# Programmieren II (Java)

## 4. Praktikum: Vertiefung Objektorientierung

Sommersemester 2024
Christopher Auer

## Abgabetermine

## Lernziele

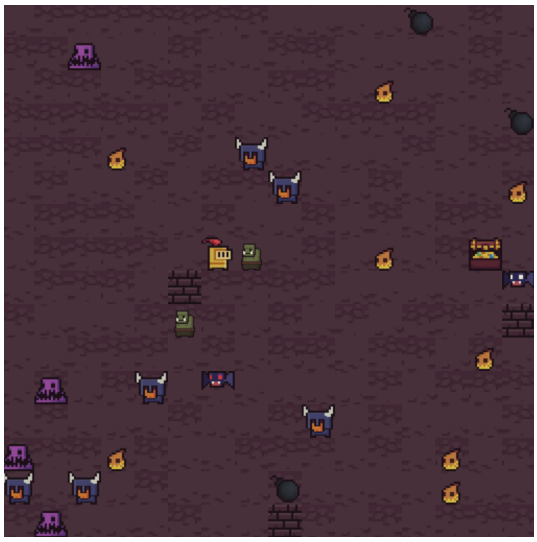
- Vertiefung Objektorientierung in Java
- Abstrakte Klasse, Ableitung, **super**-Methodenaufrufe
- Interfaces, implementieren von Interfaces

## Hinweise

- Sie dürfen die Aufgaben *alleine* oder zu *zweit* bearbeiten und abgeben
- Sie müssen *4 der 5* Praktika bestehen
- *Kommentieren* Sie Ihren Code
  - Jede *Methode* (wenn nicht vorgegeben)
  - *Wichtige* Anweisungen/Code-Blöcke
  - *Nicht kommentierter* Code führt zu *Nichtbestehen*
- Bestehen Sie eine Abgabe *nicht* haben Sie einen *zweiten Versuch*, in dem Sie Ihre Abgabe *verbessern müssen*.
- *Wichtig*: Sie sind einer *Praktikumsgruppe* zugewiesen, *nur* in dieser werden Ihre Abgaben *akzeptiert*!

## Aufgabe: Dungeon Chase

In diesem Praktikum programmieren Sie ein Spiel:[1]



In dem Spiel muss unser *Held*, der Ritter in Gold in der Mitte, die *Truhe mit Gold* auf der rechten Seite erreichen. Leider stellen sich dem Helden allerlei Monster und andere Hindernisse in den Weg, die versuchen dies zu verhindern.

# Keine Panik!

Die *grafische Ausgabe* ist bereits vorbereitet und sie müssen sich nur um die Implementierung der *Spielregeln* kümmern!

### Importieren des Projekts

Importieren Sie das *Gradle-Projekt* wie gewohnt und führen Sie den *Gradle-Task* `run` aus. Es sollte ein Fenster mit dem Boden des Dungeons erscheinen, das Sie gleich wieder schließen können. Die Anwendung basiert auf JavaFX, einer modernen Java-Bibliothek um grafische Benutzeroberflächen (GUIs) zu implementieren. Gradle wird das Paket automatisch beim Start herunterladen und einbinden.

**Terminal-Version** Sollte die grafische Ausgabe bei Ihnen nicht funktionieren (Probleme mit *JavaFX*), verwenden Sie stattdessen die *Terminal-Version*. Öffnen Sie dazu das Projekt im Verzeichnis

`SupportMaterial/terminal-dungeon-chase`

und implementieren Sie dort Ihre Klassen. Statt einer Ausgaben über Bilder (*Tiles*) werden Zeichen verwendet (siehe Beschreibung von `getImage` unten).

