UNIVERSITY OF PADOVA
DEPARTMENT OF INFORMATION ENGINEERING

Biomedical Wearable Technologies
for Healthcare and Wellbeing

Dart 101 – Part 2

A.Y. 2024-2025

Giacomo Cappon

DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA

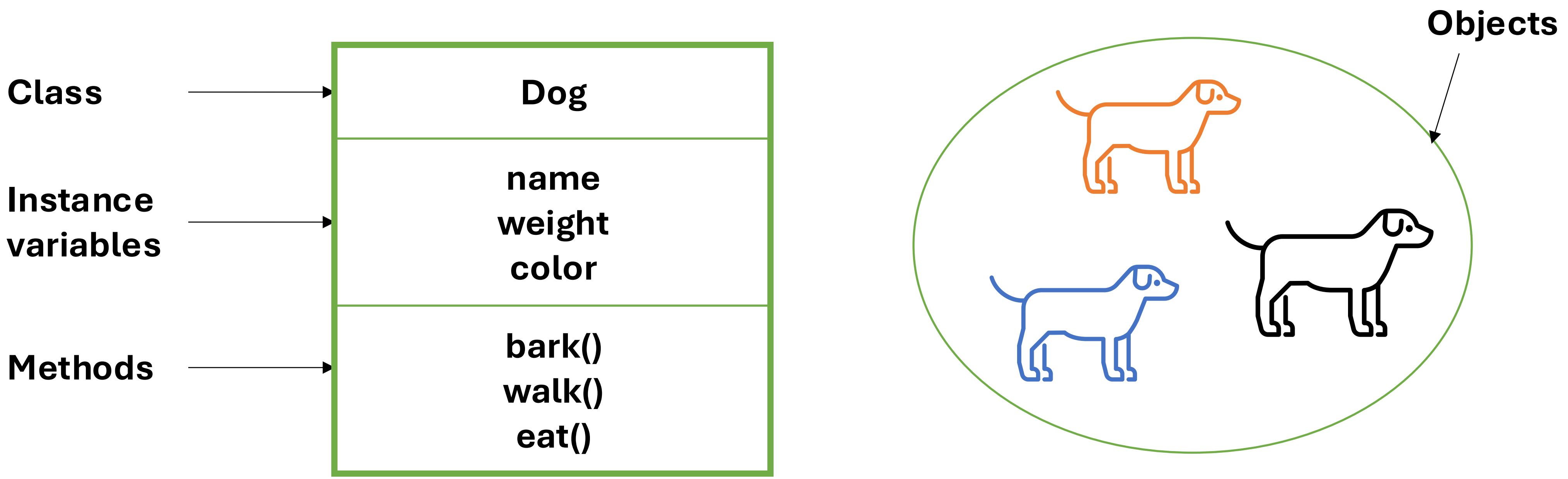
Outline

- **Goals and Principles**
- Classes and objects
- Inheritance
- Polymorphism
- Abstract data types
- Flavours of design patterns
  - Composite
  - Singleton
  - Abstract Factory
- Exercises
- Homework
- Resources

Object oriented programming

- The object-oriented programming (**OOP**) paradigm is a computer programming approach whose, as the name implies, ”main actors” are called **objects**.

- An object comes from a **class**, which is a specification of the data **fields**, also called **instance variables**, that the object contains, as well as the **methods** (operations) that the object can execute.

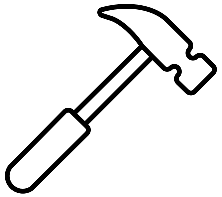


Goals of OOP

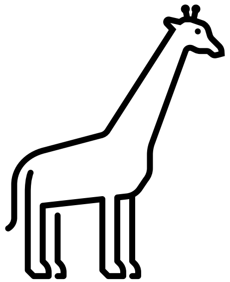
BONUS

- OOP has **three main goals**:

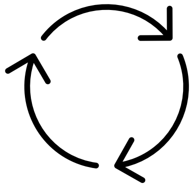
  - **Robustness**: software that produces the right output for all the anticipated inputs. In addition we want software that is able to handle unexpected inputs that are not explicitly defined for its application (e.g., a text box used to input a timestamp should be **robust** to inputs that are not timestamps and raise an error).



  - **Adaptability**: the ability of a software to adapt in response to changing conditions in its environment (e.g., updating a software to be compatible with the new OS).



  - **Reusability**: the same code should be usable as a component of different systems or software for various application (e.g., a module that allows to communicate with a smartwatch should be easily integrable in multiple apps).



Principles of OOP

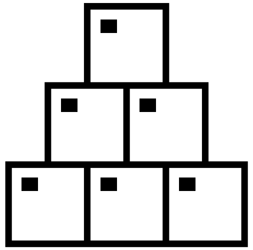
BONUS

- OOP has **three main principles**:

  - **Abstraction**: condense a complicated system down to its most fundamental parts and describe these parts and their functionalities in a simple, precise language.

  - **Encapsulation**: different components of a software system should not reveal the internal details of their respective implementations.

  - **Modularity**: organizing principle for code in which different components of a software system are divided into separate functional units.



Outline

- Goals and Principles
- **Classes and objects**
- Inheritance
- Polymorphism
- Abstract data types
- Flavours of design patterns
  - Composite
  - Singleton
  - Abstract Factory
- Exercises
- Homework
- Resources

Creating and defining a class

- Let’s analyse the basic elements of a class:

```
class Dog{

    // --- Instance variables

    // --- Constructors

    // --- Methods

}//Dog
```

Dog
name weight color
bark() walk() eat()

Creating and defining a class

- As such, the Dog class becomes something like:

```
class Dog{

    // --- Instance variables
    String? name;
    double? weight;
    String? color;

    // --- Constructors
    Dog({this.name, this.weight, this.color});
    Dog.withName({this.name});
    Dog.fuffy() : name = 'Fuffy';

    // --- Methods
    void bark(){
      print('Bark!');
    }//bark
    void walk(){
      print("I'm walking!");
    }//walk
    double eat(){
      weight = weight! + 0.1;
      print("I'm eating! I also gained 0.1 kg");
      return weight!;
    }//eat
}//Dog
```

Dog
name weight color
bark() walk() eat()

The `this` keyword

- As such, the `Dog` class becomes something like:

```
class Dog{

    // --- Instance variables
    String? name;
    double? weight;
    String? color;

    // --- Constructors
    Dog({this.name, this.weight, this.color});
    Dog.withName({this.name});
    Dog.fuffy() : name = ’Fuffy’;

    // --- Methods
    void bark(){
      print('Bark!');
    }//bark
    void walk(){
      print("I'm walking!");
    }//walk
    double eat(){
      weight = weight! + 0.1;
      print("I'm eating! I also gained 0.1 kg");
      return weight!;
    }//eat
}//Dog
```

Dog
name **weight** **color**
bark() **walk()** **eat()**

The keyword **this** is a special reference variable that referes to the current object.

Constructors

- As such, the Dog class becomes something like:

```
class Dog{

    // --- Instance variables
    String? name;
    double? weight;
    String? color;

    // --- Constructors
    Dog({this.name, this.weight, this.color});
    Dog.withName({this.name});
    Dog.fuffy() : name = 'Fuffy';

    // --- Methods
    void bark(){
      print('Bark!');
    }//bark
    void walk(){
      print("I'm walking!");
    }//walk
    double eat(){
      weight = weight! + 0.1;
      print("I'm eating! I also gained 0.1 kg");
      return weight!;
    }//eat
}//Dog
```

Dog

name
weight
color

bark()
walk()
eat()

Note: This is equivalent to write:

```
Dog(String? name, double? weight,
String? color){
    this.name = name;
    this.weight = weight;
    this.color = color;
}//Dog
```

Constructors

- As such, the Dog class becomes something like:

```
class Dog{

    // --- Instance variables
    String? name;
    double? weight;
    String? color;

    // --- Constructors
    Dog({this.name, this.weight, this.color});
    Dog.withName({this.name});
    Dog.fuffy() : name = ’Fuffy’;

    // --- Methods
    void bark(){
      print('Bark!');
    }//bark
    void walk(){
      print("I'm walking!");
    }//walk
    double eat(){
      weight = weight! + 0.1;
      print("I'm eating! I also gained 0.1 kg");
      return weight!;
    }//eat
}//Dog
```

Dog
name weight color
bark() walk() eat()

Note: It is possible to define 1 unnamed constructor, and 0 or more named constructors using the synthax:

- **Unnamed:** `ClassName(parameterList){}`
- **Named:** `ClassName.name(parameterList){}`

Constructors

- As such, the Dog class becomes something like:

```
class Dog{

    // --- Instance variables
    String? name;
    double? weight;
    String? color;

    // --- Constructors
    Dog({this.name, this.weight, this.color});
    Dog.withName({this.name});
    Dog.fuffy() : name = ’Fuffy’;

    // --- Methods
    void bark(){
      print('Bark!');
    }//bark
    void walk(){
      print("I'm walking!");
    }//walk
    double eat(){
      weight = weight! + 0.1;
      print("I'm eating! I also gained 0.1 kg");
      return weight!;
    }//eat
}//Dog
```

Note: It is possible to use the so-called ”initializer list” syntax

Dog
name **weight** **color**
bark() **walk()** **eat()**

Methods

- As such, the Dog class becomes something like:

```
class Dog{

    // --- Instance variables
    String? name;
    double? weight;
    String? color;

    // --- Constructors
    Dog({this.name, this.weight, this.color});
    Dog.withName({this.name});
    Dog.fuffy() : name = 'Fuffy';

    // --- Methods
    void bark(){
      print('Bark!');
    }//bark
    void walk(){
      print("I'm walking!");
    }//walk
    double eat(){
      weight = weight! + 0.1;
      print("I'm eating! I also gained 0.1 kg");
      return weight!;
    }//eat
}//Dog
```

Dog
name weight color
bark() walk() eat()

Note: Methods defines the behaviour of an object. Defining a method is similar to defining a function.

