImage`. We hebben de body van deze methode weggelaten omdat de details hiervan voor de opdracht niet van belang zijn. In `setImageOfCell` wordt het juiste plaatje aan het `GOLCell`-object gekoppeld.

3 punten **3.a**) Geef de definitie voor `setImageOfCell`. Gebruik voor het koppelen van een plaatje aan een de actor-methode

```
public void setImage( GreenfootImage image ).
```

De `act` methode bestaat enkel uit de aanroep van `applyRules`. Hierin wordt de nieuwe toestand van de cel bepaald aan de hand van de eerder gegeven regels. We kunnen echter de nieuwe toestand niet meteen aan de variabele `myCurrentState` toekennen. Het kan immers zijn dat voor de berekening van de toestand van een andere cel de huidige toestand nog nodig is. Eigenlijk kunnen we pas veilig de nieuw berekende toestand aan de huidige toestand toewijzen als **alle** cellen aan de beurt geweest zijn. Om die reden voegen we een nieuwe instantievariabele aan de klasse toe waarin we de nieuwe toestand opslaan.

2 punten **3.b**) Voeg een geschikte instantievariabele aan de klasse toe. Vergeet niet deze een initiële waarde te geven.

4 punten **3.c**) Maak de methode **private int** `countNeighborsAlive`() die het aantal levende buurcellen van de huidige cel bepaalt. Hint: je kunt de buurcellen van een cel opvragen met de methode `getNeighborCells`.

4 punten **3.d**) Definieer de methode `applyRules` waarin de nieuwe toestand van een cel bepaald wordt.

Het toekennen van de nieuwe toestand aan huidige toestand laten we aan de `GameOfLife` klasse (zie beneden) over. Deze heeft ook een `act`-methode waarvan je mag aannemen dat hij wordt uitgevoerd *nadat* de `act`-methodes van *alle* actoren aan de beurt geweest zijn.

3 punten **3.e**) Voeg tenslotte de (publieke) methode **`public void`** `updateState()` aan `GOLCell` toe waarmee de huidige toestand wordt geupdate met de nieuwe. Pas hierin ook, indien nodig, het plaatje aan.

De klasse `GameOfLife`

Ook nu geven we een klein deel van de klasse voor:

```
public class GameOfLife extends World
{
    public GameOfLife() {
        super( 50, 50, 20 );
        createCells();
    }

    public void act() {
        updateCells();
    }
}
```

2 punten **3.f**) Introduceer *klasseconstantes* voor de gebruikte integer-waardes. Opmerking: de afmetingen van de wereld zoals hier aangegeven komen niet overeen met die in het eerder gegeven voorbeeld.

4 punten **3.g**) De methode `createCells` maakt voor ieder veld van de wereld een `GOLCell`-object aan plaatst dat in de wereld. Geef een definitie van deze methode. Voor het plaatsen van actorobject in de wereld kun je gebruik maken van

```
public void addObject( Actor actor, int x, int y )
```

Zorg ervoor dat ongeveer een kwart van de toegevoegde cellen levend is. Hint: maak gebruik van de `Greenfoot`-methode **`int`** `getRandomNumber(` **`int`** `limit )`, waarbij je voor `limit` 4 invult.

3 punten **3.h**) Definieer de methode `updateCells` die elk `GOLCell`-object uit de wereld update. Een lijst met alle `GOLCell`-objecten kun je opvragen met de methode

```
public List getObjects( Class cls ).
```