Александр Загоруйко © 2018

OOP Introduction

Парадигмы программирования

Парадигма программирования – это комплекс концепций, принципов, идей и понятий, определяющих фундаментальный стиль написания компьютерных программ. Парадигма задаётся использованием определённых сущностей, например объектов и взаимодействий между ними (объектно-ориентированное программирование), алгоритмов и коллекций, способных работать с произвольными типами данных (обобщённое программирование) и тд.
https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F

https://lispcast.com/procedural-paradox/

Список парадигм

- **процедурное программирование**
- структурное программирование
- функциональное программирование
- аспектно-ориентированное программирование
- **объектно-ориентированное программирование**
- событийно-ориентированное программирование
- метапрограммирование
- **обобщённое программирование**
- реактивное программирование
- автоматное программирование

http://progopedia.ru/paradigm/
https://habrahabr.ru/post/223253

Современные языки программирования (вроде C++, Java, C#) поддерживают несколько парадигм одновременно.

Процедурное программирование

Согласно процедурной парадигме программирования, **код программы должен делиться на блоки** (процедуры). Процедура (иногда также называемая подпрограммой, функцией или методом) — это определённая последовательность команд, которые следует выполнить. Любая процедура может быть вызвана по имени из любой точки программы, включая другие процедуры или её же саму (т.н. рекурсивный вызов). Вся **работа программы осуществляется в терминах ДЕЙСТВИЙ**, и основной **акцент делается на составлении правильного алгоритма** (последовательности выполнения блоков).

Как выглядит ПП-программа

```
// … реализация всех функций

void main() {
      StageOptions(); // настройки окна
      Intro(); // стартовая заставка
      FirstScreen(); // показ главного меню
      GenerateLevel(); // создание игровой локации
      LogResults(); // запись игрового прогресса в файл
      Outro(); // завершение работы программы
    }
}
```

Преимущества использования

- Проект **структурно разделяется на логические части** (заставка, показ главного меню, вывод игрового поля, и тд.) – в следствие чего отладка, чтение кода, и отслеживание общей логики выполнения программы значительно упрощается.
- Спустя три месяца, глядя на свой проект, состоящий (допустим) из 5000 строк кода, вы не теряетесь, и всегда можете найти ту функцию, в которой нужно что-нибудь исправить. То есть, **обеспечивается возможность сопровождения программного продукта через длительное время** после написания кода, и даже кем-то, кроме его автора ☺
- Наличие именованных блоков кода позволяет **разделить проект на несколько файлов** (библиотек), и так 5000 строк превратится в 5 библиотек по ~1000 строк кода. К тому же, хорошо написанная функция **может оказаться полезной для** любой другой вашей **будущей программы** на языке C++.

Преимущества использования

- Существенно сокращается размер кода. Например, если игровое поле спустя какое-то время работы программы придётся показать снова - достаточно будет написать одну строчку кода с вызовом функции ShowField(); а не заново копировать код показа поля на 200 строк через copy-paste. Таким образом, **функции обеспечивают более высокоуровневый способ повторного использования кода**. https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0
- **Функции могут** принимать параметры - это позволяет им **срабатывать по-разному**. Например, метод может быть вызван так: ShowField(20, 20); или так: ShowField(30,15); - при этом размер игрового поля будет варьироваться (20x20, или 30x15).
- **Над проектом одновременно могут работать несколько человек**. Функции, написанные разными программистами, несложно собрать в один проект.

Проблема процедурного программирования

Проблема процедурного программирования

Процедурное программирование – это подход, при котором переменные и функции, по смыслу относящиеся к определённому объекту, особо не привязаны к нему на уровне кода. Например, в вашем исходнике может быть две функции, отвечающих за генерацию и показ лабиринта, и три функции, выводящих разного рода статистику в заголовок окна. Все эти пять функций могут располагаться в коде в произвольном порядке, и никто, кроме их создателя, понятия не имеет (без пристальной вычитки кода), с какими именно объектами работает та или иная функция.

