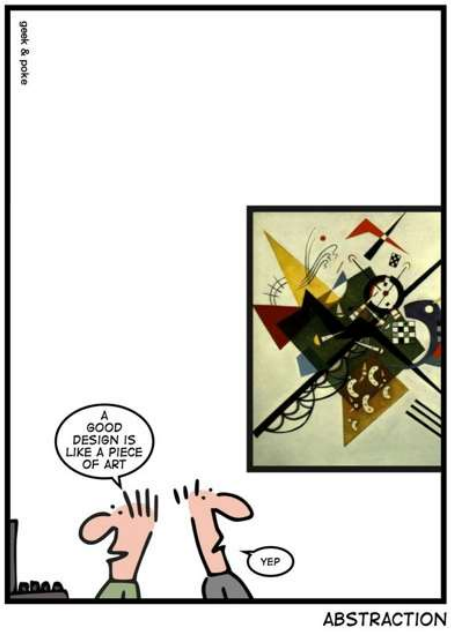


OBJECT-ORIENTED PROGRAMMING

State and Behavior of the Object

Lecture #2

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2024/2025

Encapsulation

- An abstraction focuses on the outside view of an object (**interface of the object**) and separates an object's behavior from its implementation
- Classes should be **opaque**
- Classes should **not expose** their internal implementation details

State and behavior of the object

- **Instance variables** of the class represent **state** of the object
  - Internal property which should be controlled only by the object itself
  - Manipulated only by the **behavior** of the object
- **Methods** of the class represent **behavior** of the object
  - Some behaviors are internal and controlled only within the object
  - Some behaviors represents interaction of the object with other objects
    - This forms the **interface of the object**

Practical information hiding

- General principle of information hiding: Use private members and appropriate public **accessors** and **mutators** (get/set methods) wherever possible
- Wrong implementation

```
public double price;
```

- Correct implementation

```
private double price;
public double getPrice() {
  return this.price;
}
public void setPrice(double price) {
  this.price = price;
}
```


Accessibility modificators

- Accessibility of each class member can be defined by the **accessibility modificators**
- **`public`**
  - Accessible from the outside of the object
  - Another object can freely access this member
- **`private`**
  - Accessible only from the internal members (methods) of the object and cannot be directly exposed to other objects
  - Private members are hidden inside the object

Using accessors and mutators

- You can put constraints on values

```
public void setPrice(double newPrice) {
  if (newPrice < 0.0) {
    sendErrorMessage(...);
    newPrice = Math.abs(newPrice);
  }
  this.price = newPrice;
}
```

- If users of your class accessed the fields directly, then they would each be responsible for checking constraints

Using accessors and mutators

- You can change your internal representation without changing the interface

```
public void setPriceInUSD(double newPrice) {
  this.price = convert(newPrice);
}
public void setPriceInEUR(double newPrice) {
  this.price = newPrice;
}
```

Using accessors and mutators

- You can perform arbitrary side effects

```
public void setPrice(double newPrice) {
  this.price = newPrice;
  notifyObservers();
}
```

- If users of your class accessed the fields directly, then they would each be responsible for executing side effects

Using the `this` keyword

- **`this`** is a reference to the current object (the object whose method or constructor is being called)
  - You can refer to any member of the current object from within an instance method or a constructor by using **`this`**
- Object-oriented programming languages implement reference to the current object
  - To access the shadowed class members
  - To have a reference to itself

Using `this` with instance variables

- The most common reason for using the **`this`** keyword is because an instance variable is shadowed by a method or constructor parameter
- For example, the `Article` class could be written like this