Full example in lab_03-dart_101_part2/dog.dart

Using a class

- Let’s see how to use a class in a simple snippet of code:

```
import 'dog.dart';

void main(List<String> args) {

  //Here, I am creating an instance of the class Dog using the
defined constructor.
  //In other words, d is an object of class Dog.
  final d = Dog(name: 'Bob', weight: 10, color: 'Black');

  //Here, I'm using a method of d
  d.bark();

  //Instance variables are accessible by default in Dart
  print('d weights ${d.weight} kg');
  print('d weights ${d.eat()} kg’);

}//main
```

Dog
name **weight** **color**
bark() **walk()** **eat()**

We can access to methods and instance variables through the **dot notation**

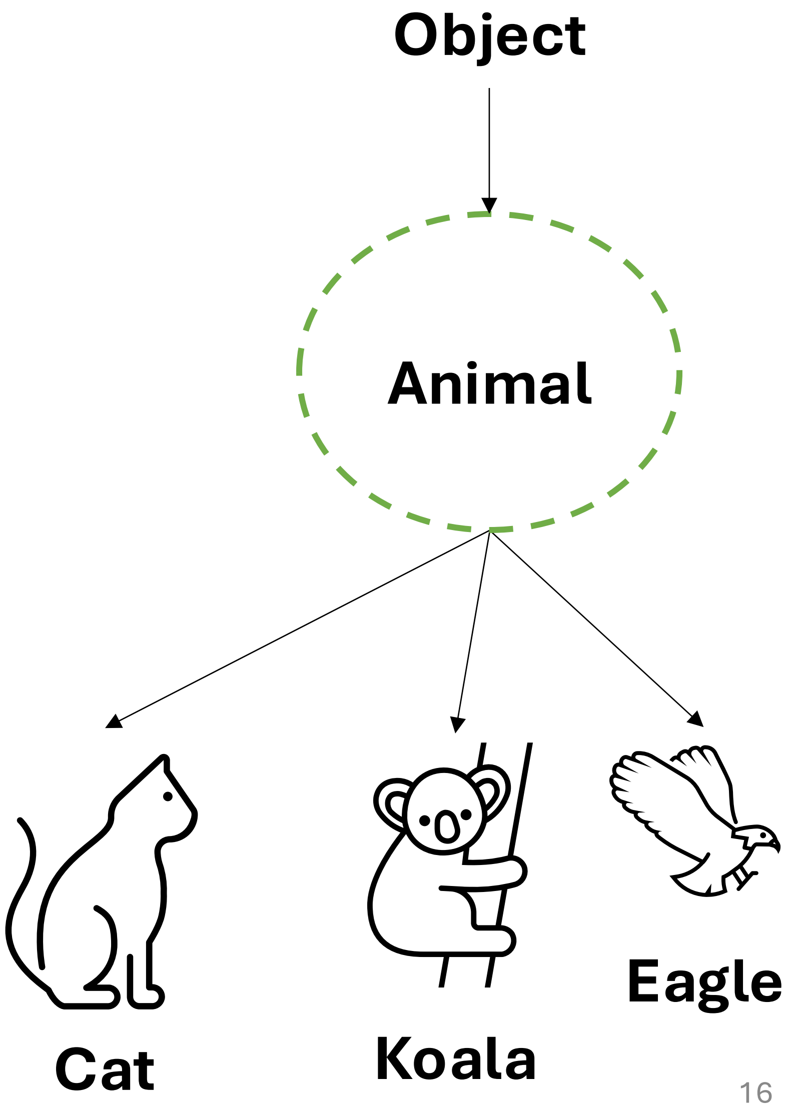
Full example in lab_03-dart_101_part2/use_dog.dart

Outline

- Goals and Principles
- Classes and objects
- **Inheritance**
- Polymorphism
- Abstract data types
- Flavours of design patterns
  - Composite
  - Singleton
  - Abstract Factory
- Exercises
- Homework
- Resources

Inheritance

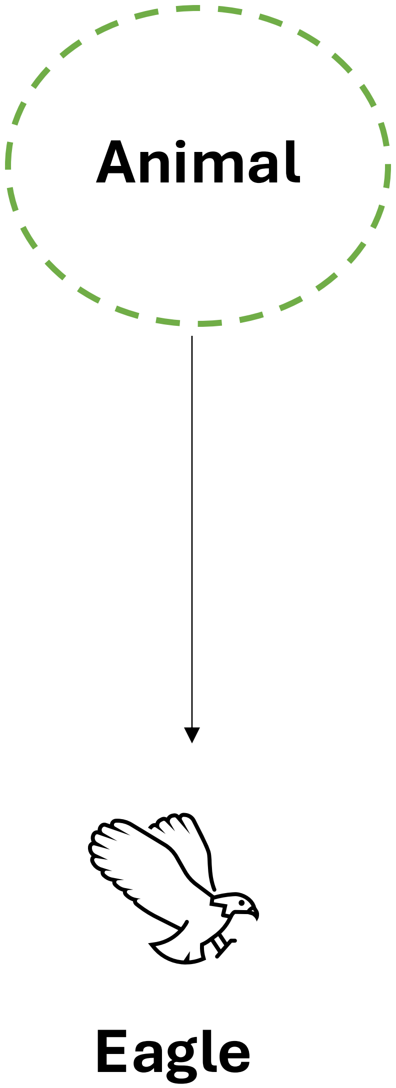
- OOP provides a modular and hierchical organizing structure for reusing code: **inheritance.**
- **Inheritance** allows to design ”general classes” (e.g., Animal) that can be specialized to more particular ones (e.g., Cat, Koala, Eagle).
- The general class is the so-called **superclass**, the specialized class is the so-called **subclass.**
- The subclass **inherits** the instance variables and methods of the superclass and **extends** it with additional instance variables and methods.
- Everything is an **`Object`** (actually **`Object?`**)



Inheritance (example)

- Let's write the superclass `Animal`:

Animal
name weight
jump() eat()

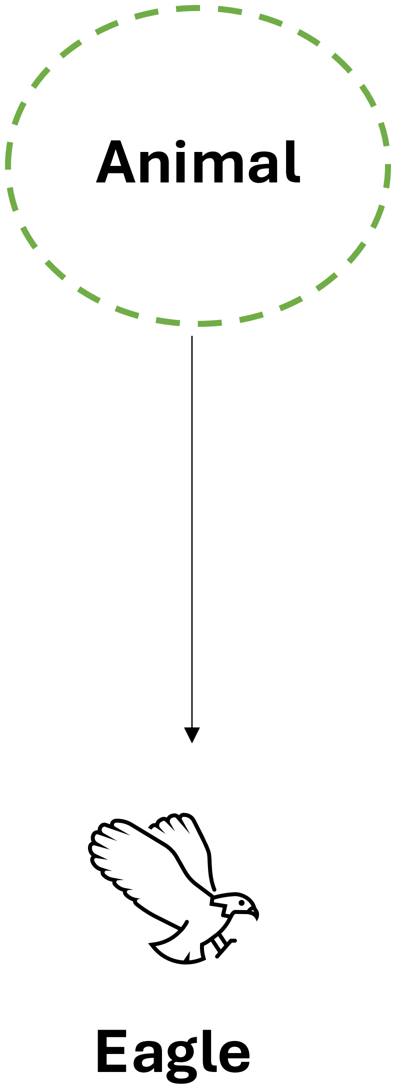


Inheritance (example)

➢ Let's write the superclass `Animal`:

```
//This is the definition of the superclass Animal
class Animal{

  // -- Instance variables
  double weight;
  String name;

  // -- Constructors
  Animal({required this.name, required this.weight});

  // -- Methods
  void jump(){
    print('Jump');
  }//jump

  @override
  String toString() {
    return '(weight: $weight, name: $name)';
  }//toString
}//Animal
```

