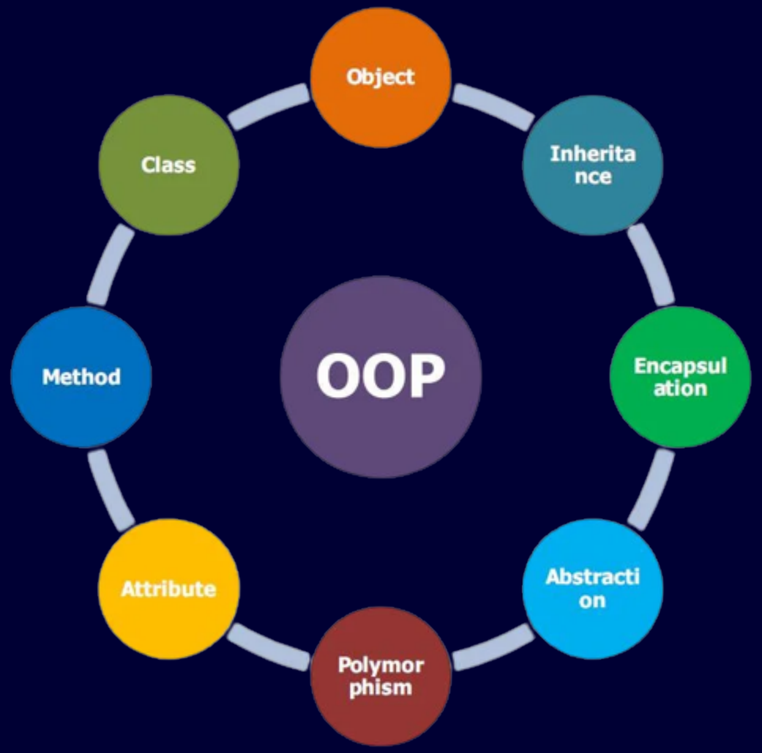




# Введение в ООП

# Что это такое?

Объектно-ориентированное программирование (ООП) - это способ написания программ, который помогает организовать код в более логичные и удобные для понимания структуры. Основная идея заключается в том, чтобы рассматривать элементы программы, такие как данные и действия с ними, как "объекты". Эти объекты могут взаимодействовать друг с другом, что делает код более понятным и легким в сопровождении. ООП позволяет создавать программы, которые моделируют реальные объекты и процессы, что делает их более интуитивными и удобными для разработчиков.

# А если простым языком?

- ИЗ ЧЕГО СОСТОИТ ЧЕЛОВЕК?
- ЧТО ОН МОЖЕТ ДЕЛАТЬ?
- КАК МЕНЯЕТСЯ ЧЕЛОВЕК, КОГДА ВЗРОСЛЕЕТ?

- ИЗ ЧЕГО СОСТОИТ АВТО?
- ЧТО МОЖЕТ ДЕЛАТЬ АВТОМОБИЛЬ?
- КАКИЕ КОМПЛЕКТУЮЩИЕ АВТО МЕНЯЮТСЯ?

# Преимущества использования ООП

- ✓ **Упрощение понимания**: ООП позволяет структурировать код, организовывая его в логические "объекты", которые представляют реальные объекты и сущности. Это делает код более понятным для разработчиков, так как они могут легко представить, какие действия выполняют объекты.
- ✓ **Модульность**: Код разбивается на маленькие, независимые модули (классы), каждый из которых выполняет определенную задачу. Это упрощает сопровождение, тестирование и расширение кода, так как изменения в одном модуле редко влияют на другие.
- ✓ **Повторное использование кода**: Вы можете создавать классы и объекты, которые могут быть использованы в разных частях вашей программы или даже в других проектах. Это экономит время и усилия, так как вы можете использовать уже написанный и отлаженный код.
- ✓ **Сокрытие деталей реализации**: ООП позволяет скрыть сложные детали реализации объектов от внешнего кода. Это повышает безопасность и предотвращает случайные изменения данных.
- ✓ **Интерфейсы и абстракции**: ООП позволяет создавать четкие интерфейсы для работы с объектами. Это упрощает взаимодействие между разными частями программы и снижает вероятность ошибок.
- ✓ **Подходит для моделирования реального мира**: ООП естественным образом подходит для моделирования объектов и событий в реальном мире, что делает его эффективным для разработки приложений, связанных с реальными объектами, такими как игры, системы управления и многие другие.

