Key Concepts of Object-Oriented Programming

Programmazione Avanzata 2025-26

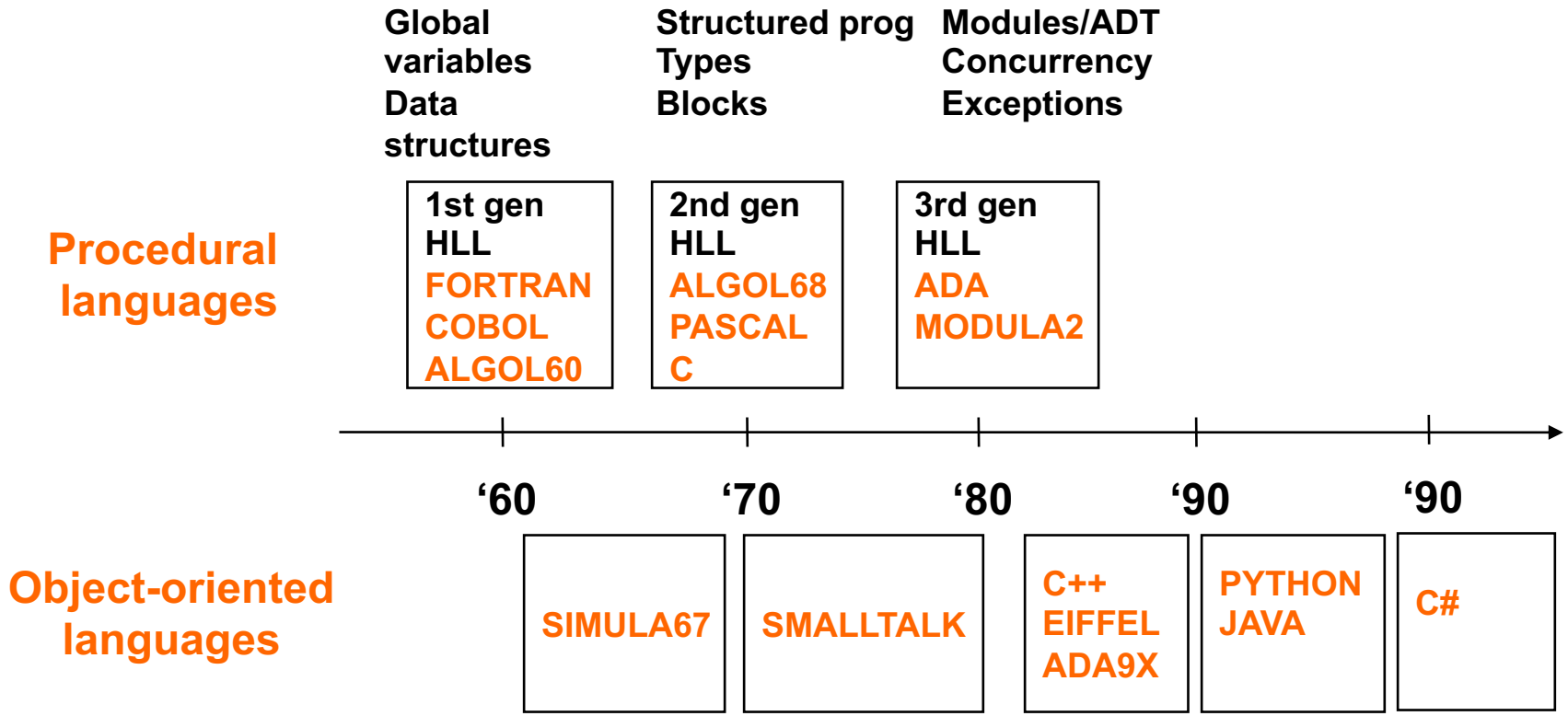
Learning objectives

- Define the object-oriented (OO) paradigm
  - What are objects and classes?
- Understand the differences between procedural approach and OO
  - What is encapsulation?
- Understand the fundamental concepts of OO
  - What are interfaces, messages, and inheritance?
- Appreciate the benefits of OO
  - What are modularity, reuse, and maintainability?

Programmig paradigms

- Procedural (Pascal, C, Python, …)
- Object-Oriented (C++, Java, C#, Python, …)
- Functional (LISP, Haskell, SQL, …)
- Logic (Prolog)

Languages timeline



Procedural (C)

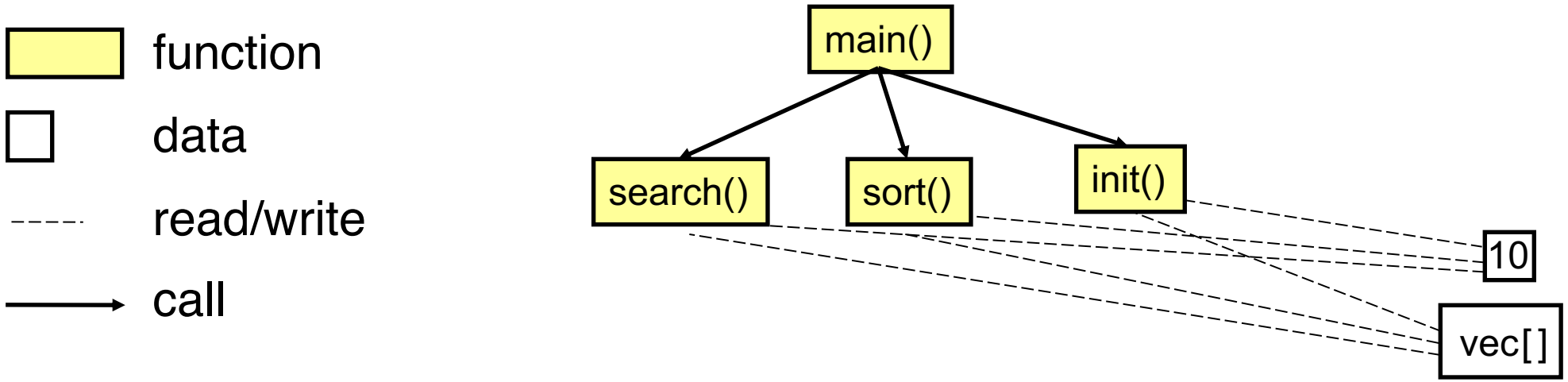
```
int vec[10]; // vector, array, or list
void sort() { /* sort the data structure */ }
int search(int n){ /* search into it */ }
void init() { /* initialize it */ }
// ...
void main(){
    init();
    sort();
    search(13);
}
```