Хорошо, что есть комменты ☺

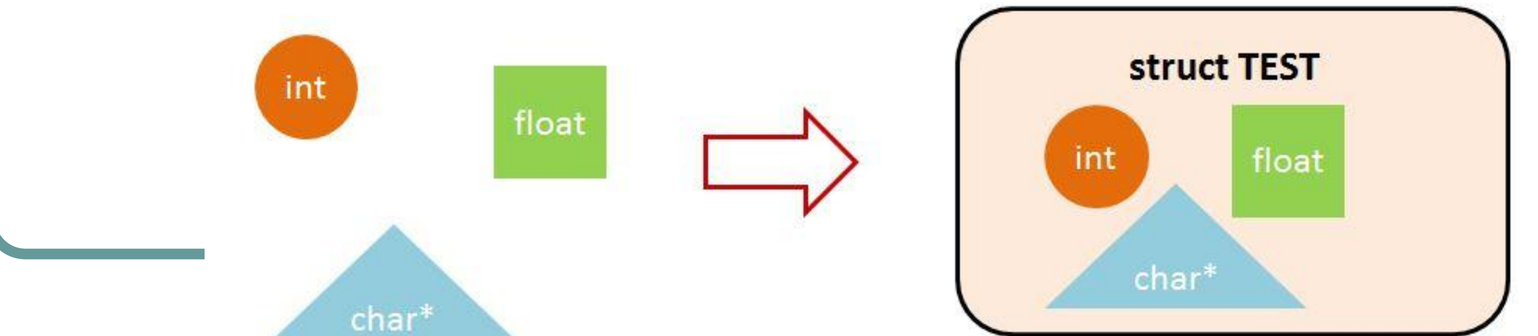
```
/// ПЕРЕМЕННЫЕ КЛАССА, которые будут доступны во всех методах:
int height = 40; // высота лабиринта (количество строк)
int width = 60; // ширина лабиринта (количество столбцов в каждой строке)
int gatheredMedals = 0; // собранные медали
int medals = 0; // число медалей в лабиринте
int health = 100; // здоровье персонажа
boolean flagMedicine = true; // можно или нет собирать здоровье, true-можно,false-нельзя
int old_x, old_y; // сюда нужно вернуть здоровье, которое нельзя собирать
int energy = 500; // первоначально 500 единиц энергии
int steps = 0; // число перемещений
boolean flagCoffee = true; // можно или нет пить кофе, true-можно,false-нельзя
boolean putCoffeeBack = false; // положить ли кофе на место
int energyForKick = 10; // затраты энергии при ударе ногой
int leftEnergy = energyForKick - 1; // пинок отнимает left_energy + 1
int coffeeEnergy = 25; // столько энергии добавляет чашка кофе
int medicinePlus = 5; // столько здоровья добавляет таблетка
int maxSteps = 10; // число перемещений после принятия лекарства, начиная с которогоможно
int energyForBomb = 49; // энергия, которая тратится на установку бомбы
int energyForBlaster = 20; // энергия, которая тратится на выстрел из бластера
int radiusKick = 1; // радиус действия пинка
int radiusBomb = 3; // радиус действия бомбы
int direction; // 0 - right, 1 - left, 2 - up, 3 - down
int enemyCount = 0; // текущее количество врагов
boolean baseMusic = true; // играет или основная фоновая музыка
String musicBomb = "/sound/bomb.mp3";
String musicKick = "/sound/kick.mp3";
String musicBase = "/sound/basicMusic.mp3";
```

Какие функции влияют на ИИ врага?

```
25   public static void main(String[] args) {...3 строк }
     public void start(Stage primaryStage) {...6 строк }
34   public void musicOn(String s) {...6 строк }
40   public void musicOff() {...3 строк }
43   public void options(Stage primaryStage) {...37 строк }
     public void make(GameObject obj, int rand, int x, int y) {...5 строк }
85   public void generateMaze() {...51 строк }
136  public void showMaze() {...31 строк }
167  public void clearCell(int x, int y) {...6 строк }
173  public void kick(int x, int y, int radius) {...16 строк }
189  public void checkMedal(int x, int y) {...6 строк }
195  public void checkEnemy(int x, int y) {...16 строк }
211  public void checkHall(int x, int y) {...18 строк }
229  public void checkMedicine(int x, int y) {...17 строк }
246  public void checkCoffee(int x, int y) {...13 строк }
259  public void checkAll(int x, int y) {...7 строк }
266  public void setSmile(int x, int y) {...8 строк }
274  public void putBomb(int x, int y) {...8 строк }
282  public void checkEnemyDead(int radiusBomb) {...19 строк }
301  public void checkHeroDead(int x, int y, int radiusBomb) {...13 строк }
314  public void checkAllMedals() {...7 строк }
321  public void bombDetonation(int x, int y, int radiusBomb) {...8 строк }
329  public void blaster(int x, int y, int direction) {...21 строк }
350  public void gameOver(String str) {...5 строк }
355  public void message(String str) {...7 строк }
362  public void gameProcess() {...69 строк }
```

