Assignment 1: Designing complex data

Goals: Practice designing the representation of complex data.

1.1 Instructions

Be very, very careful with naming! The solution files expect your submissions to be named a certain way, so that they can define their own `Examples` class with which to test your code. Therefore, whenever the assignment specifies:

- the names of classes,
- the names and types of the fields within classes,
- the names, types and order of the arguments to the constructor, or
- filenames,

...be sure that your submission uses exactly those names. Additionally, make sure you follow the course style guidelines. For now the most important ones are: using spaces instead of tabs, indenting by 2 characters, following the naming conventions (data type names start with a capital letter, interfaces begin with an uppercase I, names of fields and methods start with a lower case letter), and having spaces before curly braces.

You will submit this assignment by the deadlines using the course handin server. Follow A Complete Guide to the Handin Server for information on how to use the handin server. **You may submit as many times as you wish. Be aware of the fact that close to the deadline the server may slow down to handle many submissions, so try to finish early.** There will be a separate submission for each problem - it makes it easier to grade each problem, and to provide you with the feedback for each problem you work on.

Due Date: Thursday, May 11th, 9:00 pm

Practice Problems

Work out these problems from *How to Design Classes* on your own. Save them in an electronic portfolio, so you can show them to your instructor, review them before the exam, use them as a reference when working on the homework assignments.

- Problem 2.4 on page 17
- Problem 3.1 on page 25
- Problem 4.4 on page 34

- Problem 5.3 on page 43
- Problem 10.6 on page 102
- Problem 11.2 on page 113
- Problem 14.7 on page 140

> Everywhere in this assignment that you see *`italic, fixed-width text`*, it is intended to be the name of a field, identifier, class name or interface name you must define...but you likely must modify that name a bit to conform to our Java naming conventions: *`hyphenated-names`* are written in `camelCase`, and *`interface names`* begin with an uppercase `I`.
>
> Everywhere that you see `fixed-width text`, it is exactly the name you must use.

Problem 1: Our Canine Friends

We are designing the data collection for the American Kennel Club. For each dog we need to collect the following information:

- `name`: to be represented as a `String`
- `breed`: of dog
- `yob`: the year of birth given as a four digit number
- `state`: of residence – given as the standard two letter abbreviation
- `hypoallergenic`: a boolean value, `true`, if the dog is hypoallergenic

Design the class `Dog` that represents the information about each dog for the census.

Make at least three examples of instances of this class, in the class `ExamplesDog`. Two of the examples should be objects named `huffle` and `pearl` and should represent the following two dogs:

- Hufflepuff, a Wheaten Terrier, born in 2012, resides in TX, and is hypoallergenic
- Pearl, a Labrador Retriever, born in 2016, resides in MA, and is not hypoallergenic

---

What to submit

You should submit your data definitions and examples in a file named `Dog.java`

Remember to check the feedback for Style and Checker Tests in handins!

Problem 2: Icecream

Here is a data definition in DrRacket:

```
;; An IceCream is one of:
;; -- EmptyServing
;; -- Scooped

;;An EmptyServing is a (make-empty-serving Boolean)
(define-struct empty-serving (cone))

;;A Scooped is a (make-scooped IceCream String)
(define-struct scooped (more flavor))
```

- Draw the class diagram that represents this data definition. You may draw this as ASCII-art and include it in your submission, if you wish. Or you can just draw it on paper and not submit it. Regardless, we think it will help you in visualizing how the data is organized.
- Convert this data definition into Java. Make sure you use the same names for data types and for the fields, as are used in the DrRacket data definitions, *converted into Java style standards*. Make sure that the constructor arguments are given in the same order as shown.
- Create examples in a class `ExamplesIceCream`. Include in your examples the following two ice cream orders:

  – a cup of ice cream with scoops of `"mint chip"`, `"coffee"`, `"black raspberry"`, and `"caramel swirl"`

  – a cone with scoops of `"chocolate"`, `"vanilla"`, and `"strawberry"`

Make sure the two sample orders given above are named `order1` and `order2`. **Note: the descriptions above are listed in the order that you would order this in real life. Think carefully how this should be represented as data.**

What to submit

You should submit your data definitions and examples in a file named `IceCream.java`

Remember to check the feedback for Style and Checker Tests in handins!

Problem 3: Problem: Oh-Gi-Yu

We've been asked to help build a new, totally original, deck-building game, *Oh-Gi-Yu*. To start, we're designing representations for the resources a player can have and the actions they can take during their turn. A player can have three kinds of *`resources`*: `Monster`, `Fusion`, and `Trap`.

A `Monster` has a *`name`*, such as Bright Magician, *`hp`* (short for hit points, measured as an integer), and an *`attack`* rating (measured as an integer).

`Fusion` has a *`name`*, such as Green-Eyes Epic Dragon, *`monster1`* and a *`monster2`* of which it is comprised. Note that these can only be `Monster`s, not other `Fusion`s.

A `Trap` has some *`description`* and a flag *`continuous`* denoting whether its effect is instant or occurs over a period of time.

As the game is under construction, the player can only perform two kinds of *`action`*s right now: they can `Attack` one monster with another (be it a fusion or non-fusion monster), or they can `Activate` a trap card.

An `Attack` involves an *`attacker`* and a *`defender`*, both of which are monster resources (as in, not trap cards). To succesfully attack, the attacker's attack value must be worth more than the defender's hp. A `Fusion`'s HP and attack is derived from the sum of its monsters' HP and attack. **Note:** You do not have to implement this calculation in code.

Be sure to only define examples `Attack`s that are successful.

> We're placing a lot of restrictions on the data, such as attacks only involving monsters, and attackers needing to have more attack points than defenders have HP, but aren't actually enforcing these in the code. The ways to enforce these constraints will be further explored later in the semester.

An `Activate` has a *`trap`* (which should be a `Trap`) and a *`target`* (a monster, fusion or otherwise) it is targeting.

- Define six examples of resources, including:
  - `kuriboh`: name `"Kuriboh"`, hp 200, attack 100
  - `jinzo`: name `"Jinzo"`, hp 500, attack 400
  - `kurizo`: name `"Kurizo"`, monster1 `kuriboh`, monster2 `jinzo`
  - `trapHole`: description `"Kills a monster"`, not contiunous

  The others can be whatever you wish.
- Define four types of actions, two of each kind.

Name your action examples `attack1`, `attack2`, `activate1`, `activate2`, etc., and your examples class `ExamplesGame`.

Design:

Note that we have described a data definition which groups monsters, fusions, and trap cards all as resources. Keeping in mind the actions described above, was this a good data design? If so, why, and if not, why not? Leave a brief comment below your examples in your examples class with your answer, and if you think it's a poor design choice, briefly theorize about an alternative design that would make more sense.

What to submit

You should submit your data definitions and examples in a file named `OhGiYu.java`

Remember to check the feedback for Style and Checker Tests in handins!