Procedural (Python)

```
vec = ...# vector, array or list(vec=[0]*10)
def sort(): ...    # sort the data structure
def search(n): ...# search into it
def init(): ...    # initialize it
# ...
def main():
    init()
    sort()
    search(13)
}
```

Elements and relationships

- Elements: data and functions (procedures)
- Relationships: call and read/write



Possible problems

- There is no syntactic relationship between:
  - Complex data structures (vec)
  - Operations onto them (search, sort, init)
- There is no control over size:

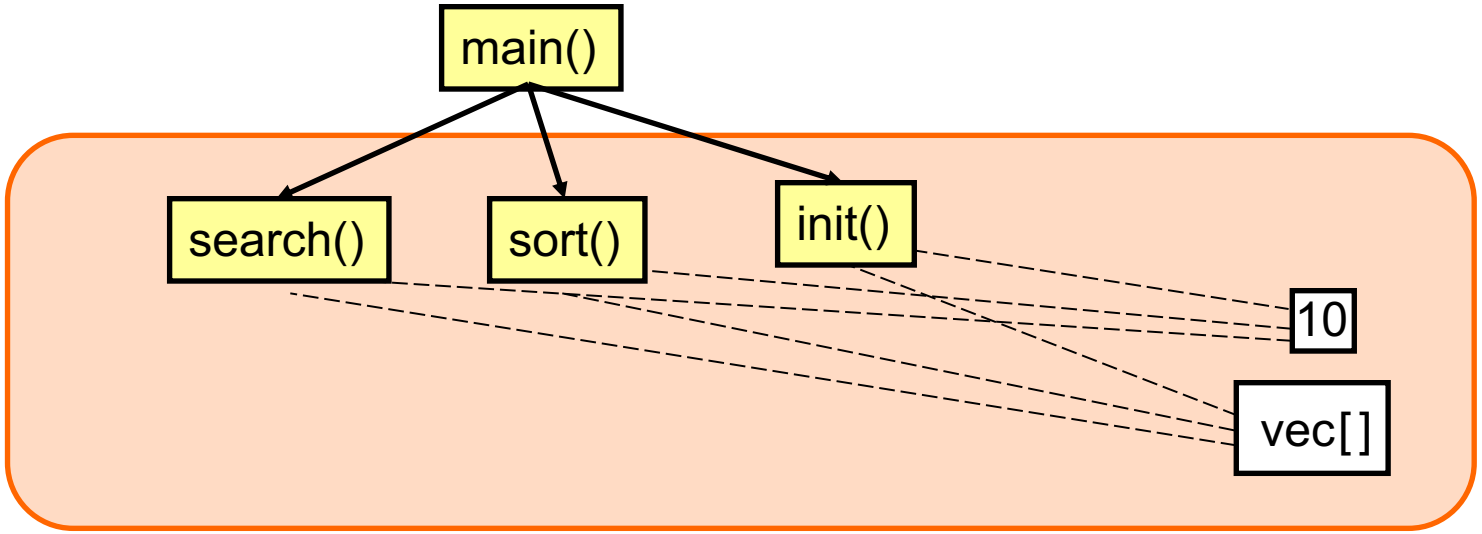
```
for (i=0; i<=11; i++) {  vec[ i ]=5; };
```

```
for i in range(0,11): vec[i] = 5
```

- Initialization
  - Actually performed?

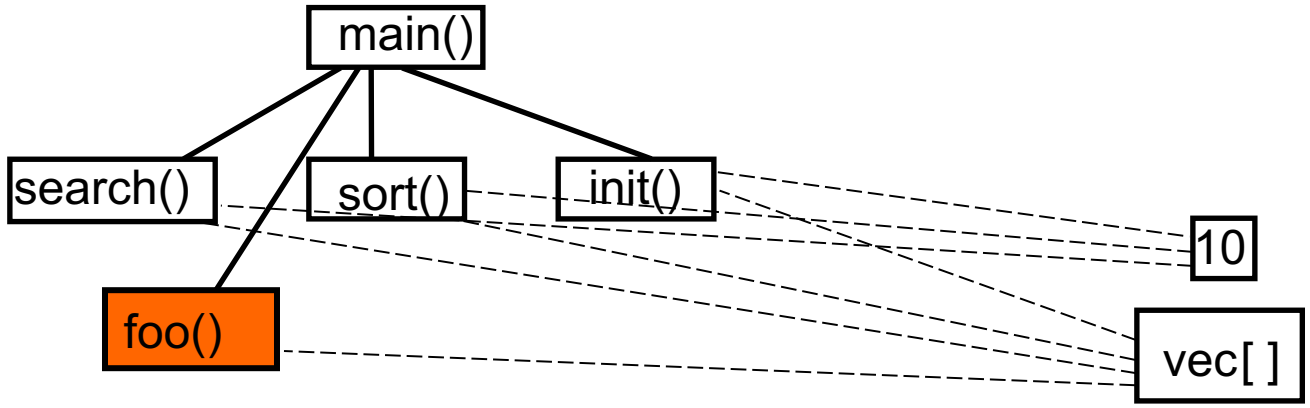
The container

- It is not possible to consider the container as a primitive and modular concept: data and functions cannot be modularized properly



