Tasks:

Das Zepter der Unterwerfung

In dieser Aufgabe übst Du, Situationen mit Klassen und Objekten zu modellieren und UML-Diagramme in Code umzusetzen.

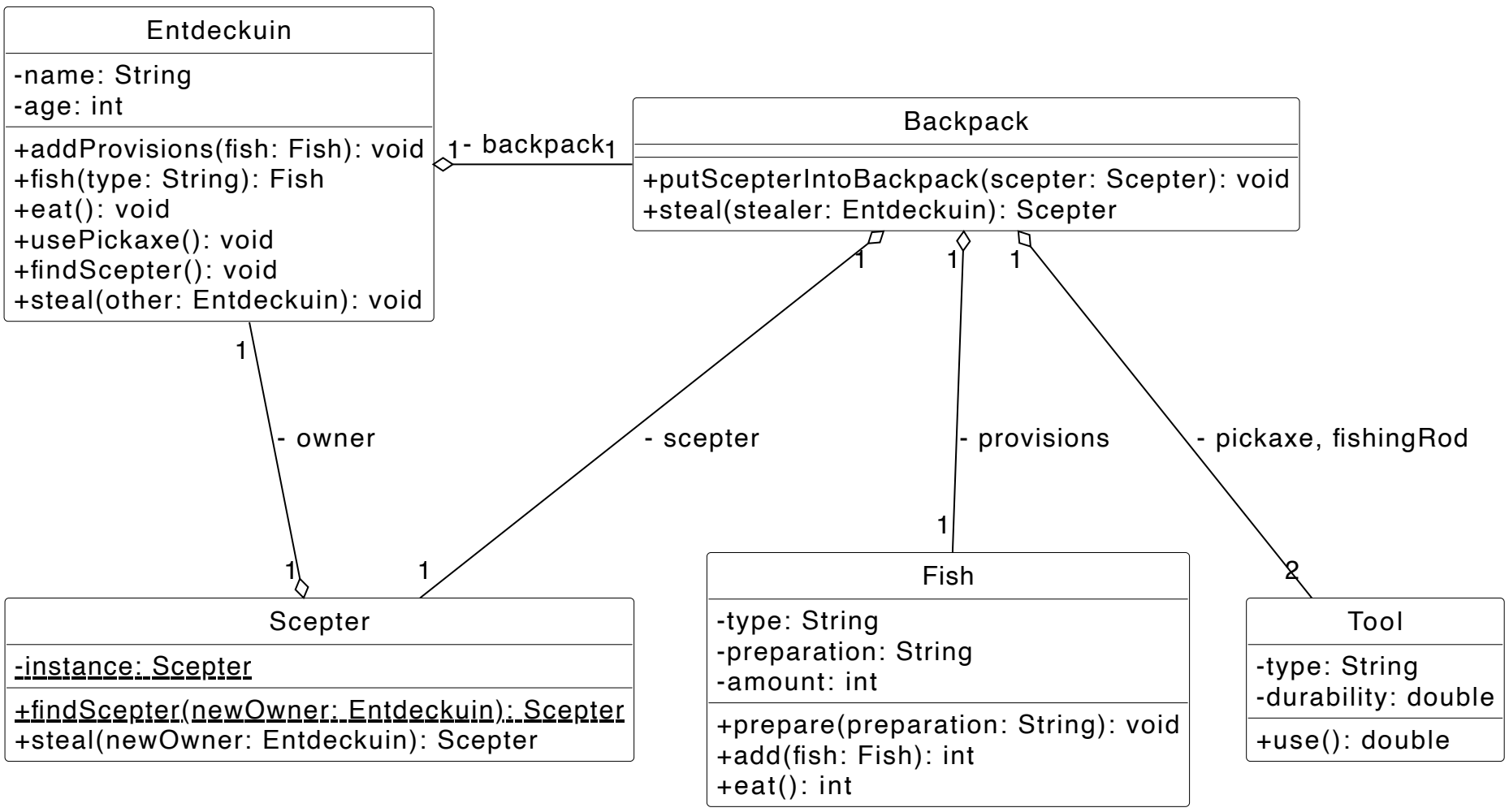
Die Entdeckuine Antonia, Christian und Jonas langweilen sich in der Antarktis. Wohin das Auge sieht gibt es nur Eis und Schnee. Eines Tages fällt Antonia eine alte Geschichte ihres Großvaters ein, die von einem sagenumwobenen Zepter der Unterwerfung erzählt. Der Legende zufolge erhält derjenige die Macht, die Menschheit zu unterwerfen, der das Zepter besitzt. Antonia erzählt ihren beiden Freunden von der Geschichte und zusammen beschließen sie nach diesem Zepter zu suchen. Um sich darauf vorzubereiten, brauchen sie einen Rucksack mit Werkzeugen und Proviant. Am liebsten essen sie natürlich Fisch. Hilfst du ihnen, sich auf die Expedition vorzubereiten?

