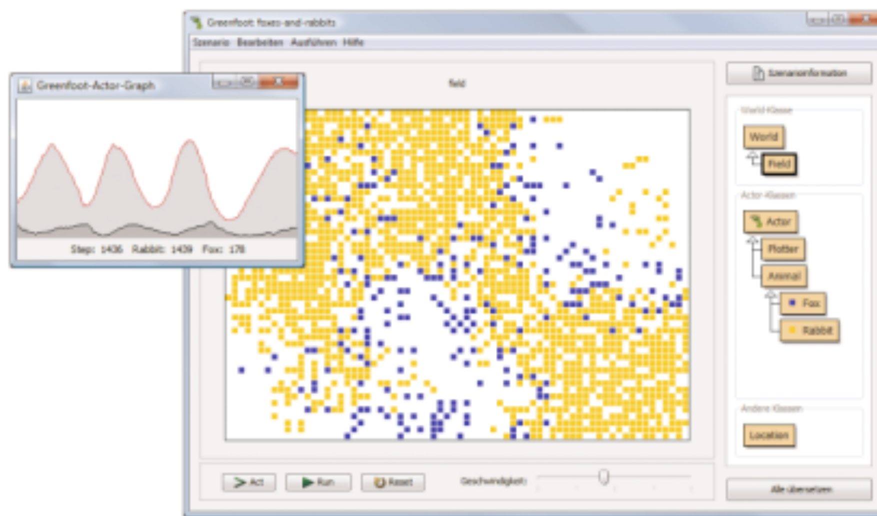
5. *Hase und Fuchs*

In dieser Simulation wirst du die Population von Hasen und Füchsen beobachten, die in einem Gehege bestimmter Größe leben. Dies ist ein einfaches Beispiel für so genannte Räuber-Beute-Simulationen, in der eine Kreatur eine andere jagt (und frisst). Der Fuchs ist der Räuber und der Hase ist die Beute. Eine hohe Population der Beutetiere liefert Nahrung im Überfluss für die Raubtiere. Zu viele Raubtiere andererseits fressen alle Beutetiere und sterben deshalb letztendlich an Nahrungsmangel. Meist besteht ein empfindliches Gleichgewicht zwischen diesen beiden Arten. Die Anzahl einer Population wird beispielsweise aber auch beeinflusst durch die Größe des Geheges, durch Krankheiten oder durch Umweltverschmutzung bzw. -gifte, die auf die Fortpflanzungs- bzw. Sterblichkeitsrate Auswirkungen haben.



In dieser einfachen Simulation werden die Hasen durch gelbe Quadrate und die Füchse durch blaue Quadrate dargestellt. Die Anzahl der Hasen und Füchse in dem Gehege wird in einem kleinen Fenster mit anhand der Bevölkerungskurve eingeblendet. Hasen bewegen sich und zeugen, wenn sie alt genug sind, Nachwuchs. Sie sterben entweder aufgrund ihres Alters oder weil sie von einem Fuchs gefressen werden. Füchse bewegen sich und vermehren sich, natürlich haben sie weniger Nachkommen als Hasen. Wenn sie hungrig sind, jagen sie Hasen. Sie sterben entweder aus Altersgründen oder sie verhungern.

5.1. *Untersuchung der Ausgangslage*

Die Grundidee ist recht einfach: Die Füchse und Hasen werden als Akteure in einer Sammlung gehalten. In jeden Schritt wird diese Sammlung durchlaufen und somit jedem Akteur die Gelegenheit gegeben, sich zu bewegen. Nach jedem Schritt, wenn alle Akteure sich also einmal bewegt haben, wird der aktuelle Zustand untersucht und auf dem Bildschirm angezeigt.

Übung 5.1:

Öffne das Szenario *haseFuchs1*. Untersuche die bereits gegebenen Klassen *Position* und *Feld*.

Die Klassen *World* und *Actor* werden bereits von den Greenfoot-Entwicklern zur Verfügung gestellt. Mit Hilfe eines Doppelklicks öffnet sich die Dokumentation.

5.1.1. Die Klasse Position

Die Klasse *Position* repräsentiert eine zweidimensionale Position innerhalb des Feldes. Eine Position wird durch eine x-Koordinate und eine y-Koordinate definiert.

5.1.2. Die Klasse Feld

Die Klasse *Feld* erzeugt ein zweidimensionales begrenztes Feld, das das Gehege darstellt. Das Feld besteht aus einer festgelegten Anzahl von Positionen, die in Spalten (x-Werte) und Zeilen (y-Werte) angelegt sind. Mit Hilfe der (zur Zeit noch leeren Methode) *bevoelkereFeld()* kann eine Position im Feld von genau einem Akteur eingenommen werden oder ist leer.

Zur Bewegung der Akteure auf Nachbarpositionen wird die Methode *gibFreieNachbarPosition()* verwendet. In der Liste *positionen* werden zunächst alle gültigen Nachbarpositionen gesammelt. Anschließen durchläuft der Iterator *nachbarn* diese Liste und liefert die erste gefundene Nachbarposition. Ansonsten wird die übergegebene Position (Elterntier könnte gestorben sein) oder (im schlimmsten Fall wegen Überbevölkerung) *null* geliefert.

5.2. Im Paradies

Nun wirst Du in dieses Gehege Tiere setzen: Hasen und Füchse.

Übung 5.2:

Finde Gemeinsamkeiten und Unterschiede zwischen der Klasse *Hase* und der Klasse *Fuchs*. Erstelle separate Listen der Eigenschaften (Datenfelder) und der Fähigkeiten (Methoden).

Zunächst simulierst du eine noch paradiesische Welt (die Vertreibung aus dem Paradies kommt später). Die Hasen und die Füchse leben friedvoll miteinander, sie bewegen sich und sterben schließlich aufgrund ihres Alters. Irgendwann werden dann zwar keine Tiere mehr existieren, aber um die Vermehrung wirst du dich später kümmern.