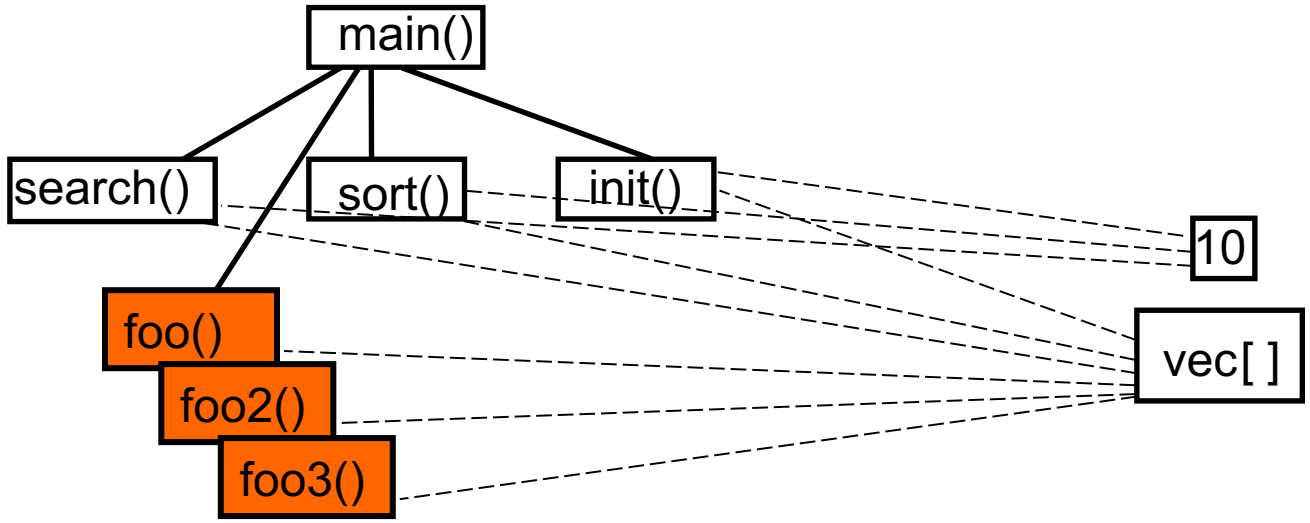
Procedural

- No constraints on read/write relationships
- External functions can read/write container's data



Procedural

- (All) functions may read/write (all) data
- As time goes by, growing number of relationships
- Source code becomes difficult to understand and maintain
  - Problem known as “spaghetti code”



What is OO?

- Procedural paradigm
  - Program defines data and then calls subprograms acting on data
- OO paradigm
  - Program creates objects that encapsulate the *data* and the *procedures* operating onto them (hiding internals)
- OO is “simply” a new way of organizing a program
  - Cannot do anything using OO that can’t be done using procedural paradigm

Why OO?

- Programs are getting too large to be fully comprehensible by any person
- There is need of a way of managing very-large projects
- OO paradigm allows:
  - Programmers to (re)use large blocks of code …
  - … without knowing all the picture
- Makes code reuse a real possibility
- Simplifies maintenance and evolution of code

When OO?

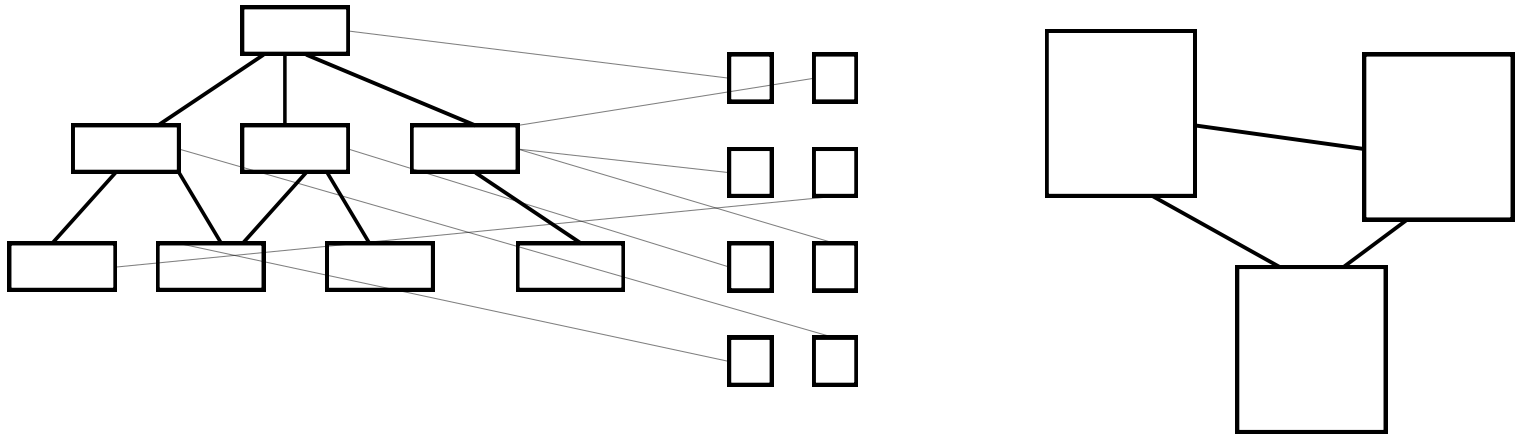
- Benefits only occur in larger programs
- Analogous to structured programming
  - Programs < 30 lines, spaghetti is as understandable and faster to write than structured
  - Programs > 1000 lines, spaghetti is incomprehensible, probably doesn’t work, not maintainable
- Only programs > 1000 lines benefit from OO really