Animal
name **weight**
jump() **eat()**

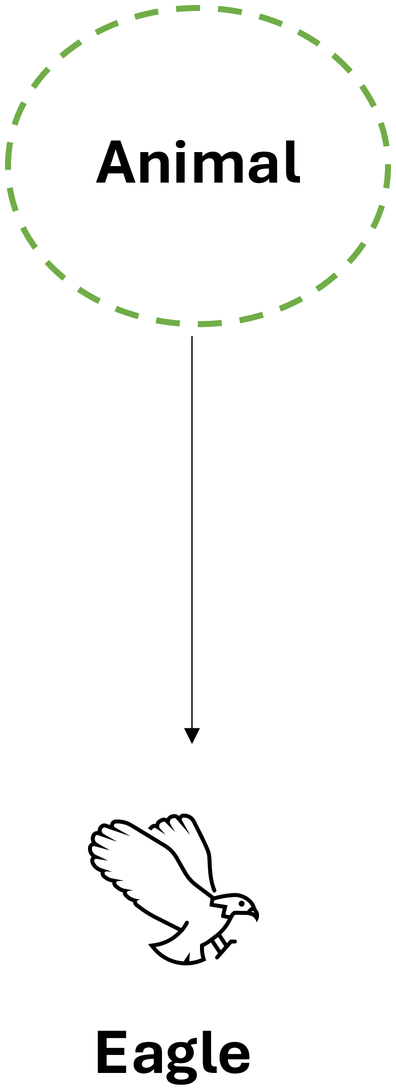


Inheritance (example)

- Let’s write the superclass `Animal`:

```
//This is the definition of the superclass Animal
class Animal{

  // -- Instance variables
  double weight;
  String name;

  // -- Constructors
  Animal({required this.name, required this.weight});

  // -- Methods
  void jump(){
    print('Jump');
  }//jump

  @override
  String toString() {
    return '(weight: $weight, name: $name)';
  }//toString
}//Animal
```

Animal
name **weight**
jump() **eat()**



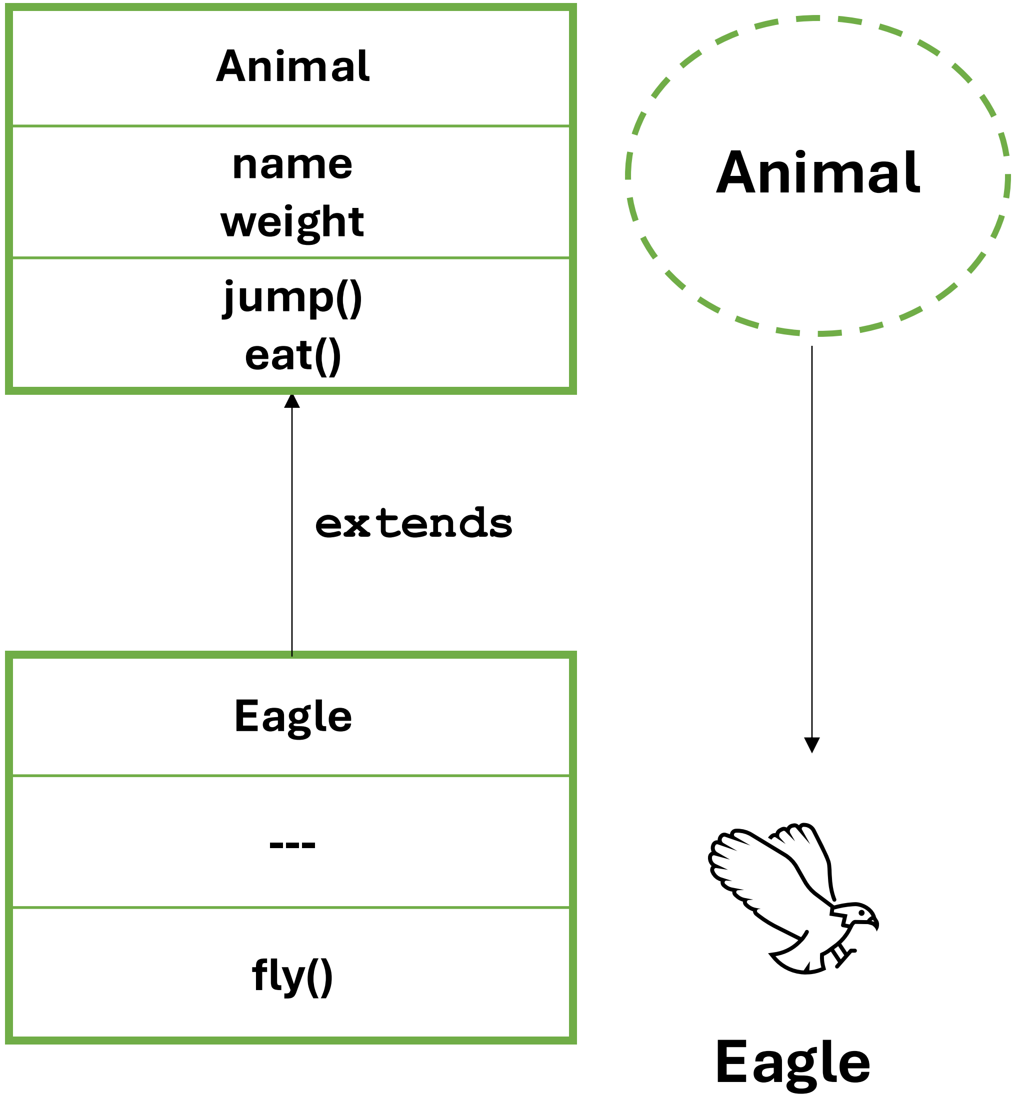
We will learn about the **@override** decorator later.

toString() is a special method that is invoked when we want to ”print” an object of the class. It provides a String representation of the object (it is the same as **__repr__** in Python)

Full example in lab_03-dart_101_part2 /animal.dart

Inheritance (example)

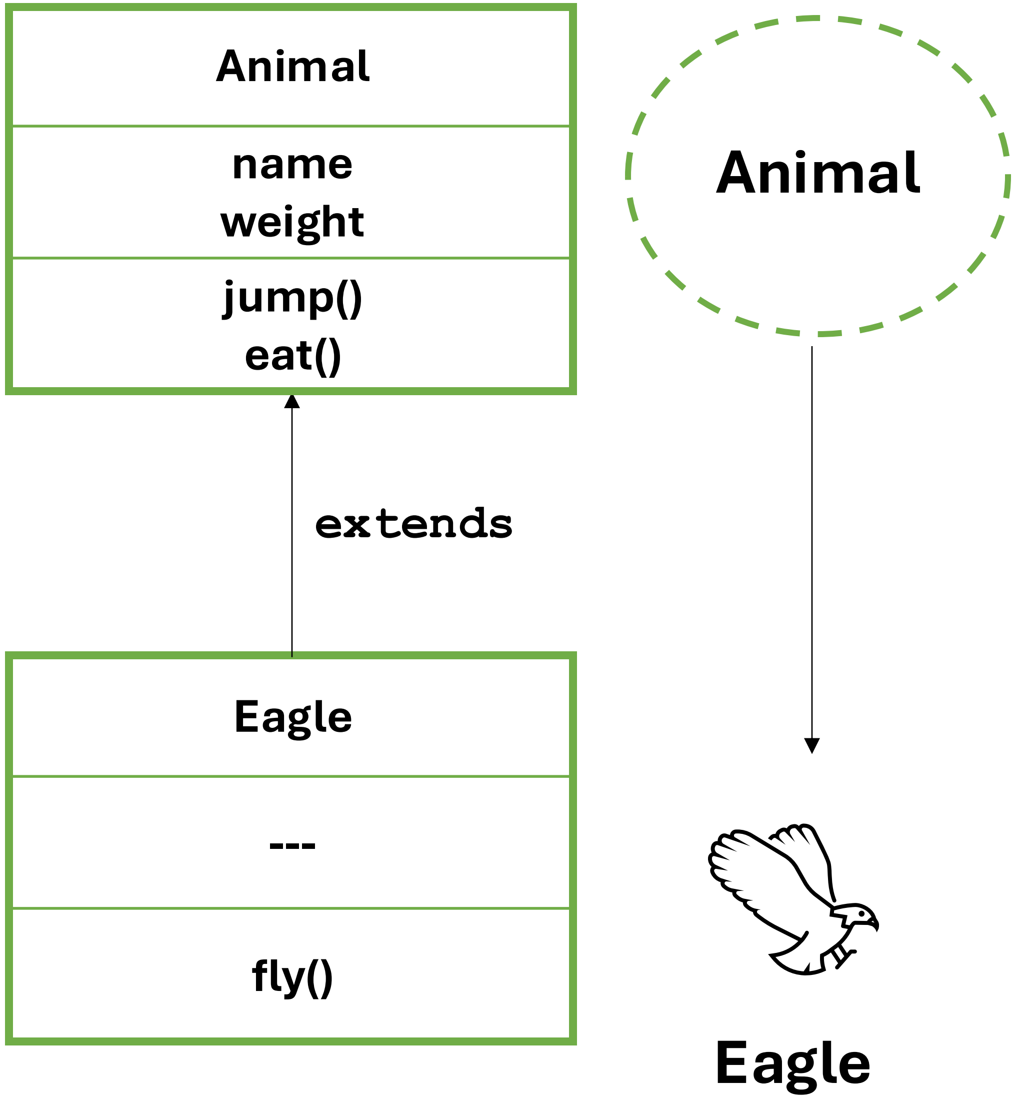
- Let's write the subclass `Eagle`:



