Object Oriented System Design

Omid Abbaszadeh

University of Zanjan
o.abbaszadeh@znu.ac.ir

Syllabus

- Introduction (1 session)
- Object oriented computing concepts (2 sessions)
  - Abstract data types, classes, methods
  - Message passing
  - Inheritance, Polymorphism, Dynamic binding
- A review of UML (1 session)
- Software Development Methodologies (2 sessions)
- Principle of object-oriented design (3 sessions)
  - Basic principles
  - GRASP patterns
  - Design by Contract
- GoF patterns and MVC pattern (5 sessions)
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- Software testing (4 sessions)
  - Acceptance testing, Integration testing
  - Unit testing, Functional testing
  - Performance testing, Stress testing, Usability testing
- Refactoring and Code Reviews (3 sessions)

Introduction

- Definition
- Seminal works
- Motivations
- A brief history of Object Oriented
- Programming paradigms
  - Functional Programming (FP)
  - Event-Driven Programming (EDP)
  - Reactive Programming (RP)
  - Aspect-Oriented Programming (AOP)
  - Data-Oriented Design (DOD)
- Future

Definitions

Definition (Martin Fowler)

Object-oriented programming is a paradigm that encourages **modular design, separation of concerns, and code reusability** through the **use of objects, classes, and inheritance**.

Definitions

Definition (Rebecca Wirfs-Brock)

Object-oriented programming is a way of **decomposing a problem into a set of objects that interact** to solve the problem. Each object represents some **entity, physical or abstract, and has a set of responsibilities to perform**.

Definitions

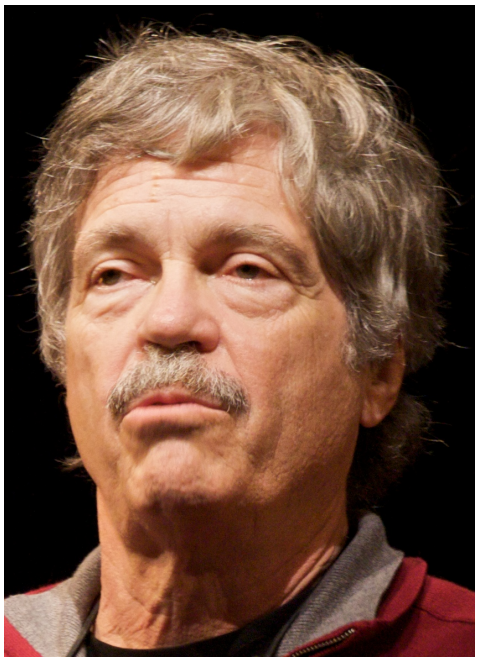
Definition (James Gosling)

Object-oriented programming is a programming methodology that focuses on objects and the interactions between them. It is based on the **principles of encapsulation, inheritance, and polymorphism.**

Definitions

Definition (Alan Kay)

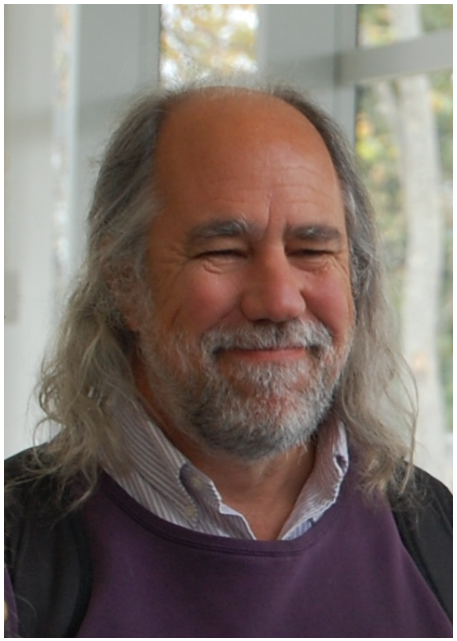
Object-oriented programming to me means only **messaging, local retention, and protection and hiding of state-process, and extreme late-binding of all things**.



Definitions

Definition (Grady Booch)

Object-oriented programming is about organizing code in a way that **reflects the real world, with objects that have properties and methods**.



Definitions

Definition (Barbara Liskov)

Object-oriented programming is a way of modeling real-world objects as software objects that interact with each other, where **each object has its own state and behavior**.