An engineering approach

- Given a system, with components and relationships among them:
  - Identify the components
  - Define component interfaces
  - Define how components interact each other through their interfaces
  - Minimize relationships among components

An engineering approach

- Objects introduce an additional abstraction layer
- More complex systems can be built



OO approach

- Defines a new component type
  - Object (and class)
  - Data and functions on data are within the same module (*encapsulated*)
  - Allows defining a more precise interface
- Defines a new kind of relationship
  - Message passing
  - Read/write operations limited to *object scope*

Class and object

- A class represents a set of objects
  - Common properties (attributes and methods)
  - Autonomous existence
  - E.g., facts, things, people, etc.
- Abstraction
- An instance of a class is an object of the type that the class represents (the creation of an object is called instantiation)
  - A class is like a *type definition* (for languages that mandates it)
    - No data is allocated until an object is created from the class

Classes

- Goal
  - Group related information
  - Give a “structure” to it
- Fields (properties and methods)
- Accessing fields
- Lists of structured data

Class and object

- In an application for a commercial organization, **`City`**, **`Department`**, **`Employee`**, **`Purchase`** and **`Sale`** could be examples of typical classes
- No limit to the number of objects that can be created from a class
- Each object is independent
  - Changing one object doesn't change the others

Class and object

- Class (the description of object structure, i.e., its type):
  - Data (attributes or properties)
  - Functions (methods or operations)
- Object (class instance)
  - State and identity

References

- Reference vs. Object
  - Creation
  - Object names and aliasing

Example

```
class Car:
    def __init__(self):
        self.licensePlate = 0
        self.bodyColor = ''
        self.turnedOn = False

    def paint(self, color):
        self.bodyColor = color

    def turnOn(self):
        self.turnedOn = True

c = Car()
c.licensePlate = 'AB123CD’
c.paint('red')
c.turnOn()
```

Procedural vs OO approach

```
vect = [0]*10
def sort(v, size):
    print('Sorting ...')
def search(v, size, c): ...
def main():
   for i in range(0,10):
      vect[i] = 5
   sort(vect, 10)
   search(vect, 10, 33)
}
```

```
class Vector:
    def __init__(self):
        self.v = [5]*10

    def sort(self):
        print('Sorting ...')

    def search(self, c):
        print(f"Searching {c}")

v1 = Vector()
v1.sort()
v1.search(33)
```

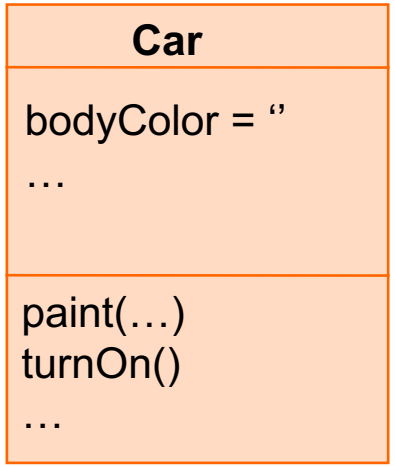
UML

- Unified Modeling Language
- Standardized modeling and specification language
  - Defined by the Object Management Group (OMG)
- Graphical notation to specify, visualize, construct and document an object-oriented system
- Several diagrams
  - Class diagram, Activity diagram, Use Case diagram, Sequence diagram, State diagrams, etc.

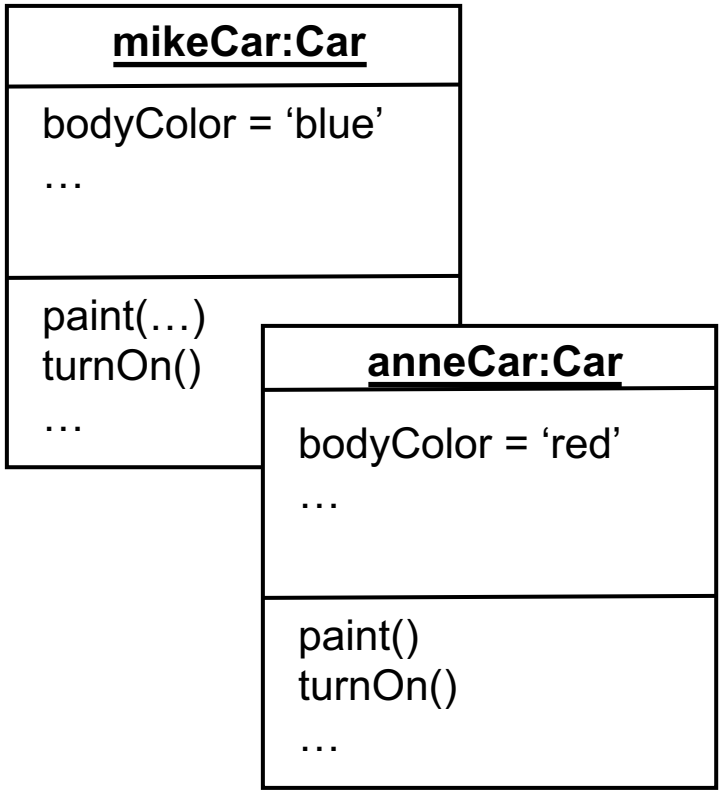
UML Class diagram

- Captures
  - Main concepts (classes)
  - Characteristics of the concepts
    - Data associated to the concepts
  - Relationships between concepts
  - Behavior of concepts
    - Operations associated to the concepts (functions)

