UML Class Diagram Style Guide

Sam Scott, COMP 10062, Mohawk College, May 2018

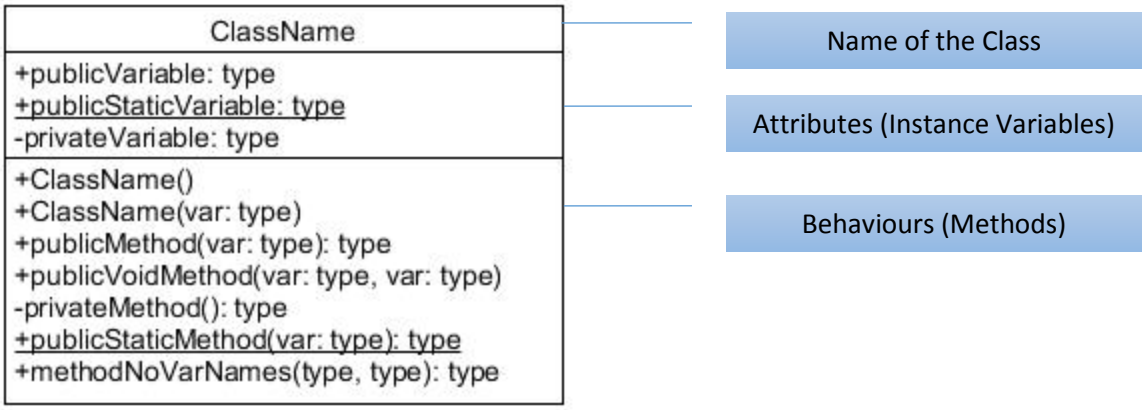
0. What is this?

The UML style we have adopted for this course is a mixture of standard and non-standard elements, and it is not quite the same as the style used in the textbook. When in doubt, use this guide to create UML class diagrams for your assignments.

UML diagrams can be neatly hand drawn and scanned, or created with UMLet (http://umlet.com/) or draw.io(https://draw.io/). If you want to use a different piece of software, you must clear it with your instructor first.

1. Basic Class Diagrams (Weeks 3 – 4)

A class diagram specifies the instance variables and method signatures contained in the class. All variable types, parameter types, and return types.



Access

Every method and instance variable should be marked + for public or – for private.

Types

Use standard Java primitive type names (`int`, `double`, `boolean`, `char`, `long`, etc.) and commonly used class names (`String`, `Date`, `Scanner`, etc.)

Parameters

You can show the name and type for each parameter, or if you think it is clear what the parameters are for, you can just list their types, as in the `methodNoVarNames` method in the diagram above.

Return Types

Constructors and methods of type void don't need to have a return type listed (see `publicVoidMethod` above).

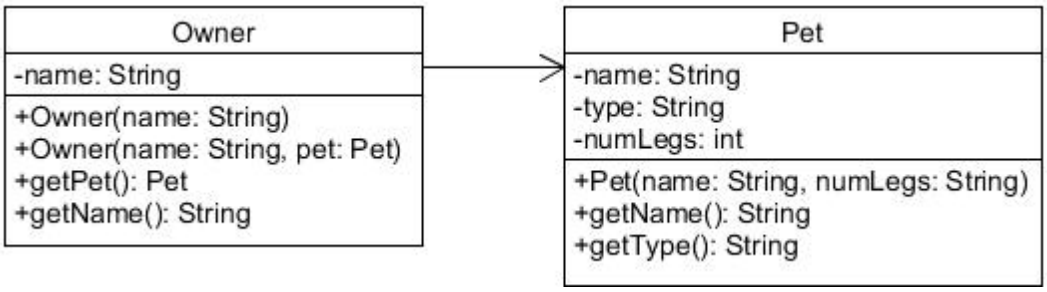
Static

Static variables and methods are underlined. Instance variables and methods are not.

Association (Weeks 4 – 7)

Association (or "has a") relationships are shown with an open-headed arrow, as in the diagrams below. Association arrows represent private instance variables.

