Classes and Objects

COSC346

Overview

- OO Concepts
  - classes and objects
  - instances, encapsulation, behaviours, state
  - visibility
- Swift implementation
- Design Patterns

Description of OOP

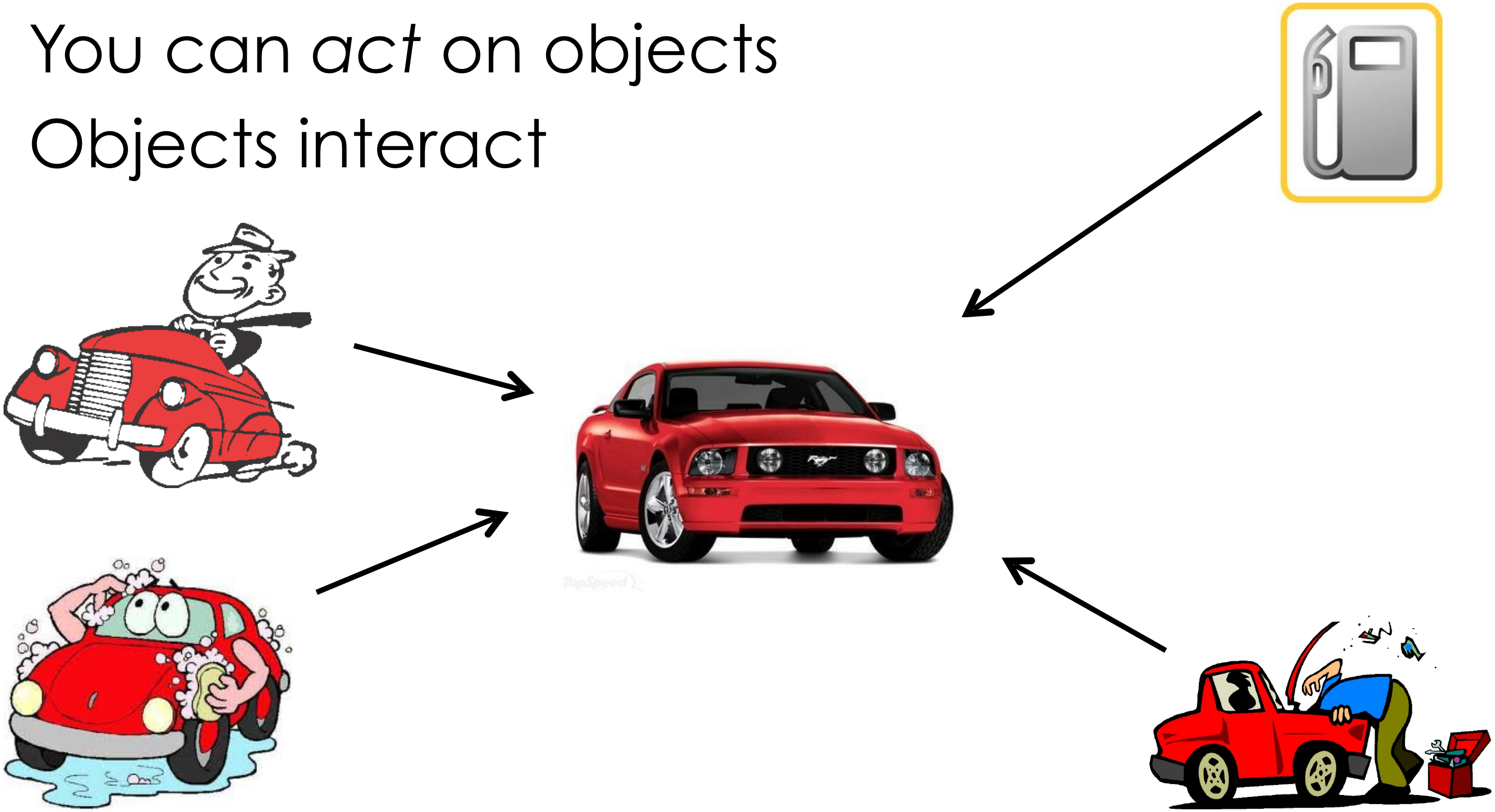
1. Everything is an object
2. Objects perform computation by making requests of each other by passing messages
3. Every object has its own memory, which consists of other objects
4. Every object is an instance of a class. A class groups similar objects.
5. The class is the repository for the behaviour associated with an object.
6. Classes are organised into a singularly rooted tree structure, called an inheritance hierarchy