UML Class diagram

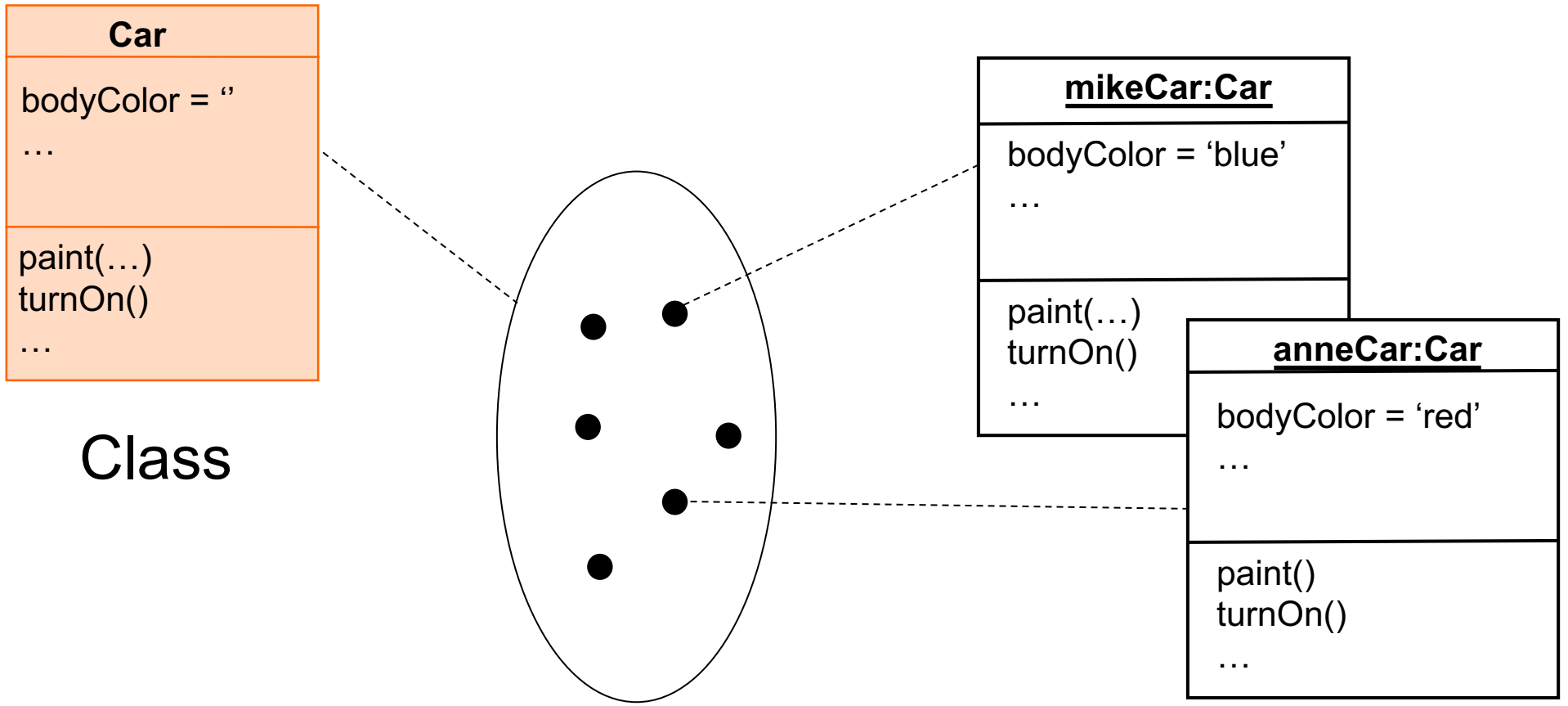


Class



Objects (i.e., instances)

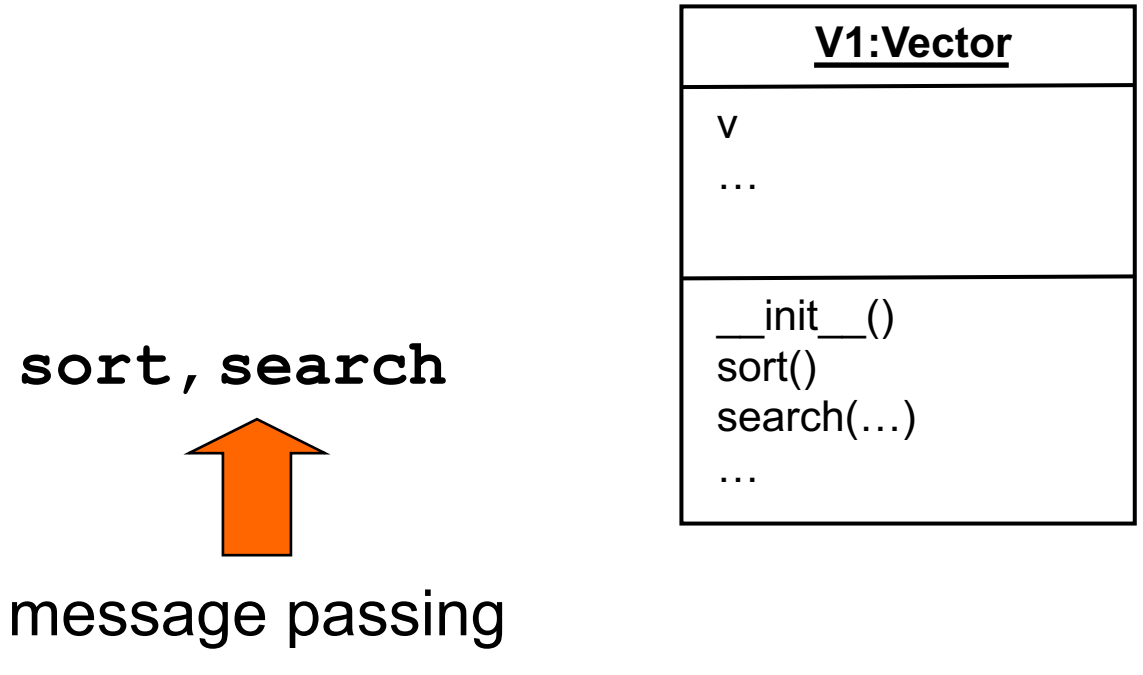
UML Class diagram



Message passing

- Objects communicate by message passing
  - Not by procedure call
  - Not by direct access to object's local data

```
v1 = Vector()
v1.sort()
v1.search(33)
```