Du erkennst, dass die beiden Tierarten einige Eigenschaften und Fähigkeiten gemeinsam haben. Solche gemeinsamen Datenfelder und Methoden sind offensichtlich Kandidaten, die in eine Superklasse *Tier* verlagert werden sollten.

Bemerkung:

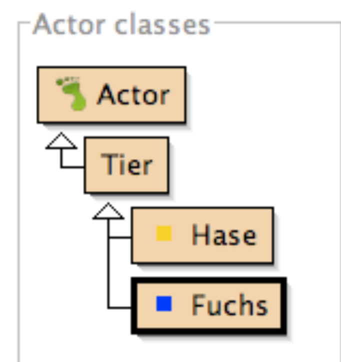
Die Klassen mit englischen Namen sind die von den Greenfoot-Entwicklern zu Verfügung gestellten Klassen. Diese stellen immer Superklassen dar und du wirst in diesen Klassen keinen Quelltext schreiben.

Die Klassen mit deutschem Namen sind die Klassen, mit denen du arbeiten wirst.

Das gleiche Prinzip wendest du auch mit den Namen bei Methoden und Datenfeldern an.

Die rechts stehende Abbildung zeigt die Superklasse *Tier* und die Subklasse *Hase*, die die Klasse *Tier* erweitert. Im Klassendiagramm sind diese Klassen mit Pfeilen verbunden. Dies weist auf eine *Ist-ein*-Beziehung hin. Eine Hase *ist ein* Tier und ein Tier *ist ein* Actor. Daraus folgt natürlich, dass ein Hase auch ein Actor ist.

Die gleichen Beziehungen gelten auch für die Klasse *Fuchs*.



5.2.1. Die Superklasse Tier

Da beide Tierarten sich bewegen und irgendwann aufgrund ihres Alters sterben, benötigt die Klasse *Tier* die beiden Datenfelder *alter* und *lebendig*. In dem Datenfeld *alter* wird das Alter des Tieres festgehalten und das Datenfeld *lebendig* zeigt, ob das Tier noch lebendig ist. Im Konstruktor werden diese beiden Datenfelder mit sinnvollen Werten belegt.

```
public class Tier extends Actor
{
    private int alter;
    private boolean lebendig;

    public Tier()
    {
        alter = 0;
        lebendig = true;
    }
}
```

Die beiden Datenfelder werden als *private* deklariert. Somit kann auch nicht von einer anderen Klasse auf diese Datenfelder zugreifen. Eine Möglichkeit wäre, die Datenfelder *protected* zu deklarieren, dies würde wenigstens den Subklassen vollen Zugang zu den Datenfeldern geben. Allerdings führt dies zu einer sehr engen Kopplung zwischen diesen beiden Klassen. Du kannst eine losere Kopplung erreichen, wenn du die Datenfelder *private* deklarierst und sondierende und verändernde Methoden definierst. Die Subklassen können dann diese Methoden verwenden, um die Werte dieser Datenfelder auszulesen oder zu verändern.

```
public int gibAlter()
{
    return alter;
}

public void setzeAlter(int alter)
{
    this.alter = alter;
}

public boolean istLebendig()
{
    return lebendig;
}
```

Übung 5.3:

Erzeuge eine Subklasse *Tier* der Klasse *Actor*. Implementiere die Datenfelder und den Konstruktor, sowie die drei sondierenden und verändernden Methoden. Dokumentiere deinen Quelltext.

Etwas größere Aufmerksamkeit benötigt die Methode *setzeGestorben()*, da hier nicht nur das Datenfeld *lebendig* auf *false* gesetzt wird, sondern das Tier auch aus dem Gehege entfernt werden muss.

Übung 5.4:

Suche in der Dokumentation der Superklasse *World* nach einer geeigneten Methode, mit der das Tier aus seiner Welt entfernt werden kann. Welchen Parameter erwartet diese Methode?
Um ein Objekt aus der Welt zu entfernen, benötigst du zuerst ein Objekt der Klasse *World*. Suche in der Dokumentation der Superklasse *Actor* nach einer geeigneten Methode. Welcher Rückgabetyp werden geliefert?

```
public void setzeGestorben()
{
    lebendig = false;
```

```
    World welt = getWorld();
    if (welt != null) {
        welt.removeObject(this);
    }
}
```

Übung 5.5:

Implementiere die Methode *setzeGestorben()* in der Klasse *Tier*. Weshalb könnte die *if*-Bedingung sinnvoll sein?

Bemerkung:

Es ist nicht so schlimm, wenn du jetzt noch nicht die Methode *getWorld()* verstanden hast. Im folgenden Kapitel wirst du die Klasse *Hase* erstellen und ein Objekt in das Gehege setzen. Mit Hilfe des Inspektors wirst du anschließend den Hasen genauer kennen lernen.

5.2.2. Die Subklasse Hase

Nun wirst du Hasen in das Gehege setzen. Hasen sollen bei ihrer Erzeugung ein zufälliges Alter erhalten, jedoch höchstens 50 werden. Wie alt nun Hasen tatsächlich werden, weiß ich nicht, wenn du allerdings als Einheit Monat wählst, scheint dies recht sinnvoll zu sein.

Das maximale Alter kannst du als Konstante festlegen und mit Hilfe des Konstruktors einen Hasen mit zufälligen Alter erzeugen.

```
public class Hase extends Tier
{
    private static final int MAX_ALTER = 50;

    public Hase()
    {
        this(true);
    }

    public Hase(boolean zufaelligesAlter)
    {
        super();
        if (zufaelligesAlter) {
            setzeAlter(Greenfoot.getRandomNumber(MAX_ALTER));
        }
    }
}
```

In der Klasse *Hase* werden zwei Konstuktoren zur Verfügung gestellt. Der zweite Konstuktor ist dir wahrscheinlich vertraut. Er erlaubt dir, einen Hasen mit allen Parameterwerten, die ihm übergeben werden, zu erzeugen. Hat der Parameter *zufaelligesAlter* den Wert *false*, so wird eine Hase mit dem Alter 0 geboren, wie es in der Superklasse festgelegt wird. Andernfalls hat der Hase ein zufälliges Alter, was zum Testen der Simulation bestimmt sinnvoll ist. Und genau für dieses Testen ist der erste Konstruktor einsatzbar. Mit dem Aufruf des Schlüsselwortes *this* wird der andere Konstruktor aufgerufen und der entsprechende Parameterwert übergeben.

