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

 *Edsger Dijkstra*

OBJECT-ORIENTED PROGRAMMING

Class and Object

Lecture #1

TECHNICAL UNIVERSITY OF KOŠICE

Programming concepts

- Structured programming
  - Data structures (array, record)
  - Program flow structures (sequence, selection, iteration)
  - Subroutines (macros, procedures, functions)
- 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

Object orientation

- Abstraction
  - How to deal with complexity?
  - Data abstraction: Abstract Data Types (ADT)
- Encapsulation
  - How to define the scope of data inside data structures (objects)?
- Information hiding
  - How to protect parts of the program from extensive modification based on design changes?
  - Separation of interfaces from implementation
- Inheritance
  - How to reuse existing code with little or no modification?
  - Data structures can be based on other data structures
- Polymorphism
  - How to work with variables (objects) using different types?
  - The program does not have to know the exact type of the object, and so the exact behavior is determined at run time

Abstraction

- "*Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences.*" (T. Hoare)
- "*An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.*" (G. Booch)
- Abstraction is 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 OO programming

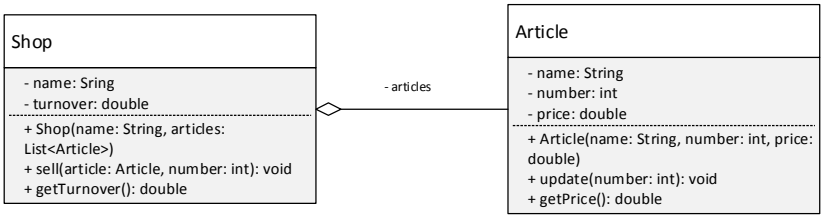
- **Class**
  - Defines abstract characteristics of things (objects)
  - It includes properties (data fields) and capabilities (functions and procedures)
- **Object**
  - One particular version (exemplar) of the class with concrete version of properties

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, number of pieces, price
    - Capabilities: create new item, update article number (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 hidden and can be maintained only by the object of that class itself (**encapsulation**)

Object example: Implementation

- For the implementation we will use Java

```
public class Shop {
  private String name;
  private List<Article> articles;
  private double turnover;

  public Shop(String name, List<Article> articles) {
    this.name = name;
    this.articles = articles;
    this.turnover = 0.0;
  }

  public void sell(Article article, int number) {
    article.update(-number);
    this.turnover += article.getPrice() * number;
  }

  public double getTurnover() {
    return this.turnover;
  }
}
```

Object example: Implementation

```
public class Article {
  private String name;
  private int number;
  private double price;

  public Article(String name, int number, double price) {
    this.name = name;
    this.number = number;
    this.price = price;
  }

  public void update(int number) {
    this.number += number;
  }

  public double getPrice() {
    return this.price;
  }
}
```


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
  - Objects are smart – they "know" how to do things
  - An object gets something done only if some other object tells it to use one of its capabilities
- 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

Object properties (state)

- **Properties** determine how an object acts
  - Some properties are constant, others variable
  - Properties themselves are objects – they also can receive messages; e.g., articles in the shop
- 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 of an object's properties (changes if any property changes)
  - Some properties do not change, e.g., name of the article
  - Others do, e.g., price 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

Class

- **Class**
  - A class is a category of object
  - Defines capabilities and properties common among a set of individual objects
    - All shops can sell articles
  - Defines template for making **object instances**
    - Particular shops may have different names, different articles in the stock, etc.
- Classes implement capabilities as **methods**
  - A method is a sequence of statements (in programming language – functions and procedures)
  - Objects cooperate by sending "messages" to each other (in programming language – function calls or procedure calls)
- Classes implement properties as **instance variables**
  - Slot of memory allocated to the object that can hold a potentially changeable value, e.g., price of the article

Class in Java

```
public class Shop {
  private String name;
  private List<Article> articles;
  private double turnover;

  public Shop(String name, List<Article> articles) {
    this.name = name;
    this.articles = articles;
    this.turnover = 0.0;
  }

  public void sell(Article article, int number) {
    article.update(-number);
    this.turnover += article.getPrice() * number;
  }

  public double getTurnover() {
    return this.turnover;
  }
}
```

Class declaration tells Java compiler that we are about to define a new class
Class definition following a declaration (using curly braces) tells the Java compiler what it means to make an instance of this class and how that instance will respond to messages

Instance variables which implement object properties

Methods which implement object capabilities

Object instance

- **Object instances** (or just **instances**) are individual objects made from class template
  - Can have many object instances of the same class
  - All will have the same attributes, but may have different values for them
  - For example two object instances of `Article` both will have a price, but one may be 1.00 €, and one may be 9.99 € however, each is still an `Article`
  - The process of creating an object instance is called **instantiating** an object
- Individual instances have individual identities (separate entities in the memory)
  - This allows other objects to send "messages" to given objects (call their methods)
  - Each instance is unique, even though they all have the same capabilities
- Note: Unlike other programming languages (as C), in Java, the value of an instance variable is always a reference to an object instance, not an instance itself
  - Dynamic memory allocation in C is very similar
- A **reference** is the address in memory where the instance is stored
  - Also called a **pointer**