Ну вообще, не всё так плохо

В рамках курса «Процедурное программирование на языке Си» вы могли познакомиться с таким понятием, как структуры. **Структура** – композитный тип данных, определяющий физически группированный список переменных (одного или разных типов), расположенный под одним именем в едином блоке памяти. Структуры позволяют собрать разные данные, относящиеся к одной сущности, под «одной крышей» для их одновременного создания, передачи в функцию или, что более значимо, возврата из неё.



Синтаксис использования

```
struct Student
{
    char surname[50];
    char name[20];
    unsigned int age;
    float rating;
    Student* leader;
};
```

```
Student CreateDefaultStudent()
{
    Student s;
    strcpy(s.surname, “Ivanov”);
    strcpy(s.name, “Ivan”);
    s.age = 25;
    s.rating = 11;
    s.leader = nullptr;
    return s;
}
```

Понятие «реализация»

Студент – сущность реального мира. В примере определены только наиболее значимые его характеристики. Это – наша **РЕАЛИЗАЦИЯ** студента, то, **ЧЕМ** конкретно он будет представлен в программе.

Структурный тип данных Student определяет лишь то, что данные, относящиеся к одному студенту, хранятся вместе. Действия над одним студентом и диапазоны значений его характеристик определяются только типами этих внутренних характеристик.

Структуры и функции

Но в реальности, свойства студента включают также **действия**, которые он может совершать сам, или же которые можно совершать над ним. Например, студенту можно назначить старосту, поднять его рейтинг, студент может учиться, отдыхать, работать и тд. Чтобы задать **ПОВЕДЕНИЕ** (**интерфейс**) студента, мы используем функции:

```
void ChangeLeader(Student* s, Student * leader) {
    if (s->leader != nullptr)
        return false;
    s->leader = leader;
    return true;
}
```

Структур не достаточно…

```
void Work(Student* s)
{
    cout << s->surname << ' ' << s->name << "is working\n";
    s->rating *= 1.01;
}
```

Но это означает, что структура сама по себе не даёт полного представления о свойствах студента. Только в комплекте с функциями рождается концепция полноценного типа данных, способного задавать диапазон значений своих переменных и действия, совершаемые над ними…

ООП-подход

Объектно-ориентированное программирование – это подход, при котором переменные и функции, относящиеся к определённому объекту, объединены в коде специальным образом и очень тесно связаны между собой. Для стороннего разработчика обычно не составляет труда разобраться, из каких основных сущностей состоит программа (лабиринт, окно, игрок, и тд.), и какие данные, а также действия относятся к той или иной сущности.

Реализация + интерфейс

ООП позволяет создавать пользовательские типы данных (классы), определяющие как **реализацию** (набор внутренних переменных), так и **интерфейс** (список функций, правила поведения) для своих экземпляров (которые называются объектами).

```
class Student {
    char surname[50];
    char name[20];
    unsigned int age;
    float rating;
    Student* leader;
    void ChangeLeader(Student * l){
        leader = l;
    }
    void Work(){
        cout << surname << ' ' << name << "is working\n";
        rating *= 1.01;
    }
};
```

Описание окружающего мира

В процедурном программировании – осуществляется в терминах ДЕЙСТВИЙ (акцент делается на **составлении правильного алгоритма**).

В объектно-ориентированном программировании – осуществляется в терминах ОБЪЕКТОВ, над которыми производятся действия (**объект – центр концепции ООП**).

Определение

ООП – это стиль программирования, который фиксирует поведение реального мира так, что при этом скрываются детали разработки. Это позволяет программисту, решающему задачу, мыслить в терминах, присущих самой задаче, а не программированию в целом.