Übung 5.6:

Erzeuge eine Subklasse *Hase* der Klasse *Tier*. Implementiere die Datenfelder und den Konstruktor. Dokumentiere deinen Quelltext.
Erzeuge einen Hasen und platziere diesen in der Welt. Erzeuge weitere Hasen. Untersuche die Hasen-Objekte mit Hilfe des Inspektors. Welche Unterschiede kannst du feststellen?
Führe das Programm aus, indem du auf den *Run*-Button klickst. Was kannst du beobachten?

Bemerkung:

Du kannst besonders schnell Objekte in deiner Welt ablegen, wenn du eine Klasse im Klassendiagramm auswählst und dann mit gedrückt gehaltener *Shift*-Taste in die Welt klickst.

Du wirst bemerkt haben, dass die Hasen nichts machen, wenn du den *Run*-Button drückst. Zwar haben alle Hasen die Methode *act()* von der Superklasse *Actor* geerbt, jedoch muss diese Methode in der Klasse *Hase* überschrieben werden.

Jeder Aufruf der Methode *act()* bewirkt, dass der Hase älter wird. Hierzu erstellst du zunächst die Methode *erhoeheAlter()* und verwendest die Methoden, die die Superklasse *Tier* bereits zur Verfügung stellt.

```
private void erhoeheAlter()
{
    setzeAlter(gibAlter() + 1);
    if (gibAlter() > MAX_ALTER) {
        setzeGestorben();
    }
}
```

Nun wird die Methode *act()* implementiert.

```
public void act()
{
    erhoeheAlter();
```

```
    if (istLebendig()) {
        //laeuft auf die naechste Nachbarposition, wenn diese frei ist
        Position neuePosition = gibFeld().gibFreieNachbarPosition(getX(),
                                                                  getY());
        //ist diese Position frei?
        if (neuePosition != null) {
            setLocation(neuePosition.gibX(), neuePosition.gibY());
        }
        else {
            setzeGestorben();
        }
    }
}
```

Zunächst wird das Alter der Hasen erhöht. Anschließend sucht er auf eine freie Nachbarposition. Gibt es eine solche, hoppelt er dorthin. Schlimmstenfalls stirbt er wegen Überbevölkerung.

Übung 5.7:

Implementiere in der Klasse *Hase* die beiden Methoden *erhoeheAlter()* und *act()*.

a) Beim Kompilieren meldet der Compiler, dass er die Methode *gibFeld()* vermisst. Implementiere hierzu in der Klasse *Tier* die Methode

```
public Feld gibFeld()
{
    return (Feld) getWorld();
}
```

(Diese Methode wird unten genauer erklärt.)

b) Kompiliere dein Szenario.

c) Erzeuge einen Hasen und setze diesen in das Gehege.

d) Untersuche den Hasen mit Hilfe des Inspektors.

Die Anweisung, in der die oben erwähnte Fehlermeldung erscheint

```
Position neuePosition = gibFeld().gibFreieNachbarPosition(getX(), getY());
```

solltest du noch etwas genauer betrachten.

Der Inspektor eines Hasen listet insgesamt 7 Datenfelder auf.

Die beiden Datenfelder *alter* und *lebendig* erbt der Hase von der Klasse *Tier*, in der du diese selbst initialisiert hast. Die restlichen Datenfelder *x*, *y*, *rotation*, *world* und *image* hat der Hase von der Superklasse *Actor* geerbt. (In dieser Simulation sind jedoch die Datenfelder *rotation* und *image* nicht von Interesse.)

Übung 5.8:

Öffne den Inspektor eines Hasen.

a) Verschiebe den Hasen im Gehege und beobachte die Werte der Datenfelder *x* und *y*. Welche Werte haben diese Datenfelder, wenn der Hase in den Eckpunkten des Geheges sich befindet?

b) Klicke auf den Referenzpfeil des Datenfeldes *world*. Interpretiere die nun gezeigten Werte der entsprechenden Datenfelder.

Jeder Akteur kennt also seine Welt (in dieser Simulation sein Gehege) und auch seine genaue Position hierin. Allerdings sind diese von der Superklasse *Actor* geerbten Datenfelder von den Greenfoot-Entwickler als *private* eingestuft worden, so dass du von den Subklassen keinen direkten Zugriff auf dieser Datenfelder hast. Jedoch stellen die Greenfoot-Entwickler dir zugehörige *get*-Methoden (*getImage(), getRotation(), getWorld(), getX()* und *getY()*) und *set*-Methoden (*setImage(), setLocation()* und *setRotation()*) zur Verfügung.

Übung 5.9:

In der Klasse *Tier* hast du bereits die Methode *setzeGestorben()* implementiert. Versuche ohne der Methode *getWorld()* das Tier direkt aus dem Gehege zu entfernen, wenn es gestorben ist. Welche Fehlermeldung zeigt der Compiler?

In der Methode *act()* sucht der Hase sich nun eine freie Nachbarposition. Mit *getX()* und *getY()* bekommst du also die aktuelle (x/y)-Position des Hasen und verwendest diese als Parameter im Aufruf der Methode *gibFreieNachbarPosition(getX(), getY())*. Diese Methode musste jedoch in der Subklasse *Feld* implementiert werden, da du auf die zugehörige Superklasse *World* keinen Einfluss hast. Und leider besitzt der Hase kein Datenfeld vom Typ *Feld*. Deswegen wird in der Klasse *Tier* eine zu *getWorld()* analoge Methode *gibFeld()* implementiert, in der ein *World*-Objekt mit Hilfe des *Cast*-Operators in den Subtyp *Feld* umgewandelt wird.

```
public Feld gibFeld()
{
    return (Feld) getWorld();
}
```

Nun hat der Hase kurzfristig (lokale Variable) sein Gehege als Feld und kann sich hierin eine freie Nachbarposition suchen.

```
Feld tempFeld = gibFeld();
Position neuePosition = tempFeld.gibFreieNachbarPosition(getX(), getY());
```

In kurzer Form lässt sich dies auch formulieren:

```
Position neuePosition = gibFeld().gibFreieNachbarPosition(getX(), getY());
```

Übung 5.10:

Setze nun einige Hasen in das Gehege und beobachte ihr Verhalten.