[1] Wir nutzen dafür das 16×16 Dungeon Tileset von 0x72 das frei zu Verwendung steht (Public Domain)

## Das `enum Direction`

Die Elemente in unserem Spiel bewegen Sie auf einem rechteckigem Gitter der Größe $16 \times 16$, wobei der Ursprung links unten liegt, die $x$-Koordinatenachse nach rechts und die $y$-Koordinatenachse nach oben zeigt. In einem Schritt gibt es *neun mögliche Bewegungen*: Entweder man bleibt auf dem Feld *stehen* oder geht in eine der folgenden *acht Richtungen*:

Dafür definieren wir ein **`enum`** `Direction`:

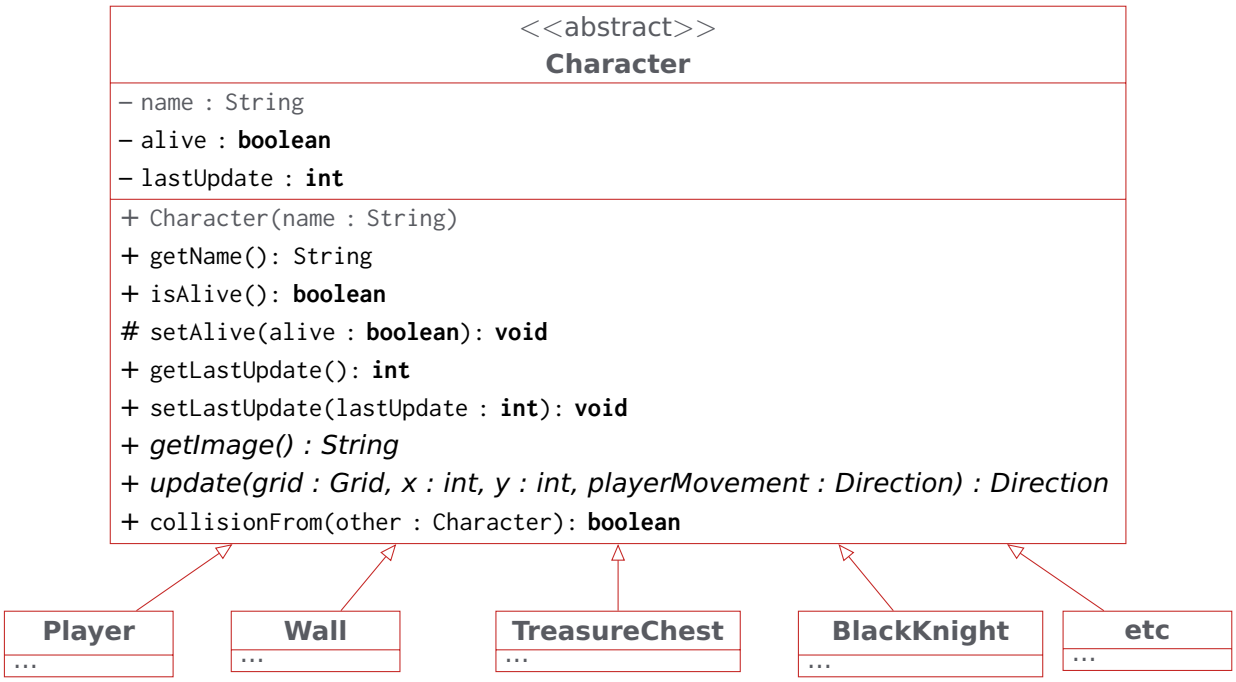
| `enum`-Wert | dx | dy | Richtung |
|---|---|---|---|
| `NONE` | `0` | `0` | keine Richtung |
| `NORTH` | `0` | `+1` | oben |
| `NORTH_EAST` | `+1` | `+1` | rechts oben |
| `EAST` | `+1` | `0` | rechts |
| `SOUTH_EAST` | `+1` | `-1` | rechts unten |
| `SOUTH` | `0` | `-1` | unten |
| `SOUTH_WEST` | `-1` | `-1` | unten links |
| `WEST` | `-1` | `0` | links |
| `NORTH_WEST` | `-1` | `+1` | links oben |

Die Attribute `dx` und `dy` definieren die Änderungen in der $x$- und $y$-Achse. Neben dem Konstruktur und den Gettern `getDX` und `getDY`, implementiert `Direction` noch folgende Methoden:

- Die *statische Methode* `Direction fromD(int dx, int dy)`, die für die Parameter `dx` und `dy` (z.B. `dx=-1`, `dy=+1`) die entsprechende Richtung zurückgibt (z.B. `NORTH_WEST`). Ein ungültiges Parameterpaar führt wie üblich zu einer `IllegalArgumentException`.
- Die Methode `Direction opposite()` gibt die *entgegengesetzte Richtung* zurück, d.h., die Richtung mit invertierten Werten von `dx` und `dy` (z.B. `SOUTH_WEST` für `NORTH_EAST`). *Tipp*: Verwenden Sie `fromD`!
- Die *statische Methode* `Direction random4()` gibt einen zufälligen Wert aus den Richtungen `NORTH`, `EAST`, `SOUTH` und `WEST` zurück.