Objects in real world

- An object is a thing
- A real-world example is a car

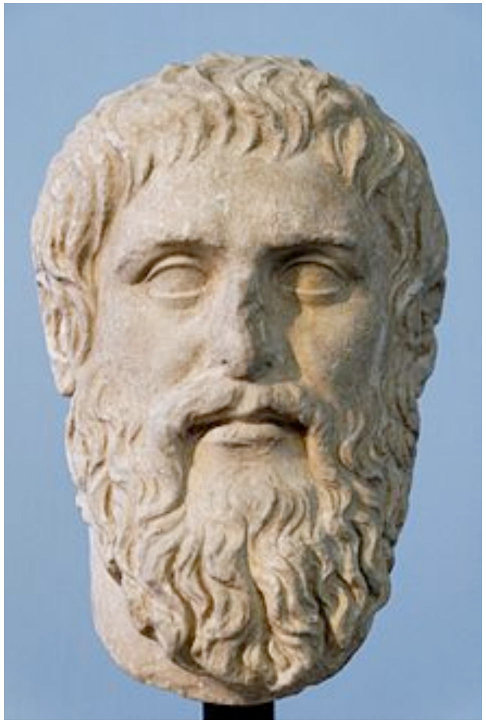
Objects in real world

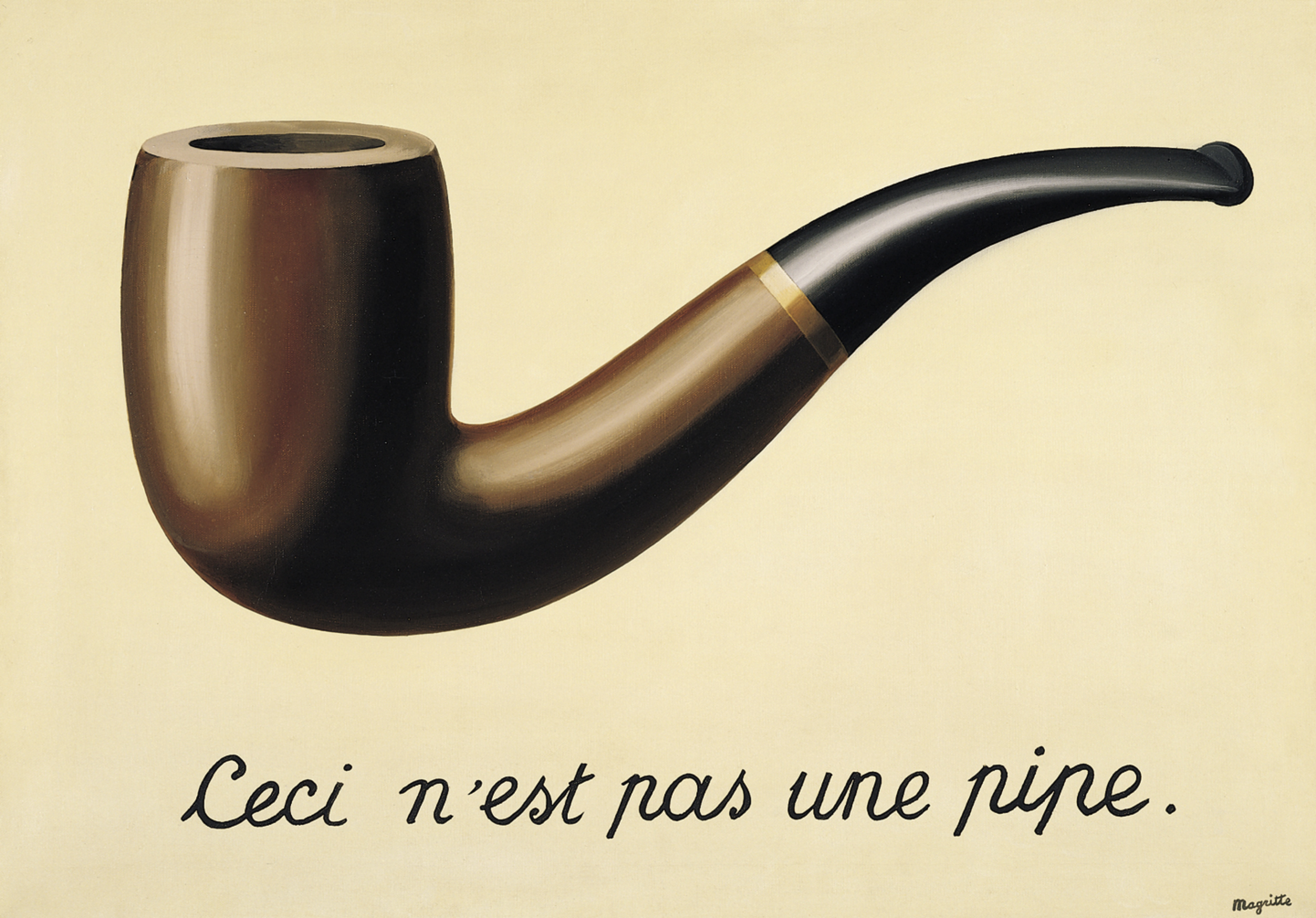
- Objects have properties
- You can *act* on objects
- Objects interact



Plato’s Theory of Forms

- Objects that we see mimic real Forms
- A Form is an idea, an abstract concept that conveys the essence of an object
- Example:
  - What is the form of a “car”?
  - How does a car you see on the street correspond to its form?

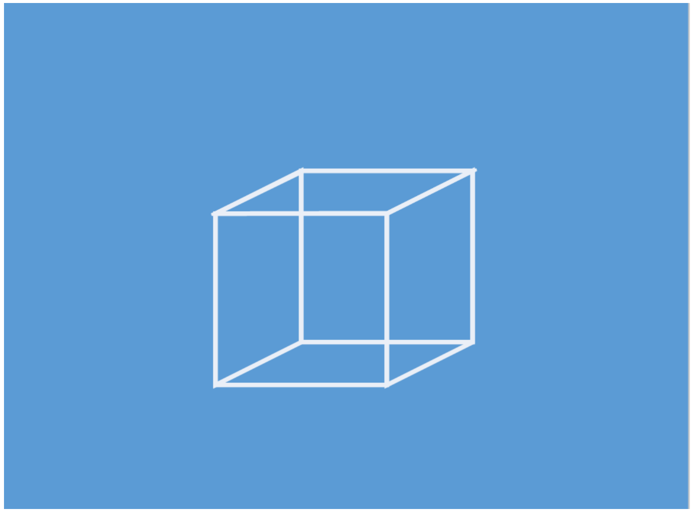


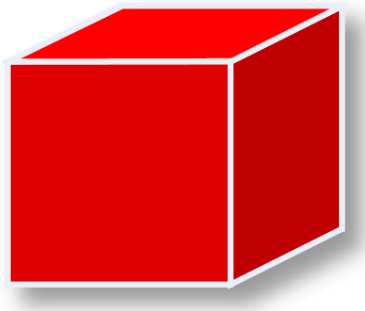


The Treachery of Images (This is not a pipe) (1928-1929)
René Magritte

Class and Object

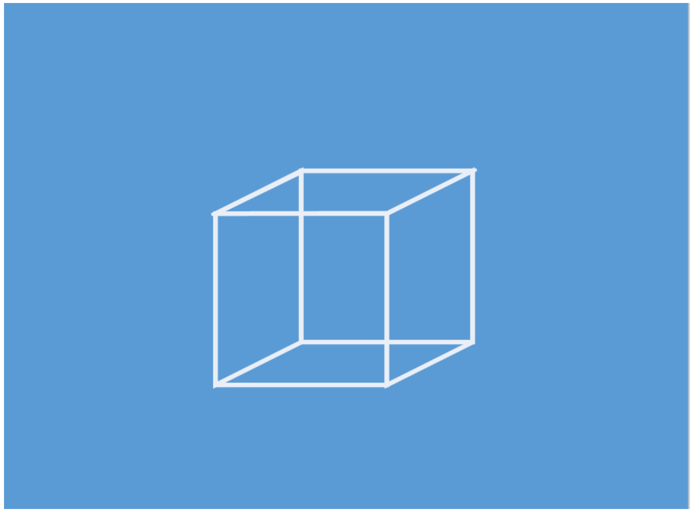
- A class is a specification of how the object is to be built (essence of an object)
- An object is an instance of a class (a "real" object…in computer memory)