Introduction

- Definition
- Seminal works
- A brief history of Object Oriented
- Motivations
- Programming paradigms
  - Functional Programming (FP)
  - Event-Driven Programming (EDP)
  - Reactive Programming (RP)
  - Aspect-Oriented Programming (AOP)
  - Data-Oriented Design (DOD)
- Future

Seminal work

1968 - Ole-Johan Dahl - Kristen Nygaard – **Truring award winners 2001**

Dahl and Nygaard at the time of Simula's development

The languages that Dahl and Nygaard developed together were, first a system simulation language called **SIMULA** (1962-1968), now usually referred to as SIMULA67.

System = set of interacting **quasiparallel processes** $\hookrightarrow$ Co-routine

SIMULA

SIMULA contains the core of the concepts now available in mainstream object-oriented languages such as C++, Eiffel, Java, and C#:

1. **Class and object**. The class concept as a *template* for creating instances (objects).
2. **Subclass**. Classes may be organized in a *classification hierarchy* by means of subclasses.
3. **Virtual methods**. A SIMULA class may define virtual methods that can be *redefined* in subclasses.
4. **Active objects**. An object in SIMULA may be the *head of an active thread*; technically it is a **co-routine**.

SIMULA

SIMULA contains the core of the concepts now available in mainstream object-oriented languages such as C++, Eiffel, Java, and C#:

5. **Processes and schedulers**. SIMULA makes it easy to write *new concurrency abstractions, including schedulers*.
6. **Frameworks**. SIMULA provided the first object-oriented framework in the form of Class Simulation—the mechanism it used to implement its simulation.
7. **Automatic memory management**, including *garbage collection*.

SIMULA - Virtual method

Definition

A virtual method is a declared class method that allows **overriding** by a method with the same derived class **signature**.

Virtual method $\Rightarrow$ ***Run-Time Polymorphism*** *(*$\neq$ *Compile-Time)*
Virtual method $\Rightarrow$ ***Late binding*** *(*$\neq$*Early binding)*

```java
public class Shape {
    public void draw() {
        System.out.println("Drawing a shape.");
    }

    public String toString() {
        return "This is a shape.";
    }
}
```

```java
public class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing a circle.");
    }

    public String toString() {
        return "This is a circle.";
    }
}
```

The main benefit of virtual methods is that they enable the use of **interfaces and abstract classes**, which are important concepts in many object-oriented programming languages.

SIMULA - Virtual methods (Today)

- In Java, **all non-static methods are virtual by default**
- In Python, **all methods are virtual by default**
- In C#, virtual methods are declared using the **virtual keyword**, and can be overridden in derived classes using the **override keyword**.
- In TypeScript, virtual methods can be implemented in classes using the **override keyword**.

SIMULA - Virtual method

C#

```
public class Shape {
    public virtual void Draw() {
        Console.WriteLine("Drawing a shape.");
    }
}
```

```
public class Circle : Shape {
    public override void Draw() {
        Console.WriteLine("Drawing a circle.");
    }
}
```

Typescript

```
class Animal {
  public makeSound(): string {
    return "The animal makes a sound";
  }
}
```

```
class Cat extends Animal {
  public override makeSound(): string {
    return "Meow";
  }
}
```

SIMULA - Active Object

Traditional OOP encapsulation, as provided by C++, C# or Java, **does not really encapsulate anything in terms of *concurrency***.