Implementieren Sie `Direction` vollständig! Für die folgenden Aufgaben gibt es *keine JUnit-Tests*. Stattdessen testen Sie (und die Übungsleiter) Ihre Implementierung indem das Programm *ausgeführt* und das Spiel *gespielt wird*.

## Die abstrakte Oberklasse `Character`




Die *abstrakte Klasse* `Character` ist die Basisklasse für alles, was sich auf dem Spielfeld befinden und eventuell bewegen wird, wie z.B. der Held, die Schatztruhe, Gegner, aber auch Wände. Die Klasse besitzt folgende Attribute:

- `name` — der Name, darf *nicht leer und* **`null`** *sein*
- `alive` — **`true`**, wenn noch am Leben, **`false`**, wenn nicht; zu Beginn **`true`**
- `lastUpdate` — der *letzte Zeitschritt* an dem der Zustand aktualisiert wurde (wird später für den Ablauf des Spiels interessant); hat den Wert `0` zu Beginn

Folgende Methoden werden von `Character` definiert:

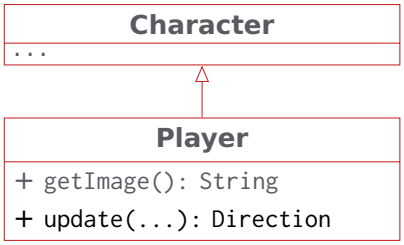
- `Character(String name)` — Konstruktor, der die Attribute initialisiert
- *Getter* für `name`, `alive` und `lastUpdate`
- *Setter* für `lastUpdate` und `alive`; achten Sie hier auf die *Sichtbarkeiten*! Was könnte der Grund sein, dass `setAlive` die Sichtbarkeit **`protected`** hat?
- *Abstrakte Methode* `String getImage(): String` — liefert den Dateinamen mit dem Bild, das *aktuell für den Character angezeigt* werden soll. Die Bilder liegen in `app/src/main/resources`. Was heißt es, dass die Methode *abstrakt* ist? Was ist wohl der Grund dafür?
  *Terminal-Version*: Liefert das Zeichen, das zur Ausgabe auf dem Terminal verwendet werden soll.
- *Abstrakte Methode* `Direction update(Grid grid, int x, int y, Direction playerMovement)` — aktualisiert den Zustand des Characters und gibt die *Richtung zurück in die sich der Character bewegen will*. Die Parameter sind:
  - `Grid grid` — das *Spielfeld* auf dem das Spiel stattfindet (die Klasse `Grid` ist vorbereitet und wird später vervollständigt)
  - **`int`** `x`, **`int`** `y` — aktuelle *Position* des Characters auf dem Spielfeld
  - `Direction playerMovement` — aktuelle Bewegung des *Spielers*. Der Held selbst nutzt den Parameter um seine Bewegungsrichtung zu bestimmen, Gegner können diese Information nutzen um den Spieler *„abzufangen“*.
- **`boolean`** `collisionFrom(Character other)` — wird verwendet um *Kollisionen* aufzulösen, d.h. auf

das Feld, auf dem der Character steht, will der Character `other` ziehen. Die Rückgabe ist **`true`**, wenn `other` auf das Feld gesetzt werden darf, sonst **`false`**. Implementieren Sie `collisionFrom` so, dass `alive` auf **`false`** gesetzt wird (Character überlebt die Kollision nicht) und **`true`** zurückgegeben wird.[2] Wir werden die Methoden später *überschreiben* um andere Kollisionsauflösungen zu implementieren.

Implementieren Sie `Character`! Warum gibt es wohl *keine JUnit-Tests* für `Character`? Nachdem Sie `Character` implementiert haben, kommentieren Sie die Codeblöcke mit `### Character` in `Grid`, `Helper` und `DungeonChaseMain` ein!