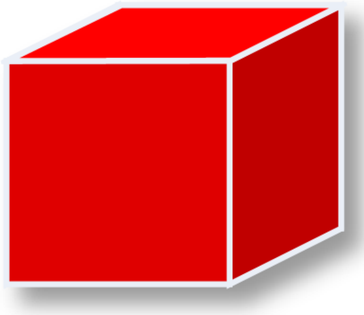




Class

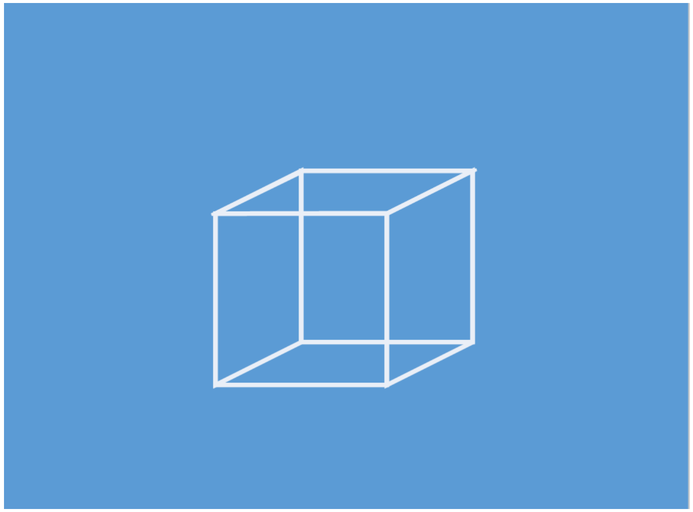
- A class defines a type by specifying:
  - What its state is composed of (its internal variables and properties)?
  - How it behaves (its methods)?

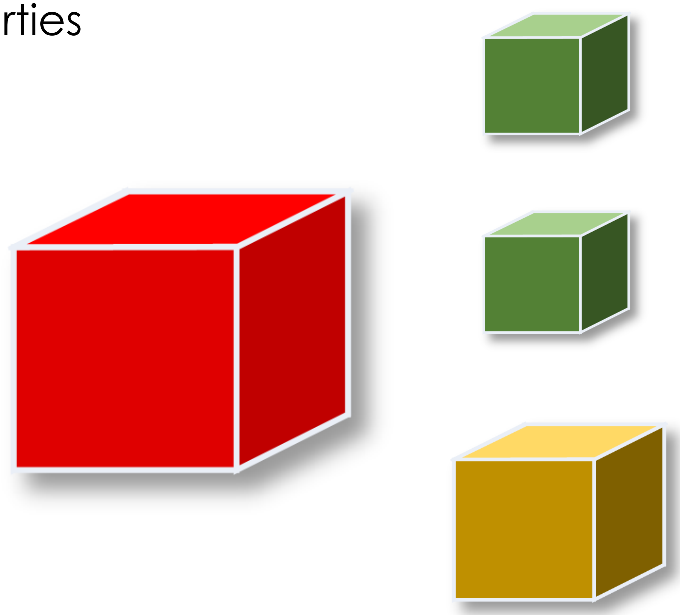




Object instances

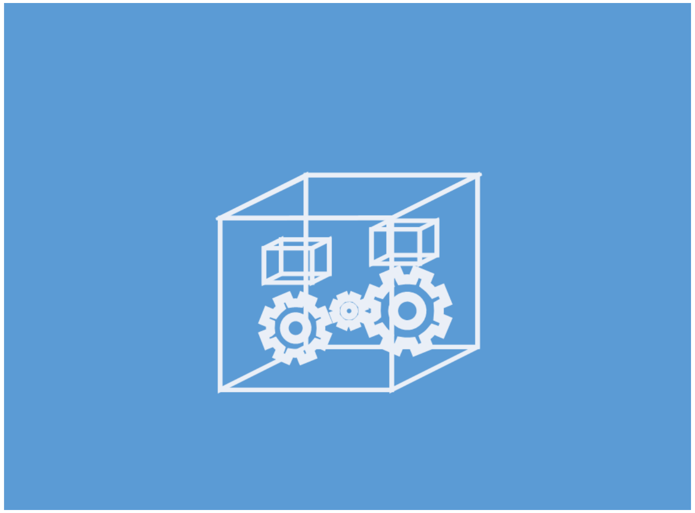
- An object instance is a particular "realisation" of a given class
  - Properties take on specific values
  - Behaviour of a given object may depend on its state and properties
  - Different instances can have different properties