Inheritance (example)

- Let's write the subclass `Eagle`:

```
import 'animal.dart';

//To extend a class, use the extends keyword
class Eagle extends Animal{

  // -- Constructors
  Eagle({required name, required weight}) : super(name :
name, weight: weight);

  void fly(){
    print('Fly');
  }//fly

  @override
  String toString() {
    return super.toString();
  }//toString
}//Eagle
```



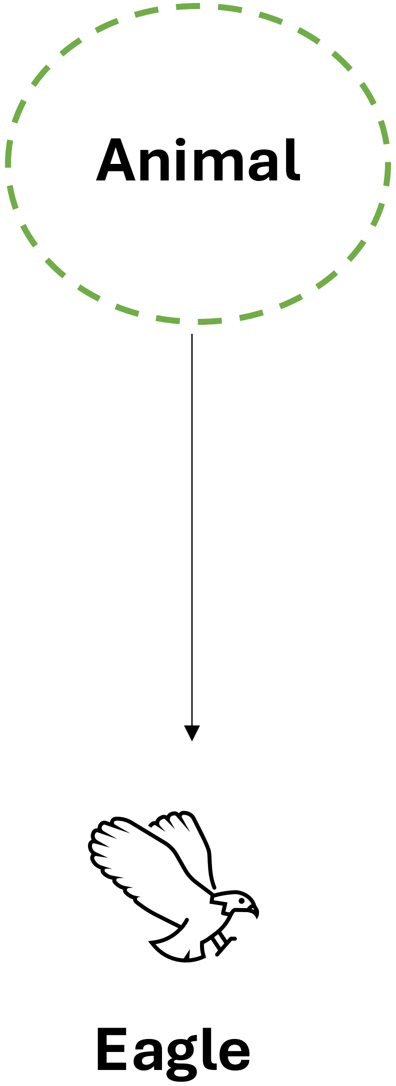
Full example in lab_03-dart_101_part2/eagle.dart

The `extends` keyword

- Let’s write the subclass `Eagle`:

```
import 'animal.dart';

//To extend a class, use the extends keyword
class Eagle extends Animal{

  // -- Constructors
  Eagle({required name, required weight}) : super(name :
name, weight: weight);

  void fly(){
    print('Fly');
  }//fly

  @override
  String toString() {
    return super.toString();
  }//toString
}//Eagle
```

The keyword **extends** specifies the superclass we are extending and thus we are inheriting from.



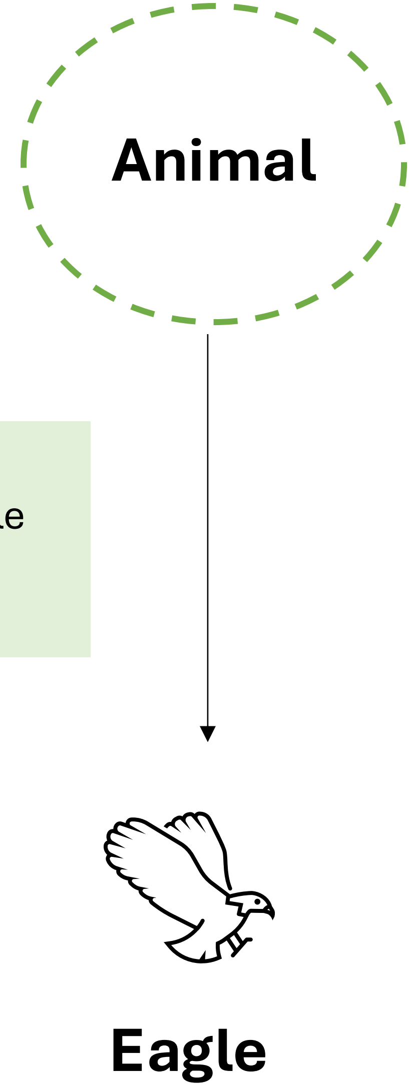
Full example in lab_03-dart_101_part2/eagle.dart

The super keyword

- Let's write the subclass Eagle:

```
import 'animal.dart';

//To extend a class, use the extends keyword
class Eagle extends Animal{

  // -- Constructors
  Eagle({required name, required weight}) : super(name :
name, weight: weight);

  void fly(){
    print('Fly');
  }//fly

  @override
  String toString() {
    return super.toString();
  }//toString
}//Eagle
```

The keyword **super** is a special reference variable that refers to the superclass.

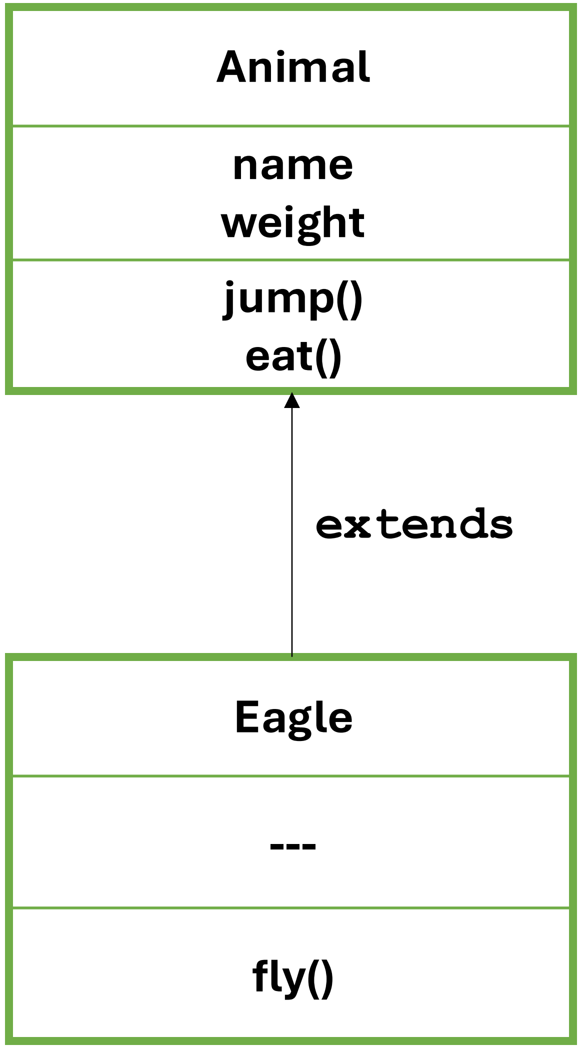


Full example in lab_03-dart_101_part2/eagle.dart

Using inheritance

- Let’s see how to use inheritance in a simple snippet of code:

```
import 'animal.dart';
import 'eagle.dart';
void main(List<String> args) {

  //Create an Animal
  Animal animal = Animal(name: 'Bob', weight: 10.0);
  //Use its methods
  animal.jump(); //this will print 'Jump'
  //Use its instance variables
  animal.weight = 30;
  //print it (it will use toString())
  print(animal); //This will print '(weight: 30.0, name: Bob)'

  //Create an Eagle
  Eagle eagle = Eagle(name: 'Jim', weight: 20.0);
  //Use its methods (inherited and its own)
  eagle.jump(); //this will print 'Jump'
  eagle.fly(); //this will print 'Fly'

  //Use its instance variables (inherited)
  eagle.name = 'Carl';

  //print it (it will use the inherited toString())
  print(eagle); //This will print '(weight: 20.0, name: Carl)'

}//main
```



Full example in lab_03-dart_101_part2/use_animal_and_eagle.dart

BONUS

How's that possible? Dynamic dispatch

- This is possible thanks to the **dynamic dispatch** mechanism.
- When a program wishes to invoke a method `met()` of some object `obj` of class `C`, i.e., `obj.met()`, the run-time:
  - Examines the class `C` checking if it implements a method `met()`
  - If it does, the method `met()` of class `C` is executed
  - If it does not, the run-time examines the superclass of `C` (let's say `S`)
  - If `S` defines a method `met()`, the method `met()` of class `S` is executed
  - Otherwise, the run-time checks the superclass of `S` (let's say `T`) and so on...

...

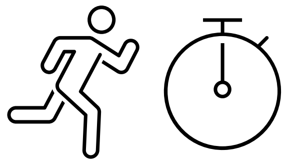
`T`, do you implement `met()`? → **T**

No! Ask to my superclass `T`.

`S`, do you implement `met()`? → **S**

No! Ask to my superclass `S`.

Run-time



→ `C`, do you implement `met()`? → **C**

Yes!