Bevor wir anfangen gibt es hier ein paar Hinweise:

- Die Aufgabe ist wieder zweigeteilt in Struktur und Funktionalität. Bevor die Tests für die Funktionalität ausgeführt werden können, muss erst die Struktur komplett stimmen.
- Im UML-Diagramm sind ein Attribut und eine Methode unterstrichen. Das bedeutet, dass diese beiden *static* sind.
- Deine IDE kann dir sehr vieles automatisch generieren, v.a. Konstruktoren, Getter und Setter. Informiere dich, welche Tastenkombination das für dich erledigt!
- In der Aufgabe wird in jedem String das Zeichen ␣ verwendet, um Leerzeichen zu kennzeichnen. Ersetze sie dann später im Code durch ein richtiges Leerzeichen.

Struktur No results

Zunächst fangen wir mit der Struktur an, indem wir das folgende UML-Diagramm implementieren. In deinem Repo findest du bereits die notwendigen Klassen. Mit Ausnahme der Klasse `Scepter` bekommt jedes Attribut in jeder Klasse einen *Getter*. Zudem bekommt jede Klasse eine `toString`-Methode, ausgenommen `Backpack`. Infos zu den Konstruktoren findest du unterhalb des Diagramms.



Konstruktoren

- **Fish** No results
  Ein `Fish` hat einen `type` (beispielsweise `"salmon"` oder `"anchovies"`),eine `preparation` (bspw. `"fresh"` oder `"dried"`) und eine Anzahl, wie viele Fische es sind. Die Klasse bekommt 2 Konstruktoren:
  - Der erste bekommt einen Parameter `type` vom Typ `String` und soll das zugehörige Attribut initialisieren. Zudem soll `preparation` mit `"fresh"` und `amount` mit `1` initialisiert werden

- Der zweite soll die beiden Parameter `type` und `preparation` vom Typ `String` in dieser Reihenfolge bekommen und die beiden zugehörigen Attribute Initialisieren. Zudem soll auch hier `amount` mit `1` initialisiert werden.

- ◉ **Tool** No results
  Die Klasse `Tool` hat einen `type` (bspw "`pickaxe`") und eine `durability`. Der Konstruktor bekommt `type` als `String` und soll das zugehörige Attribut, sowie `durability` mit `100.0` initialisieren.
- ◉ **Scepter** No results
  Der Konstruktor für `Scepter` wird etwas anders als gewohnt. Da das Zepter einzigartig ist, soll nur eine einzige Instanz dieser Klasse existieren können und andere Klassen sollen keine Möglichkeit haben, ein weiteres Objekt zu erzeugen. Dies können wir erreichen, indem wir den *default*-Konstruktor mit dem Modifier *private* hinzufügen. Somit existiert kein für andere Klassen aufrufbarer Konstruktor. Damit diese Instanz aber existiert, musst du sie bei der Deklaration von `instance` bereits initialisieren. Diese darf von nun an nie wieder neu initialisiert werden. Wir kümmern uns später darum, wie wir die Klasse sinnvoll verwenden können. Zudem besitzt das Zepter eine Referenz auf den `owner`, welche zu Beginn `null` sein soll.
- ◉ **Backpack** No results
  Unseren `Backpack` benutzen wir, um eine `pickaxe`, eine `fishingRod` und `provisions` in Form von `Fish` zu lagern und transportieren. Falls wir das Zepter gefunden haben, kommt das ebenfalls in den Rucksack. Ein neuer `Backpack` bekommt 2 `String`s namens `fishType` und `fishPreparation`. Beide werden in dieser Reihenfolge dazu benutzt, um `provisions` passend zu initialisieren. Zudem soll `pickaxe` mit "`pickaxe`" und `fishingRod` mit "`fishing␣rod`" initialisiert werden. Da wir das Zepter noch nicht gefunden haben, bleibt es erstmal `null`.
- ◉ **Entdeckuin** No results
  Die Klasse `Entdeckuin` hat die Attribute `name`, `age` und einen `backpack`. Der Konstruktor bekommt 4 Parameter: einen `String name`, einen `int age` und 2 weitere `String`s `favoriteFishType` und `favoritePreparation`. Die ersten beiden Parameter initialisieren die beiden zugehörigen Attribute während mit den restlichen beiden `backpack` passend initialisiert wird.

◉ **Getter** No results

Hier kannst du sehen, ob deine Getter richtig funktionieren.

Funktionalität

◉ **toString** No results

Als nächstes implementieren wir die ganzen `toString`-Methoden. Sie sollen für jede Klasse die folgenden `String`s zurückgeben:

- `Fish`: "<preparation>␣<type>"
- `Tool`: "<type>␣`with`␣<durability>␣`points`␣`of`␣`durability`"
- `Scepter`: "`The`␣`scepter`␣`of`␣`submission`␣`belongs`␣`to`␣<owner>␣`now.`"
- `Entdeckuin`: "`Entdeckuin`␣<name>`(`<age>`)`"

Nachdem wir das jetzt aus dem Weg haben, können wir uns um die eigentlichen Methoden kümmern. Starten wir mit der Klasse `Fish`:

Fish

- ◉ **prepare** No results
  Diese Methode bekommt einen `String` übergeben und soll sein eigenes Attribut damit überschreiben.
- ◉ **add** No results
  Diese Methode bekommt einen anderen `Fish` übergeben und soll die Menge des anderen Fisches auf die eigene addieren. Anschließend soll die neue Gesamtmenge zurückgegeben werden.
- ◉ **eat** No results
  Diese Methode soll einen Fisch von der Gesamtmenge abziehen und den neuen Wert zurückgeben.