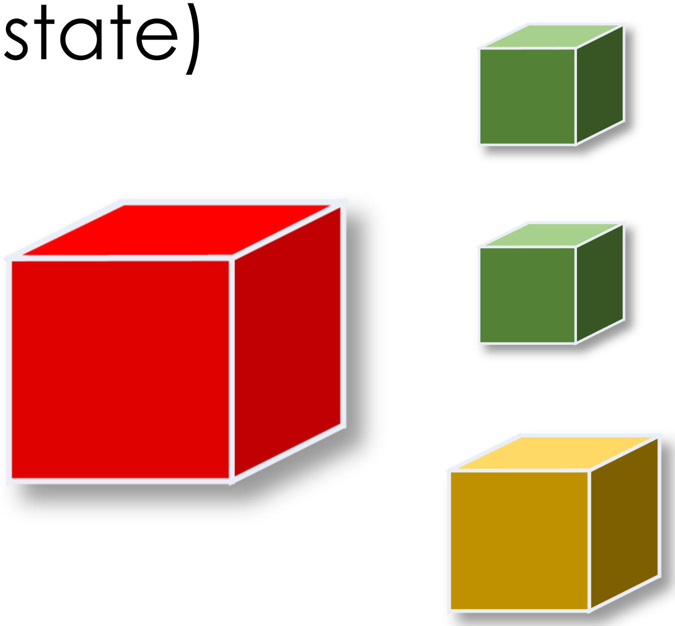




Object State

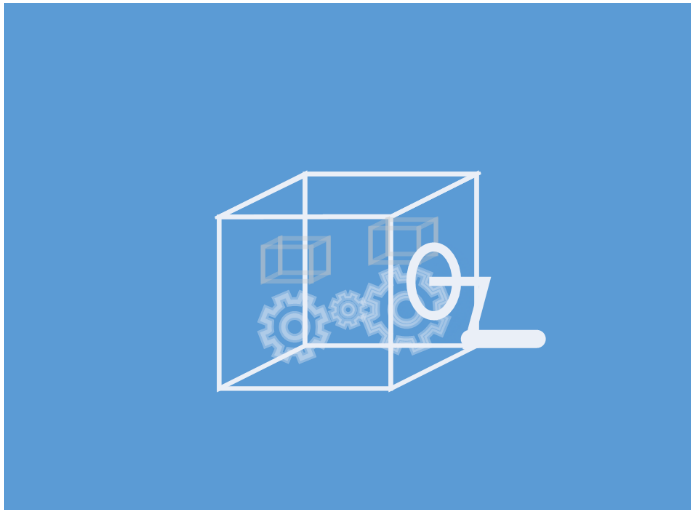
- **Instance variables** specify object's state
- Some of this state is visible to the object user (object properties) …
- … and some is not (internal state)

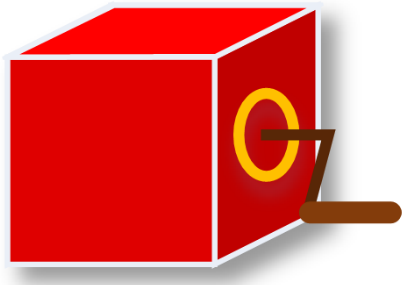




Methods

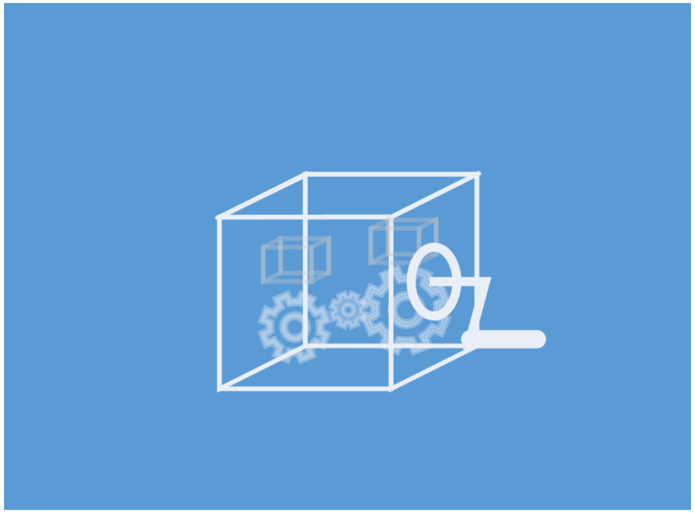
- **Methods** are class specific functions that define what the object does and how it does it





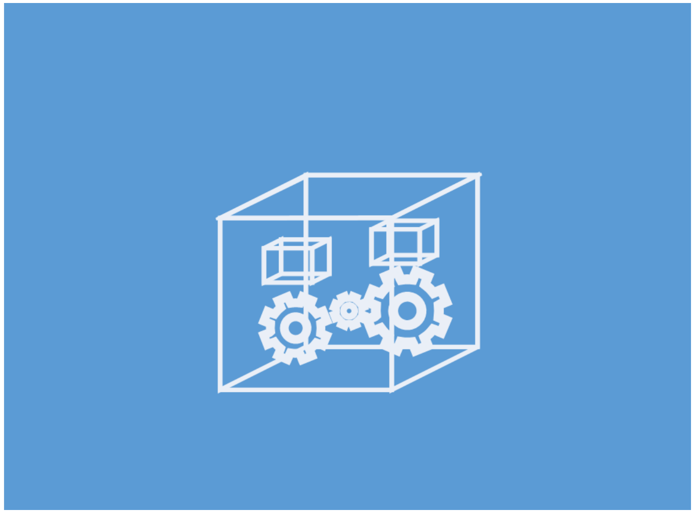
Methods

- **Methods** are class specific functions that define what the object does and how it does it



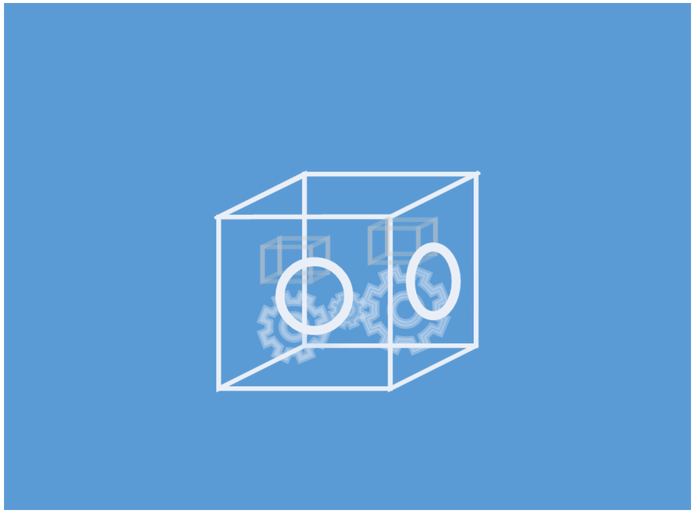
Abstraction

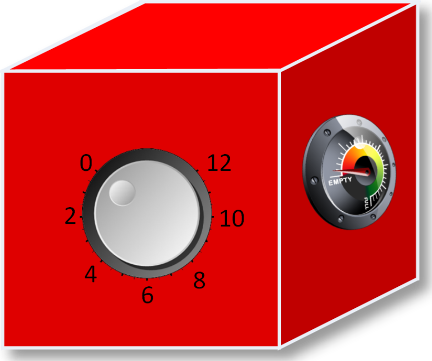
- Knowledge of the inner workings of the object is not required in order to use it
- It’s sufficient that user understands object’s properties (visible state) and how to use it