5.2.3. Die Subklasse Fuchs

Da die Füchse und Hasen im Paradies leben, ist die Klasse *Fuchs* zunächst einmal genauso aufgebaut wie die Klasse *Hase*.

Übung 5.11:

Erzeuge eine Subklasse *Fuchs* der Klasse *Tier*.

a) Kopiere den Quelltext der Klasse *Hase* in die Klasse *Fuchs* und ändere ihn an den entsprechenden Stellen. Das Höchstalter eines Fuchses soll nun 150 betragen.

b) Setze mehre Hasen und Füchse in das Gehege und beobachte ihr Verhalten.

Die Hasen und Füchse leben nun friedvoll nebeneinander. Sie bewegen sich frei in dem Gehege umher bis sie irgendwann an Altersschwäche sterben. Dann ist das Gehege leer und du kannst die Simulation beenden.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *haseFuchs2*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

5.3. *Die Vertreibung aus dem Paradies*

Nun kommt das Böse in das Paradies. Die Füchse entwickeln sich zu Räubern, sie haben die Hasen als Beutetiere erkannt und machen Jagd auf sie. Mit der Zeit sterben sie nun nicht mehr nur an Altersschwäche. Sie haben ihr Futterverhalten total umgestellt. Sie müssen nun im Gehege nach Hasen suchen und, wenn sie nicht mehr genug Hasen zum Fressen gefunden haben, sterben sie nun auch an Hunger.

Dieses neuartige Verhalten wirst du nun in der Klasse *Fuchs* implementieren. Jeder Hase, der gefressen wird, besitzt einen bestimmten Nährwert, um den der Futterlevel des Fuchses erhöht wird. Setzt du einen Fuchs in das Gehege, so hat dieser sowohl ein zufälliges Alter als auch ein zufälliges Sättigungsgefühl. Neugeborene Füchse sind grundsätzlich nicht hungrig.

```
private static final int MAX_ALTER = 150;
private static final int HASEN_NAEHRWERT = 4;

private int futterLevel;

//Standardkonstruktor ausgelassen

public Fuchs(boolean zufaelligesAlter)
{
    super();
    if (zufaelligesAlter) {
        setzeAlter(Greenfoot.getRandomNumber(MAX_ALTER));
        futterLevel = Greenfoot.getRandomNumber(HASEN_NAEHRWERT);
    }
    else {
        //Alter auf 0 belassen, Neugeborene sind nicht hungrig
        futterLevel = HASEN_NAEHRWERT;
    }
}
```

Übung 5.12:

Implementiere die Konstante *HASEN_NAEHRWERT* und das Datenfeld *futterLevel*. Erweitere den Konstuktor. Setze anschließend jeweils einen Fuchs mit dem Standardkonstruktor und einen neugeborenen Fuchs in das Gehege. Überprüfe diese beiden mit Hilfe der Inspektoren.

Mit jedem Schritt, also mit jedem Aufruf der Methode *act(),* wird der Fuchs hungriger bis er schließlich vor Hunger stirbt, falls er keinen Hasen fressen konnte. Dazu implementierst du die Methode *vergroessereHunger()*, in der *futterLevel* um eins erniedrigt wird. Hat *futterLevel* schließlich den Wert 0 erreicht, so wird die Methode *setzeGestorben()* aufgerufen, die den Fuchs aus dem Gehege entfernt.

```
private void vergroessereHunger()
{
    futterLevel--;
    if (futterLevel <= 0) {
        setzeGestorben();
    }
}
```

Anschließend musst du diese Methode *vergroessereHunger()* noch in der Methode *act()* aufrufen.

```
public void act()
{
    erhoeheAlter();
    vergroessereHunger();

    //weitere Anweisungen ausgelassen

}
```

Übung 5.13:

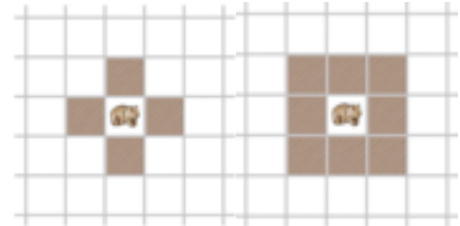
Implementiere die Methode *vergroessereHunger()*. Überprüfe mit Hilfe des Inspektors, ob sich der Futterlevel eines Fuchses mit jedem Schritt erniedrigt, bis er schließlich verhungert ist.

Im Zusammenhang mit der Ernährungsumstellung des Fuchses hat sich auch sein Lebensinhalt verändert. Er läuft nicht mehr wie bisher zufällig umher, sondern er ist nun ständig auf der Suche nach Nahrung, indem er alle 8 benachbarten Positionen nach Hasen absucht. Findet er einen Hasen, so läuft er auf die entsprechende Nachbarposition und frisst den dort stehenden Hasen auf. Findet er hingegen keine Nahrung, trottelt er auf eine zufällige Nachbarposition und sucht weiter, wobei sich mit jedem Schritt sein Hunger vergrößert.

Zuerst jedoch wirst du die Methode *findeNahrung()* kennen lernen. Hierzu muss der Fuchs seine benachbarten Positionen nach einem Hasen absuchen.

Übung 5.14:

Suche in der Dokumentation der Klasse *Actor* nach einer geeigneten Methode, um Objekte zu ermitteln, die sich in den Zellen um den aktuellen Akteur befinden. Welche Parameter erwartet diese Methode? Welchen Rückgabetyp liefert diese Methode?