Определение

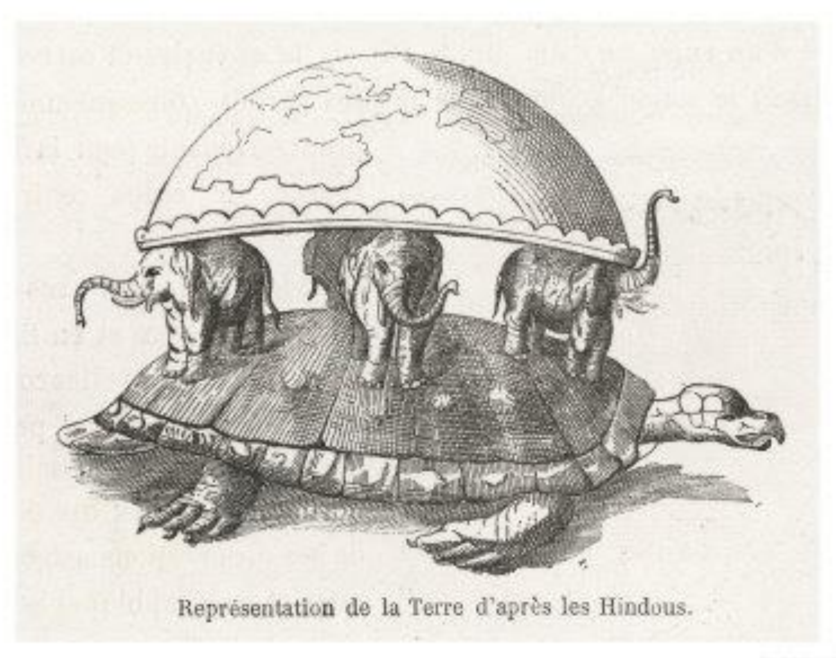
ООП - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Парадигма ООП

Программа состоит из объектов, обменивающихся сообщениями. Объекты могут обладать состоянием, единственный способ изменить состояние объекта - послать ему сообщение (вызвать метод), в ответ на которое, объект может изменить собственное состояние.

Основные понятия ООП

- Класс
- Объект
- Абстрагирование
- Инкапсуляция
- Наследование
- Полиморфизм


Représentation de la Terre d'après les Hindous.

Поиграем в ООП

- Берём любой предмет (объект)
- Отвечаем на вопросы:
- Что этот объект собой представляет? Что мы о нём знаем? Каковы его характеристики? (поля)
- Что этот объект умеет делать? Какие действия можно совершать над объектом? (методы)

Поиграем в ООП

- Тип объекта: маркер
- Поля:

- Модель: Friday
- Цвет колпачка: синий
- Длина корпуса: 138 мм

- Методы:

- Чесаться
- Ковыряться
- Метко бросать
- Писать по доске

Концепция «чёрный ящик»

- Снаружи объект принято рассматривать как чёрный ящик (прибор с кнопками).
- Известно, что будет, если нажимать на кнопки (есть инструкция).
- Неизвестно, как всё устроено внутри и почему оно работает (да и не нужно знать, почему).

Понятие класса

Класс – это общее описание состояния и поведения некоторой сущности, а также правил по взаимодействию с ней. Класс – это своего рода схема или чертёж, по которому будут строиться будущие объекты. Класс можно рассматривать как набор данных (свойств, полей, атрибутов) и функций для работы с ними (методов). Также это пользовательский структурированный тип данных, введённый программистом на основе уже существующих типов.