## Die Klasse `Player`

Die Klasse `Player` leitet von `Character` ab:




`Player` stellt unseren Helden dar, der vom Spieler gesteuert wird. Dabei werden die Methoden `getImage` und `update` *überschrieben*:

- `getImage` — liefert immer `"player.png"` (*Terminal-Version*: `"+"`)

- `update` — gibt den Parameter `playerMovement` unverändert zurück.

Implementieren Sie `Player` und überschreiben Sie die zwei Methoden, wie in der Vorlesung besprochen:

- Achten Sie dabei auf den Modifizierer `@Override`!
- `@Override` ist nicht zwingend notwendig, aber trotzdem sollte man es angeben. Warum?
- Kommentieren Sie die Codebestandteile in `Grid.java`, die mit `### Player` gekennzeichnet sind!
- Führen Sie den Gradle-Task `run` aus! Es sollte der Held sichtbar sein.

## Der Held lernt laufen!

Die Klasse `Grid` stellt das Spielfeld dar und hat folgende Attribute:

- **`int`** `GRID_WIDTH/GRID_HEIGHT` — Konstanten mit *Breite* und *Höhe* des Spielfelds
- `Character[][] grid` — das Spielfeld mit den Character-Objekten
- **`int`** `playerX/playerY` — die aktuelle Position des Spielers; `playerX=1` und `playerY=GRID_HEIGHT/2` zu Beginn
- `Player player` — Referenz auf das `Player`-Objekt
- `TreasureChest treasureChest` — Referenz auf den *Schatz* (benötigen wir später)
- **`int`** `currentIteration` — aktueller *Iterationsschritt* des Spielverlauf; `0` zu Beginn

[2] *Achtung*: Setzen Sie *nicht* `alive` von `other`, sondern nur von dem `Character`, auf dem `collisionFrom` aufgerufen wird!

Implementieren Sie die *öffentliche Methode* **`void`** `updatePlayer(Direction playerMovement)` wie folgt:

- Prüfen Sie, dass `playerMovement` *nicht* **`null`** ist!
- Rufen Sie `update` auf der `player`-Instanz auf und um die Bewegungsrichtung des Helden zu ermitteln.
- Speichern Sie die Zielposition der Bewegung mit Hilfe von `playerX`, `playerY` und `getDX`, `getDY` der Bewegungsrichtung in *lokalen Variablen*.
- Führen Sie die Bewegung *nur unter folgenden Bedingungen* aus:
  - Die *neue Position* befindet sich *innerhalb* des Spielfelds.
  - Die Zielposition ist *leer*, oder. . .
  - . . . die Zielposition ist *nicht leer* und wird von einer `Character`-Instanz belegt. In diesem Fall führen Sie die Bewegung des Helden *nur durch*, wenn der Aufruf von `collisionFrom(player)` auf der anderen `Character`-Instanz **`true`** liefert.

  Sollten keine der Bedingungen erfüllt sein, bleibt der Held an der Stelle stehen. *Hinweis*: Im Falle einer Bewegung, vergessen Sie nicht die Variablen `playerX`, `playerY` zu aktualisieren!

Kommentieren Sie in `DungeonChaseMain` den Codebestandteil der mit `### updatePlayer` markiert ist. Führen Sie das Programm danach aus: Der Held sollte sich mit den *Pfeiltaste* und *WASD* bewegen lassen! Testen Sie so die Korrektheit Ihrer Implementierung!

## Der erste Gegner: `Ogre` 

Bei unserem ersten Gegner handelt es sich um einen `Ogre` (*Oger*) mit folgendem Verhalten: `Ogre` leitet von `Character` ab mit:

- `Ogre(String name)` — setzt den Namen.
- `String getImage()` — liefert immer `"ogre.png"` (*Terminal-Version*: `"O"`)
- `update` — ermittelt jeweils den Abstand in $x$- und $y$-Richtung zum Spieler. Der Oger bewegt sich in einer der Richtungen `NORTH` oder `SOUTH` wenn der Abstand in $y$ größer ist als in $x$, sonst in eine der Richtungen `WEST` und `EAST`. Natürlich immer *in Richtung des Spielers*! Die aktuelle Position des Spielers können Sie über `grid.getPlayerX/Y()` ermiteln.

