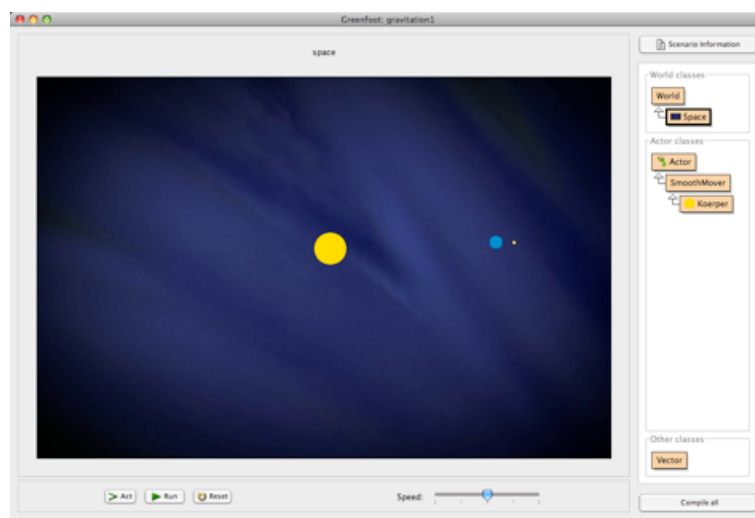


## 2. Newtons Labor

In dieser Simulation wirst du die Bewegung von Himmelskörpern (wie Sterne und Planeten) untersuchen und hierbei das Newtonsche Gravitationsgesetz anwenden. Falls du dir Gedanken machst, ob deine Physikkenntnisse ausreichen, möchte ich dich etwas beruhigen. Die Formel hierzu ist recht einfach, du wirst nicht allzu tief in die Materie einsteigen.

### 2.1. Untersuchung der Ausgangslage

Aber zunächst wirst du die Ausgangssituation untersuchen. Öffne mit **Greenfoot** das Szenario *gravitation1*.



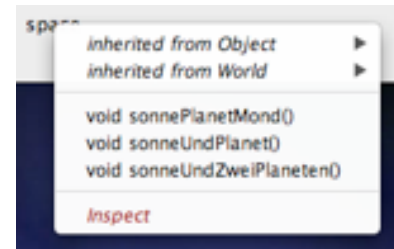
#### 2.1.1. Die Oberfläche

Du erkennst, dass die Klasse *World* bereits eine Subklasse *Space* besitzt. Zusätzlich existieren zwei weitere Subklassen: Die Klasse *Koerper* erbt von der Klasse *SmoothMover*, die wiederum die Klasse *Actor* erweitert. In einem weiteren Bereich existiert die Klasse *Vector*, die für dieses Szenario eine Hilfsklasse darstellt, von der keine Objekte erzeugt werden sollen.

#### Übung 2.1:

- Platziere einige Körper in dem Weltall. Was kannst du beobachten? Klicke einen Körper mit der rechten Maustaste an, um dich über seine öffentlichen Methoden zu informieren.

- b) Rufe die öffentliche Methode `sonnePlanetMond()` von `Space` auf. Finde heraus, welche Masse die Sonne, der Planet und der Mond hat. Klicke hierzu den Titel der Welt (das Wort `space` ganz oben) mit der rechten Maustaste an.



### 2.1.2. Die Hilfsklassen `SmoothMover` und `Vector`

In dem Szenario `gravitation1` werden noch zwei weitere Hilfsklassen verwendet: `SmoothMover` und `Vector`.

Die Klasse `SmoothMover` hat allgemein zwei Aufgaben:

- Die Darstellung der Bewegung der Körper wird weicher, da die Klasse zur Berechnung Dezimalzahlen verwendet und diese erst zum Zeichnen wieder in ganze Zahlen umwandelt.
- Sie verwaltet für jeden Körper einen Vektor, der die aktuelle Richtung und die Bewegungsgeschwindigkeit angibt.

Die zweite Klasse `Vector` implementiert einen Vektor, der von der Klasse `SmoothMover` verwendet wird. Beachte, dass `Vector` kein Akteur ist. Sie wird nur von Akteur-Objekten erzeugt und verwendet und wird nie für sich allein in der Welt erscheinen.

