Opdracht 7: Dodo's race

– Algoritmisch Denken en Gestructureerd Programmeren in Greenfoot –

1 Inleiding

In de vorige opgaven heb je al zelf aardig wat code geschreven en uitgevoerd. Nu ga je de dingen die je geleerd hebt combineren om een complexere programma te schrijven.

Je gaat een wedstrijd aan met jouw klasgenoten. In de wereld ligt één gouden ei (5 punten waard) en 15 blauwe eieren (ieder één punt waard). Het einddoel van deze opdracht is om Mimi zo te programmeren dat ze zo veel mogelijk punten verdiend door de eieren te vinden en deze uit te broeden. Het aantal stappen dat Mimi mag zetten is beperkt. Mimi moet dus met zo min mogelijk stappen zo veel mogelijke punten verdienen. Als laatste testen wij jouw programma in een nieuwe wereld. Dat scenario is nu nog geheim. Wie kan de slimste Mimi maken?

Het is de bedoeling dat je zelf tot een zo goed mogelijk algoritme komt en dit implementeert. Om je op weg te helpen gaan we eerst samen een paar eenvoudige algoritmes implementeren. Zo kan je ideeën opdoen voor verbeteringen. We geven je een aantal scenario's waar je jouw Mimi mee kan testen. Bij de uiteindelijke wedstrijd zal een nieuw scenario gebruikt worden. Het doel is dus om een algoritme te bedenken dat generiek genoeg is om in een onbekend scenario nog steeds de 'slimste' te zijn.

Om de wedstrijd mogelijk te maken moet ons programma als volgt uitgebreid worden:

- Mimi moet gaan bijhouden hoeveel stappen ze gezet heeft;
- Mimi moet bijhouden wat haar score is;
- met een scorebord, dat het aantal stappen en de score laat zien;
- het spel moet stoppen als het gegeven maximum aantal stappen gezet is;
- Mimi zo veel mogelijk punten verzamelen door eieren te vinden en uitbroeden.
- ... en Mimi moet véél slimmer worden!

2 Leerdoelen

Na het voltooien van deze opdracht kun je:

- uitleggen hoe een **`else if`** constructie werkt;
- uitleggen wat **Objecttypes** zijn;
- uitleggen wat het verschil is tussen Objecttypes en primitieve types bij het toekennen van een waarde;
- uitleggen wat **`null`** betekent en waar het voor gebruikt wordt;
- gebruik maken van willekeurige (*random*) getallen;
- uitleggen waarvoor **klasseconstanten** gebruikt worden;
- gebruik maken van klasseconstantes;

- voorbeelden geven van wanneer het handig kan zijn om een **`List`** te gebruiken;
- `List`-variabelen declareren en gebruiken;
- uitleggen dat een lijst-object naast primitieve typen ook andere objecttypen kan bevatten;
- een aantal bestaande methoden van een `List` toepassen, zoals het bekijken en verwijderen van elementen;
- Java Library Documentation gebruiken om bestaande Java methodes op te zoeken en te gebruiken;
- een *for-each-loop* gebruiken voor het doorlopen van elementen van een lijst;
- elementen in een lijst met elkaar verwisselen;

3 Instructies

In de vorige opdrachten heb je aardig wat code geschreven met als doel nieuwe principes te oefenen. Die code is wellicht wat onoverzichtelijk geworden. Verder zal je niet al die code nodig hebben voor deze opdracht. Daarom beginnen we in deze opdracht met een nieuw 'opgeschoonde' scenario. Als je straks een methode wilt gebruiken die je in een vorige opdracht al geschreven en getest hebt, mag je die natuurlijk gewoon kopiëren!

Voor deze opdracht heb je scenario **'DodoScenario7'** nodig. Een opmerking vooraf: op een kleine aanpassing in `Madagaskar` na, mag in deze opdracht alléén de klasse `MyDodo`worden aangepast.

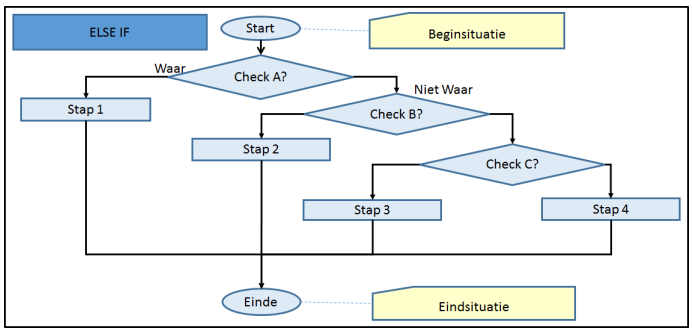
4 Uitleg en Opgaven

We beginnen eerst met twee stukken theorie. Daarna volgen de opgaven

Geneste `if` .. `then` .. `else` statements

Bij een *geneste* ***`if`*** .. `then` .. ***`else`*** worden meerdere gevallen getest. In een programma kun je dan meteen meerdere gevallen testen door **`if`** .. `then` .. **`else`** genest te gebruiken.

Stroomdiagram:
Het stroomdiagram van een geneste **`if`** .. `then` .. **`else`** ziet er als volgt uit:



Figuur 1: Stroomdiagram geneste **`if`** .. `then` .. **`else`**

Toelichting stroomdiagram:

- Eerst wordt er bij 'Check A?' gecontroleerd of de conditie in de ruit waar is.
- Als de conditie 'Waar' is, wordt de pijl 'Waar' naar links vervolgd en wordt 'Stap 1' uitgevoerd.
- Als de conditie 'Niet Waar' wordt de pijl 'Niet Waar' naar rechts vervolgd en wordt bij 'Check B?' gecontroleerd of de conditie in de ruit waar is.
- ...

Bijbehorende code:

```
if( checkA() ){
    stap1();
} else {
    if( checkB() ){
        stap2();
    } else {
        if( checkC() ){
            stap3();
        } else if (checkD() ) {
            stap4();
        }
    }
}
```

Korter omschrijven:
Deze vorm van nesting kan je in ook op een andere manier omschrijven:

- Als 'Check A?' 'Waar' is dan wordt 'Stap 1' uitgevoerd.
- Anders (dus A is 'Niet Waar') als 'Check B?' 'Waar' is dan wordt 'Stap 2' uitgevoerd.
- Anders (dus A en B zijn 'Niet Waar') als 'Check C?' 'Waar' is dan wordt 'Stap 3' uitgevoerd.
- Anders (dus A, B en C zijn 'Niet Waar') als 'Check D?' 'Waar' is dan wordt 'Stap 4' uitgevoerd.

