

I don't think object-oriented programming is a structuring paradigm that meets my standards of elegance.



Edsger W. Dijkstra

OBJECT-ORIENTED PROGRAMMING

Class and Object

Lecture #1

doc. Ing. Martin Tomášek, PhD.

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

(Imperative) Programming concepts

- Structured programming
 - Program flow structures (sequence, selection, iteration)
 - Subroutines (macros, procedures, functions)
 - Data structures (array, record)
- Modular programming
 - Separation of concerns
 - Definition of interfaces (declaration, implementation)
 - Improvement of maintainability
- Object-oriented programming
 - Objects (data fields + methods) and their interactions
 - Object-oriented techniques (**abstraction, information hiding, encapsulation, inheritance, polymorphism**)
 - **Improvement of code reuse**

Motivation for object orientation

- How to deal with complexity?
 - Abstraction
- How to define the scope of data inside data structures (objects)?
 - Encapsulation
- How to protect parts of the program from extensive modification based on design changes?
 - Information hiding
- How to reuse existing code with little or no modification?
 - Subtyping and inheritance
- How to work with variables (objects) using different types?
 - Polymorphism

Abstraction

- Simplifying complex reality by modeling objects appropriate to the problem
- An abstraction focuses on the outside view of an object and separates an object's behavior from its implementation

Abstract data types

- Mathematical model for a **class** of data structures (**objects**) that have similar behavior
- Abstract data types simplify the description of abstract algorithms
 - In programming languages usually implemented as data types, data structures, modules
- **One of the formalizing concept of object-oriented programming**

Abstract data types in object-oriented programming

- **Class**

- Defines abstract characteristics of things (objects)
- It includes **properties** (data fields) and **capabilities** (functions and procedures)

- **Object**

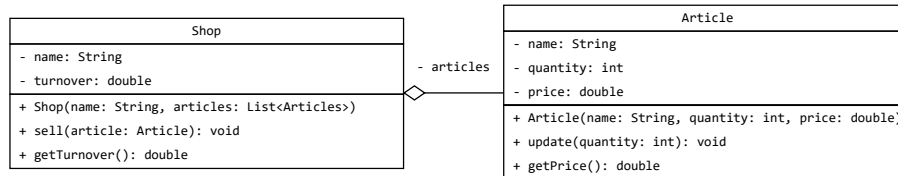
- One version (exemplar) of the class with concrete version of properties

Object example: Analysis

- Example: Let's have a shop that sales articles and keeps the track of total turnover
- What data structure do we need?
 - Shop
 - Properties: name, list of articles, total turnover
 - Capabilities: create new shop, sell article, get total turnover
 - Article
 - Properties: name, quantity, price
 - Capabilities: create new item, update article quantity (after sale), get price
- How to represent list of articles?
 - Dynamic data structures, e.g. linked list

Object example: Design

- For the design we will use UML
- We design two classes with following characteristics, behaviors and interactions



- Data fields inside classes are bound and can be maintained only by the object of that class itself (**encapsulation**)

What are software objects?

- Building blocks of software systems
 - A program is a collection of interacting objects
 - Objects cooperate to complete a task
 - To do this, they communicate by sending “messages” to each other
- Objects model **tangible** things
 - A shop
 - An article
- Objects model **conceptual** things
 - An inventory
 - An order
- Objects model **processes**
 - Calculating sales report for the shop owner
 - Sorting articles in the stock
- Objects have
 - **Capabilities**: what they can do, how they behave
 - **Properties**: features that describe them

Objects capabilities (behavior)

- Objects' **capabilities** allow them to perform specific actions
- Capabilities are also called **behaviors** and can be
 - **Constructors**: establish initial state of object's properties
 - **Commands**: change object's properties
 - **Queries**: provide responses based on object's properties
- Example: shops are capable of performing specific actions
 - **Constructor**: be created
 - **Commands**: sell article
 - **Queries**: get total turnover
- Implement capabilities as **methods**
 - Functions and procedures bound to the class

Object properties (state)

- **Properties** determine how an object acts
- Properties can be
 - **Attributes**: things that help describe an object
 - **Components**: things that are "part of" an object
 - **Associations**: things an object knows about, but are not part of that object
- **State**: collection of all an object's properties (changes if any property changes)
 - Some properties do not change, e.g., name of the article
 - Others do, e.g., quantity of the article
- Example: properties of shop
 - **Attributes**: name, total turnover
 - **Components**: articles
 - **Associations**: e.g. a shop can be associated with a chain of shops
- Implement properties as **instance variables**
 - Variables bound to the class

Object instantiation

- **Object instantiation** creates an object instance of a class
 - **Constructor** is the first message: creates the instance

```
Shop grocery = new Shop("Grocery", articles);
```

- Reserved word **new** makes a new object instance, and call its constructor
 - Full name of constructor is given after **new**; since constructors have same name as their classes, this specifies the class to instantiate
- Statements defined in constructor are executed when the constructor is called

