Объектно-ориентированное программирование в Julia

Перцев Дмитрий, Швачко Никита

группа 5030102/20202

Санкт-Петербургский политехнический университет Петра Великого

28 октября 2025 г.

Что такое объектно-ориентированное программирование?

Определение: Подход к разработке, где программа строится из объектов, объединяющих данные (состояние) и функции (поведение).

Основные принципы:

- **Инкапсуляция** — объединение данных и методов в один объект
- **Наследование** — создание иерархий типов с общим поведением
- **Полиморфизм** — использование объектов разных типов через единый интерфейс

ООП позволяет писать модульный, переиспользуемый и легко расширяемый код.

Почему в Julia нет классов?

Julia выбрала другую парадигму — **множественная диспетчеризация (multiple dispatch)**.

Философия Julia:

- **Разделение данных и функций:** Типы данных отделены от методов
- **Гибкость:** Одна функция может работать с разными типами данных
- **Производительность:** Диспетчеризация позволяет JIT-компилятору генерировать оптимальный код

Вместо классов Julia использует:

- Структуры (`struct`) для определения данных
- Типы (`type`) для создания иерархий
- Функции с типизированными аргументами для поведения

Структуры: основной способ организации данных

```
struct Point
    x::Float64
    y::Float64
end

# Создание объекта
p1 = Point(3.0, 4.0)
println(p1.x)  # 3.0
```

Неизменяемые структуры (`struct`):

- После создания поля нельзя изменить
- Безопасны в многопоточном коде
- Оптимальны для производительности

```
mutable struct MutablePoint
    x::Float64
    y::Float64
end

mp = MutablePoint(1.0, 2.0)
mp.x = 5.0  # можно изменять
```

Изменяемые структуры (`mutable struct`):

- Поля можно менять после создания
- Более гибкие, но медленнее

Добавление поведения к типам

Функции, работающие с типами, определяются **отдельно от самого типа**:

```
struct Point
    x::Float64
    y::Float64
end

function distance(p1::Point, p2::Point)::Float64 # ::Float64 – тип возвращаемого значения
    sqrt((p1.x - p2.x)^2 + (p1.y - p2.y)^2)
end

function move!(p::Point, dx::Float64, dy::Float64) # ! – означает, что функция изменяет аргумент
    Point(p.x + dx, p.y + dy)
end

p1 = Point(0.0, 0.0)
p2 = Point(3.0, 4.0)
println(distance(p1, p2))  # 5.0
```

Ключевая идея: Поведение не привязано к определению типа. Это позволяет:

- Расширять существующие типы новыми функциями
- Избежать монолитных классов с множеством методов

Множественная диспетчеризация

Julia выбирает реализацию функции на основе **типов всех аргументов**:

```
function describe(x::Int)
    println("Целое число: $x")
end

function describe(x::String)
    println("Строка: $x")
end

function describe(x::Float64)
    println("Действительное число: $x")
end

describe(42)       # "Целое число: 42"
describe("Julia")  # "Строка: Julia"
describe(3.14)     # "Действительное число: 3.14"
```

Версия с несколькими аргументами:

```
function add(x::Int, y::Int)
    x + y
end

function add(x::String, y::String)
    x * y
end

function add(x::Vector, y::Vector)
    x .+ y
end

add(1, 2)           # 3
add("Hello", " World")  # "Hello World"
add(, ) #
```

Абстрактные типы и иерархия

```
abstract type Shape end  # Абстрактный тип — только для наследования, нельзя создать экземпляр

struct Circle <: Shape  # Конкретный тип "Круг", наследует от Shape
    radius::Float64
end

struct Rectangle <: Shape  # Конкретный тип "Прямоугольник", наследует от Shape
    width::Float64
    height::Float64
end

struct Triangle <: Shape  # Конкретный тип "Треугольник", наследует от Shape
    a::Float64
    b::Float64
    c::Float64
end
```

Иерархия типов:

```
Shape (абстрактный тип)
├── Circle
├── Rectangle
└── Triangle
```

Абстрактные типы определяют интерфейс, а конкретные типы его реализуют.