Du wirst wohl die Methode *List getNeighbours(int distance, boolean diagonal, Class class)* gefunden haben. Die Parameter geben den zu berücksichtigenden Abstand von dem aufrufenden Akteur an und ob die diagonal liegenden Zellen ebenfalls zu prüfen sind. Der dritte Parameter bietet die Möglichkeit, nur die Objekte einer angegebenen Klasse zu berücksichtigen. Der Rückgabetyp ist eine Liste, die alle gesuchten Objekte auf den benachbarten Positionen enthält.

Mit Hilfe dieser Methode besitzt der Fuchs nun eine Möglichkeit, auf den benachbarten Feldern nach Hasen zu suchen. Existieren Hasen auf diesen Nachbarpositionen, so merkt er sich die Position des ersten Hasen dieser Liste, geht auf dieses Feld, frisst den Hasen und setzt dadurch sein Futterlevel wieder auf maximale Größe *HASEN_NAEHRWERT*. (Übergewicht kennen die Füchse nicht, ein bisschen Paradies ist offensichtlich doch noch übrig geblieben.)

```
import java.util.List;

private Position findeNahrung()
{
    List hasen = getNeighbours(1, true, Hase.class);
    if (hasen.isEmpty()) {
        return null;
    }
    else {
        Hase hase = (Hase) hasen.get(0);
        Position position = new Position(hase.getX(), hase.getY());
        hase.setzeGestorben();
        futterLevel = HASEN_NAEHRWERT;
        return position;
    }
}
```

Übung 5.15:

Implementiere die Methode *findeNahrung()* in der Klasse *Fuchs*. Welchen Rückgabetyp liefert diese Methode?

Die Methode f*indeNahrung()* musst du nun in der Methode *act()* aufrufen.

```
public void act()
{
    erhoeheAlter();
    vergroessereHunger();
    if (istLebendig()) {
        //Sucht eine Nachbarposition, auf der Nahrung (Hase) liegt
        Position neuePosition = findeNahrung();
        //kein Futter gefunden, nun also zufaellig weitergehen
        if (neuePosition == null) {
            neuePosition = gibFeld().gibFreieNachbarPosition(getX(), getY());
        }
        //ist diese Position frei, dann gehe darauf
        if (neuePosition != null) {
            setLocation(neuePosition.gibX(), neuePosition.gibY());
        }
        else {
            //kann sich weder bewegen noch bleiben
            //(Ueberbevoelkerung - alle Positionen belegt
            setzeGestorben();
        }
    }
}
```

Nachdem das Alter des Fuchses erhöht, sein Hunger vergrößert wurde und er immer noch lebendig ist, sucht er als neue Position nach einer Nachbarposition mit Nahrung. Fand der Fuchs keine Nahrung, so sucht er sich als neue Position eine freie Nachbarposition. Gibt es eine solche freie Nachbarposition, geht er auf diese. Falls keine freie Nachbarposition mehr existiert, so stirbt der Fuchs wegen Überbevölkerung.

Übung 5.16:

Implementiere die Methode *act()* in der Klasse *Fuchs*. Setze einen Fuchs und einen Hasen auf benachbarte Positionen in das Gehege. Kontrolliere ihre Positionen mit Hilfe der Inspektoren. Klicke auf den *Act*-Button und beobachte, ob der Fuchs den Hasen findet und diesen auch frisst. Probiere dies mit verschiedenen Nachbarpositionen aus.

Übung 5.17:

Setze nun mehrere Hasen und Füchse auf beliebige Positionen in das Gehege. Klicke auf den *Run*-Button und beobachte das Geschehen. Verringere die Geschwindigkeit, damit du feststellen kannst, ob die Füchse tatsächlich die benachbarten Hasen fressen.

Du wirst bemerkt haben, dass die Füchse sich immer in Richtung Hasen bewegen, also aktiv nach Hasen suchen. Allerdings werden wohl in deiner Simulation sehr wahrscheinlich die Füchse bald ausgestorben sein. Die Hasen kommen nur noch vereinzelt vor, deswegen finden die Füchse nicht genug Nahrung. Schließlich sterben auch die Hasen bald an Altersschwäche. Das Gehege liegt nun verlassen dar.

Bemerkung:
Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *haseFuchs3*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

5.4. … und vermehret euch

Nun wirst du dafür sorgen, dass die Hasen Nachwuchs erhalten. Sie kommen mit 5 in das gebärfähige Alter. Allerdings sollen nur 12 % der gebärfähigen Hasen auch tatsächlich Nachwuchs bekommen, wobei die maximale Wurfgröße auf 5 begrenzt ist.

Bemerkung:
Dieses Szenario entspricht jetzt nicht ganz der Realität. Die Tiere müssen keinen Geschlechtspartner finden, um sich fortzupflanzen (das können sie allein), Nahrungsquellen für Hasen sind auch nicht vorgesehen, andere Todesursachen (wie Krankheiten) werden einfach ignoriert. Wenn du dieses Kapitel durchgearbeitet hast, kannst du die Mängel der Simulation beheben.

Diese oben erwähnten Daten hältst du in den folgenden drei Datenfeldern fest:

```
private static final int GEBAER_ALTER = 5;
private static final double GEBAER_WAHRSCHEINLICHKEIT = 12;
private static final int MAX_WURFGROESSE = 5;
```

Nun ermittelst du mit Hilfe der Methode *gebaereNachwuchs()* eine Zahl für die Wurfgrösse ermitteln. Hierzu wird zuerst überprüft, ob der Hase bereits im gebärfähigen Alter ist. Anschließend wird die Gebärwahrscheinlichkeit auf 12 % begrenzt. Nun kann die Anzahl des Nachwuchses bestimmt werden und von der Methode zurück geliefert werden.

```
private int gebaereNachwuchs()
{
    int geburten = 0;
    if (gibAlter() >= GEBAER_ALTER &&
                Greenfoot.getRandomNumber(100) <= GEBAER_WAHRSCHEINLICHKEIT) {
        geburten = Greenfoot.getRandomNumber(MAX_WURFGROESSE) +1;
```

```
    }
    return geburten;
}
```

Übung 5.18:

Implementiere die Methode *gebaereNachwuchs()* in der Klasse *Hase*. Welchen Rückgabetyp liefert diese Methode?