Implementieren Sie `Ogre` und kommentieren Sie die Zeile mit der Markierung `### Ogre` in `Grid` ein, damit `Ogre`-Instanzen erstellt werden! Starten Sie das Programm: Die Oger werden Sie noch nicht bewegen! Das lösen wir in den nächsten Aufgaben. *Hinweis*: Der erste Parameter beim Aufruf von `fillGrid` gibt die Anzahl der Gegner an. Verringern und erhöhen Sie die Anzahl nach Belieben.

## Der zweite „Gegner“: `Wall` 

Sogar Wände können wir mit Hilfe einer Ableitung aus `Character` implementieren:

- Der Konstruktor `Wall()` legt den Namen als `"Wall"` fest.
- `getImage` liefert immer `"wall.png"` (*Terminal-Version*: `"#"`)
- `update` liefert immer `Direction.NONE` — Wände können sich nicht bewegen.
- Die Implementierung von `collisionFrom` müssen wir für die Wand *überschreiben*: Die Wand bleibt immer am Leben und gibt bei `collissionFrom` **`false`** zurück, da sie keinen auf ihren Platz lässt.

Implementieren Sie `Wall`, kommentieren Sie die Zeile markiert mit `### Wall` in `Grid` ein, und probieren Sie aus, ob Sie mit dem Helden durch eine Wand laufen können!

## „Oger, Marsch!“

Der Held kann sich schon bewegen, die Oger aber noch nicht. Implementieren Sie daher die *öffentliche Methode* **`void`** `updateOthers(Direction playerMovement)` in `Grid` wie folgt:

- Erhöhen Sie zunächst das Attribut `currentIteration` um `1`; eine neue Iteration beginnt.
- Durchlaufen Sie alle Koordinaten des Gitters in *zwei Schleifen* (ähnlich wie Sie es bei `Bingo`) gemacht haben.
- Sollte ein Feld **`null`** sein, vom Helden belegt sein oder schon im aktuellen Durchlauf verarbeitet worden sein (`getLastUpdate()== currentIteration`), so können Sie es überspringen.
- Sonst ist auf dem aktuellen Feld ein Character, den wir *aktualisieren* müssen: Setzen Sie `lastUpdate` auf `currentIteration`, rufen Sie `update` auf und behandeln Sie Character sonst genauso wie den Helden in der Methode `updatePlayer` (Kollisionsauflösungen, Bewegung, etc.). Vermeiden Sie *Codeduplikation*!
- Sollte der Character nach der Bewegung nicht mehr am Leben sein, setzen Sie das Feld auf **`null`**.

Kommentieren Sie die Zeilen mit der Markierung `### updateOthers` in `DungeonChaseMain` ein. In jeder Runde wird zuerst der Held bewegt und dann alle anderen Elemente. Führen Sie das Programm aus und testen Sie, ob sich Held, Oger und Wände richtig verhalten.

## Die Klasse `Bomb` 

Leiten Sie die Klasse `Bomb` von `Character` ab:

- Der Name ist immer `"Bomb"`
- `getImage` liefert immer `"bomb.png"` (*Terminal-Version*: `"@"`)
- `update` — die Bombe bleibt immer an der gleichen Position
- `collisionFrom` — ruft die Implementierung der Oberklasse `Character` auf, setzt dann `alive` bei `other` auf **`false`** (Bombe ist *hochgegangen*) und gibt **`true`** zurück.

## Die Klasse `BlackKnight` 

Leiten Sie die Klasse `BlackKnight` von `Character` ab:

- Der Name wird über den Konstruktor gesetzt.
- `getImage` liefert immer `"black-knight.png"` (*Terminal-Version*: `"X"`)

- `update` — der schwarze Ritter bewegt sich immer in die *entgegengesetzte Richtung* der Spielerbewegung, d.h. bewegt sich der Spieler nach `NORTH_EAST`, so bewegt sich der schwarze Ritter nach `SOUTH_WEST`.