Bijbehorende code:
De **`else`** en de **`if`** worden hierbij gecombineerd. Zo komen ze samen op één regel te staan. De methode van hierboven komt er dan zo uit te zien:

```
if( checkA() ){
    stap1();
} else if ( checkB() ){
    stap2();
} else if( checkC() ){
    stap3();
} else if ( checkD() ){
     stap4();
}
```

We hebben nu minder accolades en minder regels nodig. Bovendien is de structuur is veel overzichtelijker.

Toevoeging:
Bij de laatste conditie kan een **`else`** of een **`else if`** gebruikt worden. Deze hebben wel een ander betekenis! De code na een **`else if`** zal alléén uitgevoerd worden als nog aan de laatste conditie is voldaan.

Objecttypes
Objecttypes zijn types die horen bij objecten en die overeenkomen met de klassen waartoe de objecten behoren, zoals `MyDodo` en `Egg`. Java kent dus naast primitieve types (zoals **`int`** en **`boolean`**) ook *objecttypes*, soms ook wel *referentietypes* genoemd. De primitieve types zijn allemaal in Java ingebouwd. Dat betekent dat je primitieve types cadeau krijgt. Objecttypes horen bij een klasse. Deze kun je dus zelf maken door een nieuwe klasse te schrijven of krijg je zodra je een klasse importeert.

Gebruik:
Je mag objecttypes op dezelfde plekken gebruiken als primitieve types. Ter herinnering, types vind je op de volgende plekken in de code terug:

Voorkomen	**Voorbeeld**	**Type**
resultaat	`int methodeNaam( )`	`int`
parameter	`void methodeNaam( String tekstje )`	`String`
lokale variabele	`int getal = 4`	`int`
instantievariabele	`private int myNrEggsFound = 0`	`int`

Een voorbeeld dat we in opdracht 5 zijn tegengekomen, maar we stiekem overheen zijn gestapt is het volgende:

```
World myWorld = getWorld( );
```

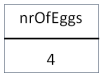
Hierin wordt een variabele met naam `myWorld` en type `World` aangemaakt (gedeclareerd) en meteen geïnitialiseerd met het resultaat van `getWorld( )`. Het resultaattype is `World`. Dit zie je aan de signatuur **`public`** `World getWorld( )`. Omdat `World` een klasse is hebben we hier te maken met een objecttype.

Waardes:
Het type bepaalt met wat voor soort waardes er gewerkt wordt. Bijvoorbeeld, in een variabele van het type **`int`**, zoals **`int`** `nrOfEggs` kun je alleen maar gehele getallen opslaan. Daar mag je geen `bool` of `String` in opslaan. Hetzelfde geldt voor objecttypes. In een variabele waarvan het type een klasse is, bijvoorbeeld `Egg`, kunnen we alleen `Egg`-objecten opslaan. Daar kan je dus geen `Fence`-objecten of een **`int`**-waarde in opslaan. Wel kunnen we in zo'n `Egg` variabele een `BlueEgg`- of `GoldenEgg`-object stoppen. Dit komt omdat ieder `BlueEgg`- of `GoldenEgg`-object ook een `Egg`-object is (zie het hoofdstuk 'Overerving' in opdracht 1).

Een voorbeeld van een variabeledeclaratie met een objecttype waar meteen een initiële waarde aan gegeven wordt is: `Egg firstEgg = new BlueEgg( );`

Toevoeging:

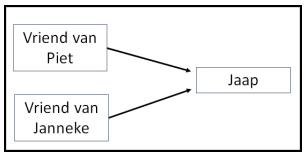
- Op het eerste gezicht lijkt er geen verschil te zijn tussen primitieve types en objecttypes. Maar dat is er wel. Een van de verschillen is hoe waardes worden opgeslagen. Als we een variabele `nrOfEggs` met een primitief type zoals **`int`** hebben, dan wordt een waarde van dat type, bijvoorbeeld 4, rechtstreeks in die variabele opgeslagen.



Figuur 2: Waardes van primitieve types worden in de variabele opgeslagen

Voor objecten geldt dat niet. Deze worden niet rechtstreeks maar in de vorm van een verwijzing (oftewel 'referentie') naar dat object opgeslagen. Dit hun je een beetje vergelijken met de manier waarop je in een programma als Facebook je *friends* bewaart.

Dat doe je natuurlijk niet rechtstreeks (je kan natuurlijk niet je vrienden echt opslaan) maar door de Facebook-login van je vriendje op te slaan. Deze login kun je zien als een verwijzing naar de echte persoon. Bovendien kunnen andere Facebook-gebruikers deze login ook gebruiken om te verwijzen naar dezelfde persoon. In het voorbeeld hieronder zie je dat zowel Piet als Janneke een verwijzing naar Jaap hebben.



Figuur 3: Waardes van objecttypes zijn verwijzingen naar objecten

In een Java programma gebeurt eigenlijk hetzelfde. Je kunt meerdere variabelen hebben die allemaal naar hetzelfde object wijzen en dus dezelfde verwijzing als waarde hebben.

- Een veelgemaakte programmeerfout is het gebruiken van het verkeerde type. Bijvoorbeeld, de compiler zal gaan mopperen over een poging om aan een `Egg`-variabele de **`int`**-waarde 3 toe te kennen:

```
// incorrect assignment
Egg thirdEgg = 3;
```

4.1 Instantievariabelen die naar objecten wijzen

In deze opdracht gaan we oefenen met objecttypes. Daarvoor breiden we eerst de klasse `MyDodo` uit met twee `Egg` instantievariabelen. We voegen ook settermethodes toe zodat Mimi aan die variabelen een waarde (een ei) kan geven om ze daarna uit te kunnen broeden. Dit doen we als volgt:

1. Zorg dat dat je scenario steeds geopend wordt met de wereld 'world_mimi_2_eggs.txt'.
2. Voeg aan de klasse `MyDodo` twee instantievariabelen toe, beide van het type `Egg`. Noem de eerste `myFirstEgg` en de tweede `mySecondEgg`.
3. Voeg voor beide variabelen een settermethode toe waarmee je deze variabelen een waarde (een gegeven ei) kunt geven. De signatuur van de settermethode voor `myFirstEgg` is:

   ```
   public void setFirstEgg( Egg newEgg )
   ```

   Vul deze aan met de juiste code om de meegegeven waarde `newEgg` aan `myFirstEgg` toe te kennen. Doe daarna hetzelfde voor `mySecondEgg`.
4. Compileer jouw code alvast om te controleren of je geen tikfouten hebt gemaakt.
5. Voeg nu een methode toe om het ei uit te broeden. Schrijf voor `myFirstEgg` de methode **`public void`** `hatchFirstEgg( )` die de toestand van het ei `myFirstEgg` wijzigt naar uitgebroed. Gebruik daarvoor `myFirstEgg.setHatched()`. Doe hetzelfde voor de andere instantievariabele.
6. Compileer jouw code.