Die Methode *gebaereNachwuchs()* musst du nun in der Methode *act()* aufrufen.

```
public void act()
{
    erhoeheAlter();
    if (istLebendig()) {
        //gebaert Nachwuchs und setzt diesen auf freie Nachbarpositionen
        int geburten = gebaereNachwuchs();
        for (int g = 0; g < geburten; g++) {
            Position position = gibFeld().gibFreieNachbarPosition(getX(),
                                                                  getY());
            if (position != null) {
                Hase neuerHase = new Hase(false);
                gibFeld().addObject(neuerHase, position.gibX(),position.gibY());
            }
        }
        //laeuft auf die naechste Nachbarposition, wenn diese frei ist
        Position neuePosition = gibFeld().gibFreieNachbarPosition(getX(),
                                                                  getY());
        //ist diese Position frei?
        if (neuePosition != null) {
            setLocation(neuePosition.gibX(), neuePosition.gibY());
        }
        else {
            setzeGestorben();
        }
    }
}
```

Falls der Hase lebendig ist, wird die Anzahl des Nachwuchses mit Hilfe der Methode *gebaereNachwuchs()* festgelegt. Da es sinnvoll ist, dass der gesamte Nachwuchs sich in der Nähe des Elterntieres aufhalten soll, wird nun für jedes Jungtier eine freie Nachbarpositionen gesucht. Existiert eine solche freie Nachbarposition, wird ein neuer Hase erzeugt und auf diese Position gesetzt. Anschließend sucht sich das Elterntier noch eine freie Nachbarposition und läuft dorthin, falls keine mehr existiert, stirbt das Elterntier wegen Überbevölkerung.

Übung 5.19:

Ergänze in der Klasse *Hase* die beiden Methode *act()*.

a) Kompiliere dein Szenario.
b) Erzeuge einen Hasen und setze diesen in das Gehege.
c) Untersuche den Hasen mit Hilfe des Inspektors.

Übung 5.20:

Setze einige Hasen und einen Fuchs in das Gehege. Öffne den Inspektors dieses Fuchses. Starte die Simulation und beobachte den Fuchs. Findet der Fuchs immer genug Nahrung? Stirbt er an Hunger oder an Altersschwäche? Probiere verschiedene Ausgangssituationen aus.

Der Fuchs sollte natürlich auch Nachwuchs gebären können. Da er jedoch ein größeres Tier ist und keine Feinde hat. kannst du das gebärfähige Alter auf 10 heraufsetzen. Die Gebärwahrscheinlichkeit setzt du auf 8 % und die maximale Wurfgröße auf 3 herab.

```
private static final int GEBAER_ALTER = 10;
private static final int GEBAER_WAHRSCHEINLICHKEIT = 8;
private static final int MAX_WURFGROESSE = 3;
```

Übung 5.21:

Erstelle in der Klasse *Fuchs* die drei Konstanten. Implementiere anschließend die Methode *gebaereNachwuchs()* und ergänze die Methode *act()*. Setze einige Hasen und einige Füchse in das Gehege. Starte die Simulation und beobachte die Hasen- und Fuchspopulation. Probiere verschiedene Ausgangssituationen aus.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *haseFuchs4*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

5.5. *Zurück zu den Anfängen*

Vielleicht erscheint es dir recht mühevoll, bei jedem Start der Simulation so viele Füchse und Hasen manuell in das Gehege zu setzen. Diese Aufgabe sollte beim Programmstart die Simulation selbstständig übernehmen, indem es das Gehege mit einer zufälligen Anzahl von Füchsen und Hasen bevölkert. Der beste Ort für eine solche Methode b*evoelkereFeld()* ist die Klasse *Feld*.

```
private static final int FUCHSGEBURT_WAHRSCHEINLICHKEIT = 2;
private static final int HASENGEBURT_WAHRSCHEINLICHKEIT = 8;
```

In diesen beiden Konstanten wird die Wahrscheinlichkeit für die Geburt eines Fuchses bzw. eines Hasen festgelegt.

```
private void bevoelkereFeld()
{
    for (int zeile = 0; zeile < HOEHE; zeile++) {
        for (int spalte = 0; spalte < BREITE; spalte++) {
            if (Greenfoot.getRandomNumber(100) <=
                                   FUCHSGEBURT_WAHRSCHEINLICHKEIT) {
                Fuchs fuchs = new Fuchs(true);
                addObject(fuchs, spalte, zeile);
            }
            else if (Greenfoot.getRandomNumber(100) <=
                                   HASENGEBURT_WAHRSCHEINLICHKEIT) {
                Hase hase = new Hase(true);
                addObject(hase, spalte, zeile);
            }
            // andernfalls die Position leer lassen
        }
    }
}
```

Das gesamte Gehege wird nun zeilen- und spaltenweise abgetastet. Jede Zelle wird nun entsprechend der Fuchswahrscheinlichkeit mit einem Fuchs mit zufälligen Alter und Sättigungsgrad belegt. Wenn kein Fuchs platziert werden kann, wird ein Hase mit zufälligem Alter entsprechend der Hasenwahrscheinlichkeit in das Gehege gesetzt. Und es besteht auch noch die Möglichkeit, dass die Zelle leer bleibt.