## Die Schatztruhe: `TreasureChest` 

Es fehlt noch der Schatz, den unser Held erreichen müssen: Die Klasse `TreasureChest` leitet ebenfalls von `Character` ab und ist wie folgt implementiert:

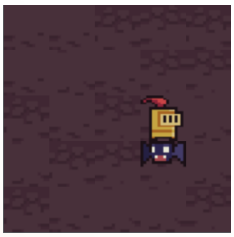
- Der Konstruktor setzt den Namen auf `"Treasure Chest"`
- `getImage` liefert `"treasure-chest.png"` (*Terminal-Version*: `"$"`)
- `update` liefert immer `NONE`; die Schatztruhe sollte nicht weglaufen können
- `collisionFrom(Character other)` wird *überschrieben* und liefert nur **`true`**, wenn `other` der Held ist, sonst **`false`**. Sollte `other` der Held sein, so „stirbt“ die Schatztruhe (`setAlive(`**`false`**`)`). Dies wird benötigt, damit `DungeonChaseMain` feststellen kann, wenn der Held die Schatztruhe erreicht hat.

Implementieren Sie `TreasureChest` und kommentieren Sie die Zeilen mit der Markierung `### TreasureChest` in `Grid` und `DungeonChaseMain` ein!

## Optional: Die Klasse `Bat`

Fledermäuse (`Bat`s) sind die kompliziertesten Gegner: Eine Fledermaus bewegt sich nur in jeder *zweiten Iteration*. Ob sie sich in der ersten Iteration bewegt wird dabei *zufällig entschieden*. Wenn sie sich bewegt, dann bewegt sie sich in *eine der neun möglichen Bewegungsrichtungen* in Richtung des Spielers, d.h., befindet sich `Bat` *links unterhalb* des Helden, so bewegt sich die Fledermaus in Richtung `NORTH_EAST`.

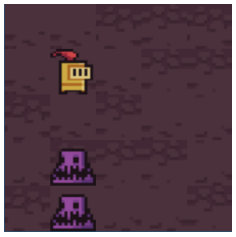




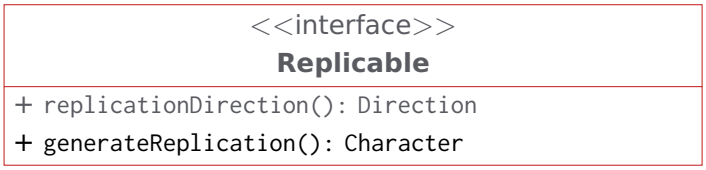
- Der Name wird über den Konstruktor `Bat(String name)` gesetzt.
- `getImage` liefert `"bat.png"` (*Terminal-Version*: `"w"`), wenn sich die Fledermaus in der nächsten Iteration *nicht* bewegt, sonst `"bat-turn.png"` (die *roten Augen* zeigen die folgende Bewegung an; *Terminal-Version*: `W`).
- `update` — bewegt die Fledermaus in jedem zweiten Schritt wie oben beschrieben

## Das Interface `Replicable`

Wir wollen manche unserer Gegner mit der Fähigkeit ausstatten, sich zu *vervielfältigen*:



Dazu definieren wir das *Interface*:

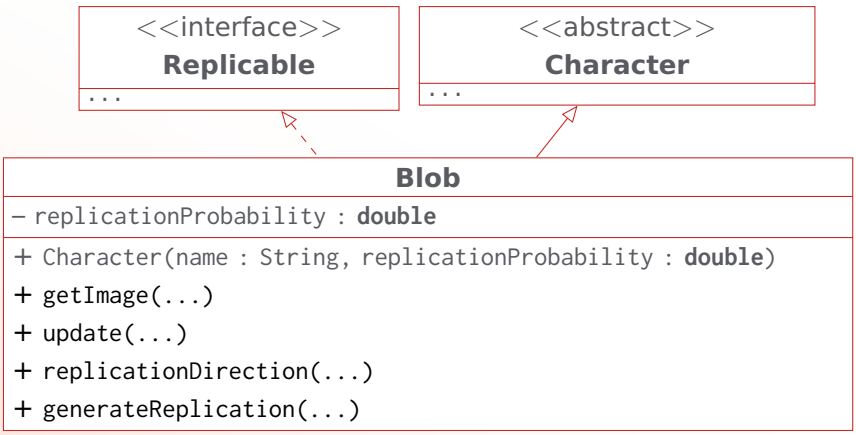



- `Direction replicationDirection()` liefert eine Richtung `!= NONE` zurück, wenn eine *Vervielfältigung* stattfinden soll; `NONE` heißt, es soll nur *eine herkömmliche Bewegung* stattfinden, keine Vervielfältigung.
- `Character generateReplication()` liefert das *Replikat*.