Tool

- ◉ **use** No results
  Diese Methode soll die `durability` dekrementieren und den neuen Wert zurückgeben.

Scepter

So, kommen wir nun zu unserer besonderen Klasse. Zum einen wollen wir das Zepter zum ersten Mal finden können, zum anderen wollen wir das Zepter auch stehlen können. Im Prinzip sind beide Methoden genau gleich außer dem kleinen Unterschied, dass eine der beiden *static* ist. Diese erlaubt uns an die Instanz zu kommen, ohne dass wir die Instanz kennen.

- ◉ **findScepter und steal** No results
  Beide Methoden sollen zuerst den `owner` der `instance` auf den übergebenen Wert setzen, dann den Text der `toString`-Methode auf der Konsole ausgeben und zum Schluss die `instance` zurückgeben.

Backpack

- ◉ **putScepterIntoBackpack** No results
  Diese Methode soll `scepter` mit dem übergebenen Parameter überschreiben.
- ◉ **steal** No results
  Diese Methode soll auf `scepter` die `steal`-Methode mit dem übergebenen Parameter aufrufen, die eigene `scepter`-Referenz auf `null` setzen und die `Scepter`-Instanz zurückgeben. Du kannst davon ausgehen, dass der Rucksack das Zepter auch wirklich enthält.

Entdeckuin

Zum Schluss müssen wir nur noch 6 Methoden von `Entdeckuin` implementieren. Bis auf die letzten beiden Methoden kannst du hier schon Punkte bekommen, auch wenn deine anderen Klassen noch nicht richtig funktionieren:

- ◉ **addProvisions** No results
  Mit dieser Methode können unsere Entdeckuine neuen Fisch in ihren Rucksack packen. Sie soll zuerst den übergebenen Fisch genauso zubereiten, wie der restliche Fisch im Rucksack (Stichwort: `preparation`). Danach wird der Fisch zu unseren `provisions` hinzugefügt. Zum Schluss soll noch folgendes auf der Konsole ausgegeben werden: "<this.toString()>␣<preparation>␣some␣<fish.getType()>␣and␣added␣it␣to␣their␣provisions."
- ◉ **fish** No results
  Mit dieser Methode können unsere Entdeckuine fischen. Zuerst soll die Angel aus dem Rucksack *benutzt* werden. Danach soll folgendes auf der Konsole ausgeben: "<this.toString()>␣used␣the␣<toString of fishingRod before usage>␣to␣catch␣some␣<type>.␣It␣has␣<durability of fishingRod after usage>␣durability␣left." Zum Schluss soll ein neuer frischer Fisch mit dem übergebenen `type` zurückgegeben werden.
- ◉ **eat** No results
  Mit dieser Methode können wir den Fisch aus unserem Rucksack *essen*. Anschließend soll folgendes auf der Konsole ausgegeben werden: "<this.toString()>␣ate␣some␣<toString of provisions before consumption>.␣There␣is␣<amount of provisions after consumption>␣<backpack.getProvisions().toString()>␣left."
- ◉ **usePickaxe** No results
  Mit dieser Methode können wir uns den Weg freischlagen. Dabei soll die `pickaxe` zunächst *benutzt* werden und anschließend soll folgender String ausgegeben werden: "<this.toString()>␣used␣the␣<toString of pickaxe before usage>␣to␣crush␣some␣ice.␣It␣has␣<durability of pickaxe after usage>␣durability␣left."
- ◉ **findScepter** No results
  Mit dieser Methode können wir das Zepter finden. Dazu nutzen wir einfach die *static* Methode, an die wir uns selber übergeben (Stichwort: `this`), und packen das Zepter in unseren Rucksack.
- ◉ **steal** No results
  Mit dieser Methode können wir von dem übergebenen Entdeckuin das Zepter klauen. Dazu greifen wir einfach in seinen Rucksack und stehlen das Zepter, welches sich da drin befindet (auch hier übergeben wir uns wieder selber beim Methodenaufruf). Anschließend packen wir auch hier das Zepter in unseren Rucksack. Du kannst davon ausgehen, dass der andere Entdeckuin das Zepter auch wirklich besitzt, wenn die Methode aufgerufen wird.

Beispiel

Geschafft! Hier findest du noch ein kleines Beispiel, welches du bereits in deiner `main`-Methode findest. Probiere ruhig selber eigene Beispiele aus und erschaffe deine eigene Geschichte der 3 Entdeckuine!

```
public static void main(String[] args) {
        Entdeckuin antonia = new Entdeckuin("Antonia", 5, "salmon", "salted");

        antonia.addProvisions(new Fish("salmon", "salted"));
    antonia.usePickaxe();
        antonia.findScepter();
        antonia.eat();
}
```

▶ Wenn du alles korrekt implementiert hast, sollte das Beispiel wie folgt aussehen (zum Ausklappen hier klicken)