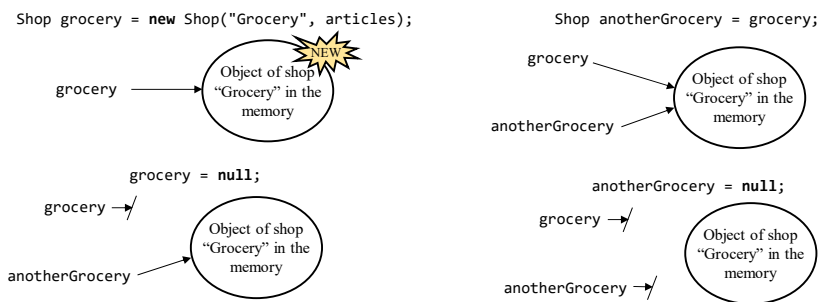
Constructor

- **Constructor** is a special method that is called whenever a class is instantiated (created)
 - Another object sends a message that calls a constructor
 - A constructor is the first message an object receives and cannot be called subsequently
 - **Establishes initial state of properties** for the object instance
 - Constructor has always the same name as class and has no return type
- If you do not define any constructors for a class, the default constructor is called
 - Default constructor will initialize each instance variable to its default value
 - This is not a good idea – always write your own constructor for each class and always give each instance variable a value

Constructors overloading

- It's common to **overload** constructors
 - Define multiple constructors which differ in number and/or types of parameters
- The compiler determines which constructor to call based on the number and the type of arguments in a call statement

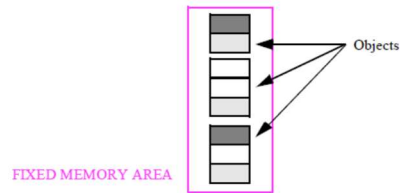
Memory management



- Remember, you are always working with references!
- There are three commonly found modes of memory management
 - **Static**
 - **Stack-based**
 - **Free (heap-based)**

Static memory mode

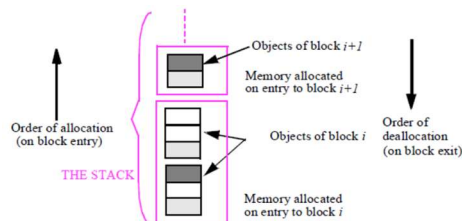
- An entity may become attached to at most one run-time object during the entire execution of the software
 - Allocate space for all objects (and attach them to the corresponding entities) once and for all, at program loading time or at the beginning of execution



- Problems
 - Recursion is permitted
 - Cannot use dynamic data structures

Stack-based mode

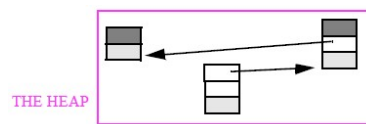
- An entity may at run-time become attached to several objects in succession, and the run-time mechanisms allocate and deallocate these objects in last-in, first-out order
 - When an object is deallocated, the corresponding entity becomes attached again to the object to which it was previously attached, if any



- Problem
 - Still cannot use dynamic data structures

Free (heap-based) mode

- An entity may become successively attached to any number of objects; the pattern of object creations is usually not predictable at compile time
 - The free mode allows us to create the sophisticated dynamic data structures

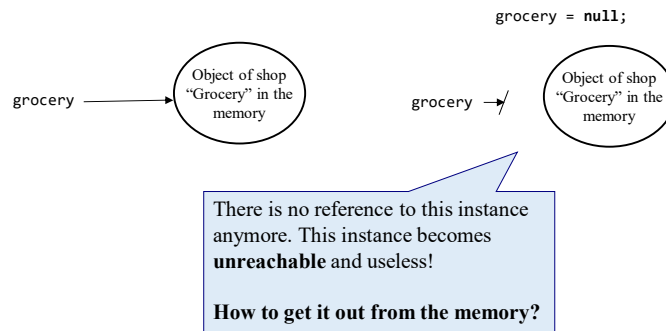


Space reclamation

- The ability to create objects dynamically, raises the question of what to do when an object becomes unused
 - Is it possible to reclaim its memory space, so as to use it again for one or more new objects in later creation instructions?
- Static mode
 - No problem with deallocation of objects
- Stack-based mode
 - Block-structured language make things particularly simple: object allocation occurs at the same time for all entities declared in a given block, allowing the use of a single stack for a whole program (remember local variables in C language)
- Free mode
 - The pattern of object creation is unknown at compile time it is not possible to predict when a given object may become useless
 - For example in C language programmers are forced to use `malloc()` and `free()` to allocate and deallocate the memory

Reachability problem

- We can easily get an unreachable problem after detachment of the reference



Memory management problem in object-oriented model

- The problem of memory management arises from the unpredictability of the operations which affect the set of reachable instances: creation and detachment
 - Such a prediction is possible in some cases, for data structures managed in a strictly controlled way (e.g. remember linked list in C language)
- Three general attitudes are possible as to objects that become unreachable
 - **Casual approach** – Ignore the problem and hope that there will be enough memory to accommodate all objects, reachable or not (hardly usable!)
 - **Manual reclamation** – Ask developers to include in every application an algorithm that looks for unreachable objects, and give them mechanisms to free the corresponding memory (e.g. C++ with command **delete**)
 - **Automatic garbage collection** – Include in the development environment (as part of the so-called runtime system) automatic mechanisms that will detect and reclaim unreachable objects (e.g. Java, C#)

Take care about memory

- *“I say a big NO! Leaving an unreferenced object around is BAD PROGRAMMING. Object pointers ARE like ordinary pointers – if you [allocate an object] you should be responsible for it, and free it when its finished with (didn't your mother always tell you to put your toys away when you'd finished with them?).” (I. Stephenson, 1991)*
- *“An object-oriented program without automatic memory management is roughly the same as a pressure cooker without a safety valve: sooner or later the thing is sure to blow up!” (M. Schweitzer, L. Strether, 1993)*