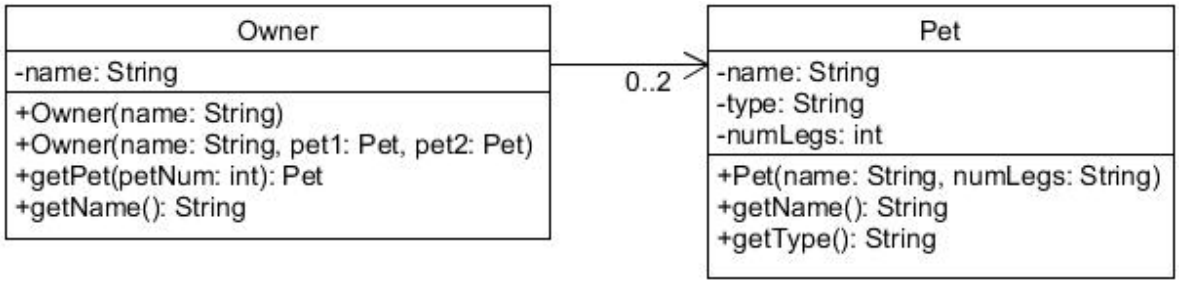
Simple Association



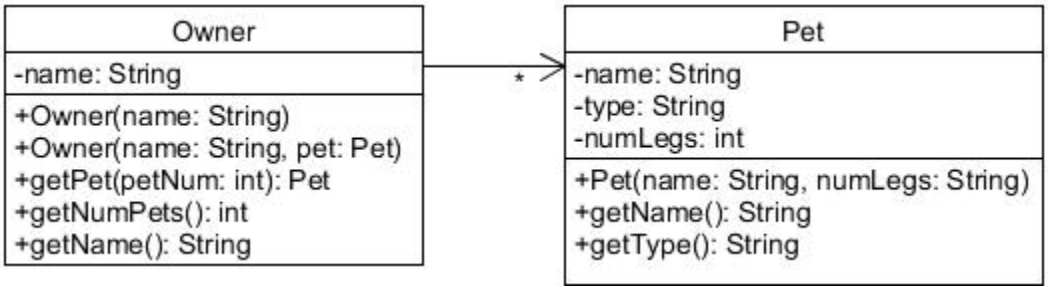
The above diagram shows that an `Owner` has exactly one `Pet`. This means that the `Owner` class contains a private instance variable of type `Pet`, even though this is not shown in the individual class diagram for `Owner`.

Note that there is one `Owner` constructor that does not accept a `Pet` parameter. This constructor will have to create a `Pet` object to satisfy the association relationship.

Multiple Association



The association arrow above has a multiplicity attached to it. It shows that every `Owner` object has 0, 1, or 2 `Pet` objects associated with it. This might mean that `Owner` contains 2 private instance variables of type `Pet`, or it might mean that `Owner` contains a private instance variable type `Pet[]`.

