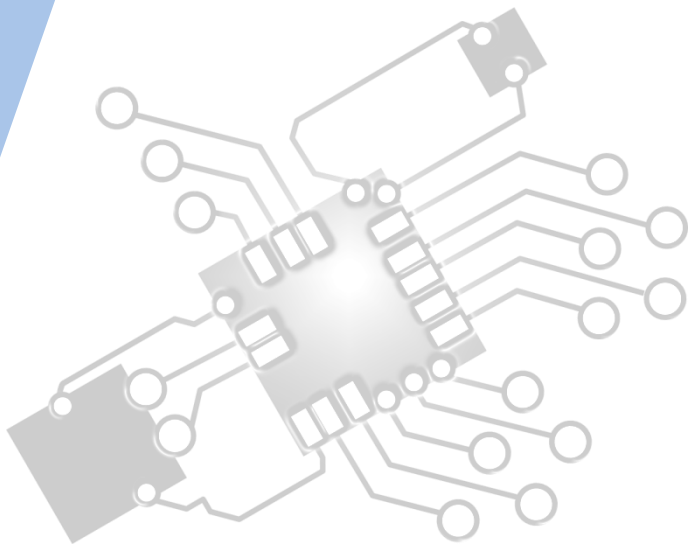
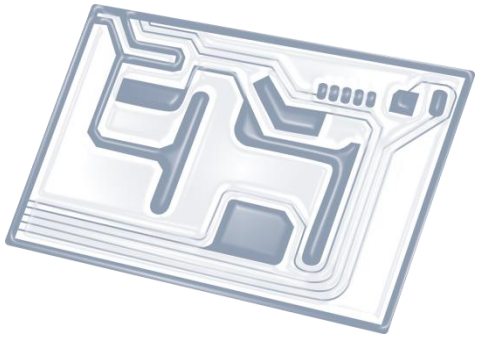
UML

IB Computer Science



Content developed by
Dartford Grammar School
Computer Science Department

HL Topics 1-7, D1-4



1: System design

2: Computer Organisation

3: Networks

4: Computational thinking

5: Abstract data structures

6: Resource management

7: Control

D: OOP

HL & SL D.1 Overview

D.1 Objects as a programming concept

D.1.1 Outline the general nature of an object

D.1.2 Distinguish between an object (definition, template or class) and instantiation

D.1.3 Construct unified modelling language (UML) diagrams to represent object designs

D.1.4 Interpret UML diagrams

D.1.5 Describe the process of decomposition into several related objects

D.1.6 Describe the relationships between objects for a given problem

D.1.7 Outline the need to reduce dependencies between objects in a given problem

D.1.8 Construct related objects for a given problem

D.1.9 Explain the need for different data types to represent data items

D.1.10 Describe how data items can be passed to and from actions as parameters

1: System design

2: Computer Organisation

3: Networks

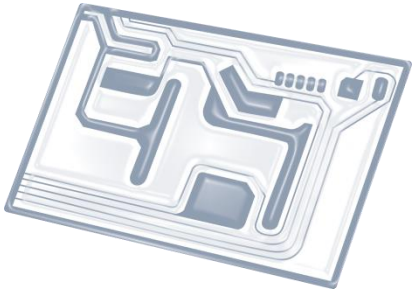
4: Computational thinking

5: Abstract data structures

6: Resource management

7: Control

D: OOP



Topic D.1.3

Construct unified modelling language (UML) diagrams to represent object designs.

