



# Парадигмы программирования

Денис С. Мигинский

# Понятие парадигмы

**Парадигма** (философия науки) – устоявшаяся система научных взглядов, в рамках которой ведутся исследования (Т. Кун)

**Парадигма программирования** – это совокупность принципов, методов и понятий, определяющих способ конструирования программ.

# Парадигма программирования

**Парадигма** в первую очередь определяется базовой программной единицей(-ами) и самим принципом достижения модульности программы.

**Цель:** достижение модульности

**Средства:** механизмы абстракции

**Вопрос:** что такое абстракция и модульность?

# Абстракция

**Абстракция (абстрагирование)** – процесс определения существенных характеристик некоторой сущности, отличающих ее от других сущностей и значимых в рамках поставленной задачи. Также **абстракция** (как процесс) включает представление выделенных характеристик моделируемой сущности и представление их в терминах языка моделирования/программирования.

**Принцип абстракции** – каждая существенная область функциональности в программе должна быть реализована всего в одном месте программного кода (Б.Пирс)

# Модульность

**Модульность** – разбиение программы (или любой другой сложной технической системы) на составные блоки с относительно замкнутой функциональностью

Единица модульности => абстракция

# Распространенные (и не очень) парадигмы

- Императивное (процедурное) программирование
- Структурное программирование
- **Функциональное программирование**
- Логическое программирование
- **Объектно-ориентированное программирование**
  
- **Аспектно-ориентированное программирование**
- **Метапрограммирование, ориентированное на языки программирования**

# Императивное программирование

Изменение состояния программы посредством  
исполнения инструкций

**Основные механизмы управления/абстракции:**

- Последовательное исполнение команд
- Ветвление
- Безусловный переход
- Вызов подпрограммы (процедурное программирование)

**Элементарные единицы модульности:**

отсутствуют

# Структурное программирование

## Основные механизмы управления/абстракции:

- Последовательное исполнение команд
- Ветвление
- Цикл
- Вызов подпрограммы
- Лексический контекст

## Элементарные единицы модульности:

- Подпрограмма с изолированным лексическим контекстом

# Лексический контекст

**Лексический контекст** определяет область видимости идентификаторов (переменных и т.д.)

Контексты могут быть вложены друг в друга

```
int main () {
    int i = 0;
    int sumSqr = 0;
    //scope 1 with vars i & sumSqr
    while(i < 10) {
        int sqrI = i*i;
        //scope 2 with inherited vars i & sumSqr
        //and self var sqrI
        sum
    }
}
//global scope with main symbol
```

# Объектно-ориентированное программирование

Представление программы в форме взаимодействующих объектов

## Основные механизмы управления/абстракции:

- Объект
- Класс
- Иерархии классов/объектов
- Полиморфизм

## Элементарные единицы модульности:

- Класс (*примечание: в наиболее распространенной интерпретации*)

# Объект в ООП

**Объект** характеризуется:

- поведением;
- состоянием;
- уникальностью (*identity*);

В трактовке Г. Буча поведение – реакция на сообщение, которая **зависит** от текущего состояния а также других факторов (полиморфизм), и выражается в изменении состояния объекта, а также посылки сообщений другим объектам.

Это ООП на основе СП.

*Примечание: существуют и другие трактовки*

# Аспектно-ориентированное программирование

Представление программы в виде набора аспектов, формирующих классы

## Основные механизмы управления/абстракции:

- Элементы ООП
- Аспекты
- Динамические контексты/контекстный «полиморфизм»

## Элементарные единицы модульности:

- Аспект

# Логическое программирование

Логический вывод на основе заданных программистов правил/фактов/предикатов

Основные механизмы управления/абстракции:

- Логические формулы
- Применение правил вывода

Элементарные единицы модульности:

- Логическая формула

# Метапрограммирование

Использование кодогенерации и/или специализированных языков, создаваемых для решения определенного узкого круга задач

## Основные механизмы управления/абстракции:

- Определение языка
- Макросы

## Элементарные единицы модульности:

- Зависит от определения языка

# Функциональное программирование

Представление программы в форме набора чистых функций, порождающих результаты на основе входных данных

## Основные механизмы управления/абстракции:

- Чистая функция, как объект первого класса
- Вызов функции (в т.ч. рекурсивный)
- Лексический контекст, замыкание

## Элементарные единицы модульности:

- Функция (в т.ч. высшего порядка, обобщенная, мультиметод и т.д.)