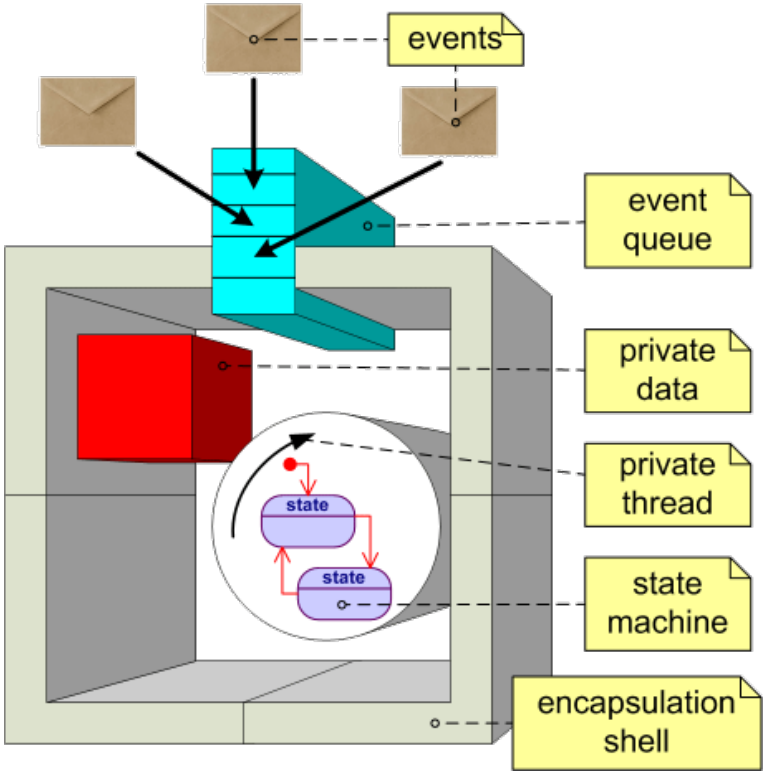
```
class MyClass {
    private double val;

    MyClass(double val) {
        this.val = val;
    }

    void setVal(double val) {
        this.val = val;
    }
}
```

Is the above code *thread-safe*?

Race Condition

SIMULA - Active Object Design Pattern

Active Object is a concurrency pattern in which we try to separate the **invocation** of a method from its **execution**.
True Encapsulation for Concurrency:



An active object provides **synchronous methods executions** and the **method invocations in an asynchronous way**.

SIMULA - Active Object Design Pattern

True Encapsulation for Concurrency

```java
import java.util.concurrent.ForkJoinPool;

public class MyClass {
    2 usages
    private double val;
    // Worker thread pool for executing tasks asynchronously
    // The pool size is set to 1 to ensure that the tasks are executed in a sequential order
    // asyncMode=true means our worker thread processes its local task queue in the FIFO order
    // only single thread may modify internal state
    2 usages
    private final ForkJoinPool fj = new ForkJoinPool( parallelism: 1, ForkJoinPool.defaultForkJoinWorkerThreadFactory, handler: null, asyncMode: true);

    // Implementation of active object method
    public void doSomething() throws InterruptedException {
        // Execute the task asynchronously using the worker thread pool
        fj.execute(() -> {
            // Update the state of the active object
            val = 1.0;
        });
    }

    // Implementation of active object method
    public void doSomethingElse() throws InterruptedException {
        // Execute the task asynchronously using the worker thread pool
        fj.execute(() -> {
            // Update the state of the active object
            val = 2.0;
        });
    }
}
```

Introduction

- Definition
- Seminal works
- A brief history of Object Oriented
- Motivations
- Programming paradigms
  - Functional Programming (FP)
  - Event-Driven Programming (EDP)
  - Reactive Programming (RP)
  - Aspect-Oriented Programming (AOP)
  - Data-Oriented Design (DOD)
- Future

A brief history

1. **Simula**
2. **Smalltalk**: In the 1970s, a team at Xerox PARC led by Alan Kay developed Smalltalk, an object-oriented programming language that popularized the use of **graphical user interfaces (GUIs)**
3. **C++**: In the early 1980s, Bjarne Stroustrup developed C++, an extension of the C programming language.
4. **Java**: In the mid-1990s, James Gosling and his team at Sun Microsystems developed Java, a **platform-independent programming language** that used object-oriented programming
5. **C#**: In the late 1990s, Microsoft developed C#, an object-oriented programming language that was designed to be **easy to learn and use, and to integrate with the Microsoft .NET Framework.**

Introduction

- Definition
- Seminal works
- A brief history of Object Oriented
- Motivations
- Programming paradigms
  - Functional Programming (FP)
  - Event-Driven Programming (EDP)
  - Reactive Programming (RP)
  - Aspect-Oriented Programming (AOP)
  - Data-Oriented Design (DOD)
- Future

Motivations - Advantages

1. **Code reuse**: OOP allows developers to create **reusable software components**, or objects. This helps to **reduce development time and costs**, and makes it easier to **maintain software systems over time**.