UML in reality

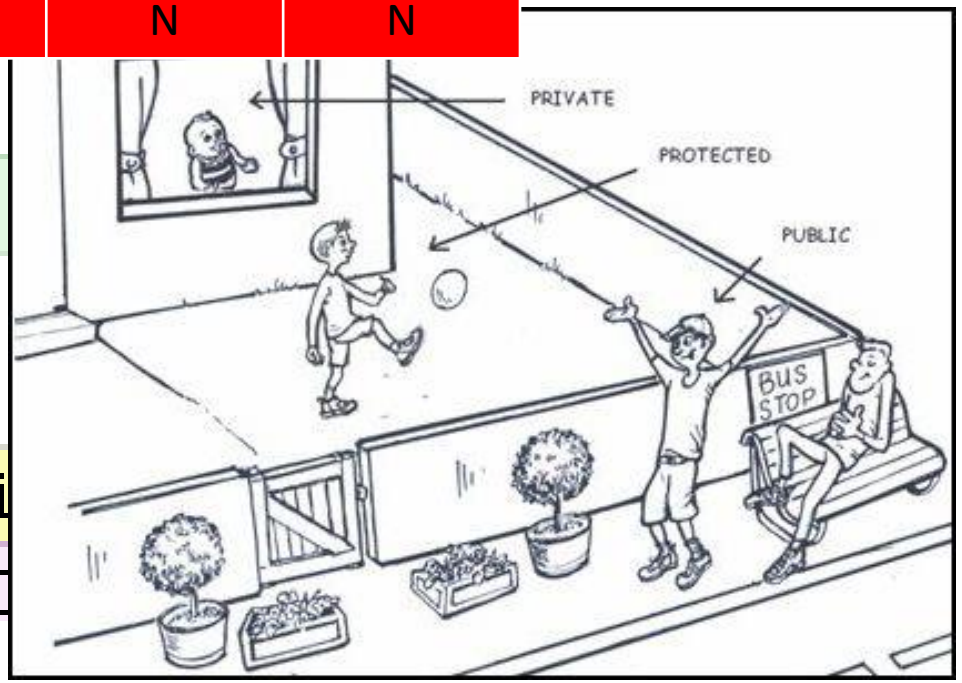
- UML helps to show interaction, behaviour and/or structure of programs and systems in diagrammatical form.
- UML has 6 different types of diagram in reality. In IB however, we only need to look at one of these:

- Component diagram
- Activity diagram
- Sequence diagram
- **Class diagram**
- Use Case diagram
- Communication diagram

Foreword: Access levels in Java

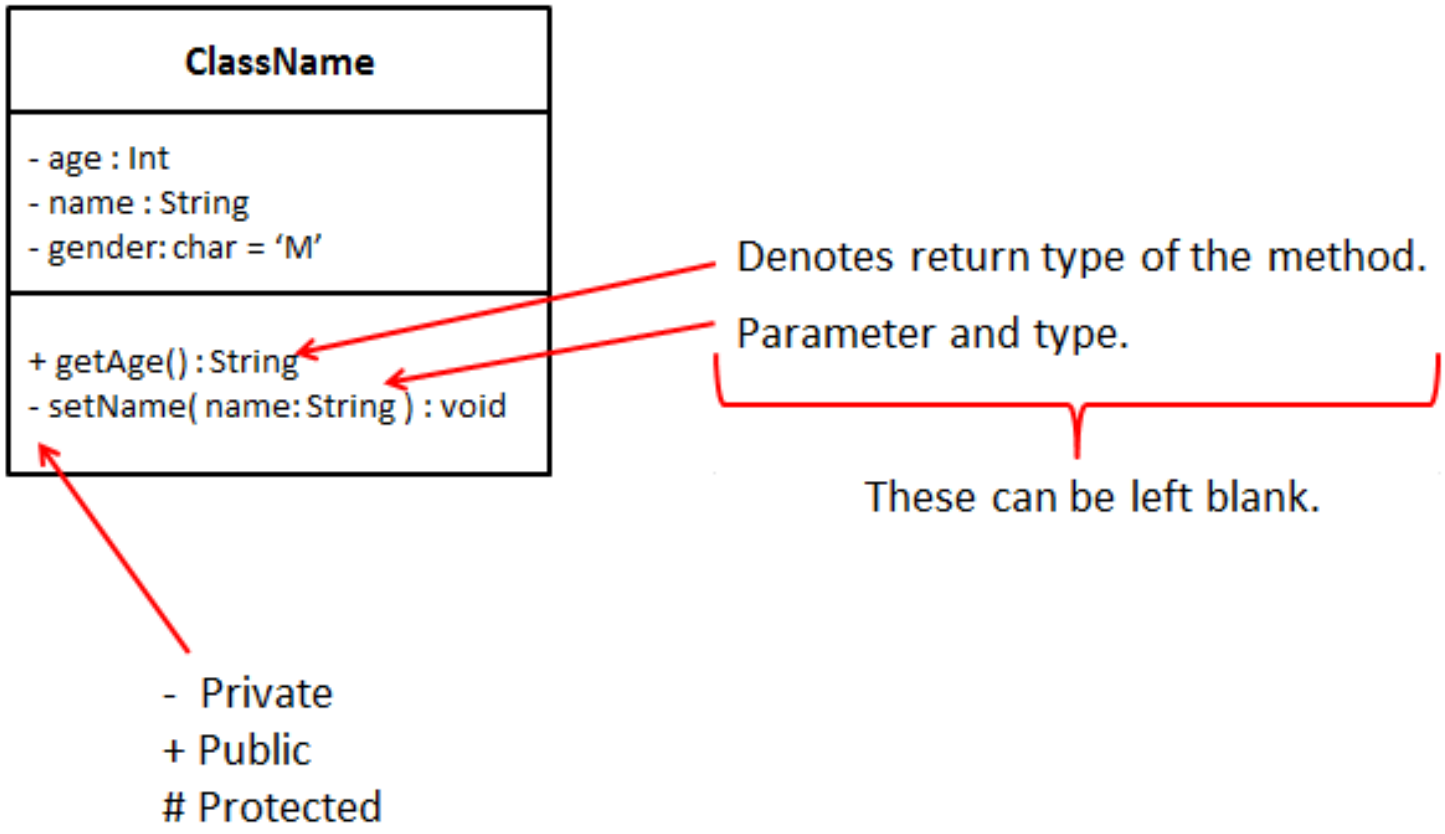
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

```
public class Dog {
    private String
    private int tri
        for(int a=0
```