# Недостатки использования ООП

- **Сложность**: ООП может привести к избыточной сложности кода, особенно если неудачно проектировать классы и их иерархии. Проектирование слишком глубокой иерархии классов может сделать код трудным для понимания и сопровождения.
- **Избыточность**: В некоторых случаях ООП может привести к избыточности кода, так как требует создания множества классов и методов, что может быть неэффективным для простых приложений.
- **Производительность**: В некоторых случаях ООП может вызывать небольшие накладные расходы по сравнению с процедурным программированием из-за дополнительных операций с объектами и вызовов методов.
- **Сложность отладки**: Отладка объектно-ориентированного кода может быть более сложной из-за сложных взаимосвязей между объектами и методами. Неправильное использование наследования или полиморфизма может вызвать трудноуловимые ошибки.
- **Инкапсуляция не всегда соблюдается**: Инкапсуляция, один из принципов ООП, иногда может быть нарушена, что может привести к неправильному доступу к данным и методам объекта.
- **Объем памяти**: Некоторые реализации ООП могут потреблять больше памяти из-за накладных расходов на хранение информации о классах и объектах.
- **Сложность обучения**: ООП может быть сложным для понимания начинающим программистам из-за абстрактных концепций, таких как наследование и полиморфизм.
- **Не всегда подходит**: ООП не является универсальным решением и не всегда подходит для всех типов приложений. Для некоторых задач процедурное программирование или другие парадигмы могут быть более подходящими.

# Что такое структуры?

- **Структуры (structures) -** это один из базовых элементов языков программирования, предназначенных для организации данных. Они позволяют объединять несколько переменных разных типов в один комплексный тип данных.
- В структурах данные называются "полями" или "членами" (members), и они могут иметь разные типы данных, включая числа, строки, другие структуры или даже указатели на функции. Структуры помогают создавать пользовательские типы данных, которые могут представлять собой набор связанных информационных элементов.

**Структуры служат для более удобной и организованной работы с данными, представления объектов и улучшения структуры кода.**

# Более подробно

- **Группировка данных:** Структуры позволяют объединить несколько переменных разных типов данных в одну логическую единицу. Это помогает структурировать данные и сделать код более организованным.
- **Представление объектов:** Структуры могут использоваться для моделирования реальных объектов, сущностей или данных. Например, структура `Person` может представлять информацию о человеке, включая имя, возраст и адрес.
- **Повышение читаемости:** Используя структуры, вы можете создавать более понятный и выразительный код. Члены структуры имеют семантические имена, что делает их использование более понятным для разработчиков.
- **Передача данных:** Структуры могут использоваться для передачи группы данных как аргументы в функции или методы. Это удобно, когда нужно передать множество связанных значений.
- **Уменьшение глобальных переменных:** Структуры позволяют уменьшить использование глобальных переменных, что способствует изоляции данных и уменьшает вероятность конфликтов и ошибок.
- **Создание пользовательских типов:** Структуры позволяют создавать пользовательские типы данных, что делает код более абстрактным и гибким.
- **Организация данных в коллекции:** Структуры могут быть использованы для создания элементов коллекций, таких как массивы или списки, где каждый элемент представляет собой группу данных.