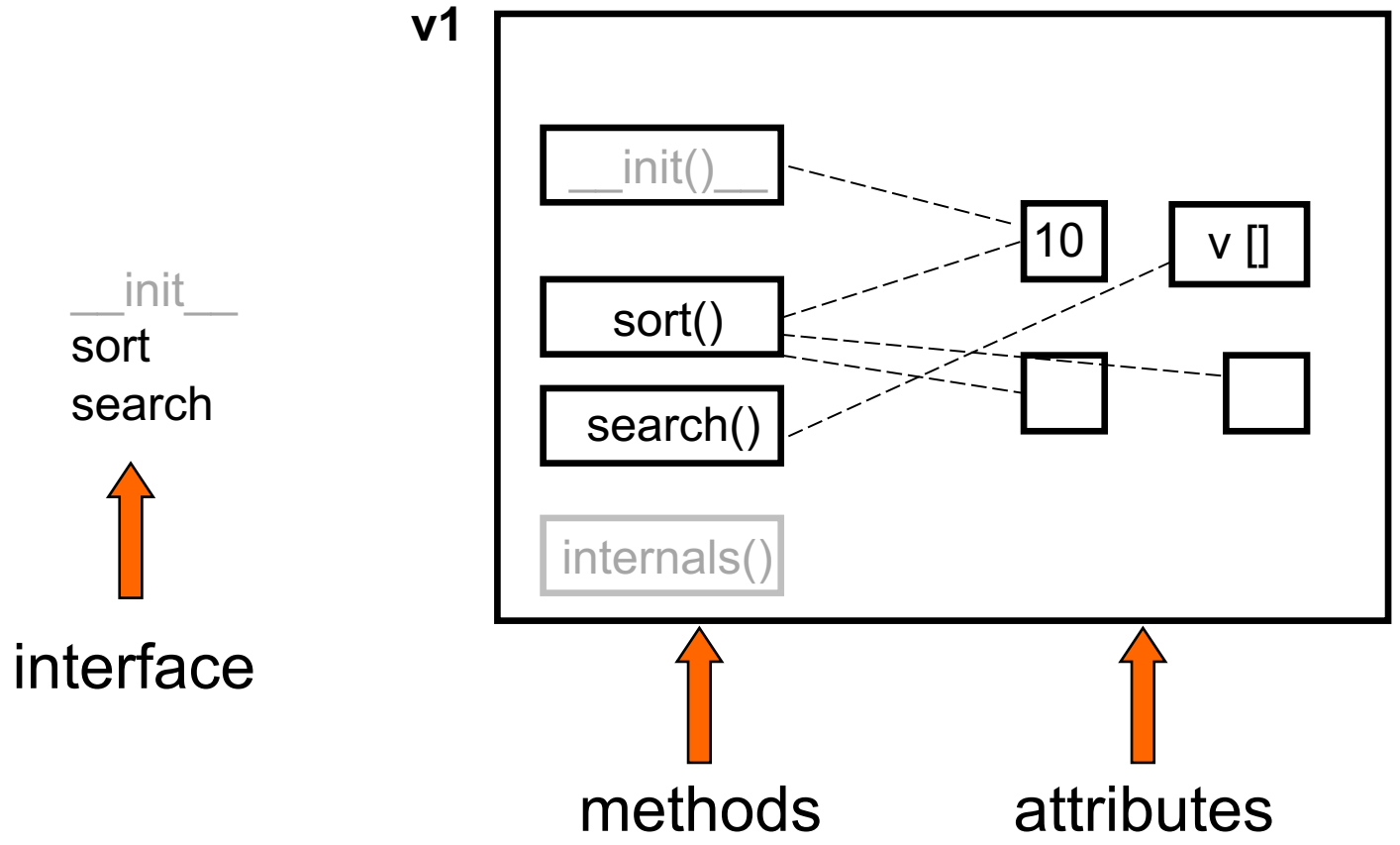
Message

- A message is a service request
  - **`search`**, **`sort`**
- A message may have arguments
- Examples
  - **`sort()`**
  - **`search(33)`**

Interface

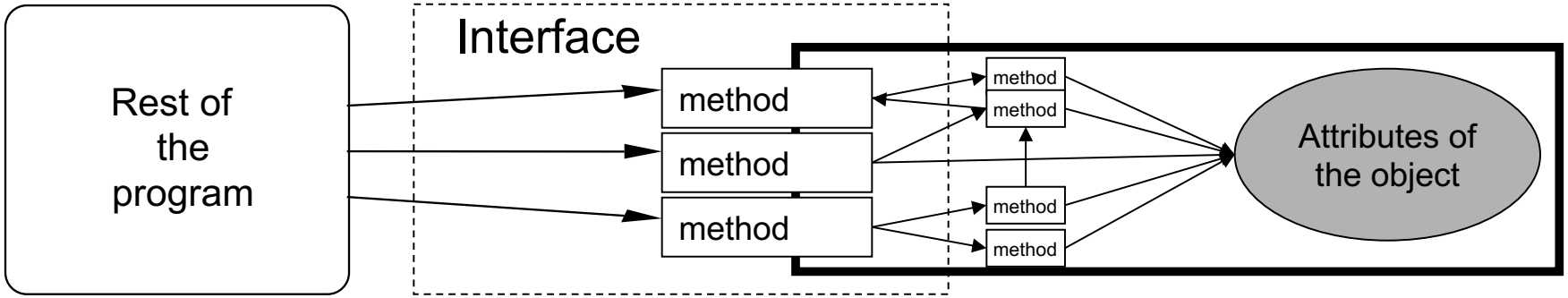
- Set of messages an object can receive
  - Any other message is illegal
  - The message is mapped to a function within the object
  - The object is responsible for the association (message, function)
- Through its interface, an object
  - Encapsulates its internals
  - Exposes a standard boundary