Übung 5.22:

Erstelle in der Klasse *Feld* die beiden Konstanten. Implementiere anschließend die Methode *bevoelkereFeld()* und ergänze den Konstruktor in geeigneter Weise. Starte die Simulation und beobachte die Hasen- und Fuchspopulation.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *haseFuchs5*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

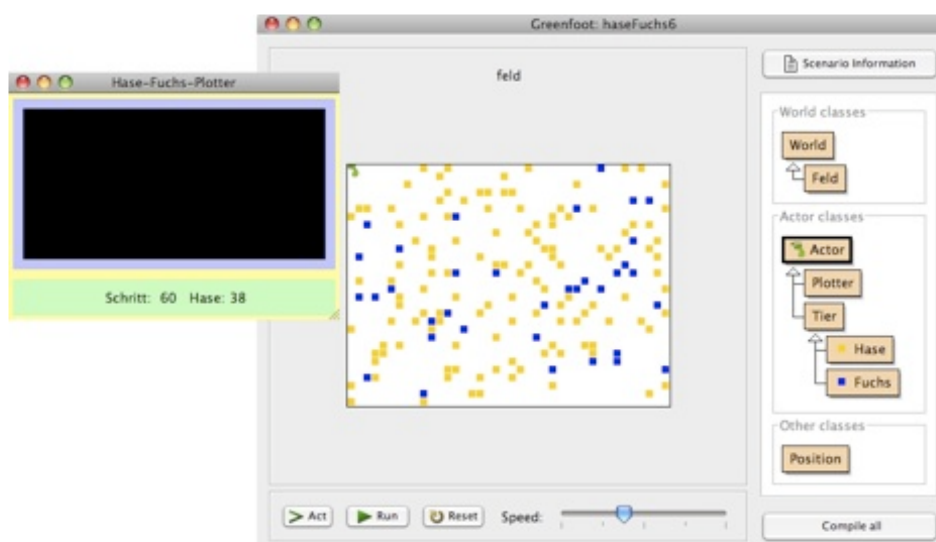
5.6. *Volkszählung*

Zum Abschluss wirst du noch etwas Volkszählung betreiben. Die Bestandsschwankungen von Hase und Fuchs lassen sich in einem Diagramm darstellen. Die hellgraue Bevölkerungskurve stellt den Bestand an Hasen dar, die dunkelgraue Kurve den Bestand an Füchsen. Sehr deutlich kannst du die zeitliche Versetzung der Maxima und Minima von Hasen und Füchsen erkennen. In dem unteren Teil wird der momentane Bestand dargestellt.

Bemerkung:
Das Erstellen einer grafischen Oberfläche soll in diesem Kapitel nicht im Vordergrund stehen, es würde zu sehr vom eigentlichen Ziel der objektorientierten Programmierung weg führen. Deswegen habe ich bereits alle wichtigen für die Darstellung von Fenstern notwendigen Anweisungen implementiert. Im Text gehe ich hauptsächlich auf die Darstellung und Aktualisierung der Kurven und Anzeigen ein.

5.6.1. Darstellung in Zahlen

Übung 5.23:
Öffne das Szenario *haseFuchs6a*. Was gibt es Neues in diesem Szenario? Öffne den Editor der Klasse *Plotter*. Versuche in groben Zügen nachzuvollziehen, was bereits implementiert worden ist.



Es öffnet sich nun ein zusätzliches Fenster *Hase-Fuchs-Plotter*. Dieses Fenster hat noch eine schwarze Fläche, in die später die beiden Bevölkerungskurven eingetragen werden. In unteren Informationsteil werden während der Simulation die aktuelle Anzahl der Schritte und der Hasen angezeigt. Im Klassendiagramm erscheint eine neue Klasse *Plotter*. Die Klasse *Feld* erzeugt ein Objekt dieser Klasse *Plotter* und setzt dieses in die linke obere Ecke des Geheges. Wenn du genau hinschaust, erkennst du an der Position (0/0) einen grünen Fuß. Im Konstruktor der Klasse *Plotter* wird die Methode *erzeugeFenster()* aufgerufen, die letztendlich alle notwendigen Komponenten zusammensetzt.

Wichtig ist nun die Methode *act()*. Hier wird mit Hilfe von *aktualisiereDaten()* dafür gesorgt, dass die Daten bei jedem Schritt aktualisiert werden.

```
private void aktualisiereDaten()
{
    List<Hase> hasenListe = welt.getObjects(classH);

    buehne.aktualisiere(schritt, hasenListe.size());

    if (hasenListe.size() == 0) {
        Greenfoot.stop();
    }
}
```

Zuerst wird eine Liste aller Hasen erstellt, die in der Welt (im Gehege) leben und der Bühne übergeben, damit diese daraus die Kurven und anderen Information aufbereitet. Falls keine Hasen mehr überlebt haben, soll die Simulation beendet werden.

```
private void aktualisiere(int schritt, int anzahlHasen)
{
    schrittL.setText("" + schritt);
    zaehlerHaseL.setText("" + anzahlHasen);
}
```

Die Bühne aktualisiert zur Zeit nur die Anzeige für die Anzahl der Schritte und der Hasen. Wird diese Methode zusätzlich auch im Konstruktor der Klasse *Plotter* aufgerufen, so wird zu Beginn die korrekte Anzahl der Hasen angezeigt.

Übung 5.24:

In dem Fenster *Hase-Fuchs-Plotte*r soll in einer weiteren Anzeige die Anzahl der Füchse dargestellt werden. Implementiere in der Klasse *Plotter* alle notwendigen Anweisungen. Versuche hierbei die bereits vorgegebenen Programmieranweisungen, die die Hasen betreffen, zu verstehen und nachzuvollziehen.

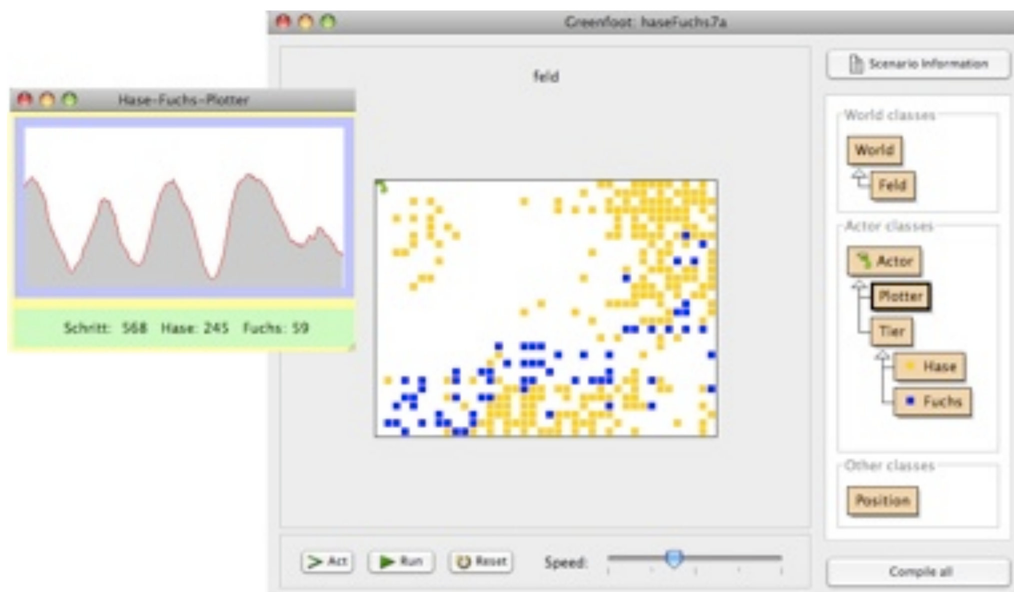
Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *haseFuchs6b*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