Class Diagrams

- Classes (*object*), variables (*state*) & methods (*behaviours*)

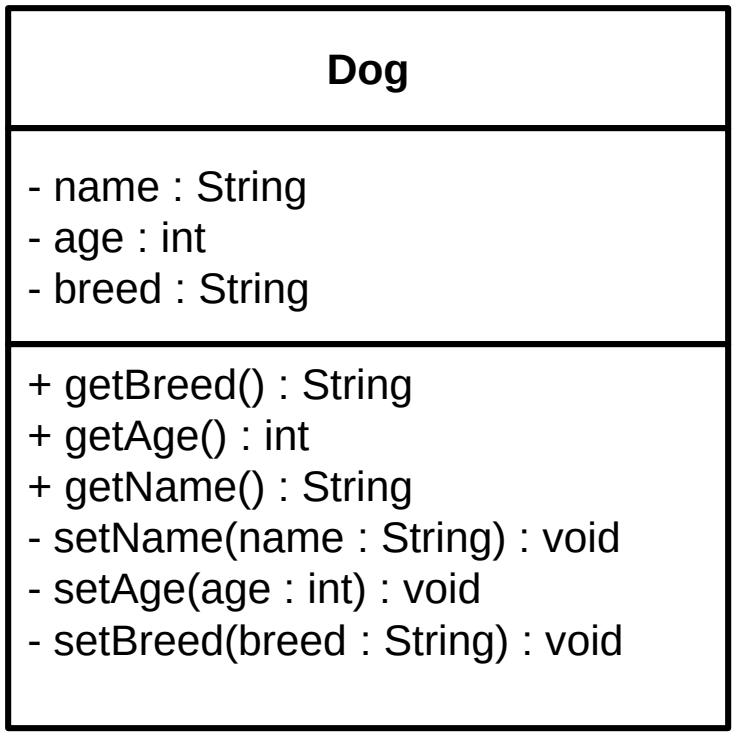


Try it

- Draw a class diagram to show the following Java scenario:

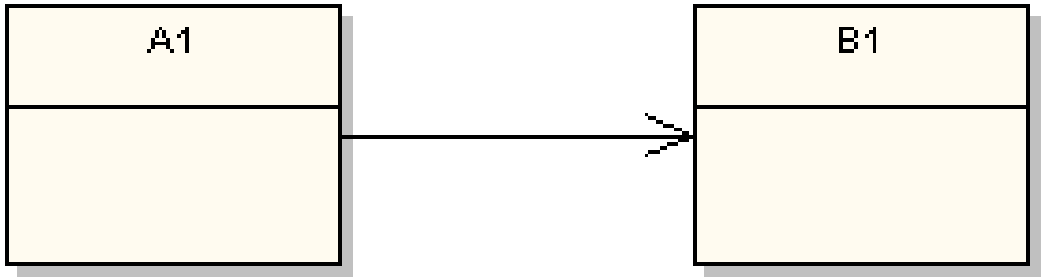
- One class called Dog
- Dog has 3 private states; breed (String), age (int) and name (string).
- Dog has 3 public behaviours: getBreed, getAge, getName. They each return their respective data type.
- Dog has 3 private behaviours: setBreed, setAge, setName. They all have void return type but take a parameter for their respective data type.

Solution


Dog
- name : String
- age : int
- breed : String
+ getBreed() : String
+ getAge() : int
+ getName() : String
- setName(name : String) : void
- setAge(age : int) : void
- setBreed(breed : String) : void

Associations

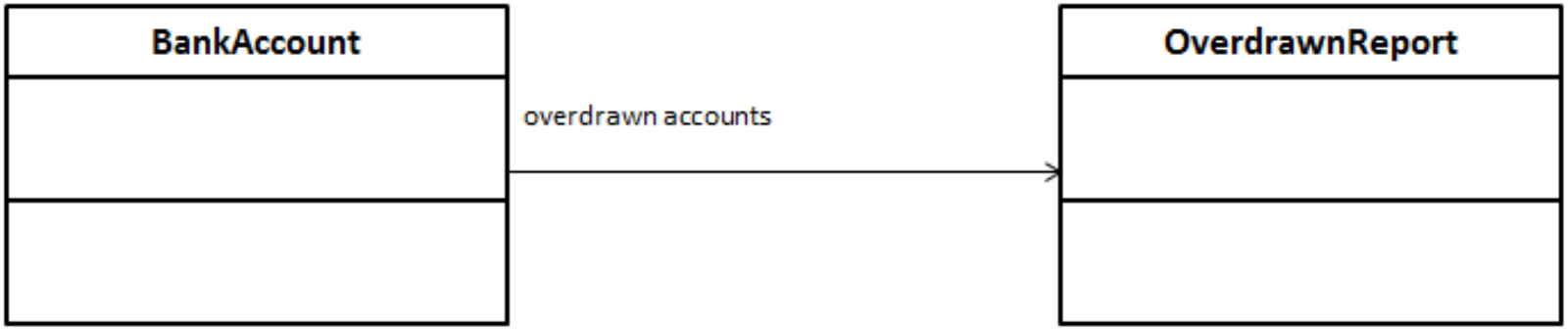
- Contain multiple classes (Driver & Providers)



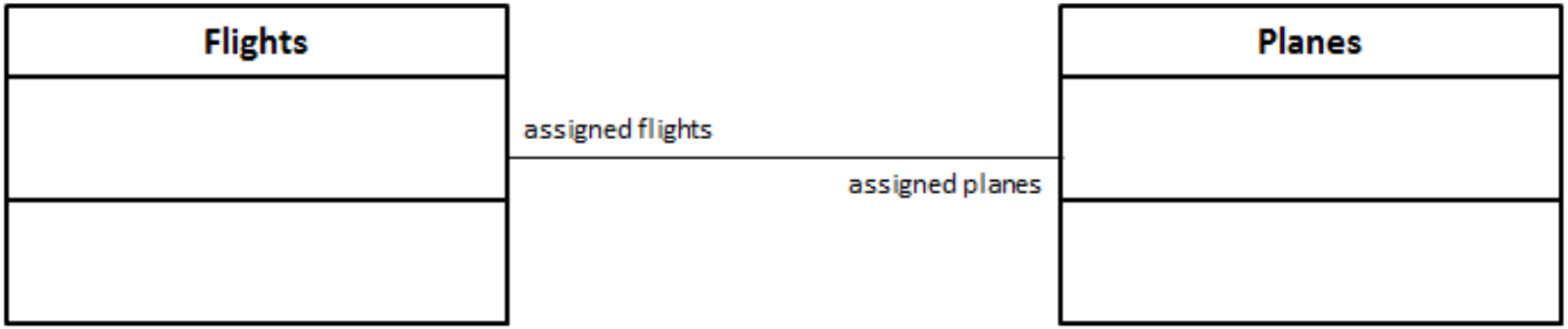
- Class A (driver) makes use of class B (provider).
- Class B does not know class A exists.