7. Klik met je rechtermuisknop op Mimi en roep de methode **void** `hatchFirstEgg( )` aan. Wat gebeurt er? Zie je niets? Kijk dan even bij het console-venster. Welke boodschap zie je? Dit gebeurt als je iets met objectvariabelen wil doen zonder dat je deze eerst een waarde hebt gegeven. Je hoeft hier nu niets mee te doen. Waarom dit gebeurt komen we zo meteen nog op terug.
8. Om dit op te lossen wijzen we eerst een ei aan. Roep de methode **void** `setFirstEgg( Egg newEgg )` aan (met de rechtermuisknop).
9. Er verschijnt een venster waarin je de parameter moet aangeven. Als parameter willen we een verwijzing naar het bovenste ei geven. Dit doe je door op het bovenste ei te klikken. Op deze manier zorg je ervoor dat de instantievariabele `myFirstEgg` wijst naar het bovenste ei-object.
10. Roep nu **void** `hatchFirstEgg( )` aan. Wat gebeurt er? Verklaar wat je ziet.
11. Doe hetzelfde met de tweede instantievariable `mySecondEgg`. Deze laat je naar het onderste ei wijzen, op dezelfde manier als dat je bij `myFirstEgg` in onderdeel 9 hebt gedaan. Controleer of wat je ziet ook overeenkomt met wat je had verwacht.
12. Druk op *Reset* om het scenario te herstarten.
13. Roep de methode **void** `setFirstEgg( Egg newEgg )` aan en geef hier een verwijzing naar het bovenste ei aan mee. Roep de methode **void** `setSecondEgg( Egg newEgg )` aan en geef hier **ook** het bovenste ei aan mee.
14. Roep nu **void** `hatchFirstEgg( )` aan. Gebeurt er wat je verwacht had?
15. Roep nu ook **void** `hatchSecondEgg( )` aan. Wat zie je gebeuren? Geef een verklaring voor wat je ziet. Gebruik daarvoor de informatie over *referentietypes* uit het bovenstaande theorieblok. Tip: teke een plaatje van de situatie die is ontstaan.

null
Een instantievariable met een objecttype die je wel al hebt gedeclareerd, maar (nog) geen initiële waarde hebt gegeven, wijst op dat moment niet naar een bepaald object. Bijvoorbeeld met:

```
private Egg myFirstEgg;
```

wordt de variabele `myFirstEgg` gedeclareerd, maar er wordt nog geen object aan toegekend: `myFirstEgg` wijst nu nog nergens naar. In Java wordt het sleutelwoord **null** gebruikt om dit aan te geven. **null** is dus een speciale waarde met als betekenis 'geen object'. Tenzij je ze zelf een initiële waarde geeft, krijgen standaard in Java alle instantievariabelen die een objecttype hebben automatisch de waarde **null**.

Als een variabele de waarde **null** heeft, dan mag je er geen methode-aanroep op doen. Methoden horen bij objecten. Ze veranderen iets aan de toestand van dat object of leveren informatie op over het object. Het object mag dus niet ontbreken.

Een veel voorkomende fout is een methode-aanroep doen op een variabele die de waarde **null** heeft. Je krijgt dan een `NullPointerException`. Dit hebben we zojuist gezien toen je in opgave 4.1 onderdeel 7 de methode **void** `hatchFirstEgg( )` aanriep zonder eerst de variabele `myFirstEgg` te initialiseren met `setFirstEgg`. Op dat moment had `myFirstEgg` nog steeds de waarde **null** en de methode-aanroep hierop leverde een `NullPointerException` op.

Afspraak:
Het is een goede gewoonte om in de constructor elke instantievariabele altijd zelf een waarde te geven (initialiseren). Weet je nog niet welke waarde die moet krijgen, gebruik dan **null**.

4.2 Dodo scoort

Help Mimi alle eieren in de wereld te vinden (er heen te lopen) en uit te broeden. Dit wordt een wedstrijdje tegen jouw medestudenten. Wie kan het beste algoritme bedenken? Mimi mag 40 stappen zetten. Met wiens algoritme kan ze de meeste eieren uitbroeden?

We gaan nu verschillende algoritmes bedenken en uitproberen waarmee Mimi in zo weinig mogelijke stappen zo veel mogelijk eieren weet uit te broeden.

Ga in de volgende opgaven gestructureerd te werk! Eerst bedenk je een **simpel** algoritme, tekent daar vervolgens een stroomdiagram bij, werkt dit uit in code en test deze. Daarna ga je stapsgewijs aan de slag om verbeteringen aan te brengen.

4.2.1 Willekeurige bewegingen

Het eerste simpele algoritme maken we samen. In deze opdracht laten we Mimi als een dronken Dodo door haar wereld lopen. Dat wil zeggen, elke keer kiest ze, voordat ze een stap zet, een willekeurige (*random*) richting.

1. Het bepalen van een willekeurige richting kan met de `Dodo`-methode:

   ```
   public int randomDirection( )
   ```

   Open de klasse `Dodo` en zoek deze methode op.

2. Deze methode maakt gebruik van een standaard methode uit de Greenfoot bibliotheek, namelijk `getRandomNumber`. Alle Greenfoot methoden staan beschreven in het 'Greenfoot Class Documentation'. Zoek deze methode op in de Greenfoot bibliotheek. Tip: Ga bovenin Greenfoot naar 'Help' en dan 'Greenfoot Class Documentation'. Via 'Index' kun je makkelijk zoeken.

3. Wat betekent de parameter '4' die aan de methode meegegeven wordt?

4. Klik met je rechtermuisknop op Mimi en kies de `Dodo` methode `randomDirection`. Roep de methode een aantal keren aan en noteer steeds het resultaat. Wat valt je op? Wat zie je?

5. Bedenk een algoritme waarin Mimi telkens een nieuwe willekeurige bewegingsrichting kiest om daarna een stap in die richting te zetten. Voorkom dat Mimi van de wereld valt of tegen een hek opbotst door eerst te controleren of ze die stap ook echt kan zetten.

6. Schrijf in `MyDodo` de bijbehorende methode `moveRandomly( )`.

7. Compileer en test jouw methode met de rechtermuisknop.

8. Doet Mimi wat je van haar verwacht verwacht had?

9. Pas de code in de `act`-methode van `MyDodo` aan zodat deze `moveRandomly( )` aanroept.

10. Compileer, run en test het programma. Werkt het niet, volg de stappen zoals beschreven in hoofdstuk 'Debuggen' van opdracht 2.

11. Breid `moveRandomly( )` uit zodat Mimi een ei uitbroedt zodra ze er eentje gevonden heeft.

Klasseconstante
Een klasseconstante is een variabele waarvan de waarde tijdens het uitvoeren van het programma niet kan veranderen. Je herkent een constante in de code aan **`static final`**. Klasseconstantes kunnen zowel **`public`** als **`private`** zijn.