#### Übung 2.2:

- a) Mache dich mit den Methoden der Klassen `SmoothMover` und `Vector` vertraut, indem du den Editor öffnest und ihre Definition und Methoden in der Ansicht Dokumentation liest.
- b) Platziere ein Objekt der Klasse `Koerper` in der Welt. Rufe das Kontextmenü auf und untersuche die Methoden, welche dieses Objekt von der Klasse `SmoothMover` geerbt hat.

#### Bemerkungen:

1. Von der Klasse `SmoothMover` kannst du keine Objekte erzeugen, da kein Konstruktor angezeigt wird. Im Quelltext findest du das Schlüsselwort ***abstract***, was die Erzeugung von Objekten verhindert. Abstrakte Klassen dienen hauptsächlich als Superklassen für andere Klassen, nicht um direkt Objekte davon zu erzeugen.
2. In der Klasse `SmoothMover` existieren zwei Methoden `setLocation()`. Java erlaubt die Definition zweier gleichnamiger Methoden, solange diese sich in der Parameterliste unterscheiden. Dieses Verfahren nennt man ***Überladung***.

### 2.1.3. Die Klasse *Koerper*

#### Übung 2.3:

Öffne den Quelltext der Klasse *Koerper* und untersuche diesen.

Die Klasse *Koerper* besitzt zwei Konstruktoren. Da Konstruktoren eine besondere Art von Methoden darstellen, ist auch bei diesen eine Überladung erlaubt, solange sie sich in der Parameterliste unterscheiden.

```
public Koerper()
{
    this(20, 300, new Vector(0, 1.0), STANDARD_FARBE);
}

public Koerper(int groesse, double masse, Vector bewegung, Color farbe)
{
    this.masse = masse;
    ..... weitere Anweisungen .....
}
```

Der erste so genannte Standardkonstruktor besitzt keine Parameterliste und dient eigentlich nur zum Testen. Mit diesem kannst du interaktiv Standardkörper erzeugen, ohne Details angeben zu müssen. Alle Objekte, die mit diesem Konstruktor erzeugt werden, haben eine fest vorgegebene Standardgröße, -masse, -bewegung und -farbe. Durch den Aufruf des Schlüsselwortes *this* werden diese vier Standardwerte dem anderen Konstruktor übergeben, der schließlich das Objekt erzeugt.

Der zweite Konstruktor zeigt eine weitere Einsatzmöglichkeit für das Schlüsselwortes *this*. Auch hier handelt sich um eine Überladung, da der gleiche Name für zwei Variablen (eine lokaler Parameter innerhalb des Konstruktors und ein Datenfeld) verwendet wird. Wenn du *masse* (ohne *this*) schreibst, wird automatisch die nächst liegende Definition einer Variablen dieses Namens verwendet - in diesem Fall der Parameter. Schreibst du hingegen *this.masse*, so gibst du damit an, dass du das *masse*-Datenfeld des aktuellen Objekts meinst.

Interessant ist auch die Deklaration der Konstanten:

```
private static final double GRAVITY = 5.8;
private static final Color STANDARD_FARBE = new Color(255, 216, 0);
```

Mit dem Schlüsselwort *final* bezeichnest du ein Datenfeld, dessen Wert von den Objekten nicht geändert werden kann. Es ist somit eine Konstante. Das Schlüsselwort *static* hat den Effekt, dass diese Konstante von allen Objekten dieser Klasse genutzt werden kann - schließlich benötigst du nicht einzelne Kopien davon in jedem Objekt. So genannte statische Felder gehören zu der Klasse und erlauben den Zugriff aus den Objekten heraus.

### Übung 2.4:

Entferne im zweiten Konstruktor das Schlüsselwort *this* vor *masse*, so dass du die Zeile `masse = masse` erhältst.

- Kannst du den Quelltext kompilieren?
- Kannst du ein Objekt der Klasse *Koerper* erzeugen?
- Welchen Effekt hat dieser neue Quelltext? Untersuche hierzu mit Hilfe des Inspektors das Datenfeld *masse*.
- Wenn du fertig bist, mache deine Änderungen wieder rückgängig.

## 2.2. Bewegung

Nun wird es Zeit, dass sich etwas tut. Die Hilfsklasse *SmoothMover* vererbt an ein Objekt der Klasse *Koerper* die Methode *move()*. Somit hat jeder Körper Zugriff auf diese Methode.

### Übung 2.5:

Füge in die Methode *act()* der Klasse *Koerper* die Methode *move()* ein.