Instance in Java

```
public class ShopApp {

  public static void main(String[] args) {

    List<Article> articleList = new Vector<Article>();

    Article article1 = new Article("Gin", 100, 10.25);
    Article article2 = new Article("Whisky", 150, 12.40);

    artilceList.add(article1);
    artilceList.add(article2);

    Shop shop = new Shop("Alco", articleList);

    shop.sell(article1, 2);

    System.out.println("Shop turnover is " + shop.getTurnover());
  }

}
```

Object instance `articleList` of type `List<Article>` based on class `Vector`

Object instances `article1` and `article2` of type `Article` based on class `Article`

Object instance `shop` of type `Shop` based on class `Shop`

Two bottles of Gin were sold

Object instantiation

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

```
Shop shop = new Shop("Alco", articleList);
```

- 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
  - in this case, a new instance of `Shop` class is created
- 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

```
public class Shop {
  private String name;
  private List<Article> articles;
  private double turnover;

  public Shop() {
    this.name = "";
    this.articles = new Vector<Article>();
    this.turnover = 0.0;
  }

  public Shop(String name, List<Article> articles) {
    this.name = name;
    this.articles = articles;
    this.turnover = 0.0;
  }
  ...
}
```

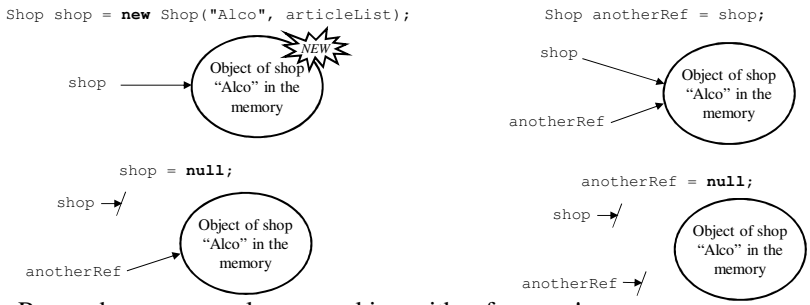
Constructor has no arguments. It provides default values for all instance variables.

Constructor has two arguments. It provides initial values for all instance variables given by arguments.

- The compiler determines which constructor to call based on the number and the type of arguments

```
Shop shop1 = new Shop();
Shop shop2 = new Shop("Kwik-E-Mart", articleList);
```

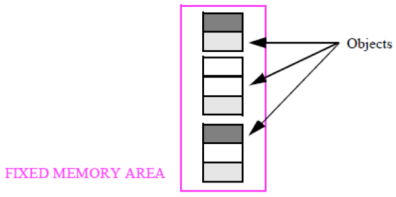

Memory management



- Remember, you are always working with references!
- There are three commonly found modes
  - Static
  - Stack-based
  - Free

Static memory mode

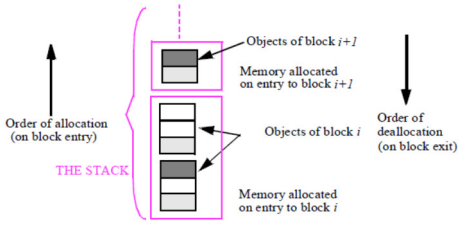
- An entity may become attached to at most one run-time object during the entire execution of the software
  - Allocate space for all objects (and attach them to the corresponding entities) once and for all, at program loading time or at the beginning of execution



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

Stack-based mode

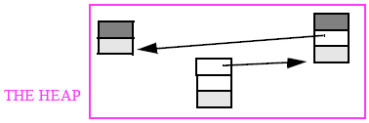
- An entity may at run time become attached to several objects in succession, and the run-time mechanisms allocate and deallocate these objects in last-in, first-out order
  - When an object is deallocated, the corresponding entity becomes attached again to the object to which it was previously attached,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

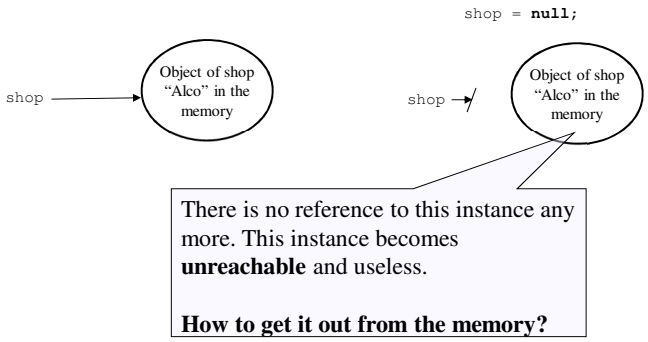


Space reclamation

- The ability to create objects dynamically (stack-based and free modes), 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)

Readings

- ECKEL, B.: *Thinking in Java*. 4th Edition, Prentice Hall, 2006
  - Introduction to Objects (pp. 15 – 40)
  - Everything Is an Object (pp. 107 – 144)