Execute `met()`

I'm here →

Dart

```
...
obj.met()
...
```

Outline

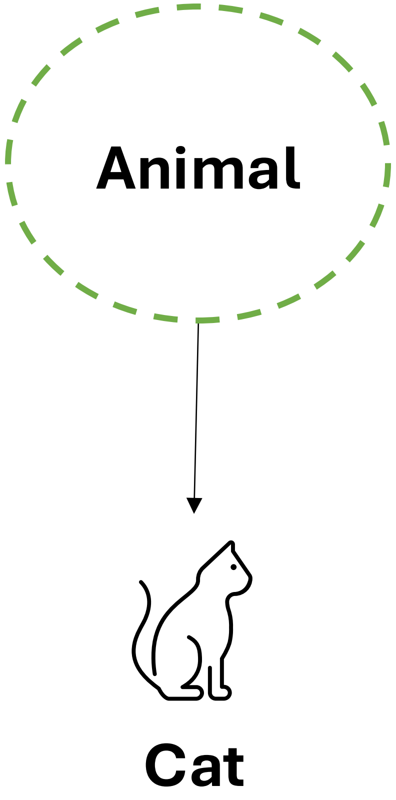
- Goals and Principles
- Classes and objects
- Inheritance
- **Polymorphism**
- Abstract data types
- Flavours of design patterns
  - Composite
  - Singleton
  - Abstract Factory
- Exercises
- Homework
- Resources

Polymorphism

- Inheritance and dynamic dispatch unlock the concept of **polymorphism**: the ability of an object of taking many forms (or types).
- A `Cat` is an `Animal`, as such the following is fine:

```
Animal cat = Cat(...);
```

- An object of type “superclass” can refer to both a type ”superclass” but also to a type “subclass”!



Polymorphism – Method invocation and override

- Moreover, the method invocation chain will always start from the most restrictive class that applies.

- Let's say that both `Animal` and `Cat` define a method `eat()`:

```
Animal cat1 = Cat(...);

Cat cat2 = Cat(...);
```

This will use the `eat()` method of `Cat`

Again, this will use the `eat()` method of `Cat`

- Technically speaking, `Cat` is **overriding** the method `eat()` from `Animal`

Note: In the previous example, you saw that class `Eagle` overrode `toString()` from `Animal` using the **`@override`** decorator.

Polymorphism – Casting

- How polymorphism affect type casting?

- **Widening casts** are always ok and there is no need of an explicit task, e.g.:

```
Cat cat = Cat(...);
Animal animal = cat;
```

Note: in this example `cat` and `animal` will refer to the same object!

- On the other hand, **narrowing casts** must be explicit and can fail, e.g.:

```
Animal animal1 = Cat(...);
Animal animal2 = Koala(...);
Cat c1 = animal1 as Cat; //This is ok
Cat c2 = animal2 as Cat; //This will fail
```

The **`as`** keyword is used to cast an object to another type

Full example in lab_03-dart_101_part2/polymorphism.dart

Outline

- Goals and Principles
- Classes and objects
- Inheritance
- Polymorphism
- **Abstract data types**
- Flavours of design patterns
  - Composite
  - Singleton
  - Abstract Factory
- Exercises
- Homework
- Resources

Abstract data types

- Finally, let's talk about **abstract data types** (ADT).

- ADTs allow to enforce the abstraction principle of OOP.

- An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them and the types of parameters of the operations.

- An ADT specifies **what** each operation does, but not **how** the operation is performed.

Interfaces

- An ADT can be specified via an **interface**, that defines the collection of methods and instance variables supported by that class (i.e., what, not how).
  - For example, this diagram expresses the interface of a Drinkable thing, i.e., a drink (we are saying that a Drinkable can be poured)

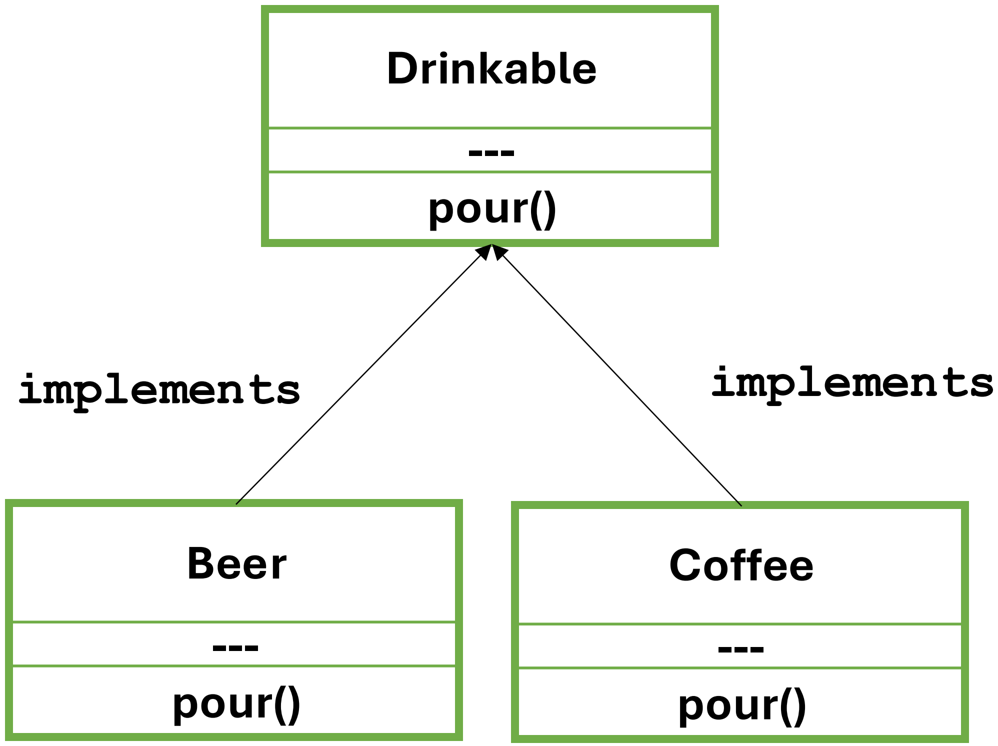
An interface cannot be instantiated since it provides only what each operation does, not how.

Interfaces

- Then, we can use an interface to force a new classes to conform to the behavior and characteristics defined by the interface itself.

- A class that **implements** an interface **must provide how** the methods expressed by the interface behave.

- In the example, Beer and Coffee **are implementing** Drinkable. As such, the programmer must define a body for the pour() methods of both Beer and Coffee



Abstract classes

- An **abstract class** is something in between an interface and a concrete class.

- **Like an interface**, an abstract class cannot be instantiated, that is, no objects can be created from an abstract class.

- **Like a concrete class**, an abstract class can extend another abstract class, and can contain concrete method body definitions.

Interfaces and abstract classes in Dart

- **In Dart, ADTs are defined via abstract classes only**. To create an interface, simply define an abstract class with no concrete method definition.

- How to define an interface? Here's an example:

```
abstract class Drinkable {
  void pour();
}//Drinkable
```

The **`abstract`** keyword is used to say that the class defines an ADT (interface or abstract class)

- How to define an abstract class?

```
abstract class Drinkable {
  void pour();
  bool isLiquid() => true;
}//Drinkable
```

Practically very similar, they just differ by the fact that the former contains a concrete implementation.

From now on, we will refer to interfaces as abstract classes without loss of generality

The `implements` keyword

- How to `implement` an abstract class? Here's an example:

```
abstract class Drinkable {
  void pour();
}//Drinkable
```

`implements`

```
class Beer implements Drinkable {
  void pour() {
    print("Pouring a beer!");
  }//pour
}//Beer
```

The **`implements`** keyword is used to say that the class is implementing another abstract class

Using abstract classes in Dart

- We can use the same approach used for superclasses and subclasses:

```
void main(List<String> args) {
  //Same as superclasses and subclasses
  Drinkable drink = Beer();
  //This will print 'Pouring a beer!'
  drink.pour();
}//main
```


Full example in lab_03-dart_101_part2/abstract_data_types.dart

Outline

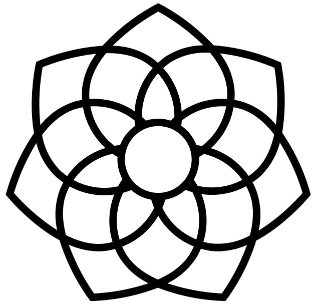
BONUS

- Goals and Principles
- Classes and objects
- Inheritance
- Polymorphism
- Abstract data types
- **Flavours of design patterns**
  - Composite
  - Singleton
  - Abstract Factory
- Exercises
- Homework
- Resources

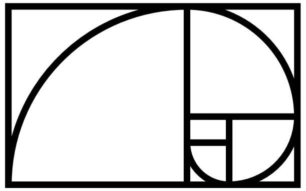
Design Patterns

BONUS

- A **Design Patterns** ”describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” – Christopher Alexander



- In few words: **they are efficient design solutions to recurring problems**.