This is achieved through the use of

- **Inheritance**
- **Composition**

Motivations - Advantages

2. **Abstraction**: OOP enables developers to model complex **real-world systems using abstract concepts and classes**, which can be used to represent a wide range of objects and behaviors.

Abstraction is the process of hiding complex implementation details and only exposing necessary information to the user.

```
public interface Vehicle {
    void start();
    void stop();
    double getFuelLevel();
}
```

Motivations - Advantages

2. **Polymorphism**: OOP enables developers to create objects that can **behave in different ways depending on the context in which they are used**.

```java
public class Animal {
    public void makeSound() {
        System.out.println("Generic animal sound");
    }
}

public class Dog extends Animal {
    public void makeSound() {
        System.out.println("Bark!");
    }
}
```

Motivations - Advantages

3. **Encapsulation**: OOP allows developers to encapsulate **data and behavior within objects**, which helps to protect the integrity of data and **prevent unwanted access or modification**.

Encapsulation hides the internal workings of an object and only exposing a public interface for interaction with the object.

```
public class BankAccount {
    private String accountNumber;
    private double balance;

    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
```

Motivations - Advantages

4. **Modularity**: OOP encourages developers to **break down software systems into smaller**, more manageable modules or objects, which **can be developed and tested independently**.

This helps to improve code quality and reduce the risk of errors or bugs in the system.

Motivations - Advantages

5. **Improved collaboration**: OOP provides a common language and **set of design patterns that can be used across teams and organizations**. This helps to improve collaboration and communication between developers, and makes it **easier to maintain and update code over time**.

Motivations - Advantages

6. **Scalability**: OOP provides a flexible and adaptable approach to software design and development, which makes it easier to scale software systems up or down as needed. This is particularly important for large-scale projects or applications that may need to handle a high volume of data or users.

Disadvantages

1. **Complexity**: OOP can lead to complex and convoluted code, especially for large-scale projects or applications. This can make it more difficult to understand and debug code and can lead to longer development times and higher costs.

Example:

- Complex inheritance hierarchy

Disadvantages

1. **Steep learning curve**: OOP can be more difficult to learn and master than other programming paradigms, especially for new programmers who are not familiar with the concepts of abstraction, encapsulation, and inheritance.

Disadvantages

1. **Performance overhead**: OOP can introduce a performance overhead due to the **use of dynamic binding, virtual functions, and other features that may require additional processing time and memory**.

- Virtual Method Table (VMT)

Disadvantages

1. **Tight coupling**: OOP can lead to tight coupling between objects, which can make it **more difficult to modify or update the system without affecting other parts of the code**. This can lead to software that is difficult to maintain and update over time.

Disadvantages

1. **Over-reliance on inheritance**: OOP can encourage developers to overuse inheritance, which can lead to code that is tightly coupled and difficult to modify. This can also lead to issues with code duplication and poor code reuse.

Introduction

- Definition
- Seminal works
- A brief history of Object Oriented
- Motivations
- Programming paradigms
  - Functional Programming (FP)
  - Event-Driven Programming (EDP)
  - Reactive Programming (RP)
  - Aspect-Oriented Programming (AOP)
  - Data-Oriented Design (DOD)
- Future

Future

1. **Increased use of functional programming techniques**: Functional programming and OOP are not mutually exclusive, and many modern programming languages allow for a hybrid approach.

In Java 8:

- lambdas
- streams

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

List<Integer> evenNumbers = numbers.stream()
                                   .filter(n -> n % 2 == 0)
                                   .map(n -> n * 2)
                                   .collect(Collectors.toList());

System.out.println(evenNumbers); // [4, 8, 12]
```

Future

1. **More emphasis on distributed systems and concurrency**: As more applications move to the cloud and distributed systems become more prevalent, OOP languages may need to adapt to better support these environments.

```java
public class BankAccount {
    private double balance;

    public synchronized void deposit(double amount) {
        balance += amount;
    }

    public synchronized void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

Future

1. **Greater focus on security and privacy**: With the increasing importance of data security and privacy, OOP languages may need to incorporate new security features and programming paradigms. This could include better support for encryption, secure coding practices, and more robust error handling.

Future

1. **Continued evolution of the Java ecosystem**: Java remains one of the most widely used OOP languages, and the Java ecosystem is constantly evolving to meet changing demands.