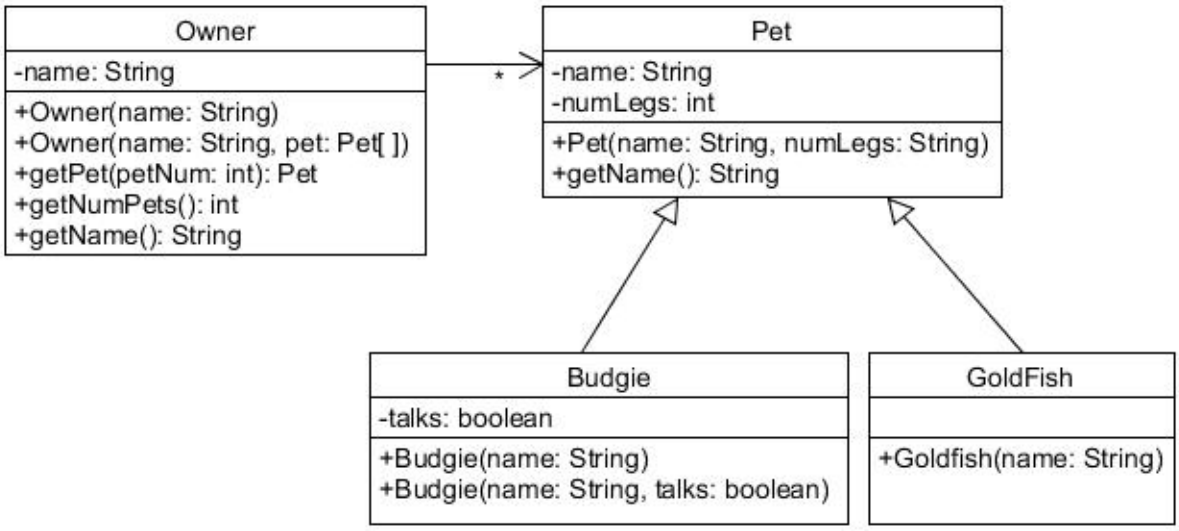


The association arrow above has a multiplicity of `*`. The `*` operator is known as the Kleene Star (https://en.wikipedia.org/wiki/Kleene_star). It means "zero or more". This probably means that `Owner` contains a instance variable of type `Pet[]` or of type `ArrayList<Pet>` (see weeks 11 and 12).

Inheritance (Weeks 8 – 9)

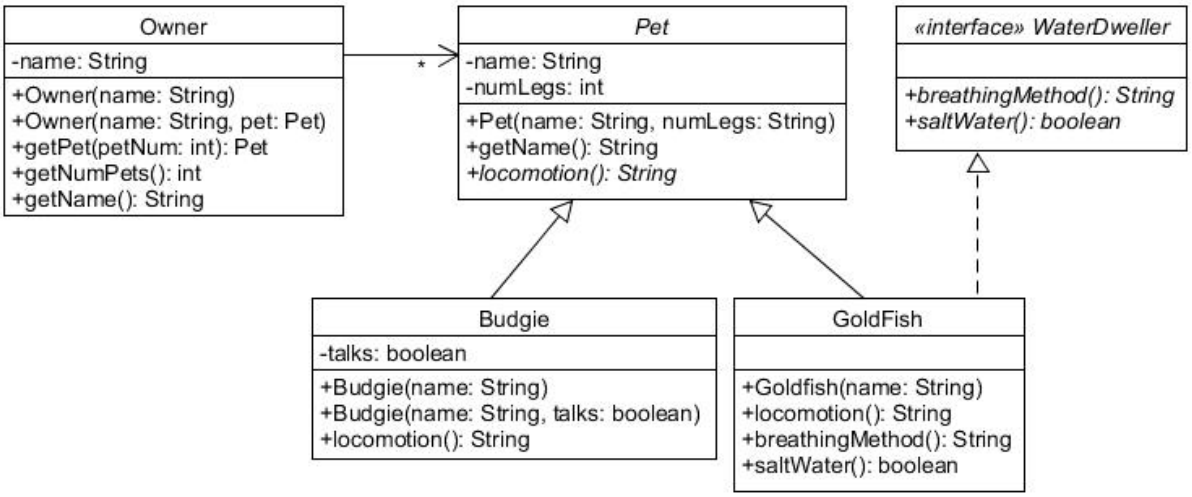
Inheritance (or "is a") relationships are shown with a triangular arrow head. This arrow means that one class extends a class or implements an interface.

Basic Inheritance



In the diagram above, the classes `Budgie` and `Goldfish` both extend or inherit from the `Pet` class.

Abstract Classes and Interfaces



In the diagram above, the italics on the name `Pet` indicate that this is an abstract class. The italics on the `locomotion` method show that it is an abstract method which both `Budgie` and `Goldfish` must override.

The italics and the tag <<interface>> show that `WaterDweller` is not a class, but an interface. The dashed inheritance arrow shows that `Goldfish` implements the `WaterDweller` interface. This means that it must override `breathingMethod` and `saltWater`.