Voorbeeld
Een voorbeeld van de declaratie van een constante (bovenaan de klasse `Madagaskar`) is:

```
private static final int MAXWIDTH=12;
```

Deze constante is **private**. Daardoor is deze alleen te gebruiken binnen de klasse `Madagaskar` zelf. Door de toevoeging **final**:

- is het onmogelijk om de waarde ergens anders in het programma te veranderen;
- ben je verplicht om zo'n klasseconsante meteen een waarde te geven, dus direct bij de declaratie.

De klasseconstante `MAXWIDTH` heeft de waarde 12 omdat we een wereld van 12 cellen breed willen hebben. Als je een spelletje maakt en een wereld van 40 bij 40 wil hebben kun je deze waarde eenvoudig aanpassen door in de declaratie 12 te vervangen door 40. Als je overal in de code verwijst naar `MAXWIDTH` in plaats van direct gebruik te maken van 12, dan zal jouw programma prima werken in een grotere wereld.

Naamgevingsafspraken van een constante:
De naam van een constante:

- is betekenisvol: het komt overeen met waar de constante voor gebruikt wordt;
- bestaat uit één of meer woorden;
- bestaat uit letters, cijfers en '_': bevat geen spaties, komma's of andere 'rare' tekens;
- is geschreven in hoofdletters: woorden worden gescheiden door een '_';
- bijvoorbeeld: `WORLD_FILE`.

Publieke constantes
Klasseconstantes waarvan je hebt aangegeven dat ze **public** zijn kun je vanuit andere klassen aanroepen. Bovenin de klasse `Dodo` zie je vier publieke klasseconstantes staan. De eerste declaratie is:

```
public static final int NORTH = 0;
```

Deze constante wordt in de klasse `Dodo` op verschillende plekken gebruikt. Omdat hij **public** is kun je hem dus ook in andere klassen gebruiken. Wil je hem vanuit een andere klasse aanroepen, zoals bijvoorbeeld de `Egg` klasse, dan moet je aangeven uit welke klasse de constante afkomstig is. Dat doe je door de klassenaam voor de constante te zetten, gescheiden door een punt '.'. In dit geval: `Dodo.NORTH`. Bekijk bijvoorbeeld de methode `public void push ( int direction )` uit de klasse `Egg`. Daar wordt de conditie `direction == Dodo.NORTH` gebruikt om te testen of `direction` van `Dodo` gelijk is aan `NORTH`.

4.2.2 Scorebord

Het spel is afgelopen zodra er een maximaal aantal stappen gezet is. We passen het programma aan zodat het stopt als dit aantal bereikt is. Daarnaast gebruiken we een scorebord om aan te geven hoeveel stappen er nog gezet mogen worden en wat het puntenaantal (score aan de hand van uitgebroede eieren) is:

1. Het maximaal aantal stappen noemen we `MAXSTEPS`. Omdat we het maximum aantal stappen later misschien makkelijk willen veranderen gaan we niet de waarde '40' zelf gebruiken, maar een klasseconstante `MAXSTEPS`. De klasseconstante `MAXSTEPS` is al voor je toegevoegd aan de klasse `Madagaskar`. Pas de waarde aan zodat die juist is.
2. We voegen een scorebord toe aan het scenario. Dat doe je als volgt:

(a) Open de klasse `Madagaskar` en zoek in de constructor de aanroep van de methode `addScoreboard( )`. Deze staat in commentaar.

(b) Verwijder het commentaar zodat een scorebord wordt toegevoegd als de wereld gecreëerd wordt.

(c) Compileer.

(d) Zoals je in het scenario ziet is er onderin een scorebord toegevoegd. Hoeveel waardes staan er op het scorebord?

3. Er is ook een `Dodo`-methode `updateScores` waarmee Mimi de score op het scoreboard kan aanpassen. Dat doet Mimi zodra ze een ei gevonden heeft of een stap heeft gezet. Klik met je rechtermuisknop op Mimi en selecteer

   ```
   void updateScores( int score1, int score2) (inherited from Dodo)
   ```

   Vul nu twee getallen als parameters in. Wat zie je gebeuren op het scorebord? Tip: De methode zit misschien in het menu verscholen achter 'meer methodes'.

4. Het tellen van het aantal stappen doet Mimi zelf. In de klasse `MyDodo` staan daarvoor al twee instantievariabelen `myNrOfStepsTaken` en `myEggScore`. De variabele `myNrOfStepsTaken` houdt het aantal gezette stappen bij, en `myEggScore` het puntentotaal (score). Om deze te gebruiken passen we de volgende stappen toe:

   (a) Geef deze instantievariabelen in de constructor de juiste initiële waarde.

   (b) Zorg er voor dat `myNrOfStepsTaken` wordt aangepast zodra Mimi een stap zet. Tip: Doe dit in `move( )`.

   (c) Zorg er voor dat `myEggScore` wordt aangepast zodra Mimi een ei uitgebroed. Tips:
   - Doe dit in de methode `hatchEgg( )`;
   - De waarde van een bepaald ei (zijn score) kun je opvragen met `getValue( )`.
   - Weet je niet hoe je de waarde van `MAXSTEPS` kunt gebruiken? Lees dan het bovenstaande theorieblok opnieuw door.

   (d) Roep ook **`void`** `updateScores` aan om de juiste waardes op het scorebord weer te geven. Tips:
   - Het aanpassen van de score doe je direct nadat deze waardes veranderd zijn.
   - Let erop dat de eerste waarde aangeeft hoeveel stappen Mimi nog mag zetten (en dus niet hoeveel ze er al gezet heeft).

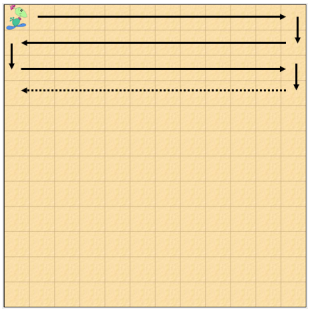
5. Test het programma door `moveRandomly( )` methode in `act` aan te roepen. Test het ook met gouden eieren (5 punten) en uitgebroede eieren (0 punten).

6. Wat gebeurt er nadat Mimi 40 stappen gezet heeft?

7. Pas het programma aan zodat dit, voordat Mimi een stap wil zetten, controleert of dat nog kan. Als Mimi het maximaal aantal stappen al gezet heeft, toon dan een compliment met haar behaalde score en stop het programma.