Interface


v1
__init()__
sort()
search()
internals()
10
v []
__init__
sort
search
interface
methods
attributes

Interface

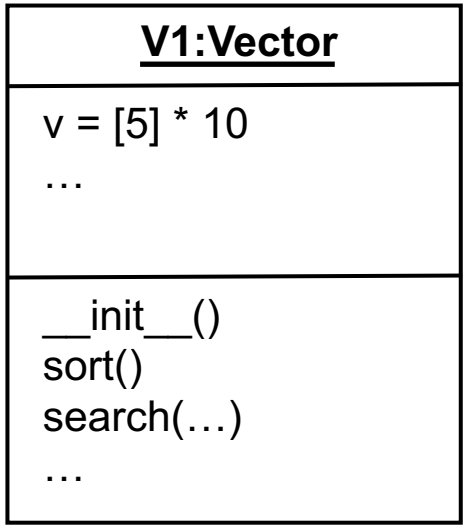
- The interface of an object is simply the subset of methods that other “program parts” are allowed to call
  - Stable (assumed to be, over time)



Encapsulation

- Read/write operations can only be performed by an object on its own data
- Between two objects, data are exchanged through message passing (methods)

```
v1 = Vector()
v1.sort()
v1.search(33)
v1.v[0] = 12 NO!
```



Benefits of encapsulation

- Simplified access
  - To use an object, the user need only to understand the interface: no knowledge of the internals is necessary
- Self-containment
  - Once the interface is defined, the programmer can implement the interface (write the code defining the object) without interference of others

Benefits of encapsulation

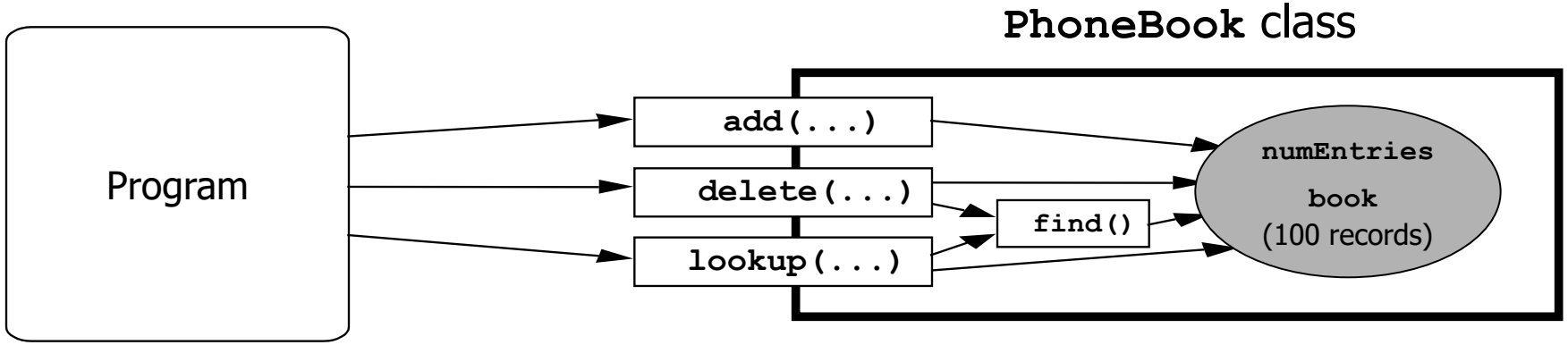
- Ease of evolution
  - Implementation can change at a later time without rewriting any other part of the program (as long as the interface does not change)
- Single point of change
  - Changes in the data mean changing code in one location, rather than code scattered around the program (error prone)

Encapsulation in real life

- Phone book
  - Allows user to enter, look up and delete names and phone numbers
  - Maximum 100 names in the phone book
    - Can be implemented using a data structure with fixed size
- **`PhoneBook`** class
  - Hidden data
    - In languages which have fixed size data structures, like C, an array
  - Interface
    - Methods **`add(...)`**, **`delete(...)`**, **`lookup(...)`**

Encapsulation in real life

- **`PhoneBook`**
  - Allows user to enter, look up and delete contacts (e.g., phone numbers)
  - Implemented using an array named **`book`**
  - Maximum 100 records in the phone book



Encapsulation in real life

- The **`PhoneBook`** class is successful, it is used in hundreds of applications across the company… but it only holds 100 records!
- It now must upgraded to hold unlimited number of records
- How can one do that without breaking all the other programs in the company (relying on that class)?

Encapsulation in real life

- The interface does not need to change, only the internals, thus there is no need to change any of the programs using **`PhoneBook`** class
- If it had been programmed in the procedural paradigm, instead, each program that used the phone book would have had a copy of the data array and would have to have been extensively modified to be upgraded

Inheritance in OOP

- A class can be a sub-type of another class
- The inheriting class contains all the methods and attributes of the class it inherited from plus any methods and attributes it defines
- The inheriting class can override (change) the definition of existing methods by providing its own implementation
- The code of the inheriting class consists only of the changes and additions to the base class