- Design patterns are 23 (as identified by the Gang of Four*) and allow to build better software leveraging the principles of OOP.

- Here, I will present you 3 popular design patterns.

*The Gang of Four – Design Patterns (see resources)

Outline

BONUS

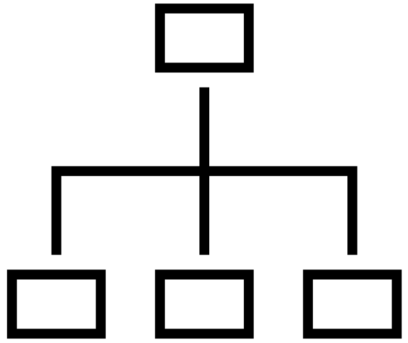
- Goals and Principles
- Classes and objects
- Inheritance
- Polymorphism
- Abstract data types
- **Flavours of design patterns**
  - **Composite**
  - Singleton
  - Abstract Factory
- Exercises
- Homework
- Resources

Composite

BONUS

- The **Composite** design pattern “composes objects into tree structures to represent part-whole hierarachies. Composite lets clients treats individual objects and compositions of objects uniformly” - GoF.

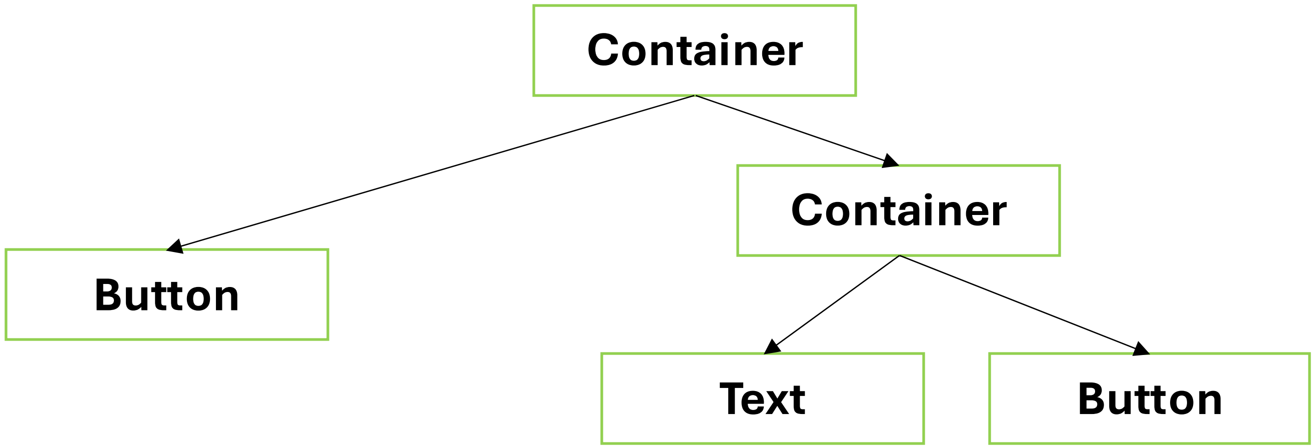
- The Composite design pattern lets users build complex diagrams out of simple components.

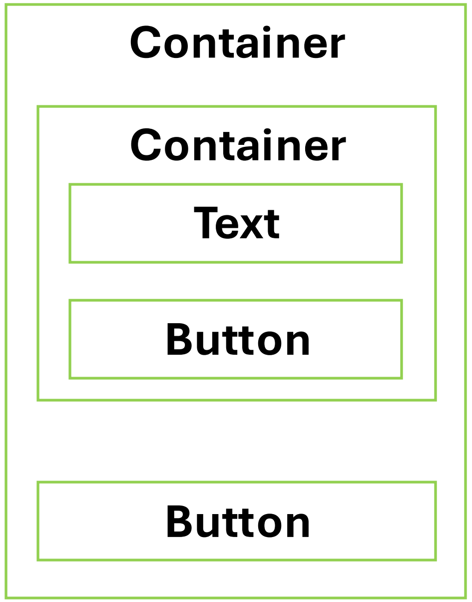


Composite (Example)

BONUS

- Let's think about a screen of your app. It is made of multiple **graphical** components, each of which can be composed by other graphical components:

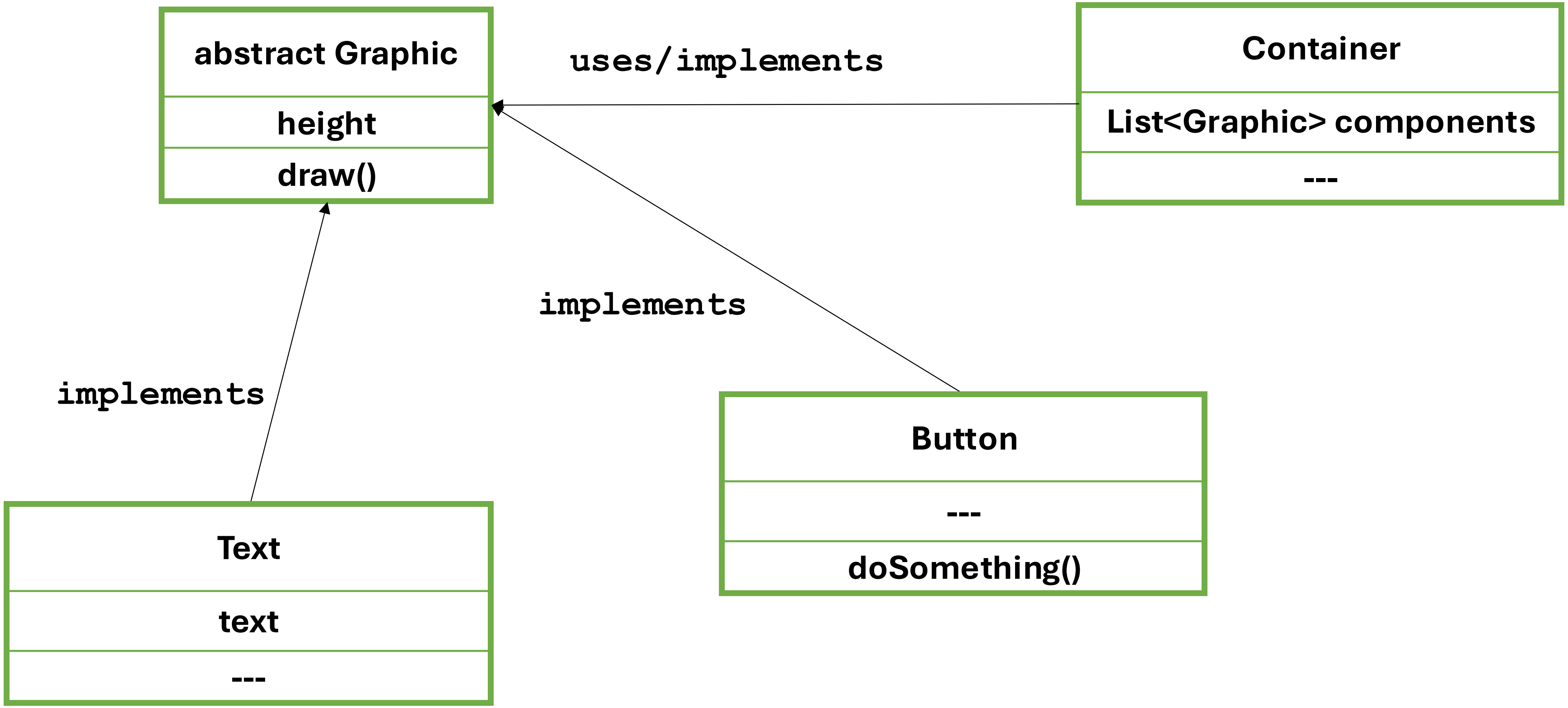




- We can think of using the composite pattern to implement this screen!