4.3 Lijsten van eitjes doorlopen

De methode `moveRandomly( )` heeft Mimi nog niet echt slimmer gemaakt. We hebben slimmere strategieën uitgewerkt om eieren te vinden. Bijvoorbeeld, jouw methode `walkThroughWorldAndCountEggs( )` in opgave 'Aantal eieren in de wereld tellen' uit opdracht 5. Daarbij liep Mimi alle rijen systematisch af zoals in het volgende plaatje is aangegeven:



Figuur 4: Doorloop de hele wereld

Het zoeken en uitbroeden van de eieren kan echter nog slimmer. In Java kun je namelijk gebruik maken van lijsten. Stel je zou een lijst hebben van alle ei-objecten in de wereld. Zou je dan een slimmer algoritme kunnen verzinnen om met minder stappen alle eieren te vinden? In deze opgave leer je werken met lijsten.

Lists
De objectvariabelen die we tot nog toe zijn tegengekomen konden precies één object bevatten. Soms is het handig om niet één maar een hele reeks van objecten bij te houden. Natuurlijk kunnen we voor ieder object uit die reeks een aparte variabele aanmaken, maar dat is in de praktijk niet erg handig. Wat we dus eigenlijk willen is een algemene manier om objecten te groeperen. Dat kan in Java op verschillende manieren. Een van de eenvoudigste en meest voorkomende wijzen is door gebruik te maken van lijsten.

Een lijst (`List` in Java) is eigenlijk een hele reeks van variabelen. We noemen die variabelen de *elementen* van de lijst. Deze elementen worden opeenvolgend bijgehouden. Je kunt ze dus nummeren en dat is ook de manier waarop je bij ieder element kunt komen, namelijk, door gebruik te maken van zijn positie. Bij het bepalen van de positie van een element in een lijst beginnen we te tellen bij 0. Dus het eerste element heeft nummer 0, het tweede 1 enz. De positie noemen we de *index* van een element. Het tiende element van een lijst heeft dus index 9.

Elementen van lijsten:
De elementen van een lijst dienen allemaal van hetzelfde type te zijn. We kunnen in één en dezelfde lijst geen appels en peren opslaan; het is óf een lijst van alléén appels óf een lijst van alléén peren.

Operaties op lijsten:
Lijsten hebben een grootte ('size') die aangeeft hoeveel elementen er in de lijst zitten. Een lijst kan groeien door er elementen aan toe te voegen of korter worden door er elementen uit te verwijderen. Het toevoegen of verwijderen kan op willekeurige posities in de lijst gebeuren: vooraan, achteraan of ergens in het midden. Ook kan een lijst leeg zijn. Dan bevat hij geen elementen.

Lijsten in Java:
De Java klasse voor lijsten heet `List`. Als je lijsten wil gebruiken moet je daarvoor een special bibliotheek importeren. Dat doe je door bovenaan het klassedocument het volgende toe te voegen: `import java.util.List;`. De Java bibliotheek voor lijsten bevat allerlei handige methodes, zoals:

Lengte: `size( )`. Geeft aan hoeveel elementen er in de lijst zitten.

Selecteer: `get( int index )`. Levert het element op positie 'index' op.

Voeg toe: `add( int index, E element )`. Voegt 'element' toe aan de lijst op positie 'index'.

Verwijder: `remove( E element )`. Verwijdert 'element' uit de lijst.

Is leeg?: `isEmpty( )`. Geeft aan of een lijst leeg is. Deze gebruik je om te controleren of een lijst nog elementen bevat; je kunt namelijk geen elementen uit een lege lijst halen.

Bevat: `contains( Object o )`. Kijkt of object 'o' in de lijst voorkomt.

Bekijk zelf de Java bibliotheek van `List` voor een compleet overzicht van alle methoden door één van de volgende twee stappen uit te voeren:

- `https://docs.oracle.com/javase/7/docs/api/java/util/List.html`
- Ga naar het Greenfoot menu en kies 'Help', en dan 'Java Library Documentation'. Zoek daarna op 'List'.

Het is belangrijk om in te zien dat lijsten zelf ook weer objecten zijn. Speciaal aan zo'n lijst-object is dat het andere objecten bevat. Verder gelden voor lijst-objecten dezelfde spelregels als voor andere objecten. Allereerst gebruik je meestal een lijstvariabele om aan een lijst-object een naam te geven zodat je dit lijst-object kunt gebruiken. Een variabele die een lijst bevat (of eigenlijk, als we heel precies willen zijn, die naar een lijst-object wijst) kunnen we als volgt declareren:

```
List<...> lijstNaam;
```

Hierin is `lijstNaam` de naam van de variable en dient op de puntjes een type te worden ingevuld dat overeenkomt met de soort van elementen die je er in wil opslaan. Als je bijvoorbeeld een lijst van teksten wil dan vul je `String` op de puntjes in. Bijvoorbeeld een lijstje van complimentjes voor Mimi:

```
List<String> complimentTexts;
```

Een ander voorbeeld vind je terug in de `MyDodo` klasse. Hierin staat een methode die een lijst van alle eieren in de wereld oplevert:

```
public List<Egg> getListOfEggsInWorld()
```

Je zult het resultaat van de methode nodig hebben in deze opdracht. Daarvoor is het handig dat je een variable declareert waarin dat resultaat kan worden opgeslagen. Dat zou je als volgt kunnen doen:

```
List<Egg>  eggsInTheWorld = getListOfEggsInWorld( );
```

Hiermee declareer je dus een variabele met de naam `eggsInTheWorld` die als type `List<Egg>` heeft en die als initiële waarde het resultaat van de aanroep van de methode `getListOfEggsInWorld` heeft.

De for-each-loop:
De *for-each-loop* wordt in Java gebruikt om alle elementen van een lijst stuk voor stuk te bekijken.

```
void methodeDoeIetsMetLijst( ) {
    for ( ElementType elemVariabele: lijstVanElementen ) {
        doIetsMetHetElement( elemVariabele );
    }
}
```

Hierin betekent `ElementType` het type van de elementen in de lijst. De loop wordt even vaak doorlopen als dat de lijst lang is. De variabele `elemVariabele` krijgt bij iedere herhaling het volgende object uit de lijst als waarde. Dit object kan dan gebruikt worden in `doIetsMetHetElement`.

Voorbeeld:
De volgende code loopt door de lijst heen en broedt ieder ei-element uit:

```
public void hatchEachEggInWorld() {
    List<Egg> eggList = getListOfEggsInWorld( );
    for ( Egg egg: eggList ) {       // get the egg in the list
        egg.setHatched( true );      // and hatch that egg
    }
}
```

Na afloop zullen alle eieren de toestand 'uitgebroed' hebben.

In deze opgaven hoef je zelf geen lijstobjecten aan te maken. De lijsten die we gaan gebruiken zullen steeds door voorgegeven methodes worden opgeleverd. Wel heb je lijstvariabelen nodig om het resultaat van zulke methodes in op te kunnen slaan. Dan kan je daarna iets met deze lijsten doen, zoals er doorheen lopen.