Example

Uni-directional

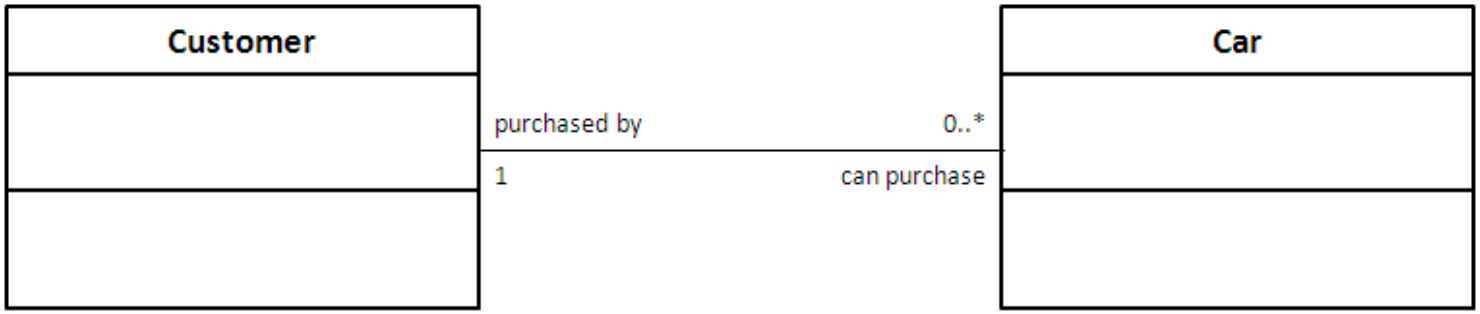


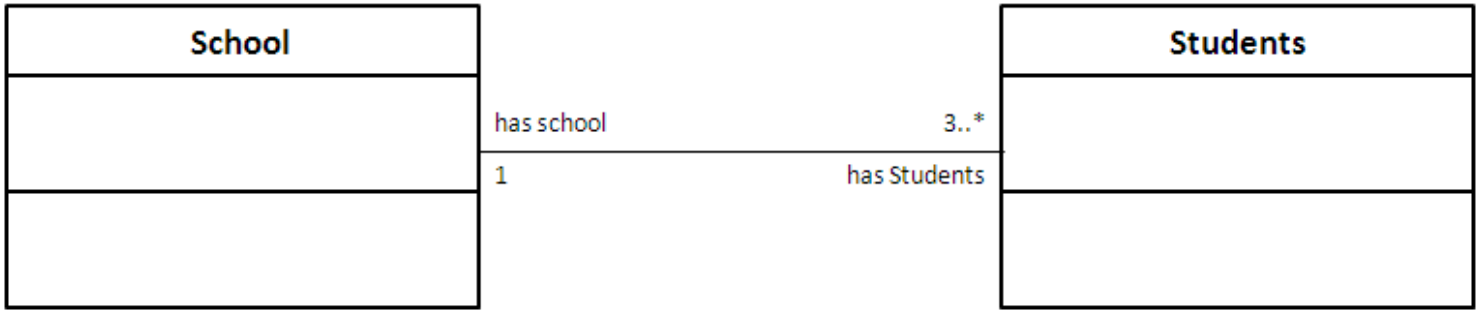
Bi-directional



Multiplicity

- Multiplicity helps define the relationship between classes that was created previously with associations.





Multiplicity

- Multiplicity follows similar patterns to that of database technology.