- Erzeuge ein Objekt. Welchen Wert hat seine Geschwindigkeit. Welchen Wert hat seine Bewegungsrichtung? Ändere die Standardrichtung eines Körpers nach *links*.
- Erzeuge mehrere *Koerper*-Objekte. Wie verhalten sie sich?
- Rufe die öffentlichen *Space*-Methoden (beispielsweise *sonneUndPlanet()*) auf. Wie bewegen sich diese Objekte? Wo sind deren anfängliche Bewegungsrichtung und Geschwindigkeit definiert?

Du verwendest die Methode *move()*, um den Körpern mitzuteilen, dass sie sich bewegen sollen. Sie bewegen sich auch, allerdings in einer geraden Linie. Die Bewegung der Körper, also die Richtung und die Geschwindigkeit, ist durch ihren Bewegungsvektor vorgegeben. Da dieser nicht geändert wird, bleibt die Bewegung konstant. Dieses Verhalten entspricht leider nicht der Bewegung der Planeten um die Sonne.

### Bemerkung:



Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *gravitation2*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

## 2.3. Gravitation

Die Gravitationskraft hat einen entscheidenden Einfluss auf die Bewegung von Himmelskörpern. Wenn du mehr als einen Körper in den Weltraum platzierst, sollen die Gravitationskräfte zwischen diesen Körpern deren Bewegungen (Richtung und Geschwindigkeit) beeinflussen. Bevor dieser Körper sich also bewegt, sollten die Kräfte aller anderen Körper im Weltraum berechnet werden.

Im Pseudocode lässt sich diese Aufgabe etwa wie folgt formulieren:

```
wende die Kräfte der anderen Körper an()
{
    ermittle alle anderen Körper im Weltraum;
    für jeden dieser Körper {
        wende die Gravitationskraft dieses Körpers auf den eigenen an;
    }
}
```

Es gibt also einiges zu tun. Deswegen wird das Problem in kleinere Teilprobleme zerlegt.

### Übung 2.6:

Erstelle eine neue (anfänglich noch leere) Methode *anwendeKraefte()* und rufe diese in der Methode *act()* auf.

```
public void act()
{
    anwendeKraefte();
    move();
}

private void anwendeKraefte()
{
    //leer
}
```

**Bemerkung:**

Methoden können *public* oder auch *private* sein. Wenn Methoden von außerhalb der Klasse aufgerufen werden sollen, dann muss die Zugriffsstufe auf *public* gesetzt werden. Wenn die Methode nur von anderen Methoden derselben Klasse aufgerufen werden sollen, dann sollten diese besser als *private* deklariert werden.

Als nächstes musst du herausfinden, wie du auf alle Objekte, die sich in der Welt befinden, zugreifen kannst. Die Klasse *World* verfügt hierfür über einige Methoden.

### Übung 2.7:

Öffne die Dokumentation über die Klasse *World*. Suche alle Methoden, die dir einen Zugriff auf Objekte in der Welt erlauben.

Die Methode *getObjects()* scheint die interessanteste zu sein:

---

#### **getObjects**

```
public java.util.List getObjects(java.lang.Class cls)
```

Get all the objects in the world, or all the objects of a particular class.

If a class is specified as a parameter, only objects of that class (or its subclasses) will be returned.

**Parameters:**

`cls` - Class of objects to look for ('null' will find all objects).

**Returns:**

A list of objects.

---

Laut Beschreibung erhältst du mit dieser Methode eine Liste aller Objekte einer bestimmten Klasse, die du als Parameter der Methode mitgibst. Übergibst du den Parameter *null*, so wird sogar eine Liste aller Objekte von allen Klassen in dieser Welt zurück.

```
getObjects(Koerper.class);
```

Allerdings liegt die Methode *getObjects()* in der Klasse *World*. Da du jedoch in der Klasse *Koerper* diese Methode verwenden willst, musst du dir mit *getWorld()* das Welt-Objekt liefern lassen, in der der Körper lebt.

### Übung 2.8:

Überlege, in welcher Klasse die Methode *getWorld()* liegen muss. Lies die Beschreibung. Was macht diese Methode?

---

## getWorld

```
public World getWorld()
```

Return the world that this object lives in.

**Returns:**

The world.

