Классы

Лекция 3-4

Понятие объектно-ориентированного программирования

- Объе́ктно-ориенти́рованное программи́рование (сокр. ООП) — методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.
- Идеологически, ООП — подход к программированию как к моделированию информационных объектов, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования и программирования, что особенно важно для крупных проектов.

Определение

- Класс — в объектно-ориентированном программировании, модель для создания объектов определённого типа, описывающая их структуру (набор полей и их начальное состояние) и определяющая алгоритмы (функции или методы) для работы с этими объектами.

Класс и объект (экземпляр)

- Каждый объект является «экземпляром» некоторого конкретного класса. То есть «экземпляр класса» в данном случае означает не «пример некоторого класса» или «отдельно взятый класс», а «объект, типом которого является какой-то класс».

Поля / Свойства (Переменные)

- Поля — переменные, принадлежащие экземплярам класса (находящиеся в области видимости конкретного объекта), которые создаются для каждого объекта этого класса.

```
class Car {
    float rate; // расход л/км
    float fuel; // топливо в баке
    int mileage; // пробег
}


Car kia();
kia.rate = 8.5; // присвоение полю
Car lada();
lada.rate = 9.2;
print(kia.rate); // доступ к полю
```

Методы (Функции)

- При использовании классов весь исполняемый код программы (алгоритмы) оформляется в виде «методов», «функций» или «процедур», что соответствует обычному структурному программированию, однако теперь они принадлежат конкретному классу.
- Метод (функция), принадлежащий классу, должен быть вызван только для конкретного объекта (а не сам по себе) и имеет доступ ко всем полям/свойствам и методам этого объекта.

```
class Car {
    int mileage = 0;
    void drive() { // ехать
        mileage += dist;
    }
}


Car kia();
kia.mileage = 0; // новая
Car lada();
lada.mileage = 80000;
print(kia.mileage); // -> 0
kia.drive(150);
print(kia.mileage); // -> 150
print(lada.mileage); // -> 80000
```

Области доступа

- *private* (закрытый, внутренний член класса) — обращения к члену допускаются только из методов того класса, в котором этот член определён. Любые наследники класса уже не смогут получить доступ к этому члену.
- *protected* (защищённый, внутренний член иерархии классов) — обращения к члену допускаются из методов того класса, в котором этот член определён, а также из любых методов его классов-наследников.
- *public* (открытый член класса) — обращения к члену допускаются из любой части кода.
- Назначение областей доступа – предотвратить рассогласование значений переменных внутри объекта, запретив прямой доступ к ним.

```
class Car {
private:
    float fuel = 0;
public:
    float fuel_level() {
        return fuel;
    }
    void refuel(float value) {
        fuel += value;
    }
}


Car kia();
print(kia.fuel); // ошибка: недоступно
print(kia.fuel_level()); // kia.fuel = 0
Kia.refuel(30);
print(kia.fuel_level()); // kia.fuel += 30
```

Конструктор

- Проблема поддержания правильного состояния переменных актуальна и для самого первого момента выставления начальных значений. Для этого в классах предусмотрены специальные методы/функции, называемые конструкторами. Ни один объект (экземпляр класса) не может быть создан иначе, как путём вызова на исполнение кода конструктора, который вернет вызывающей стороне созданный и правильно заполненный экземпляр класса.

```
class Car {
    ...
    Car(float i_rate, int i_mileage = 0,
            int i_fuel = 0) {
        rate = i_rate;
        mileage = i_mileage;
        fuel = 0;
        refuel(i_fuel);
    }
}


Car kia(8.5);
Car lada(8.5, 80000);
Car nissan(8.5, 2000, 45);
```

Наследование

- Наследование (англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

Диаграмма классов: грузовой/пассажирский автомобиль/вагон

```
class Car {
    ...
}
class Nissan: public Car {
public:
    void beep() {
        ...
    }
}


Car kia();
kia.drive(150);
kia.beep(); // ошибка: нет метода
Nissan qashqai();
Qashqai.drive(1); // в пробке
Qashqai.beep();
```

Класс “Автомобиль”

```
class Car {
private:
    int left_dist() { // оставшееся расстояние
        return (int) fuel / rate;
    float rate; // расход л/км
public:
    float fuel; // топливо в баке
    int mileage; // пробег
    void refuel(float value) { // заправить
        fuel += value;
    }
    float get_rate() {
        return rate * 100;
    }
    int drive(int distance) { // ехать
        int left = left_dist();
        int dist; // сможем проехать
        if distance < left {
            dist = distance;
        }
        else {
            dist = left;
        }
        fuel -= dist * rate;
        mileage += dist;
        return dist;
    }
    Car(float rate100, int i_mileage = 0) { // конструктор
        rate = rate100 / 100;
        mileage = i_mileage;
        fuel = 0;
    }
}
```

Использование класса

```
Car kia(8.0, 1500); // почти новая
Car lada(9, 80000); // не новая
Car nissan(8.3); // новая

print(kia.get_rate()); // узнать расход
print(kia.mileage); // узнать пробег
if(kia.mileage > 20000) {
    print(“Пора на ТО”);
}
```

```
int dist = 1000;
int done = 0;
kia.refuel(60);
while(done < dist) {
    done += kia.drive(dist - done);
    print(kia.fuel);
    kia.refuel(30);
}
```