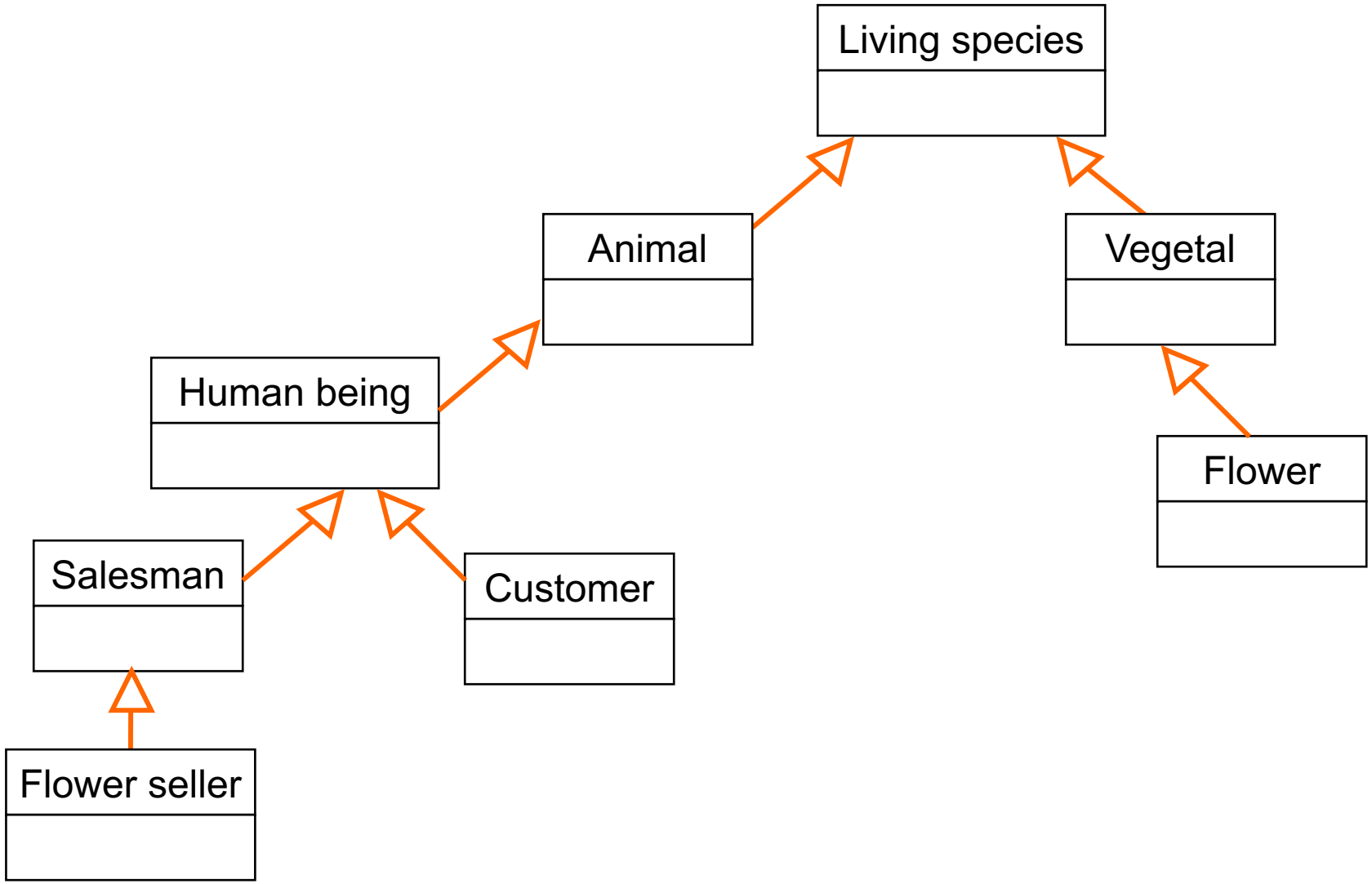
Why inheritance

- Frequently, a class is merely a modification of another class; in this way, there is minimal repetition of the same code
- Localization of code
  - Fixing a bug in the base class automatically fixes it in the subclasses
  - Adding functionality in the base class automatically adds it in the subclasses
  - Less chances of different (and inconsistent) implementations of the same operation

Inheritance in real life

- A new design created by the modification of an already existing design
  - The new design consists of only the changes or additions from the base design
- **`CoolPhoneBook`** inherits **`PhoneBook`**
  - Add email address and photo besides phone number

Inheritance tree



Inheritance terminology

- Class one above
  - Parent class
- Class one below
  - Child class
- Class one or more above
  - Superclass, ancestor class, base class
- Class one or more below
  - Subclass, descendent class, derived class

Specialization / Generalization

- B specializes A means that objects described by B have the same characteristics of objects described by A
- Objects described by B may have additional characteristics
- B is a special case of A
- A is a generalization of B (and possible other classes)

Wrap-up

- Class
  - Data structure (most likely private)
  - Private methods
  - Public interface
- Objects are class instances
  - State
  - Identity

Wrap-up

- The key role of interfaces
- Objects communicate by means of messages
- Each object manages its own state (data access)

Wrap-up

- Abstraction
  - The ability for a program to ignore some aspects of the information it is manipulating, i.e., the ability to focus on the essential
  - Each object in the system serves as a model can perform work, report on and change its state, and “communicate” with other objects in the system, without revealing *how* these features are implemented
- Example
  - Container of integers implemented as an array, or a list, …

Wrap-up

- Encapsulation
  - Together with information hiding ensures that objects cannot change the internal state of other objects in unexpected ways
  - Only the object's own methods are allowed to access its state
  - Each type of object exposes an interface to other objects that specifies how other objects may interact with it
- Do not break it, never ever … unless you know what you are doing !!!
  - Loosens up relationships among components

Wrap-up

- Inheritance
  - Objects defined as sub-types of already existing objects: they share the parent data/methods without having to re-implement
- Specialization/Generalization
  - Child class augments parent (e.g., adds an attribute/method)
- Overriding
  - Child class redefines parent method

Wrap-up

- Benefits of OO
  - Modularity (no “spaghetti code”)
  - Maintainability
  - Reusability