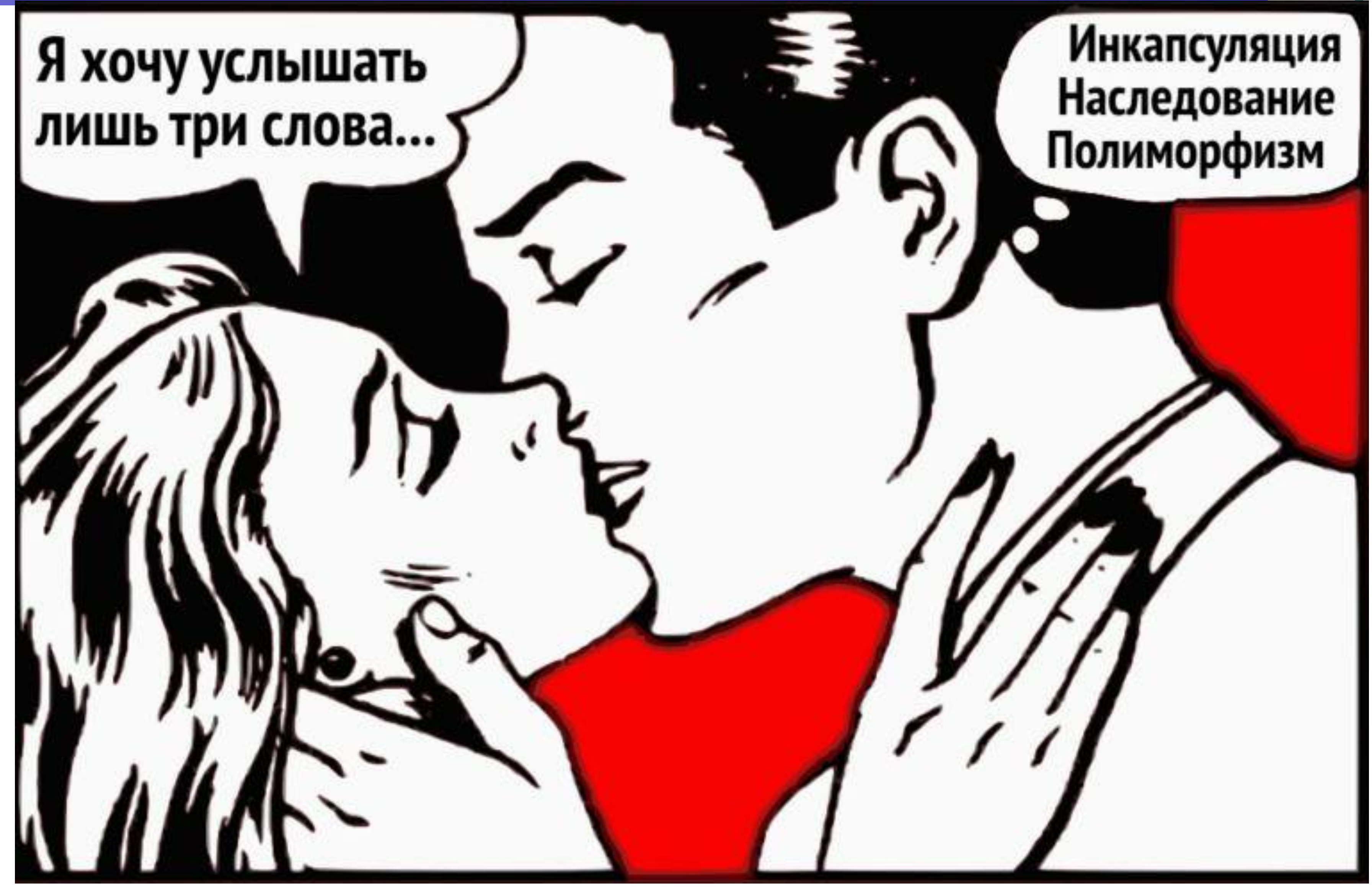
Объект. Поля и методы

Объект (экземпляр класса) – это отдельный представитель класса, имеющий своё конкретное состояние, и поведение, полностью определяемое классом.

Поле – это переменная, связанная с классом или объектом. Состояние объекта хранится в его полях. Тип данных поля задаётся при описании класса.

Метод – это функция или процедура, принадлежащая классу или объекту.