Encapsulation

- Internal state may not be directly visible to user, but interface (methods) may be provided to allow user to modify the state
  - Ability to control access to the inner state of the object
- Accessor methods:
  - **Setters** – methods that allow writing to *internal variables*
  - **Getters** – methods that allow reading of *internal variables*





Visibility

Mechanic

- Works on the engine, so that car is drivable
- Engine internals are hidden away under the hood

Driver

- Does not need to understand how the engine works
- Does not need to look at the engine
- Needs to use the interface skilfully in order to control the engine and drive

Visibility

Toolmaker

- Works on the implementation of the class, so that its object is usable
- Class internals can be hidden away

Builder

- Does not need to understand details of the class internals
- Does not need to look at the internals
- Needs to instantiate objects of the class and use their methods skilfully in order to co produce desired program logic

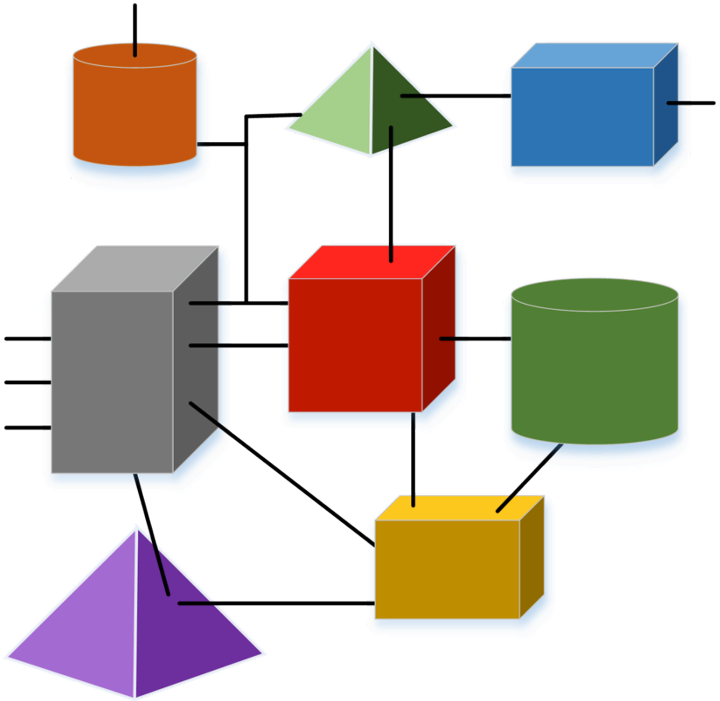
Visibility

- Class creator can decide the degree of visibility into its internals

- Access Control:
  - **Private** – only visible from within class implementation (*internal* use)
  - **Public** – visible to the object user (internal and external use)

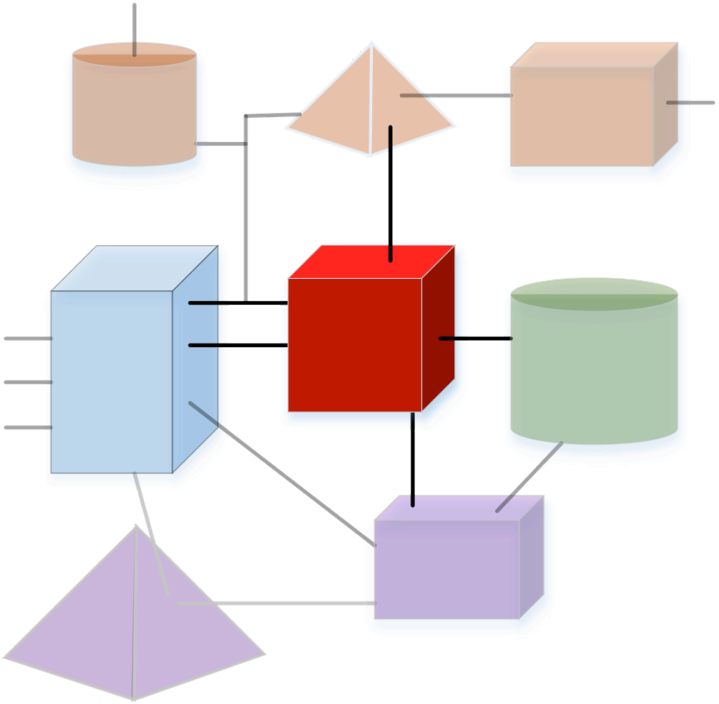
Interchangeability

- Ability to change inner working of an object without affecting its interface and the code depending upon it