# Пример структуры

Пример структуры на языке C++:

Не имеет встроенный инициализатор (20-)

```
struct Person {
    string name;
    int age;
    double height;
};
```

```
Person person1;
person1.name = "John";
person1.age = 30;
person1.height = 175.5;
```

Пример структуры на языке Swift:

Имеет встроенный инициализатор

```
struct Person {
    var name: String
    var age: Int
    var height: Double
}
```

```
var person1 = Person(name: "John", age: 30, height: 175.5)
print(person1.name)    // Выводит "John"
print(person1.age)     // Выводит 30
print(person1.height)  // Выводит 175.5
```

# Что такое классы?

- **Абстракция объектов:** Классы позволяют абстрагировать (моделировать) реальные объекты и сущности, представляя их с помощью данных и методов.
- **Организация данных и функциональности:** Классы группируют данные (переменные) и функциональность (методы) в одну логическую единицу, что способствует четкой структуре кода.
- **Инкапсуляция:** Классы обеспечивают уровень инкапсуляции, скрывая детали реализации и предоставляя интерфейс для взаимодействия с объектами.
- **Повторное использование кода:** Классы позволяют создавать шаблоны объектов, которые можно повторно использовать в различных частях программы или в разных проектах.
- **Наследование:** Классы поддерживают механизм наследования, который позволяет создавать новые классы на основе существующих, наследуя их свойства и методы.
- **Полиморфизм:** Классы могут использовать полиморфизм, что позволяет объектам классов вести себя по-разному при общем интерфейсе.
- **Улучшение читаемости и сопровождаемости:** Использование классов делает код более понятным и облегчает его сопровождение и расширение.
- **Модульность:** Классы способствуют разбиению кода на модули, что улучшает организацию проекта и совместную разработку.

**Классы служат для моделирования объектов и данных, обеспечивают структурирование кода и улучшают его читаемость и гибкость.**

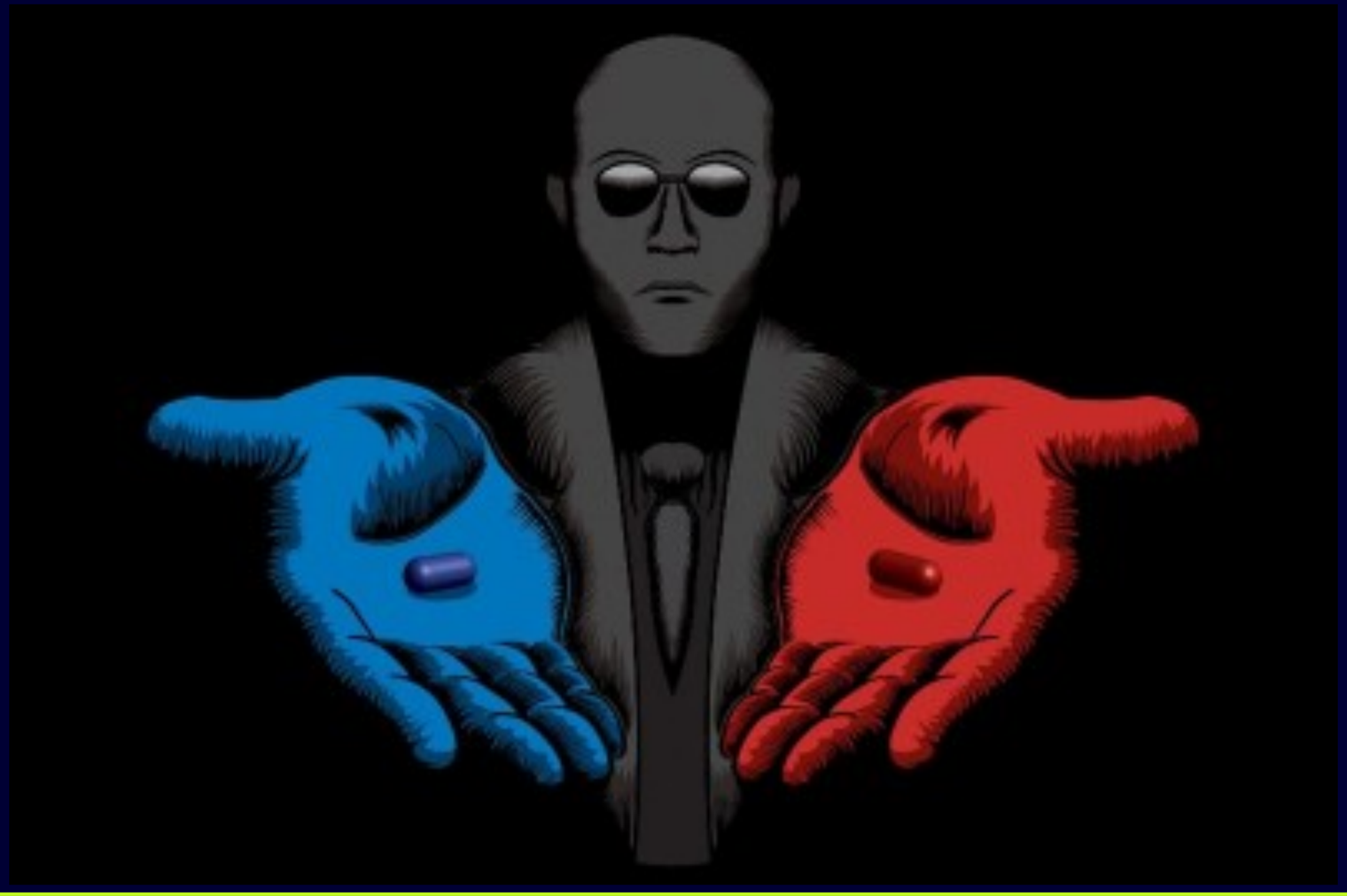
# Используйте классы, когда:

- **Требуется управление состоянием:** Если вам нужно создавать объекты, которые имеют внутреннее состояние, которое может изменяться, используйте классы. Классы поддерживают инкапсуляцию и управление данными объекта.
- **Есть сложная логика и методы:** Если ваши объекты должны обладать сложной функциональностью или методами, классы предоставляют пространство для реализации такой логики.
- **Требуется наследование и полиморфизм:** Если вам нужно использовать наследование для создания иерархии классов и реализации полиморфизма, классы являются более подходящим выбором.
- **Создание объектов с длительным временем жизни:** Классы лучше подходят для объектов, которые существуют в течение продолжительного времени и управляются динамически.

# Используйте структуры, когда:

- **Простые данные:** Если вам нужно представить простые данные или структуры данных без сложной логики и методов, структуры подходят лучше.
- **Копирование и передача значений:** Структуры копируются при передаче и присваивании, что может быть полезно, когда требуется избежать изменений в исходных данных.
- **Неизменяемые данные:** Если данные внутри объекта не должны изменяться после создания, структуры могут обеспечить неизменяемость данных.
- **Оптимизация по памяти:** Структуры могут быть эффективными с точки зрения использования памяти, особенно для небольших объектов.
- **Создание объектов с коротким временем жизни:** Структуры могут быть полезными для создания объектов, которые существуют только в пределах ограниченной области видимости.

# Так, что же тогда выбрать?



# Всё просто

В БОЛЬШИНСТВЕ СОВРЕМЕННЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ, ВКЛЮЧАЯ C++, C#, И SWIFT, КЛАССЫ И СТРУКТУРЫ МОГУТ БЫТЬ ИСПОЛЬЗОВАНЫ ВМЕСТЕ, И ВЫБОР ЗАВИСИТ ОТ КОНКРЕТНЫХ ТРЕБОВАНИЙ ВАШЕГО ПРОЕКТА!

Если заинтересованы в дисциплине, напишите на почту, чтобы получить лекции и дополнительные материалы!

**rvoltigo@gmail.com**