4.3.1 Lijst van eieren maken en coördinaten afdrukken

In deze opgave ga je oefenen met lijsten. Help Mimi om een lijst met alle eieren te maken en daarna deze één voor één te doorlopen en de coördinaten van de eieren af te drukken.

1. Open de wereld 'world_eggs'.
2. Bekijk de Java bibliotheek van `List`.
   (a) Zoek in de Java Library Documentation een bestaande methode op die een element op een bepaalde positie (of `index`) in de lijst verwijdert.
   (b) Hoe roep je deze methode aan om een object vooraan de lijst te verwijderen?
   (c) En hoe verwijder je het derde object?
3. Om een lijst van alle eieren te maken gebruik je `getListOfEggsInWorld()` uit het bovenstaande theorieblok. Zoek de methode op en voorzie deze van commentaar.
4. Schrijf een methode die de lijst één voor één doorloopt en van ieder ei de coördinaten afdrukt in de console:
   (a) Open de code voor `MyDodo` in de editor. Als het nog niet gebeurt is, voeg het volgende bovenaan toe:

   ```
   import java.util.List;
   ```

   onder de `import greenfoot.*;`.
   (b) Kies een geschikte methodenaam voor het afdrukken van de coördinaten van de eieren.
   (c) Maak een lijst gevuld met alle `Egg`-objecten in de wereld. Tip: Gebruik onderdeel 3.
   (d) Gebruik een *for-each-loop* om de lijst te doorlopen.
   (e) Druk voor ieder ei in de lijst de coördinaten af. Maak hiervoor gebruik van de Java `println` methode. Tip: De coördinaten van een ei genaamd `egg` kun je met `egg.getX( )` en `egg.getY( )` achterhalen.

5. Test jouw methode met de rechtermuisknop. Werkt deze ook als er geen eieren in de wereld liggen?

4.3.2 Java methoden voor lijsten

Stel je hebt een lijst met eieren genaamd `myEggList`. Bekijk de Java bibliotheek van `List`. Met welke code kun je:

1. Controleren of die lijst bestaat (dat wil zeggen, of `myEggList` ook echt naar een lijst wijst)?
2. Controleren of de lijst leeg is?
3. Het derde ei uit de lijst krijgen?
4. Het tweede ei uit de lijst verwijderen?
5. Een ei genaamd `egg` uit de lijst verwijderen?
6. Een ei genaamd `egg` vooraan in de lijst toevoegen?

4.3.3 Omkeren van lijsten

Nieuw aan het scenario zijn surprise-eieren. De klasse `SurpriseEgg` breidt net als de blauwe en gouden eieren de basisklasse `Egg` uit. Blauwe eieren zijn één punt waard, en gouden zijn ieder vijf punten waard. De waarde van een surprise-ei is een verrassing. Dat wil zeggen, deze wordt in de constructor op een willekeurige waarde gezet. Hiervoor wordt de Greenfoot methode `getRandomNumber` gebruikt. Ieder surprise-ei heeft dus een willekeurige waarde.

Met de `SurpriseEgg` methode `generateListOfSurpriseEggs` kun je een hele rij van `SurpriseEgg`-objecten aanmaken. Deze methode verwacht als parameter een getal: het aantal eieren die in het lijst moet komen te staan. Een lijst van 10 surprise-eieren maak je met de volgende aanroep:

```
SurpriseEgg.generateListOfSurpriseEggs( 10 );
```

Zoals je ziet moet je de aanroep vooraf laten gaan door de naam van de klasse waarin deze methode staat, namelijk `SurpriseEgg`. Het resultaat is een lijst van surprise-eieren: `List<SurpriseEgg>`.

In de volgende opdrachten ga je een aantal methodes schrijven waarin je de elementen uit de lijst niet alleen bekijkt, maar ook in de lijst op een andere plek zet. Op deze manier zou je bijvoorbeeld lijsten kunnen sorteren.

1. **Lijst aanmaken en afdrukken:** We beginnen met het maken van de lijst met surprise-eieren en deze afdrukken.
   (a) Bedenk een geschikte naam voor deze methode.
   (b) Genereer met `generateListOfSurpriseEggs` een lijst van 10 surprise-eieren.
   (c) Druk van ieder surprise-ei uit deze lijst de waarde af in de console met `println`. Maak hiervan een aparte submethode waarin je een *for-each-loop* gebruikt om de lijst te doorlopen.
2. **Meest waardevolle ei:** Het eerste algoritme is vrij eenvoudig: het vinden van het meest waardevolle ei.
   (a) Bedenk een algoritme om het ei met de hoogste waarde uit bovenstaande lijst te bepalen. Tip: Terwijl je het lijst doorloopt hou je in een (lokale) variabele bij wat de hoogste waarde is die je tegen bent gekomen. Zoiets heb je ook al gedaan in opdracht 5 bij 'Bepaal de rij met de meeste eieren'.

(b) Implementeer dit algoritme.

(c) Compileer en test het algoritme door de gevonden waarde af te drukken.

(d) Klopt jouw uitkomst? Voer het een aantal keren uit en controleer steeds of de juiste waarde wordt gevonden. Zitten er nog fouten in je algoritme, verbeter deze totdat het goed werkt.

3. **Voor wie zich uitgedaagd voelt (dus niet verplicht): lijsten omkeren:** Schrijf een methode die de lijst van surprise-eieren omkeert. Dat wil zeggen, de elementen uit die lijst zo herrangschikt dat het laatste element vooraan komt te staan, gevolgd door het één na laatste element, enz.

(a) Bedenk een geschikte naam voor deze methode.

(b) Genereer opnieuw een lijst van 10 surprise-eieren.

(c) Druk de waardes van de eieren in deze lijst af. Roep hiervoor de submethode die je in de vorige opgave bij onderdeel 1c geschreven hebt.

(d) Bedenk een algoritme om de elementen van een lijst om te keren. Tips:

i. Het verwisselen van 2 waardes heb je al in opdracht 5 'variabelen nasporen' onderdeel 13 gezien.

ii. Bekijk de verschillende beschikbare methodes in de Java Library Documentation (zoals `add`, `get`, `set`, `remove`)en beslis welke je wilt gebruiken voor jouw algoritme.

iii. Implementeer eerst het algoritme voor het verwisselen voor de eerste en het laatste element.

iv. Druk de lijst opnieuw af.

v. Compileer en test dit. Check ook of je nog steeds evenveel eieren in jouw lijst hebt.

(e) Implementeer het algoritme om de lijst om te keren. Tips:

- Gebruik een **`while`**−`loop` (en geen *for-each-loop*).
- Gebruik een lokale variabele `index` die bijhoudt waar je in de lijst bent. Vergeet niet om deze variable binnen de **`while`**-loop steeds te verhogen.

(f) Compileer en test het algoritme door nogmaals de waarde van ieder element af te drukken.