5.6.2. Darstellung als Diagramm

Übung 5.25:

Öffne das Szenario *haseFuchs7a*. Was gibt es Neues in diesem Szenario? Öffne den Editor der Klasse *Plotter*. Versuche in groben Zügen nachzuvollziehen, was bereits implementiert worden ist.



In Fenster *Hase-Fuchs-Plotte*r erscheint nun zusätzlich eine Kurve, die den Verlauf der Hasenpopulation abgebildet. Während der Simulation wandert diese Kurve ständig nach links, so dass am rechten Rand immer die aktuelle Anzahl der Hasen neu dargestellt wird.

Um diese Kurve zu normieren, erhält der Plotter im Konstruktor einen neuen Parameter *anzahlMaxTiere*, der die größte Anzahl der im Gehege lebenden Tiere angibt. Im Grunde wird dieser Parameter bis zur eingebetteten Klasse *Buehne* durchgereicht, die diesen Parameter schließlich zum Zeichnen des Graphen verwendet.

Übung 5.26:

Verfolge in der Klasse *Plotter*, wie der Parameter *anzahlMaxTiere* vom Konstruktor dieser Klasse bis zur eingebetteten Klasse *Buehne* weiter gereicht wird. Wozu wird dieser Parameter in der Klasse *Buehne* verwendet? Woher erhält dieser Parameter seinen Wert?

```
public Buehne(int breite, int hoehe, int anzahlMaxTiere)
{
    buehneImage = new BufferedImage(breite, hoehe, BufferedImage.TYPE_INT_RGB);

    Graphics g = buehneImage.getGraphics();
    g.setColor(Color.WHITE);
    g.fillRect(0, 0, buehneImage.getWidth(), buehneImage.getHeight());
    repaint();

    this.anzahlMaxTiere = anzahlMaxTiere;

    altYHase = hoehe;
}
```

Im Konstruktor der Klasse *Buehne* wird nun auf die bisher schwarze Fläche ein weißes Rechteck gezeichnet. Anschließend wird das Datenfeld *anzahlMaxTiere* belegt und in *altYHase* der y-Startwert des Diagramms, also die Höhe des Diagrammfensters, festgelegt.

```
private void aktualisiere(int schritt, int anzahlHasen, int anzahlFuechse)
{
    Graphics g = buehneImage.getGraphics();
    int hoehe = buehneImage.getHeight();
    int breite = buehneImage.getWidth();
    g.copyArea(1, 0, breite - 1, hoehe, -1, 0);

    int y = hoehe - (anzahlHasen * hoehe) / anzahlMaxTiere;
    g.setColor(Color.LIGHT_GRAY);
    g.drawLine(breite - 2, y, breite - 2, hoehe);
    g.setColor(Color.RED);
    g.drawLine(breite - 3, altYHase, breite - 2, y);
    altYHase = y;

    paintImmediately(0, 0, breite, hoehe);

    //weitere Anweisungen

}
```

Die größten Veränderung werden in der Methode *aktualisiere()* vorgenommen

Im ersten Block wird der Grafikkontext der Bühne geholt und anschließend dieser um 1 Pixel verkleinerte Bereich um 1 Pixel nach links versetzt.

Im zweiten Block wird die Hasenanzahl in einen geeigneten y-Wert umgerechnet und anschließend als hellgraue senkrechte Linie gezeichnet. Der aktuelle Hasen-y-Wert wird als rote, 1 Pixel breite waagrechte Linie dargestellt.

Zum Abschluss müssen diese Veränderungen wieder in die Bühne gezeichnet werden.

Übung 5.27:

In dem Fenster *Hase-Fuchs-Plotte*r soll in einer weiteren dunkelgrauen Kurve der Populationsverlauf der Füchse dargestellt werden. Implementiere in der Klasse *Plotter* alle notwendigen Anweisungen. Versuche hierbei die bereits vorgegebenen Programmieranweisungen, die die Hasen betreffen, zu verstehen und nachzuvollziehen.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *haseFuchs7b*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

5.7. Und nun …

Diese Simulation ist recht einfach. Wie Parameter, die in der Natur einen Einfluss haben, sind hier nicht berücksichtigt worden:

- Die Tiere müssen keinen Geschlechtspartner finden
- Hasen benötigen keine Nahrung und können deswegen nicht verhungern.
- Es gibt keine Krankheiten, die zum Tod führen.
- …

Trotzdem lassen sich mit dieser Simulation bereits einige Untersuchungen durchführen:

- Welche Rolle spielt die Größe des Geheges?
- Welchen Einfluss haben die Parameter, die das Höchstalter, das Fortpflanzungsalter, die Fortpflanzungshäufigkeit und die maximale Wurfgröße bestimmen?
- Welchen Einfluss hat der Nährwert eines Hasen?
- …

Im Beispiel, das die Greenfoot-Entwickler vorstellen, wird im Diagramm zusätzlich eine Skalierungsmethode vorgestellt, die dafür sorgt, dass der Populationsverlauf deutlicher dargestellt wird

Du siehst, hier gibt es noch einige Verbesserungen.