---

Diese Methode *getWorld()* liefert also ein Welt-Objekt zurück, auf das du dann die Methode *getObjects()* aufrufen kannst:

```
getWorld().getObjects(Koerper.class);
```

Der Rückgabetyt dieser Methode wird als *java.util.List* angegeben. Diesen neuen Datentyp *List* wirst du nun etwas genauer kennen lernen.

### 2.3.1. Der Typ List

Wie bereit erwähnt, bekommst du mit

```
getWorld().getObjects(Koerper.class);
```

eine Liste aller Objekte einer bestimmten Klasse in der Welt. Allgemein versteht man unter einer Liste eine Sammlung, die eine beliebige Anzahl anderer Objekte enthalten kann.

```
List<Koerper> koerperListe = getWorld().getObjects(Koerper.class);
```

Die obige Anweisung besagt, dass *koerperListe* eine *List* hält, in der *Koerper*-Objekte gespeichert werden können, die sich zur Zeit im Weltraum befinden.

Die Klasse (genauer das Interface) *List* befindet sich im Paket *java.util* und muss deshalb mit

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

zuerst importiert werden, um diese im Szenario verwenden zu können.

#### Bemerkungen:

1. Im Menü *Help* findest du den Menüpunkt *Java Library Documentation*. Hierunter findest die Java-Standardklassenbibliothek, in der Tausende von Klassen beschrieben werden. Um etwas Ordnung in diese Klassen zu bringen, wurden diese in Paketen zusammengefasst. *List* findest du im Paket namens *java.util*.

2. Wenn du im Paket *java.util* nach *List* gesucht hast, wirst du festgestellt haben, dass der Typ *List* keine Klasse ist, sondern ein Interface. Betrachte ein Interface vorläufig wie eine Superklasse, die allerdings keine Methoden vererben kann, sondern nur so eine Art Vereinbarung ist, dass deren Subklassen diese Methoden tatsächlich besitzen. Es reicht zu wissen, dass du den Typ *List* genauso wie alle anderen Typen verwenden kannst.

### 2.3.2. Die for-each-Schleife

Zum Durchlaufen der *koerperListe* wird eine *for-each*-Schleife verwendet. Hier wird eine Schleifenvariable *koerper* vom Typ *Koerper* deklariert, die dann nacheinander für alle Elemente der Sammlung *koerperListe* verwendet wird, um irgendetwas zu tun.

```
for (Koerper koerper : koerperListe) {  
    //tue etwas  
    //tue noch etwas  
}
```

#### Bemerkung:

Du kannst die *for-each*-Schleife etwas leichter lesen, wenn du das Schlüsselwort *for* als „für jeden“ liest, den Doppelpunkt als „in“ und die öffnende geschweifte Klammer als „tue“. Dann wird aus der Schleife:

Für jeden *koerper* in *koerperListe* tue: ...

Somit wird also jede Anweisung in der geschweiften Klammer für jedes Element in der Liste *koerperListe* einmal ausgeführt.

#### Übung 2.9:

- a) Implementiere in der Methode *anwendeKraefte()* eine lokale Variable *alleKoerper* vom Datentyp *List* und die *for-each*-Schleife.
- b) Wende auf alle Elemente der *koerperListe* die Anweisung *move()* an. Teste dein Szenario, indem du mehrere Körper in das Weltall setzt.

Nun kannst du diese *for-each*-Schleife dazu nutzen, um die Gravitationskräfte von allen Körpern auf den aktuellen Körpern anzuwenden. Du nimmst einfach jedes Element, das in der lokalen Variablen *koerper* gespeichert ist und übergibst es einer Methode namens *anwendeGravitation()*.

```
for (Koerper koerper : koerperListe) {  
    angewendeGravitation(koerper)  
}
```

Allerdings so einfach ist es nicht ganz. Du musst beachten, dass in der *koerperListe* sich auch das aktuelle Objekt befindet, auf das du die Gravitation anwendest. Es ist jedoch nicht sinnvoll, die Gravitation eines Objekts auf sich selbst anzuwenden. Sobald also das aktuelle Objekt beim Durchlaufen der Liste erscheint, wird es einfach übergangen. Der Quelltext lautet somit folgendermaßen:

```
private void anwendeKraefte()
{
    List<Koerper> koerperListe = getWorld().getObjects(Koerper.class);

    for (Koerper koerper : koerperListe) {
        if (koerper != this) {
            anwendeGravitation(koerper);
        }
    }
}

private void anwendeGravitation(Koerper andererKoerper)
{
    //noch zu erledigen
}
```

### Übung 2.10:

Implementiere in der Methode *anwendeKraefte()* und die noch leere Methode *anwendeGravitation()*. Dokumentiere dein Methoden!

### Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *gravitation3*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

## 2.3.3. Gravitationskraft anwenden

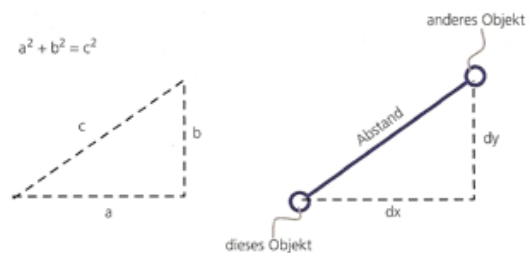
Nun musst du etwas Physik betreiben. Dies ist jedoch nicht allzu schwierig, da nur eine Formel verwendet wird:

$$Kraft = G \cdot \frac{Masse_1 \cdot Masse_2}{Entfernung^2}$$

Um die Kraft zu berechnen, die auf das aktuelle Objekt anzuwenden ist, musst du die Masse dieses Objekts mit der Masse des anderen Körpers multiplizieren und durch das Quadrat der Entfernung dividieren. Anschließend muss dieser Wert noch mit der Gravitationskonstanten  $G$  multipliziert werden. Du benötigst nur noch eine Formel für die Entfernung (Pythagoras:  $a^2 + b^2 = c^2$ ) und die Beschleunigung (Kraftgesetz:  $a = \frac{F}{m}$ ). Das war schon die ganze Physik.

Diese drei Formeln werden in der Methode `anwendeGravitation()` implementiert. Dies ist jetzt relativ einfach, da du nur noch mit zwei Objekten arbeiten musst, nämlich dieser aktuelle Körper und der andere Körper, der als Parameter übergeben wird. Die Gravitationskraft des anderen (übergebenen) Körpers wird auf diesen aktuellen Körper angewendet.

Zuerst berechnest du den Abstand zwischen den beiden Körpern. Hierzu wird aus den x/y-Koordinaten die waagrechte und senkrechten Entfernungen ermittelt und mit Hilfe des Satzes von Pythagoras der Abstand berechnet.



```
//Berechnung der Entfernung
double dx = andererKoerper.getExactX() - this.getExactX();
double dy = andererKoerper.getExactY() - this.getExactY();
double abstand = Math.sqrt(dx * dx + dy * dy);
```

Anschließend wird mit Hilfe des Gravitationsgesetzes und des Kraftgesetzes die Stärke der Gravitationskraft und die Größe der dadurch hervorgerufenen Beschleunigung ermittelt.

```
//Berechnung der Staerke der Kraft
double kraftStaerke = GRAVITATION * (this.masse * andererKoerper.masse) /
    (abstand * abstand);

//Berechnung der Groesse der Beschleunigung
double beschleunigung = kraftStaerke / this.masse;
```

Nun benötigst du noch einen Kraftvektor. Dessen Richtung wird durch die aktuellen Positionen der beiden Körper und dessen Länge wird durch die Größe der Beschleunigung festgelegt.

```
//Berechnung des aktuellen Bewegungsvektors
Vector bewegung = new Vector(dx, dy);
bewegung.setLength(beschleunigung);
```

Im letzten Schritt wird dieser Vektor zur aktuellen Bewegung addiert.

```
//Berechnung des neuen Bewegungsvektors  
addForce(bewegung);
```

### Übung 2.11:

Implementiere in der Methode *anwendeGravitation()*. Dokumentiere dein Methoden!

### Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *gravitation4*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

## 2.4. Testen

Nun solltest du das Szenario ausführlich testen. Für den Anfang kannst du erst einmal die drei vordefinierten Szenarien der Klasse *Space* verwenden.

### Übung 2.12:

- a) Teste dein Szenario mit Hilfe der öffentlichen Methoden *sonneUndPlanet()*, *sonneUndZweiPlaneten()* und *sonnePlanetMond()*.
- b) Experimentiere mit unterschiedlichen Werten für die Gravitationskraft.
- c) Experimentiere mit unterschiedlichen Werten für die Masse und Anfangsbewegung der Körper.
- d) Erzeugen neue Sterne-Planeten-Konstellationen und beobachte, wie sich verhalten. Findest du ein System, das stabil ist.