Deklarieren Sie das Interface `Replicable`!

## Der vervielfältigende Blob 

Der `Blob` leitet von `Character` ab und *implementiert* das Interface `Replicable`




- Das Attribut `replicationProbability` ist die Wahrscheinlichkeit, dass in einer Iteration des Spiels eine *Vervielfältigung* stattfindet; muss $\geq 0$ und $\leq 1$ sein.
- Der Konstruktor setzt den Namen und die Vervielfältigungswahrscheinlichkeit.
- `getImage()` liefert `"blob.png"` (*Terminal-Version*: `"o"`)
- `update` lässt den `Blob` zufällig in eine der Richtungen `NORTH`, `EAST`, `SOUTH` oder `WEST` laufen. Nutzen Sie dafür die Methode `Direction.randomDirection4`.
- `replicationDirection()` liefert mit Wahrscheinlichkeit `replicationProbability` zufällig eine der vier Richtungen `NORTH`, `EAST`, `SOUTH` oder `WEST` (nutzen Sie dafür wieder `Direction.randomDirection4`).

Sonst wird `NONE` zurückgegeben. *Tipp*: Nutzen Sie den Ausdruck `Math.random()< p`, der mit einer Wahrscheinlichkeit von `p` **`true`** ist.
- `generateReplication()` liefert eine *Kopie* des `Blob`s.

Implementieren Sie `Blob` und kommentieren Sie die Zeile mit der Markierung `### Blob` in `Grid` ein!

## Replizieren in `updateOthers`

Modifizieren Sie `Grid.updateOthers` wie folgt:

- Nachdem Sie `setLastUpdate` aufgerufen haben, prüfen Sie ob der `Character` auf dem aktuellen Feld das `Replicable`-Interface *implementiert*.
- Wenn ja, dann rufen Sie `replicationDirection` auf und falls dieser Aufruf `!= NONE` liefert, rufen Sie `generateReplication` auf um eine neue Instanz des `Character`s zu erstellen. Bewegen Sie die *neue Instanz* auf das Feld, wie durch die Rückgabe von `replicationDirection` definiert. Die Regeln für diese Bewegung sind die gleichen, wie bei den bisherigen Bewegungen. Fahren Sie danach mit dem nächsten Feld im Gitter fort.
- Sollte es sich nicht um einen `Character` handeln, der das `Replicable`-Interface implementiert, oder sollte `replicationDirection` `NONE` liefern, so wird der `Character` wie gehabt aktualisiert und bewegt.

Führen Sie das Programm aus und testen Sie das Verhalten des `Blob`s! Die Wahrscheinlichkeit für eine Vervielfältigung ist auf `0.05` festgelegt. Sie können Sie Wahrscheinlichkeit zum Testen erhöhen.

## Optional: Feuer breitet sich aus: `Fire` 

`Fire` ist eine einfache Variante von `Blob`:

- `getImage()` liefert `"fire.png"` (*Terminal-Version*: `"*"`)
- `update()` liefert immer `None`
- `collisionFrom` verhält sich wie `Bomb`
- `replicationDirection` und `generateReplication` verhalten sich wie `Blob`

Implementieren Sie `Fire` und kommentieren Sie die Zeile mit der Markierung `### Fire` in `Grid` ein! Testen Sie das Verhalten von `Fire`!

## Optional: Implementieren Sie eigene Gegner!

Sie dürfen natürlich auch noch eigene Gegner-Varianten implementieren! Sie können dafür die Tiles (Bilder) in

`SupportMaterial/dungeon-chase/app/src/main/resources/dungeon-tileset`

nutzen. Suchen Sie sich ein passendes Tile aus, kopieren Sie es in das Verzeichnis `resources` und nennen Sie es ggfsüm. In Ihrer Implementierung von `getImage` geben Sie dann den Dateinamen zurück. *Terminal-Version*: Geben Sie das gewünschte Zeichen zurück.