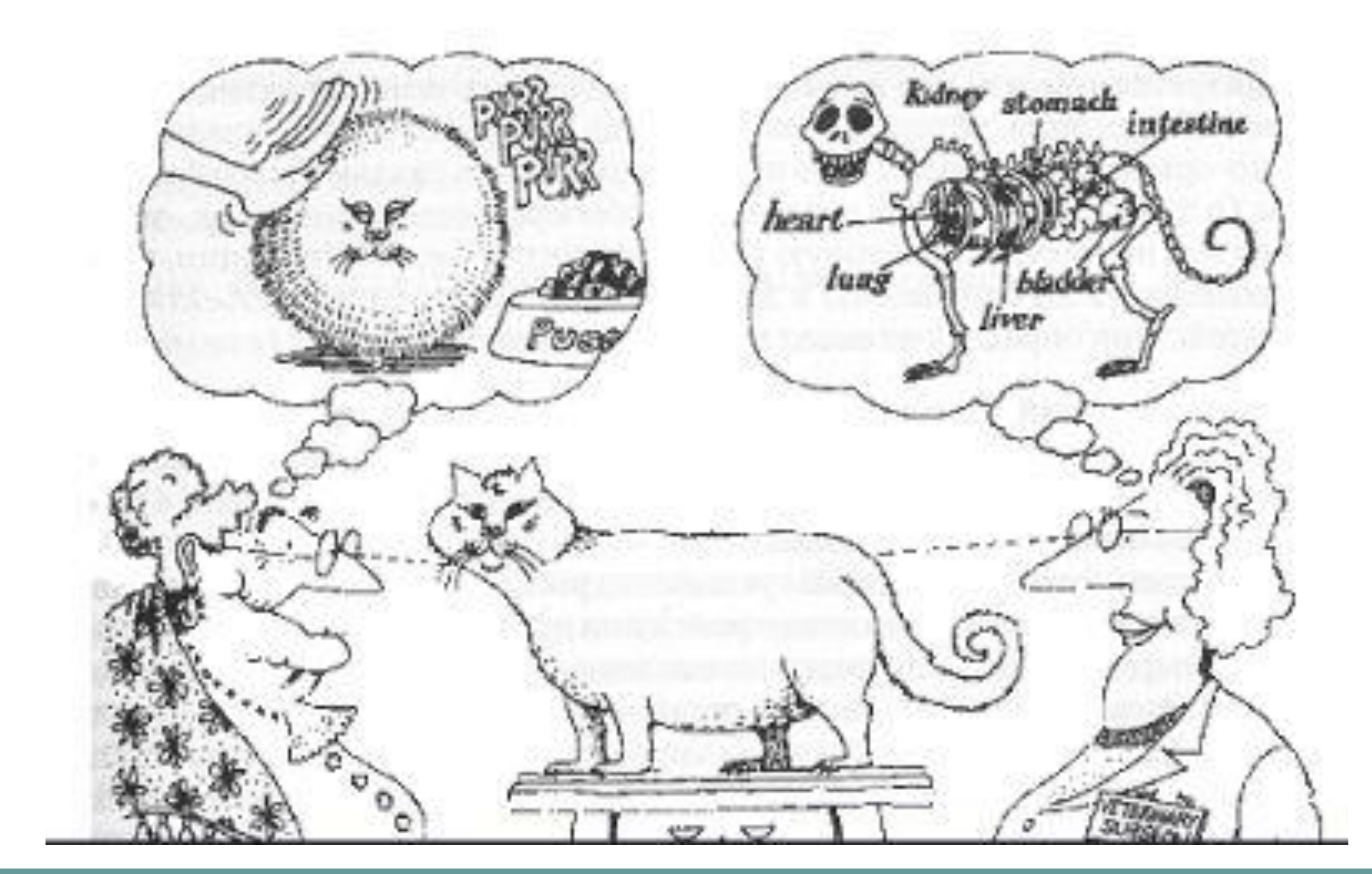
Киты ООП



Абстракция (нулевой кит ☺)

Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. Абстракция – это набор значимых характеристик объекта, которые отличают его от других объектов. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно смоделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной.

Абстракция данных



Инкапсуляция

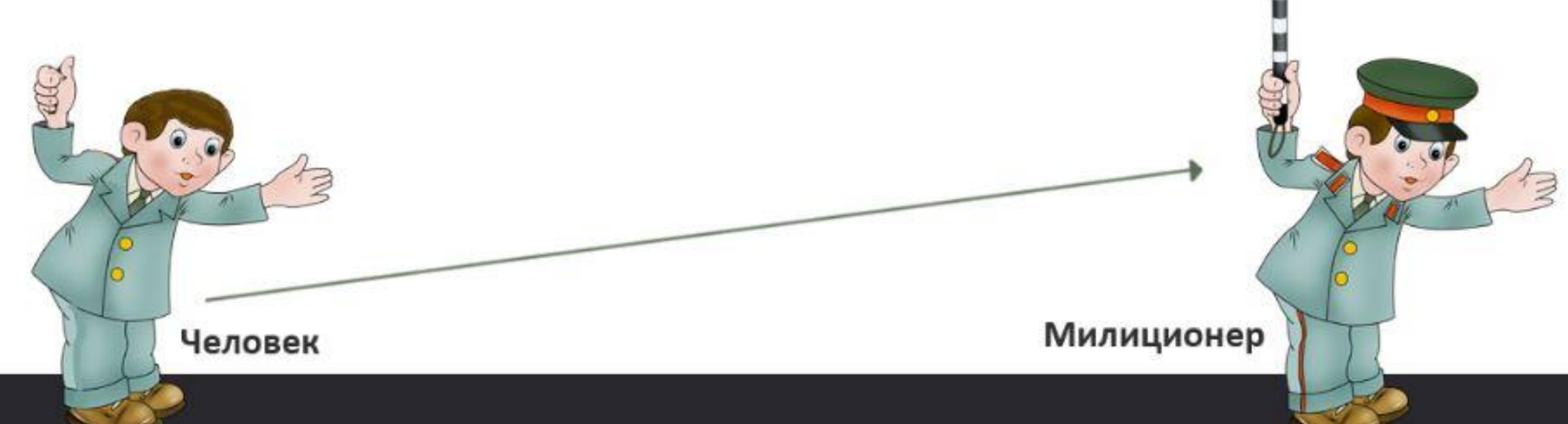
Инкапсуляция – это принцип, согласно которому данные о свойствах объекта и методы для работы с этими данными объединены в единое целое - класс. При этом главным требованием инкапсуляции является санкционированный доступ к полям класса, т.е. доступ не на прямую, а через интерфейс класса (его открытые методы).