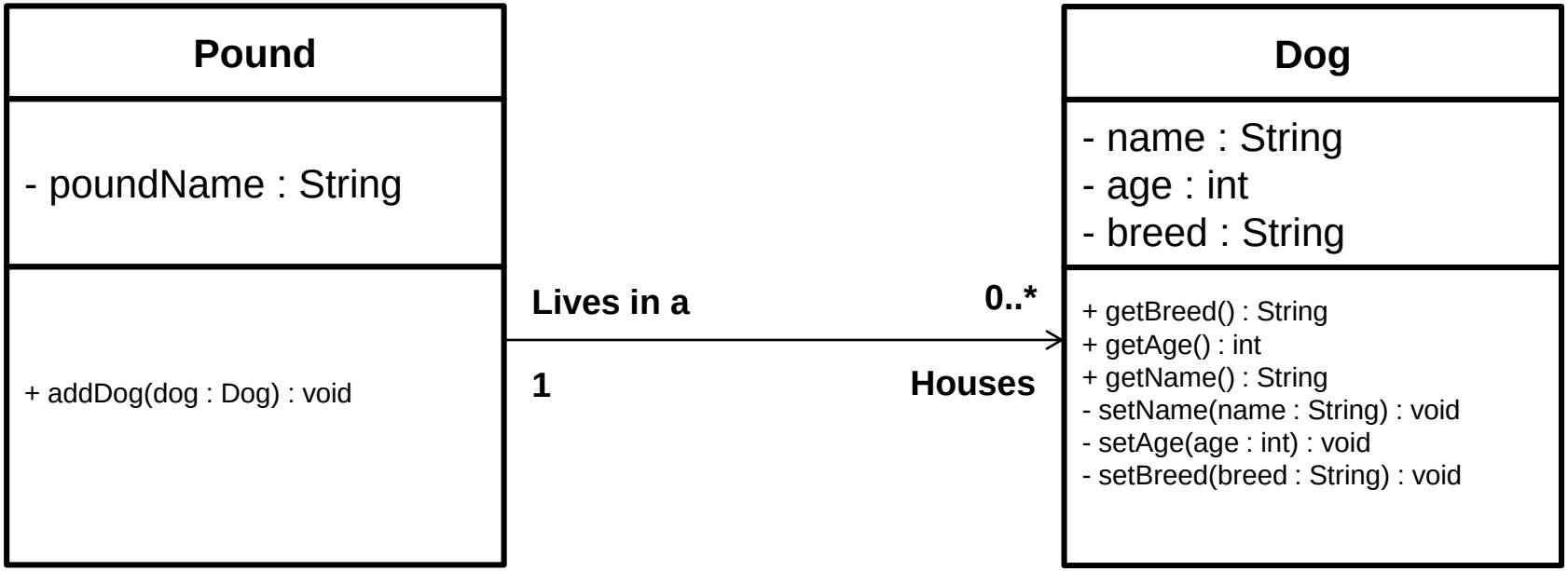
- 0
- 0..1
- 3
- 4..6
- 1..*

Try it

- Expand upon your dog class earlier:

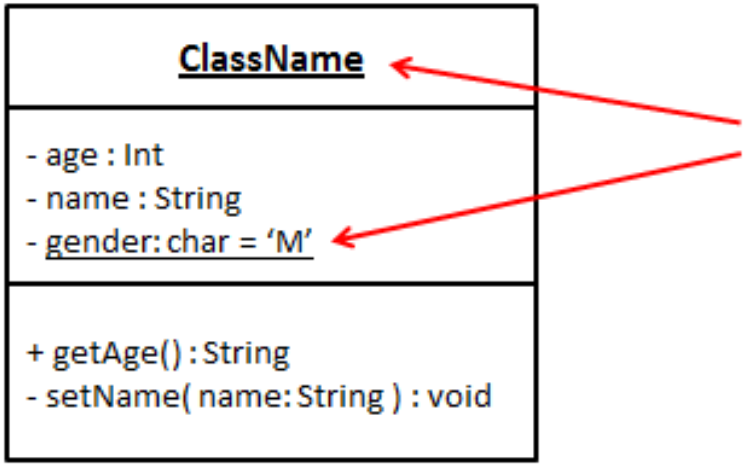
- Assume the dog class is the provider class, and the driver class is a new class called Pound. The Pound class has one private state called “poundName”, and one public behaviour called “addDog” which adds a new instance of Dog.
- Once you have drawn this new class, connect the classes together with a suitable association and multiplicity.