(g) Werkt je programma als verwacht? Zo niet, bedenk waarom het doet wat het doet. Druk eventueel tussentijds waarden af om te zien wat er precies gebeurt. Verbeter dan jouw programma.

(h) Werkt het programma ook goed als er 11 eieren in het lijst zitten? En ook als er 0 eieren in zitten?

4.3.4 Eerste ei in lijst vinden en uitbroeden

Laat Mimi het eerste ei uit een lijst halen, er naar toe lopen en dit uitbroeden. Dit algoritme zal uit de volgende stappen/taken bestaan:

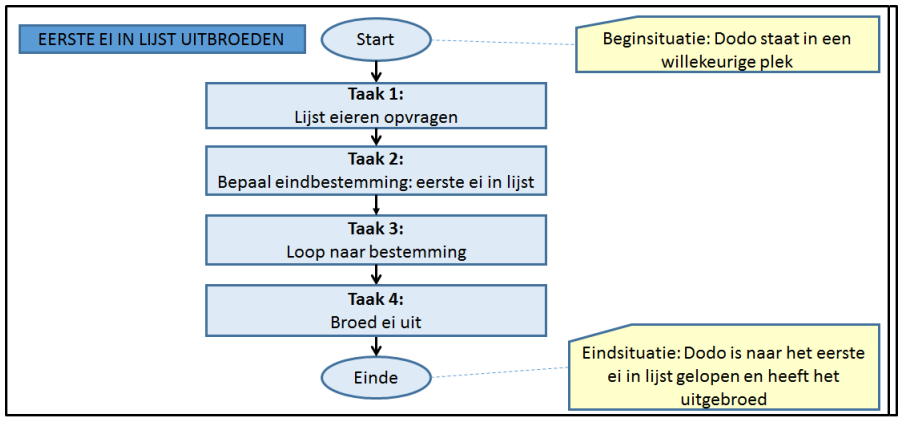
Taak 1: Vraag de lijst met eieren op (en druk deze af);

Taak 2: Haal het eerste element uit deze lijst en gebruik dit om Mimi's eindbestemming te bepalen;

Taak 3: Laat Mimi naar het ei lopen;

Taak 4: Laat Mimi het ei uitbroeden.

Het bijbehorende stroomschema ziet er dan zo uit:



Figuur 5: Stroomdiagram voor 'Eerste ei in lijst vinden en uitbroeden'

Je hebt hier te maken hebt met een algoritme dat uit verschillende deeltaken bestaat die na elkaar moeten worden uitgevoerd.

1. Bedenk geschikte methodenaam.
2. In de body van deze methode worden de vier deeltaken uitgevoerd:

   Taak 1: (a) Maak een lijst gevuld met alle `Egg`-objecten in de wereld. Gebruik hiervoor de klassevariabele `eggsInTheWorld`. Tip: Net zoals bij de opgave 4.3.1 onderdeel 3, gebruik je weer `getListOfEggsInWorld()` om een lijst te krijgen van alle eieren die zich in de wereld bevinden.

   (b) Druk de lijst af.

   Taak 2: (a) Controleer in de code of de lijst niet leeg is. Je mag namelijk niet iets uit een lege lijst proberen te halen.

   (b) Haal het eerste ei uit de lijst.

   (c) Druk de coördinaten van dat ei af met `println`.

   Taak 3: Ga naar dat ei toe. Tip: Gebruik hiervoor de methode

   ```
   public void goToLocation( int coordX, int coordY )
   ```

   die je eerder in opdracht 5 geschreven hebt.

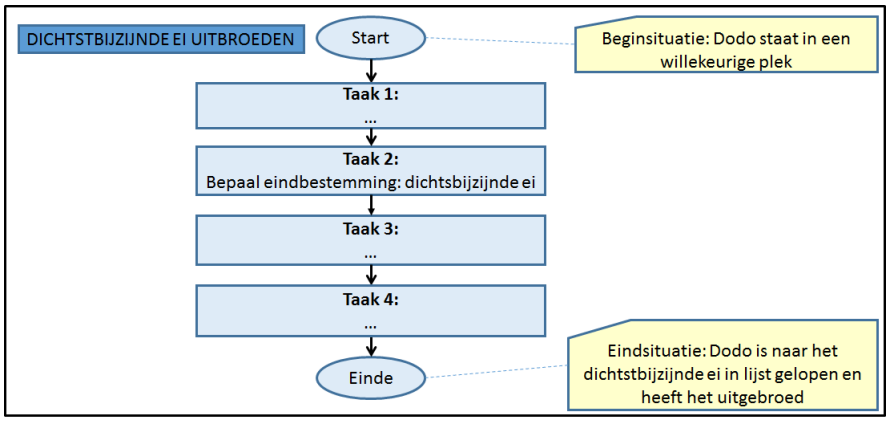
   Taak 4: Broed het ei uit. Tip: maak ook hierbij gebruik van een bestaande methode.

3. Compileer en test jouw methode. Dit laatste doe je door met de rechtermuisknop de hoofdmethode van je algoritme te selecteren.
4. Verklaar wat er in de console gebeurt.

4.3.5 Zoek het dichtstbijzijnde ei

We willen dat Mimi met zo weinig mogelijk moeite een eitje vindt. Het makkelijkst voor haar is om naar het dichtstbijzijnde ei te lopen. Help jij haar dat te vinden? Schrijf een methode die de dichtstbijzijnde `Egg` oplevert.

1. Gegeven het (deels ingevulde) stroomdiagram.



Figuur 6: Stroomdiagram voor 'Dichtstbijzijnde ei in lijst vinden en uitbroeden'

Vul de deeltaken in volgende lijst aan:

Taak 1:

Taak 2:

Taak 3:

Taak 4:

2. Bedenk een geschikte methodenaam.
3. Voer in de body de vier deeltaken uit, net zoals bij de vorige opgave.

   Taak 1: Maak een lijst met de eieren in de wereld. Gebruik de instantievariabele `eggsInTheWorld` om deze op te slaan. Tip: Gebruik hiervoor een bestaande methode.

   Taak 2: (a) Bepaal de variabelen die je voor je oplossing nodig hebt.

   (b) Schrijf een submethode die de afstand bepaalt tussen Mimi en een ei. De afstand reken je niet uit met Pythogaras, maar bestaat uit het aantal stappen die Mimi moet zetten om bij het ei te komen. Tip: Om de absolute waarde van een getal te krijgen maak je gebruik van `Math.abs( getal )`.

   (c) Maak in je code gebruik van jouw methode om de lijst met eieren af te drukken en gebruik hierin `println` om afstanden en coördinaten af te drukken. Dit maakt het makkelijker om te testen of jouw methode juist werkt.

   (d) Compileer en test deze methode met de rechtermuisknop.