Наследование

Наследование - это процесс, посредством которого один класс может наследовать свойства другого класса и добавлять к ним особенности, характерные только для него.

Определение новых классов на основе уже имеющихся позволяет построить целую иерархию классов. Имеющиеся классы обычно называют базовыми (родительскими). А новые классы, формируемые на основе базовых - производными (классами-потомками, дочерними классами).

Смысл наследования заключается в том, что не надо каждый раз заново описывать новый объект, а можно указать класс-предок и в новом классе описать только то, чем он отличается от предка, а всё остальное от него просто унаследовать.



Полиморфизм

Полиморфизм – это принцип, согласно которому есть возможность использовать одну и ту же запись в коде для работы с объектами различных типов данных. Кратко: «Один интерфейс, множество реализаций». Полиморфизм позволяет единообразно работать с объектами различных типов, подменяя только сами объекты, а не код по их обработке.

Пример полиморфизма

```
Transport** traffic = new Transport*[3];
traffic[0] = new Car();
traffic[1] = new Bike();
traffic[2] = new Marshrutka();
for (int i = 0; i < 3; i++) {
   traffic[i]->drive();
}
```

Синтаксис создания класса

```
/* class documentation comment */
class AlwaysNounsNotVerbs {
    static variables;
    instance variables;
    constructors;
    methods;
};
```

Пример создания класса

```
class Cat {
    // поля:
    char* name; // кличка кота
    double weight; // вес
    bool is_hungry; // голодный или нет?
    // метод:
    void Sleep() {
        cout << name << " спит.";
    }
};
```

https://git.io/vrjV5

Синтаксис создания объектов

тип идентификатор; // создание на стеке

тип идентификатор(параметры);

тип* указатель = new тип(параметры); // создание безымянного объекта в куче

```
Cat a; // но не Cat a(); !!!
Cat b("Васька");
Cat* pc = new Cat("Васька");
Cat* pd = new Cat();          Cat* pd2 = new Cat;
```

Массив объектов

```
Cat cats[3]; // статический массив
Cat* cats = new Cat[3]; // дин.массив
Cat** cats = new Cat*[3]; // разреженный массив
cats[0] = new Cat();
cats[1] = new Cat();
cats[2] = new Cat();
cats[0]->name = "Грампи";
cats[0]->Sleep();
```

Сравнение классов и структур

Экземпляры структур

```
Cat a, b;
. . .
sleep(&a);
hunt(&a);
```

Объекты

```
Cat a, b;
. . .
a.sleep();
a.hunt();
```

Как видно, функции объявляются внутри тела класса, что меняет синтаксис их вызова – не объект передаётся в функцию, а функция вызывается у объекта. Чтобы отделить такие «внутриклассовые» функции от классических (вроде main), первые называют **методами**. Внутри метода больше не уточняется имя объекта, для которого он был вызван.

Важно запомнить

Каждый объект хранит уникальный набор полей (копию), методы же существуют в единственном экземпляре в рамках класса, но в отдельный момент времени метод имеет доступ только к полям того объекта, для которого был вызван.

Домашнее задание

- Реализовать класс **Pet**, который описывает характеристики и поведение домашнего питомца (собаки, рыбки, хомячка, удава, и тп).
- Создать классы, описывающие 5 любых предметов (например - электрочайник, книгу, телефон, гелевую ручку, купюру в 10 гривен, и тп).

Спасибо за внимание!