# Функциональное программирование: основные концепции

Неподвижное состояние объектов

Чистые функции

Функции, как объекты первого класса

Функции высших порядков

Функции как замыкания

# Неподвижность состояния/ однократное присваивание

При инициализации идентификатора (переменной) ему сопоставляется некоторое значение (объект), которое не может быть изменено.

При этом идентификатор с тем же самым именем может быть определен в другом лексическом контексте с другим значением.

```
int main (void) {  
    const int i = 1;  
    //...  
    {  
        const double i = 2.5;  
        //...  
    }  
    //i == 1  
    return 0;  
}
```

# Чистые функции

**Чистая функция (pure function)** – функция не имеющая побочных эффектов, т.е. единственным эффектом ее применение является порождение результата, зависящего только от аргументов.

**К побочным эффектам относятся:**

- изменение переменных вне контекста функции
- изменение параметров
- ввод-вывод

Чистая функция всегда имеет возвращаемое значение. Чистая функция гарантирует воспроизводимость результата при повторном вызове с теми же аргументами.

# Лексическое замыкание (lexical closure)

**Замыкание (лексическое замыкание)** – функция в совокупности с тем лексическим контекстом в рамках которого объявлена.

Функция может использовать (быть замкнута на) переменные из этого контекста.

Имеет смысл, если функция является объектом первого класса.

```
def f
  yield
end
val=5
f{puts val}
```

# Функции высших порядков

**Функции высших порядков** принимают в качестве параметров и/или возвращают другие функции

*Частные случаи (см. функ.ан.):*

**Функционал** – функция, принимающая функцию и возвращающая скаляр.

*Примеры:* определенный интеграл, map, reduce(inject), filter(select)

**Оператор** – функция, принимающая и возвращающая функцию.

*Примеры:* производная, карринг (частичное применение)

# Объекты первого класса

Объект первого класса (**first-class object/citizen**) – объект, который:

- может быть сохранен в переменной;
- может быть передан в функцию как параметр;
- может быть возвращен из функции как результат;
- может быть создан во время выполнения программы;
- внутренне самоидентифицируем (независим от именованя).

# Объекты первого класса языках

## C++/Java:

объект

## C#:

объект, функция

## Ruby:

объект, функция

класс, модуль, методы (через элементы МОР)

произвольный код в форме строки (через eval)

## Clojure:

объект, функция (в т.ч. метод/мультиметод)

управляющие конструкции языка (через макросы)

Java-класс (начиная с v1.2)

произвольный код в форме S-выражения (через eval)

# Объекты в ФП

**Объект** характеризуется:

- поведением;
- неподвижным состоянием == уникальность;

ООП на основе ФП:

Поведение - реакция на сообщение, которая **зависит** от текущего состояния а также других факторов (полиморфизм). Метод объекта является чистой функцией от самого объекта и дополнительных параметров.

Функциональный мутатор порождает новый объект, а не меняет состояния текущего в отличие от структурного ООП.

# Механизмы построения функций высших порядков в языках

## C++:

указатели на функции

класс-функции (через перегрузку operator () и template)

## Java:

через анонимные и вложенные классы (замыкания)

## C#, Ruby, Clojure:

lambda-выражения (замыкания)

*Вывод:* для построения функций высших порядков поддержка функций как объектов первого класса и замыканий не является необходимой но крайне желательна

# Преимущества функциональных языков

Функциональные программы содержат гораздо меньше скрытых ошибок, легче поддаются формальному анализу и автоматической оптимизации

Очень выразительны для алгоритмических абстракций

Эффективная поддержка параллелизма

Поддержка мемоизации, отложенных вычислений

Эффективная поддержка символьных вычислений

# Языки с поддержкой ФП

## Статические:

Семейство ML: OCaml, Haskell, F#  
Scala, C#

## Динамические:

Семейство Lisp: Common Lisp, Scheme, Clojure  
Erlang  
Python, Ruby, Groovy

## Специализированные:

XSLT, XQuery, EMACS Lisp, Mathematica, Maple