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

  public Article(String newName, int newQuantity, double newPrice) {
    name = newName;
    quantity = newQuantity;
    price = newPrice;
  }
  ...
}
```


Using `this` with instance variables

- But it has been written like this

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

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

- Each argument to the constructor shadows one of the instance variables – inside the constructor `price` is a local copy of the constructor's first argument
- To refer to the `Article` member `price`, the constructor must use `this.price`

Using `this` with constructor

- From within a constructor, you can also use the **`this`** keyword to call another constructor in the same class
  - Doing so is called an **explicit constructor invocation**

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

  public Article() {
    this("", 0, 0.0);
  }

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


Using `this` with constructor

- There are two constructors for this class to initialize instance variables
- The no-argument constructor provides default values for all instance variables
  - Because initial values are not provided by arguments
  - The no-argument constructor calls the three-arguments constructor with three default values
- If present, the invocation of another constructor **must be the first line** in the constructor

Tends to complicate

- The existence of **static** members tends to break up the simple structures that we have discussed in previous lectures
- Static memory mode contrasts with heap memory mode
  - Similar in other languages (e.g. C, static variables vs. dynamic memory allocation)
- Support of **global variables** and **global methods** that can be accessed without creating objects of a class

Class members

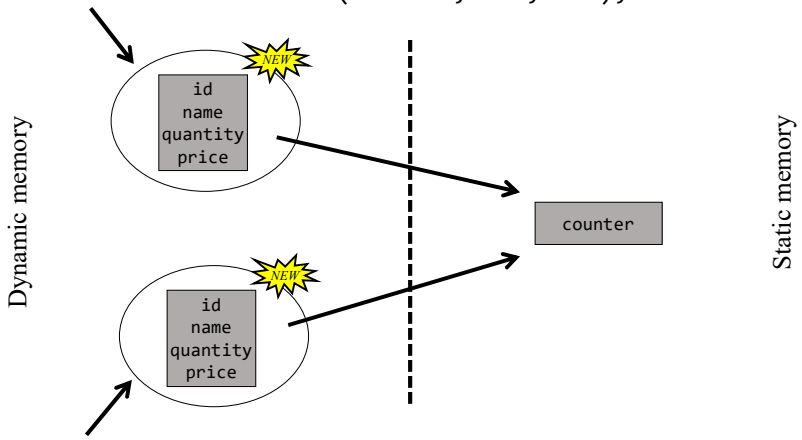
- Static variables
  - Use **`static`** keyword to define a static variable of a class
  - Static variable uses static memory mode
- Static methods
  - Use **`static`** keyword to define a static method of a class
  - Static method can be called directly, out of any class instances

Why using static variables?

- Static variable is like a **global variable**
- Value of the static variable is stored in static memory which is common for all objects of the class
  - Java creates only one copy for a static variable which can be used even if the class is never instantiated
  - Encapsulation is not broken, it is common only for objects of the class
- This feature is useful when we want to create a variable common to all instances of a class
- Example
  - One of the most common example is to have a variable that could keep a count of how many objects of a class have been created

Static variables – Example

```
Article bread = new Article("Bread", 100, 1.5);
```




```
Article milk = new Article("Milk", 150, 2.0);
```

Instance vs. static variables

- Instance variables
  - One copy per **object**
  - Every object **has its own** instance variable, e.g. `id`, `name`, `quantity`, `price`
- Static variables
  - One copy per **class** (all instances)
  - Every object **use the same** static variable, e.g. `counter`

Constants in Java

- Static variables are mostly used as constants with **`final`** keyword in the declaration

```
public class MaxUnits {
  public static final int MAX_UNITS = 25;
}
```

- For example
  - `Math.PI`
  - `Math.E`
  - `Double.POSITIVE_INFINITY`
  - `Double.NEGATIVE_INFINITY`

Static methods

- A class can have methods that are defined as static (e.g. `main` method)
- Static methods can be accessed without using objects
  - Also, there is no need to create instances
- Static methods are generally used to group related **library functions** that don't depend on data members of its class
  - For example, `Math` library functions

Static methods restrictions

- They can only call other static methods
- They can only access static data
- They cannot refer to **`this`** or **`super`** in anyway

Guidelines for use of static variables

- Do not ever use static variables without declaring them final unless you understand exactly why you are declaring them static
  - Static final variables, or constants, are often very appropriate
- There are only few situations where the use of a non-final static variable (global variable) might be appropriate
  - One appropriate use might be to count the number of objects instantiated from a specific class
  - I suspect there are a few other appropriate uses as well
  - Always reduce the usage of global variables as much as possible

Guidelines for use of static methods

- Do not declare methods static if there is any requirement for the method to remember anything from one invocation to the next
  - There is no way to use the instance variables in static methods
- The method should probably also be self-contained
  - All information that the method needs to do its job should either come from incoming parameters or from final static member variables (constants)
- The method probably should not depend on the values stored in non-final static member variables, which are subject to change over time
  - A static method only has access to other static members of the class, so it cannot depend on instance variables defined in the class
  - An example of a static method is the `sqrt()` method of the `Math` class. This method computes and "returns the correctly rounded positive square root of a `double`" where the `double` value is provided as a parameter to the method. Each time the method is invoked, it completes its task and does not attempt to save any values from that invocation to the next. Furthermore, it gets all the information that it needs to do its job from an incoming parameter

Example: Implementation of Singleton design pattern

- **Singleton** is a design pattern that restricts the instantiation of a class to one object
  - This is useful when exactly one object is needed to coordinate actions across the system

```
public class MyClass {
  private static final MyClass uniqueInstance = new MyClass();

  ...

  private MyClass() { ... }

  public static MyClass getInstance() {
    return uniqueInstance;
  }

  ...
}
```