Полиморфизм через множественную диспетчеризацию

```julia
function area(c::Circle)::Float64
    π * c.radius^2
end

function area(r::Rectangle)::Float64
    r.width * r.height
end

function area(t::Triangle)::Float64
    # Формула Герона
    s = (t.a + t.b + t.c) / 2
    sqrt(s * (s - t.a) * (s - t.b) * (s - t.c))
end

function perimeter(c::Circle)::Float64
    2π * c.radius
end

function perimeter(r::Rectangle)::Float64
    2 * (r.width + r.height)
end

# Использование
shapes = [Circle(2.0), Rectangle(3.0, 4.0), Triangle(3.0, 4.0, 5.0)]

for shape in shapes
    println("Площадь: $(area(shape))")
end
```

Julia выбирает нужную функцию в зависимости от типа объекта — это **полиморфизм**.

Инкапсуляция и соглашения

```
mutable struct BankAccount
    balance::Float64           # баланс счета
    _pin::Int                  # приватное поле PIN (префикс _ — соглашение для приватности)
end

# Функция для пополнения счета
function deposit!(account::BankAccount, amount::Float64) # ! — для изменения
    if amount > 0
        account.balance += amount           # увеличиваем баланс
    else
        error("Сумма должна быть положительной")  # ошибка, если сумма отрицательна
    end
end

# Функция для снятия средств
function withdraw!(account::BankAccount, amount::Float64) # ! — для изменения
    if amount <= account.balance
        account.balance -= amount          # уменьшаем баланс
    else
        error("Недостаточно средств")      # ошибка, если средств недостаточно
    end
end

# Геттер для получения баланса
function get_balance(account::BankAccount)::Float64
    account.balance
end

# Пример использования
acc = BankAccount(1000.0, 1234)   # создаём новый аккаунт с балансом 1000.0 и PIN 1234
deposit!(acc, 500.0)              # пополняем счет на 500.0
println(get_balance(acc))         # выводим баланс (1500.0)
```

Julia не имеет встроенных модификаторов доступа (`private`, `public`), вместо этого используются **соглашения**:

Соглашения:

- Начало с `_` обозначает приватное поле
- Функции с `!` в конце изменяют состояние
- Getter-функции для доступа к данным

Сравнение с классическим ООП

Аспект	Класс (Python/Java/C++)	Julia
Определение	Класс объединяет данные и методы	Структура хранит данные, функции добавляют поведение
Наследование	Механизм `class Child(Parent)`	Иерархия абстрактных типов `Child <: Parent`
Полиморфизм	Переопределение методов класса	Множественная диспетчеризация по типам
Инкапсуляция	`private`, `public` ключевые слова	Соглашения (`_` префикс)
Гибкость	Ограничена классом	Функция может работать с любыми типами
Производительность	Зависит от JIT	Оптимальна благодаря специализации типов

Практические примеры: иерархия животных

```
# Абстрактный тип Animal — задает общий интерфейс для всех животных
abstract type Animal end

# Структура Dog наследует Animal, содержит имя и возраст
struct Dog <: Animal
    name::String
    age::Int
end

# Структура Cat также наследует Animal, содержит имя и возраст
struct Cat <: Animal
    name::String
    age::Int
end

# Функция speak для Dog: персонализованное поведение
function speak(d::Dog)
    println("$(d.name) гавкает: Гав!")
end

# Функция speak для Cat: персонализованное поведение
function speak(c::Cat)
    println("$(c.name) мяукает: Мяу!")
end

# Функция describe для любого Animal — выводит описание
function describe(a::Animal)
    println("Животное: $(a.name), возраст: $(a.age)")
end

# Список животных разных типов
animals = [Dog("Барон", 5), Cat("Мурзик", 3)]

# Перебираем всех животных, описываем и "заставляем говорить"
for animal in animals
    describe(animal)
    speak(animal)
end
## Output:
Животное: Барон, возраст: 5
Барон гавкает: Гав!
Животное: Мурзик, возраст: 3
Мурзик мяукает: Мяу!
```

Итоги

Julia не использует классический ООП, но предоставляет **мощные инструменты** для объектно-ориентированного дизайна:

✓ **Структуры** — определение данных
✓ **Абстрактные типы** — создание иерархий
✓ **Множественная диспетчеризация** — гибкий полиморфизм
✓ **Композиция и соглашения** — инкапсуляция

Преимущества подхода Julia:

- Большая гибкость и композабельность
- Отличная производительность через JIT-компиляцию
- Легко расширять существующие типы новыми функциями
- Более выразительный полиморфизм

Julia показывает, что **ООП — это не только классы**, а набор принципов, которые можно реализовать множеством способов.

Источники

- Julia Official Documentation: https://docs.julialang.org/