   Taak 3: Implementeer deze taak.

   Taak 4: Implementeer deze taak.

4. Compileer en test jouw methode.

4.3.6 Lijst doorlopen en eieren uitbroeden

Als uitbreiding op de opgave 4.3.4 laten we Mimi nu alle eieren uit de lijst met eieren één voor één aflopen en uitbroeden. De oplossing voor dit probleem lijkt natuurlijk erg veel op die voor het probleem uit 4.3.4. Alleen is Mimi nu niet klaar op het moment dat ze het eerste ei uit de lijst heeft gevonden maar moet ze daarna doorgaan met het volgende, enz.

We kunnen dit algoritme als volgt globaal omschrijven:

Taak 1: Vraag de lijst met eieren op.

Taak 2: Controleer of er nog eieren in de lijst zitten.

Indien 'Waar', dan klaar.

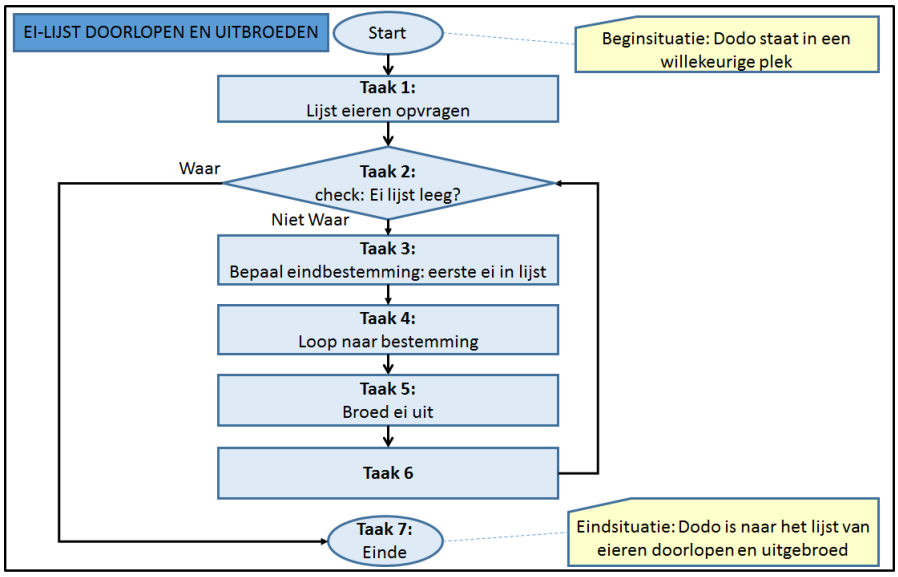
Indien 'Niet Waar', ga door naar de volgende taak.

Taak 2a: Kies het eerstvolgende ei als bestemming

Taak 2b: Loop naar bestemming.

Taak 2c: Broed het ei uit en ga verder met taak 2

Het bijbehorende stroomschema ziet er dan zo uit:



Figuur 7: Stroomdiagram 'Lijst doorlopen en eieren uitbroeden'

We gaan dit algoritme nu implementeren, echter nu zonder uitvoerige aanwijzingen.

1. Bedenk een geschikte naam voor je methode.
2. Werk de bovenbeschreven deeltaken uit in de body.
3. Compileer en test jouw methode.

4.3.7 Steeds het dichtstbijzijnde eitje zoeken

Schrijf nu een methode waarbij Mimi steeds naar het dichtstbijzijnde ei loopt en dit uitbroedt.

4.4 Dodo's Race

Wie gaat de wedstrijd winnen? Om te winnen zul je nog waarschijnlijk nog wat extra slimmigheden in moeten bouwen. Hoe kun je Mimi nog slimmer maken?
Uitgangspunten:

- Er staan geen hekken of andere objecten in de wereld, alleen één `MyDodo` object en een aantal `Egg`-objecten.
- In de wereld liggen 15 blauwe eieren en één gouden ei.
- Een gouden ei is vijf punten waard.
- Een blauw ei is één punt waard.
- Mimi mag hooguit 40 stappen zetten.

- Tijdens de echte wedstrijd wordt een (nu nog) onbekende wereld gebruikt.

Regels voor de Race: Om de competitie zo eerlijk mogelijk te maken gelden er voor deze opdracht een aantal regels.

- Zo mag Mimi alleen verplaatst worden door gebruik te maken van van de `MyDodo`-methode **`void`** `move()`.
- Wie het hoogste score bereikt wint de wedstrijd.
- Mimi mag hooguit 40 stappen zetten. Zorg ervoor dat je in `Madagaskar` de constante **`int`** `MAXSTEPS` op 40 hebt staan.

Voorbeeld werelden: Je kunt jouw algoritme uitproberen op de volgende werelden:

- world_dodoRace1
- world_dodoRace2
- world_dodoRace3
- world_dodoRace4

Reflectie: Als je klaar ben met het implementeren van jouw oplossing, kijk er dan even op terug:

1. Wat vind je van de kwaliteit van jouw oplossing? Kijk hiervoor terug in opdracht 4 hoofdstuk 'Kwaliteit.'
2. Wat vind je van de kwaliteit van jouw code?
3. In welke situaties zal jouw algoritme minder goed werken? Wat is voor jou de 'worst-case' scenario? Oftewel: welke scenario hoop je echt niet te krijgen tijdens de wedstrijd?
4. In welke situaties zal jouw algoritme goed werken? Wat is voor jou de 'best-case' scenario?
5. Waarom is jouw algoritme beter dan de algoritmes die je in deze opdracht uitgewerkt hebt?
6. Hoe is het gegaan? Wat vond je lastig om te doen?
7. Wat heb je geleerd? Wat zou je volgende keer anders doen?

5 Samenvatting

Je hebt geleerd:

- wat objectvariabelen en objecttypes zijn
- wat **`null`** betekent en wat dit met het aanroepen van methodes te maken heeft;
- wat een **`else if`** constructie is;
- gebruik te maken van klassenconstantes om waarden te onthouden;
- gebruik te maken van lijsten;
- *for-each-loop*s gebruiken.
- bestaande methodes in de Java Library Documentation op te zoeken en te gebruiken, hiermee kun je nu makkelijk nieuwe andere methodes opzoeken en toepassen;
- een complex algoritme op te splitsen in deeltaken en daarna te implementeren.

6 Jouw werk opslaan

Je bent klaar met de zevende opdracht. Sla je werk op.

1. Kies in Greenfoot 'Scenario' in het bovenste menu, en dan 'Save As ...'.
2. Vul de bestandsnaam aan met jouw eigen naam en het opgavenummer, bijvoorbeeld: `Opdr7_Michel`.

Alle onderdelen van het scenario bevinden zich nu in een map die dezelfde naam heeft als de naam die je hebt gekozen bij 'Save As ...'.