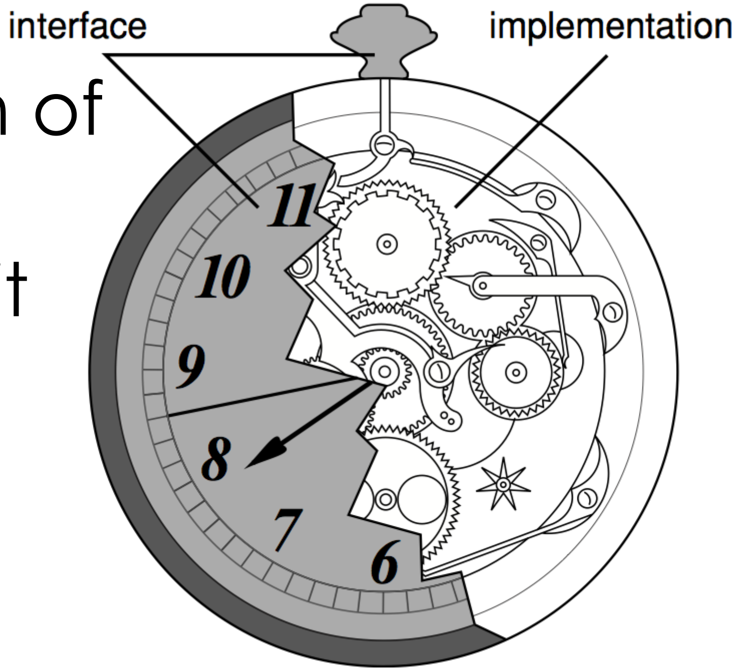
Interchangeability

- Ability to change inner working of an object without affecting its interface and the code depending upon it



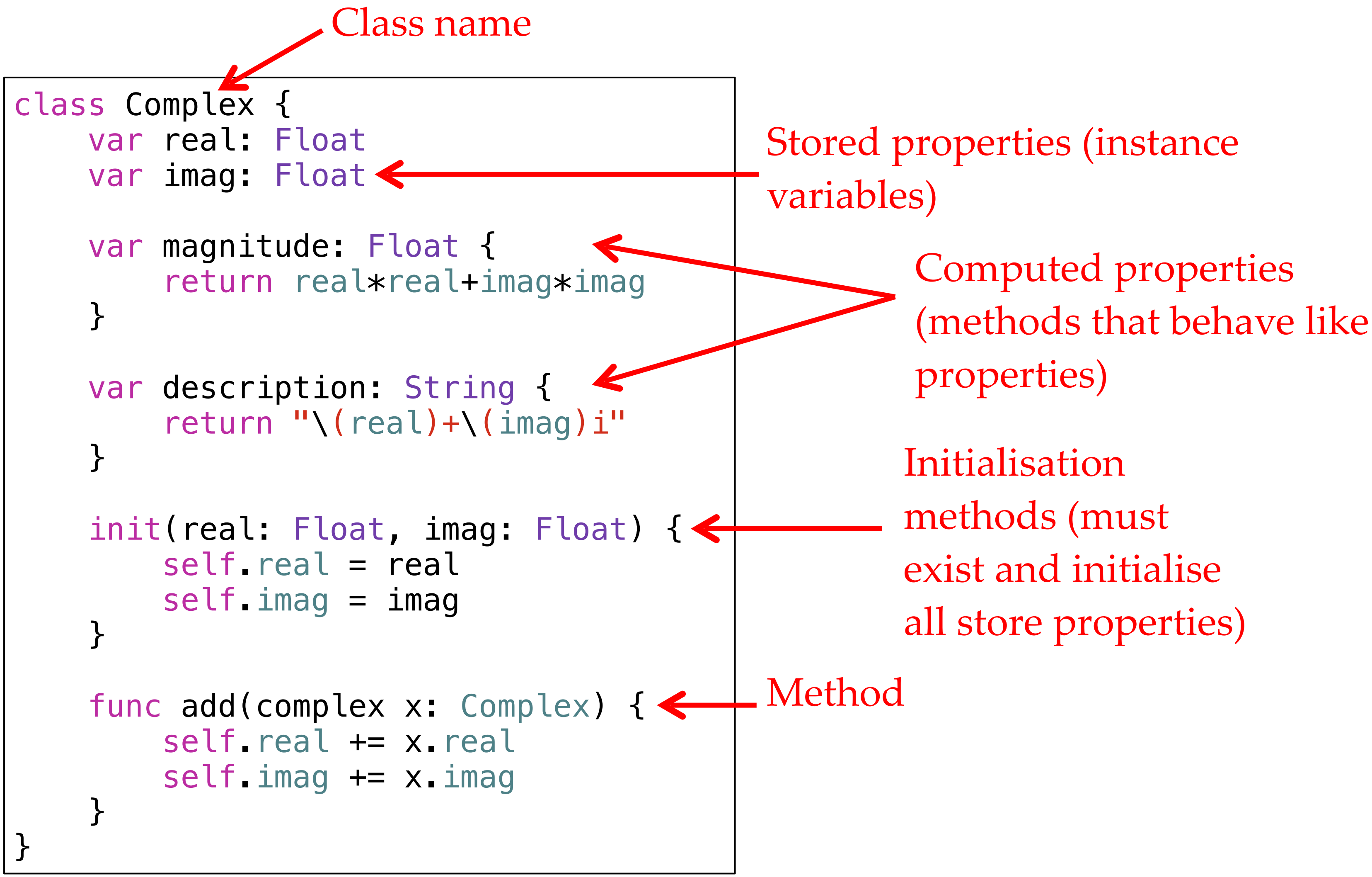
Interface and implementation

- **Interface**—declaration of what the object is and what can be done to it
- **Implementation**—the code that defines the behaviour of the class object
- In many languages class interface and implementation are specified separately (header and implementation files)



How to define a class

Swift



Class name

```
class Complex {
    var real: Float
    var imag: Float

    var magnitude: Float {
        return real*real+imag*imag
    }

    var description: String {
        return "\(real)+\(imag)i"
    }

    init(real: Float, imag: Float) {
        self.real = real
        self.imag = imag
    }

    func add(complex x: Complex) {
        self.real += x.real
        self.imag += x.imag
    }
}
```

Stored properties (instance variables)

Computed properties (methods that behave like properties)

Initialisation methods (must exist and initialise all store properties)

Method

How to define a class

Swift

```
class Complex {
    var real: Float
    var imag: Float

    var magnitude: Float {
        return real*real+imag*imag
    }

    var description: String {
        return "\(real)+\(imag)i"
    }

    init(real: Float, imag: Float) {
        self.real = real
        self.imag = imag
    }

    func add(complex x: Complex) {
        self.real += x.real
        self.imag += x.imag
    }
}
```

- Swift doesn’t separate interface and implementation: it’s all in one place
- Setters & getters can be defined within computed property
- The default setting for access control makes class internals visible to all files in the module/project