Composite (Example)

BONUS



Composite (Example in Dart)

BONUS

```dart
abstract class Graphic{
  final double height;
  Graphic({required this.height});
  void draw();
}//Graphic
```

uses/implements

```dart
class Container implements Graphic{
  final List<Graphic> components;
  @override
  final double height;
  Container({required this.components, required this.height});

  void draw(){
    print('Drawing the container, height: $height...');
    for (var item in components) {
      item.draw();
    }//for
    print('Done!');
  }//draw
}//Container
```

implements

implements

```dart
class Text implements Graphic{
  final String text;
  @override
  final double height;
  Text({required this.text, required this.height});

  void draw(){
    print('Drawing the text: \'$text\', height: $height!');
  }//draw

}//Text
```

```dart
class Button implements Graphic{
  @override
  final double height;
  Button({required this.height});

  void draw(){
    print('Drawing the button, height: $height!');
  }//draw
  void doSomething(){
    print('Do something when clicked!');
  }//doSomething

}//Button
```

Using Composite (Example in Dart)

BONUS

```
void main(List<String> args) {

  //Let's compose the box
  List<Graphic> boxComponents = [
    Text(text: 'Hello', height: 100),
    Button(height: 50),
  ];
  Graphic box = Container(components: boxComponents, height: 500);

  //Then, let's compose the whole screen
  List<Graphic> screenComponents = [
    box,
    Button(height: 150),
  ];
  Graphic screen = Container(components: screenComponents, height: 1000);

  //Finally, let's draw the screen
  screen.draw();

}//main
```

```
Drawing the container, height: 1000.0...
Drawing the container, height: 500.0...
Drawing the text: 'Hello', height: 100.0!
Drawing the button, height: 50.0!
Done!
Drawing the button, height: 150.0!
Done!
```


Full example in lab_03-dart_101_part2/composite_example.dart

Composite

BONUS

- Consequences:

  - **It is easy to build complex structures out of simple components**: by definition.
  - **Makes the client simple**: it avoids having to write specific code for the different kind of components.
  - **It is easier to add new kind of components**: new kind of components will work automatically with the existing code.
  - **Can make your design overly general**: sometimes you want your composite to have only certain components. With this pattern this is difficult since all is ”general.

Note: Flutter uses the Composite pattern to build the UI

Outline

BONUS

- Goals and Principles
- Classes and objects
- Inheritance
- Polymorphism
- Abstract data types
- **Flavours of design patterns**
  - Composite
  - **Singleton**
  - Abstract Factory
- Exercises
- Homework
- Resources

Singleton

BONUS

- The **Singleton** design pattern “ensure a class only has one instance, and provide a global point of access to it” - GoF.

- As such, it can be used when there must be exactly one instance of a class.

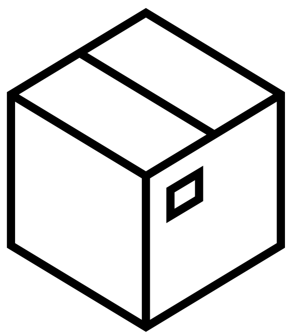
- We also have to provide to clients a clear access point to that instance.

Singleton (Example)

BONUS

- Let’s think about the storage manager of your phone.

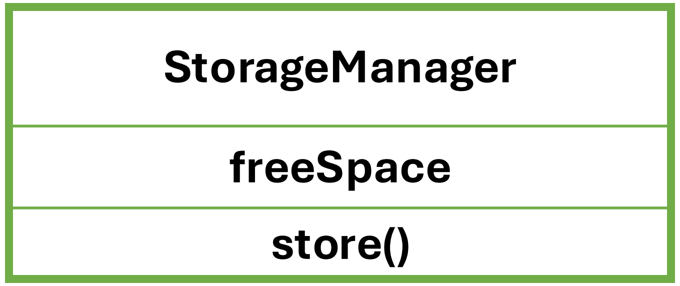
- There is only and only one storage manager through the all application and the access point to it should be clear:



- The singleton pattern is the right way to implement it

Singleton (Example)

BONUS



Singleton (Example in Dart)

BONUS

```
class StorageManager {
  static final StorageManager _singleton = StorageManager._internal();
  static final double _totalSpace = 2048;
  static double _spaceOccupied = 0;

  factory StorageManager() {
    return _singleton;
  }//StorageManager

  static double get freeSpace => _totalSpace - _spaceOccupied;

  void store(double space){
    if((freeSpace - space) < 0){
      print('Not enough space!');
    }//if
    else{
      _spaceOccupied += space;
      print('Stored $space MB');
    }//else
  }//space
  @override
  String toString() => "Space: $_spaceOccupied (of $_totalSpace) MB";

  StorageManager._internal();
}//StorageManager
```

Singleton (Example in Dart)

BONUS

```
class StorageManager {
  static final StorageManager _singleton = StorageManager._internal();
  static final double _totalSpace = 2048;
  static double _spaceOccupied = 0;

  factory StorageManager() {
    return _singleton;
  }//StorageManager

  static double get freeSpace => _totalSpace - _spaceOccupied;

  void store(double space){
    if((freeSpace - space) < 0){
      print('Not enough space!');
    }//if
    else{
      _spaceOccupied += space;
      print('Stored $space MB');
    }//else
  }//space
  @override
  String toString() => "Space: $_spaceOccupied (of $_totalSpace) MB";

  StorageManager._internal();
}//StorageManager
```

The _ before the name of a variable marks it as private (only accessible inside the class)

The **static** keyword marks a static variable or method, i.e., variable or method proper of the class (not of an instance object).

The **factory** keyword marks a constructor that does not always return a new instance of the class

The **get** keyword defines a getter: special methods that provide read acces to a variable.

Using Singleton (Example in Dart)

BONUS

```
void main(List<String> args) {

  //Here, I'm getting an instance of StorageManager
  StorageManager sm = StorageManager();

  //Let's use it
  sm.store(1000);
  print(sm); // This will print 'Space: 1000.0 (0f 2048) MB'
  sm.store(500);
  print(sm); // This will print 'Space: 1500.0 (0f 2048) MB'

  //Let's get a new instance of StorageManager
  StorageManager smBis = StorageManager();

  //This will fail since the Singleton pattern is ensuring that smBis is the same as sm
  //thus the storage has not enough space.
  smBis.store(1000);
  print(smBis); // This will print 'Space: 1500.0 (0f 2048) MB'

}//main
```


Full example in lab_03-dart_101_part2/singleton_example.dart

Singleton

BONUS

- Consequences:

  - **Easy to control the access to the sole instance**: by definition.

  - **Reduced name space**: singleton is an improvement with respect to global variables and avoid “polluting” the name space.

  - **Permits a variable number of instances**: it is easy to adapt the pattern and define an exact number of instances allowed.

Outline

BONUS

- Goals and Principles
- Classes and objects
- Inheritance
- Polymorphism
- Abstract data types
- **Flavours of design patterns**
  - Composite
  - Singleton
  - **Abstract Factory**
- Exercises
- Homework
- Resources

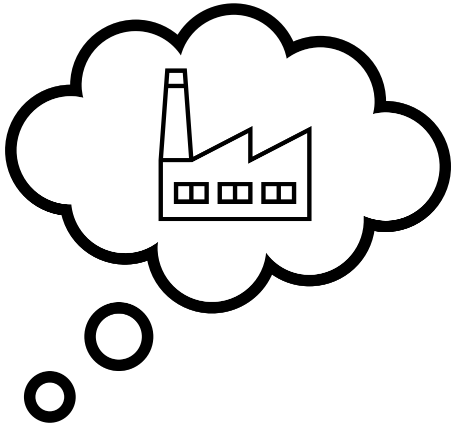
Abstract Factory

BONUS

- The **Abstract Factory** design pattern “provides an interface for creating families of related or dependent objects without specifying their concrete classes” - GoF.

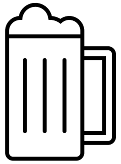
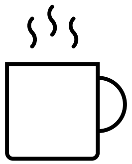
- The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.

- The abstract factory produces abstract products.



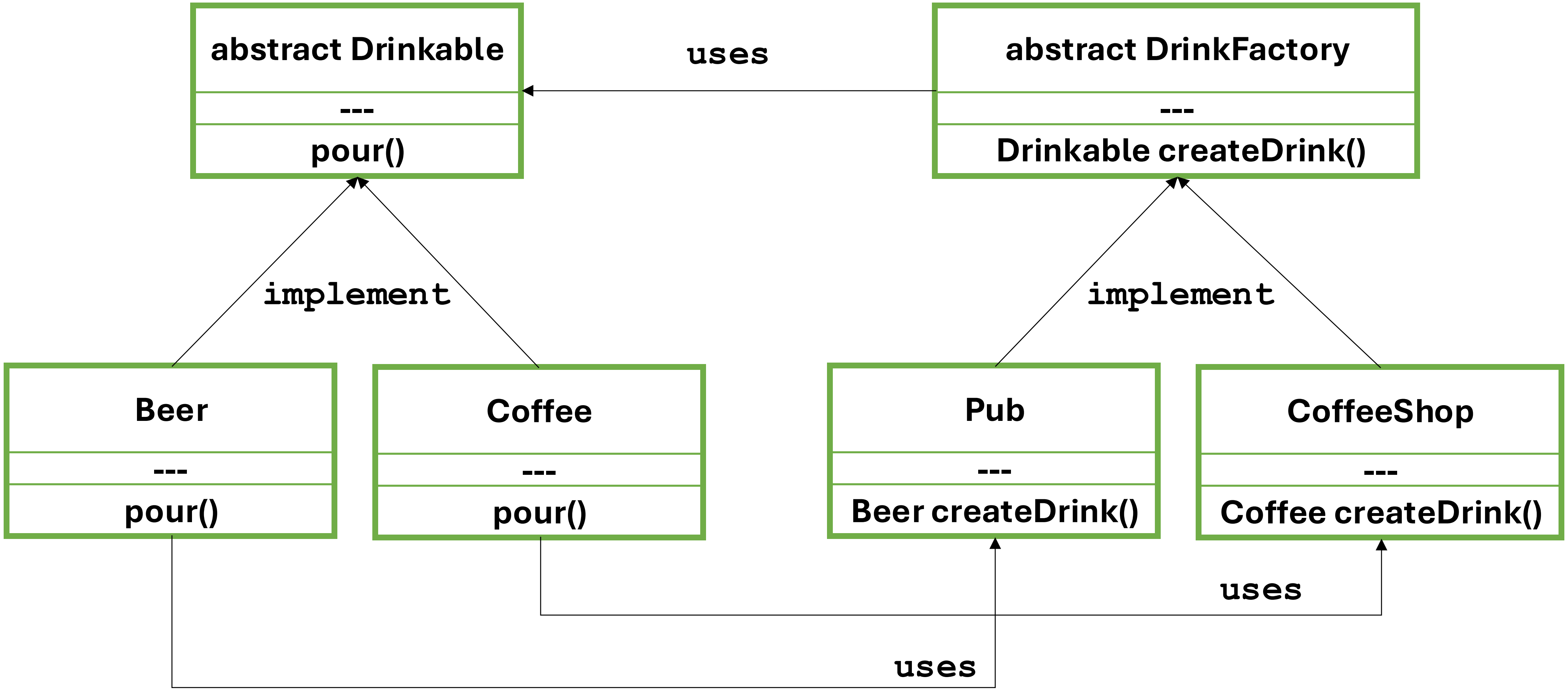
Abstract Factory (Example)