Solution


Pound
- poundName : String
+ addDog(dog : Dog) : void
Lives in a
1
0..*
Houses
Dog
- name : String
- age : int
- breed : String
+ getBreed() : String
+ getAge() : int
+ getName() : String
- setName(name : String) : void
- setAge(age : int) : void
- setBreed(breed : String) : void

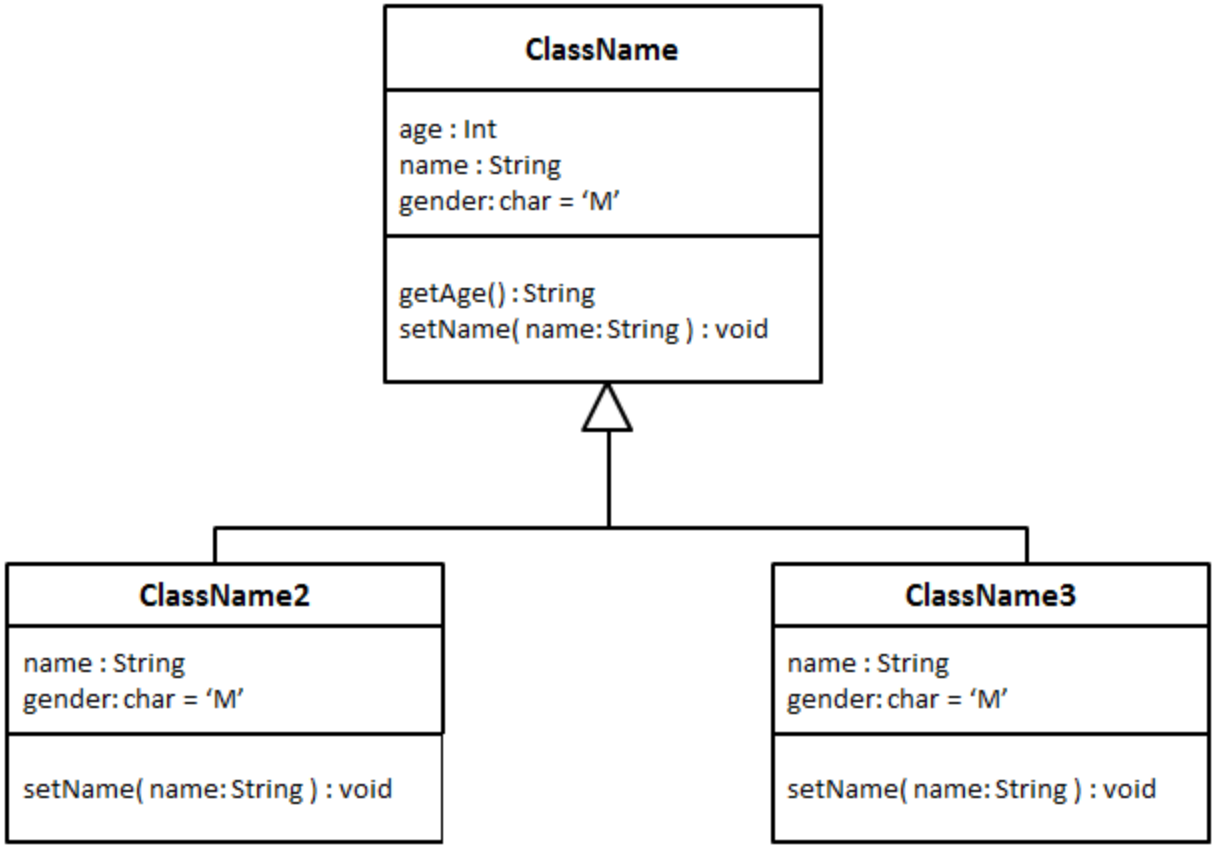
Class Diagrams

- Static



Class Diagrams

- Inheritance



Class Diagrams

- Abstraction