How to create an object instance

Create two instances of Complex objects with different internal state

```swift
var m: Complex = Complex(real: 3.1, imag: -0.5)
var n: Complex = Complex(real: 1.0, imag: 2.3)



print("The magnitude of \(m.real)+\(m.imag)i is \(m.magnitude)")
print("The magnitude of \(n.description) is \(n.magnitude)")



print("(\(m.description))+(\(n.description))=", terminator: "")
m.add(complex:n)
print("\(m.description)")
```

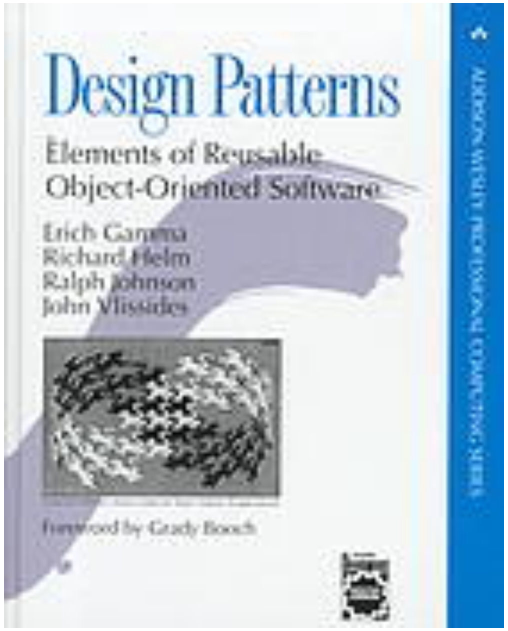
Stored properties

Computed properties

Invoke a method on ‘m’ with ‘n’ passed in as a parameter

Design Patterns

- Reusable solution to a commonly occurring problem
- Lies between a paradigm and an algorithm
- First book appeared in 1994
  - The "Gang of Four" (GoF)
  - Language features make some patterns unnecessary
  - Can unnecessarily increase complexity

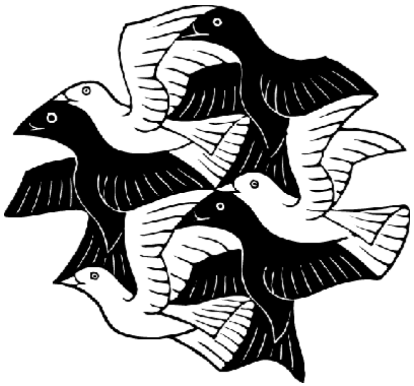

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

USE WITH CAUTION!

Three types of patterns: **Creational, Behavioural, Structural**

Why design patterns?

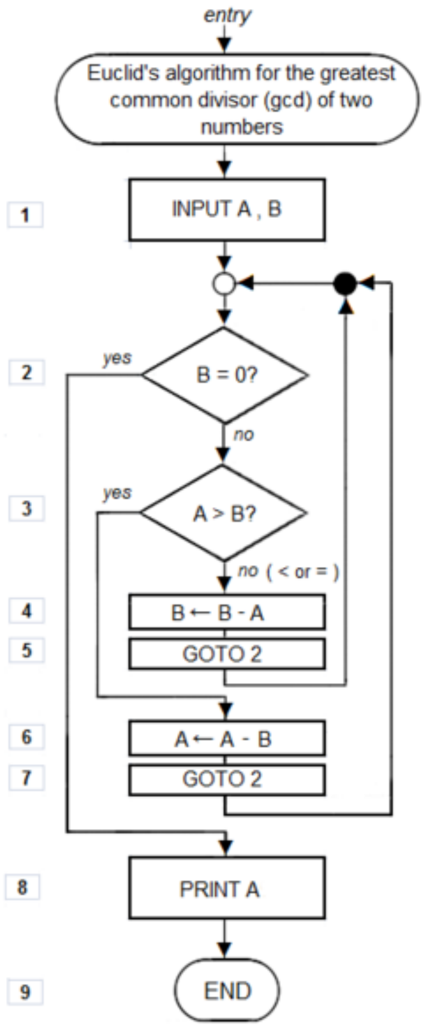
- Knowing OOP basics does not automatically make you a good OOP designer
- Patterns show you how to build systems with good OO design qualities
  - Patterns don't give you code, but general solutions to design problems
  - Patterns aren't invented, they're discovered
  - Most patterns and principles address issues of change in software
  - Most patterns allow some part of a system to vary independently of other parts



From Head First Design patterns, O'Reilly Media

Algorithms versus design patterns

- An **algorithm** provides a set of step-by-step instructions that can be described in pseudo-code then implemented directly
    - Euclid's method for finding the greatest common divisor of two numbers
- A **design pattern** describes a solution to a common, but generic, problem
    - It is like a meta-algorithm, or a generic approach
    - It typically concerns interactions between objects in OOP
    - It must generally be re-implemented each time it is used
- Algorithms are specific, design patterns are general



Pattern of the Day - Iterator

- **Behaviour**
  - Access each element of a container in order
- Don't want to know details of the container
- Traversing a LinkedList and an Array should look the same

Iterator Pattern

Pattern of the Day - Iterator

- **Behaviour**
  - Access each element of a container in order
- Don't want to know details of the container
- Traversing a LinkedList and an Array should look the same

Iterator Pattern

```
protocol Iterator{
     func next() -> Int
     func hasNext() -> Bool
}
```

Pattern of the Day - Iterator

Swift

Toolmaker

```
class ArrayIterator:Iterator {
    var pos: Int
    var cntr: ArrayContainer

    func next() -> Int {
        let val = cntr.get(pos)
        pos += 1
        return val
    }
    func hasNext() -> Bool {
        return pos < cntr.size()
    }
}
```

```
class ListIterator:Iterator {
    var curr: ListNode?
    var cntr: ListContainer

    func next() -> Int {
        let val = self.curr?.val
        self.curr = self.curr?.next
        return val
    }
    func hasNext() -> Bool {
        return self.curr != nil
    }
}
```