BONUS

- Let's think about a bar. A bar is, the-facto, a "factory" of drinks which are the products. Bars can be of different types:
  - Pub: a factory of beers 
  - Coffee shop: a factory of coffees 
- We can think of using the abstract factory pattern to implement this scenario:
  - Pub and coffee shop will be type of the abstract "drink factory" factories
  - Beer and coffee will be type of the abstract "drinkable" products

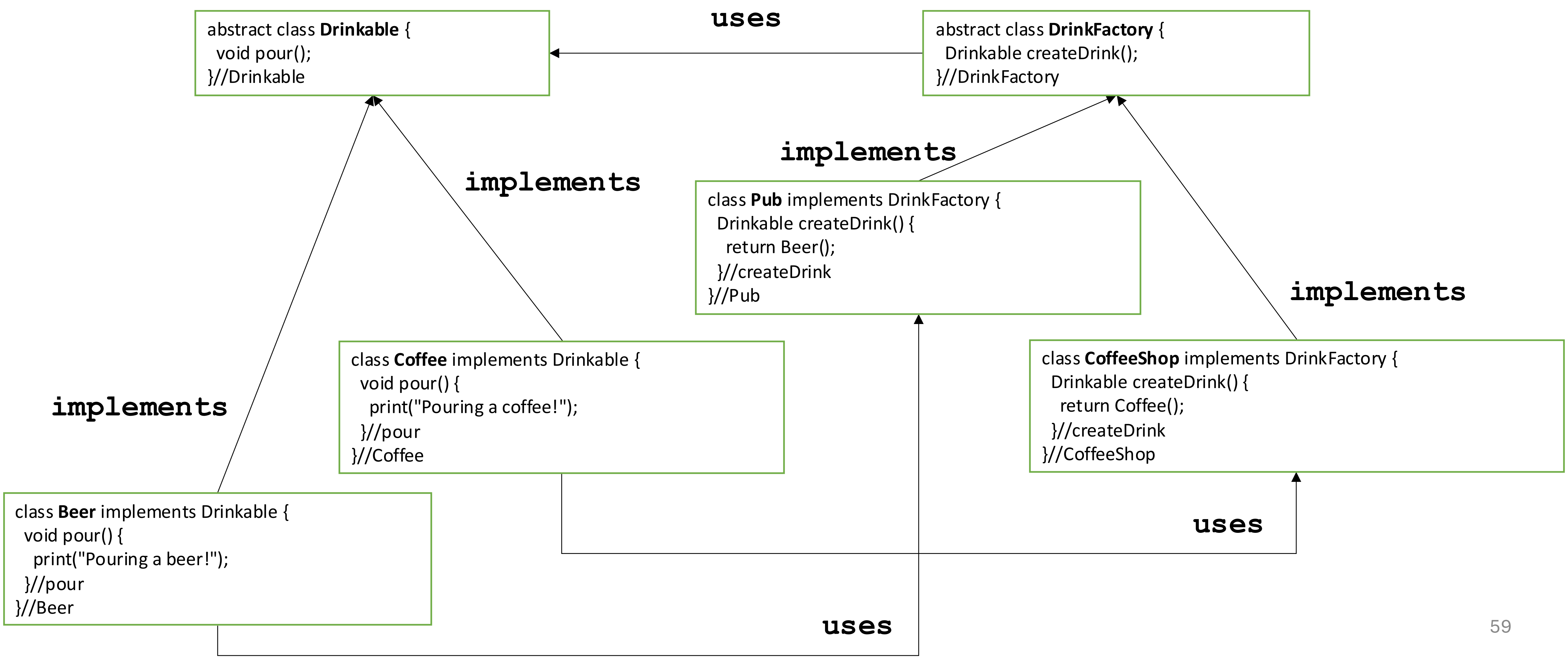
Abstract Factory (Example)

BONUS



Abstract Factory (Example in Dart)

BONUS



Using Abstract Factory (Example in Dart)

BONUS

```
void main() {
  var mood = "sleepy";

  //Here we are instantiating the abstract factory
  DrinkFactory destination;

  //We can leverage polymorphism to use the same instance object
  if(mood == "sleepy") {
      destination = CoffeeShop();
  }//if
  else{
      destination = Pub();
  }//else

  //This will assign to myBeverage a Coffee
  var myBeverage = destination.createDrink();

  //This will print "Pouring a coffee!"
  myBeverage.pour();
}//main
```


Full example in lab_03-dart_101_part2/abstract_factory_example.dart

Abstract Factory

BONUS

- Consequences:
  - **We isolated the concrete classes**: the factory isolates users from implementation classes of the products since the factory encapsulate the responsibility of creating products objects.
  - **We made exchanging product families easy**: we can just create a factory once and simply change the class that implements it to obtain different behaviors.
  - **We promoted consistency among products**: it enforces the fact that when products of the same theme are designed to work together, the application should use objects from only one family at a time.
  - **Supporting new products is difficult**: the abstract factory fixes the kind of products that can be created. To support new products, we need to extend is.

Outline

- Goals and Principles
- Classes and objects
- Inheritance
- Polymorphism
- Abstract data types
- Flavours of design patterns
  - Composite
  - Singleton
  - Abstract Factory
- **Exercises**
- Homework
- Resources

Exercises

- Exercise 02.01
  - Create a class Vehicle with max_speed, is_moving and mileage instance variables (properly choose the type of the variables). max_speed is constant. is_moving and milage must be properly initiatilized.
  - Create an unnamed constructor with the minimum amount of input arguments.
  - Create also a named constructor Vehicle.used that creates a new Vehicle with a given mileage.
  - Implement two methods start and stop that properly set is_moving
  - Implement also the toString() method of the Vehicle class.
  - Create a method addMiles that takes a named parameter miles, adds that value to the current mileage, and return the new mileage.
  - Properly test the created class capabilities in the main function.

- Exercise 02.02
  - Create a Bus class that extends the Vehicle class and inherit everything from it.
  - Properly inherit the superclass constructors
  - A bus must retain also the current_number_of_passengers and the max_number_of_passengers.
  - Each Bus has a constant max_number_of_passengers equal to 20 and the initial current_number_of_passengers is always 0.
  - Implement a method board that increments the number of passengers by a given value (as much as possible) and return the new number of passengers.
  - Remember to correctly manage the toString() method.
  - Properly test the created class capabilities in the main function.

Outline

- Goals and Principles
- Classes and objects
- Inheritance
- Polymorphism
- Abstract data types
- Flavours of design patterns
  - Composite
  - Singleton
  - Abstract Factory
- Exercises
- **Homework**
- Resources

Homework

- (At least try to) Do all the exercises
- Get familiar with Dart 101 (part 1 & 2)
- Get familiar with OOP

Outline

- Goals and Principles
- Classes and objects
- Inheritance
- Polymorphism
- Abstract data types
- Flavours of design patterns
  - Composite
  - Singleton
  - Abstract Factory
- Exercises
- Homework
- **Resources**

Resources

- Code repository of today's lesson and exercises solution
  - https://github.com/gcappon/bwthw/tree/master/lab_03-dart_101_part_2
- Dart language tour
  - https://dart.dev/guides/language/language-tour
- The Gang Of Four – Design Patterns
  - https://github.com/amilajack/reading/blob/master/Design/GOF%20Design%20Patterns.pdf
- Exhaustive examples of design patterns implemented in Dart
  - https://scottt2.github.io/design-patterns-in-dart/