Builder

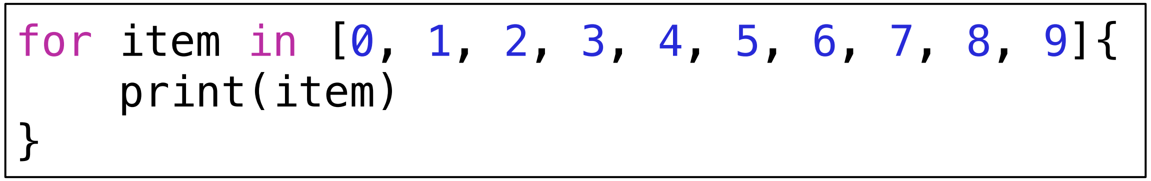
```
let array = ArrayContainer([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
let list = ListContainer([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
var iter: Iterator
```

```
print("array iterator")
iter = array.getIterator()
while iter.hasNext() {
    print(iter.next())
}
```

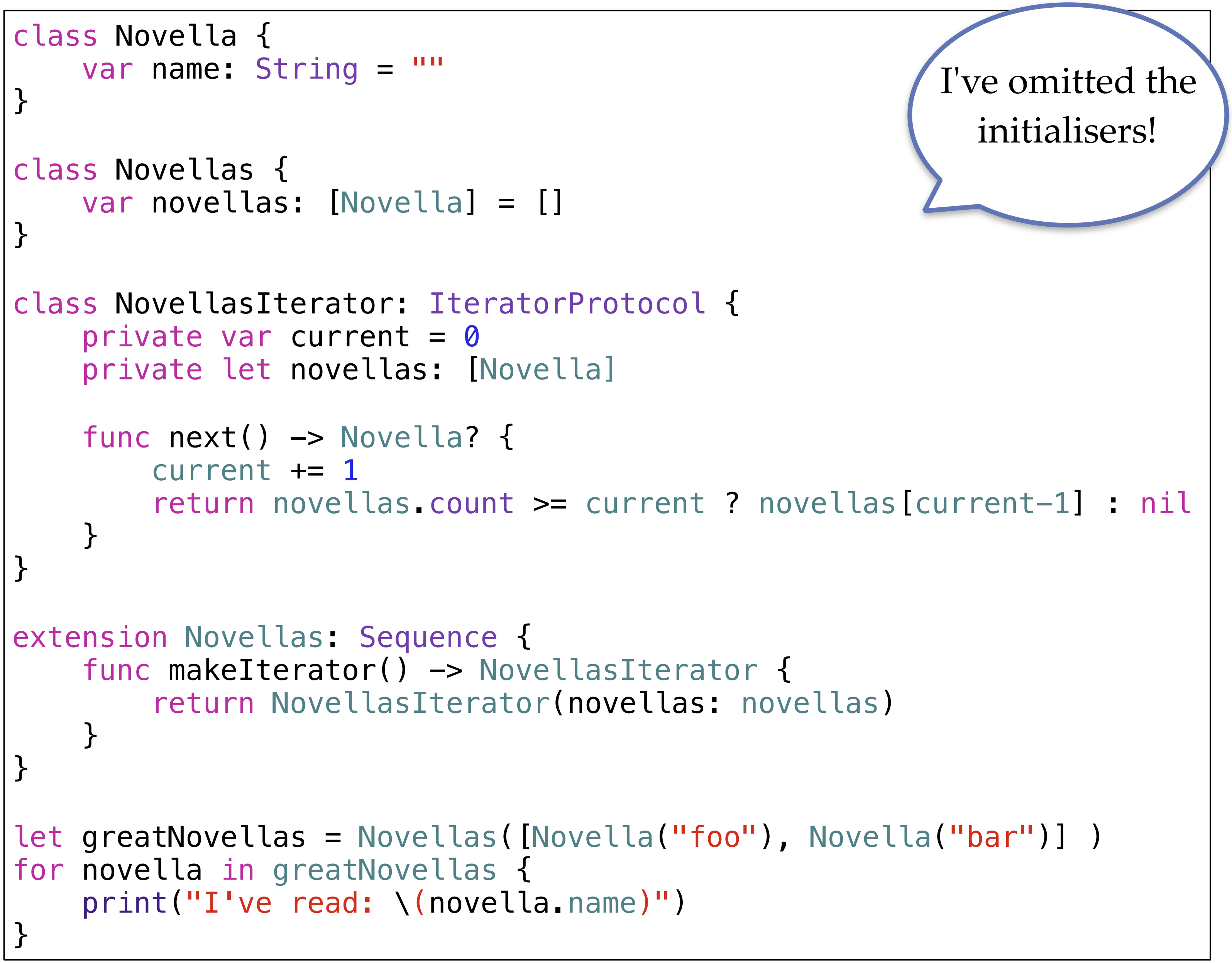
```
print("list iterator")
iter = list.getIterator()
while iter.hasNext() {
    print(iter.next())
}
```

Pattern of the Day - Iterator

- Rendered obsolete by modern language constructs



```
for item in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]{
    print(item)
}
```



```swift
class Novella {
    var name: String = ""
}

class Novellas {
    var novellas: [Novella] = []
}

class NovellasIterator: IteratorProtocol {
    private var current = 0
    private let novellas: [Novella]

    func next() -> Novella? {
        current += 1
        return novellas.count >= current ? novellas[current-1] : nil
    }
}

extension Novellas: Sequence {
    func makeIterator() -> NovellasIterator {
        return NovellasIterator(novellas: novellas)
    }
}

let greatNovellas = Novellas([Novella("foo"), Novella("bar")] )
for novella in greatNovellas {
    print("I've read: \(novella.name)")
}
```

I've omitted the initialisers!

https://github.com/ochococo/Design-Patterns-In-Swift/blob/master/source/behavioral/iterator.swift

Summary?

Summary?

- Classes and Objects
  - Classes - Blueprint
  - Objects - Realisation
- State vs Behaviour
- Visibility
  - Toolmaker vs Builder
- Interchangeability
- Interface and Implementation
- Design Pattern
  - Iterator