ППОИС

Часть 1

Абстрактные типы данных

Содержание

1. Что такое тип данных. Абстрактный и конкретный тип данных.
2. Совместимость данных. Вывод типов.
3. Классы в C++ как способы описания абстрактных типов данных. Синтаксические возможности
    a. Базовые возможности
    b. Реализация инкапсуляции - модификаторы доступа
    c. Иерархичность - наследование

Тип данных

Множество значений и операций над этими значениями

или еще вариант: *допустимое множество значений*

Теория типов - математически формализованная база для проектирования, анализа и изучения систем типов данных в теории языков программирования

Виды типов данных

Примитивные (скалярные): логический тип, целочисленный тип, вещественный тип

Структурные: массивы, строки, указатели, абстрактные типы данных

Подтипы и полиморфность типов

Совместимость типов

- идентичность и совместимость типов
- явное и неявное преобразование типов

- строгая и нестрогая типизация
- статическая и динамическая типизация

Абстрактный тип данных

Это математическая модель для типов данных, где тип данных определяется поведением (семантикой) с точки зрения пользователя данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.

В языке C++ реализуется через объявление классов (структур)

Объявление класса

```
1   class Point {
2   public:
3       Point();
4       Point(float x, float y);
5       ~Point();
6
7       float distance(const Point& other);
8       float distance(float x, float y);
9
10  private:
11      float x_;
12      float y_;
13  };
```

Google code style - https://google.github.io/styleguide/cppguide.html

Виды методов класса

- Конструкторы: по-умолчанию, “с параметрами”, копирования, перемещения
- Деструктор
- Обычный метод
- Виртуальный метод
- Абстрактный метод (pure virtual method)
- Статический метод
- Перегрузка оператора
- Шаблонный метод

Конструктор

- всегда вызывается при создании экземпляра класса - объекта
- имя конструктора совпадает с именем класса
- может принимать параметры, но не возвращает значение
- класс может содержать несколько конструкторов
- если конструктор в классе не объявлен, то компилятор предоставляет конструктор по-умолчанию без параметров и который ничего не делает

Виды конструкторов

- Конструктор по-умолчанию - A()
- Конструктор преобразования - A(int) // конструктор с один параметром
- Конструктор копирования A(const A&)
- Конструктор перемещения A(A&&)

Неявное преобразование

```
class SomeString
{
private:
    std::string m_string;
public:
    SomeString(int a) // выделяем строку размером a
    {
        m_string.resize(a);
    }

    SomeString(const char *string) // выделяем строку для хранения значения типа string
    {
        m_string = string;
    }
};

int main()
{
    SomeString mystring = 'a'; // выполняется копирующая инициализация
    return 0;
}
```

Ключевое слово explicit

```
class SomeString
{
private:
    std::string m_string;
public:
    explicit SomeString(int a)
    {
        m_string.resize(a);
    }

    SomeString(const char *string)
    {
        m_string = string;
    }
};

int main()
{
    SomeString mystring = 'a'; // ошибка компиляции
    return 0;
}
```

Еще модификаторы конструкторов

- Ключевое слово **delete** :
  ```
  SomeString(char) = delete;
  ```

- Ключевое слово **default**:
  ```
  SomeString(char) = default;
  ```

Делегирующие конструкторы

```
class Employee
{
private:
    int m_id;
    std::string m_name;

public:
    Employee(int id, const std::string& name):
        m_id(id), m_name(name)
    { }

    // Используем делегирующие конструкторы для сокращения дублированного кода
    Employee(const std::string &name) : Employee(0, name) { }
};
```

Деструктор

- вызывает автоматически когда нужно уничтожить объект
- не принимает параметры и не возвращает значение
- может быть только один в классе
- имя деструктора состоит из ‘~’ + имя класса
- если деструктор в классе не объявлен, то компилятор вставляет деструктор по-умолчанию, который ничего не делает

Модификаторы

- модификаторы доступа: public, protected, private и по-умолчанию
- static
- const, mutable
- virtual
- explicit для конструкторов

Объявление данных класса

Модификаторы: const, static, mutable

```
class S {
    int d1; // non-static data member
    int a[10] = {1,2}; // non-static data member with initializer (C++11)
    static const int d2 = 1; // static data member with initializer
    virtual void f1(int) = 0; // pure virtual member function
    std::string d3, *d4, f2(int); // two data members and a member function
    enum {NORTH, SOUTH, EAST, WEST};
    struct NestedS {
        std::string s;
    } d5, *d6;
    typedef NestedS value_type, *pointer_type;
};
```

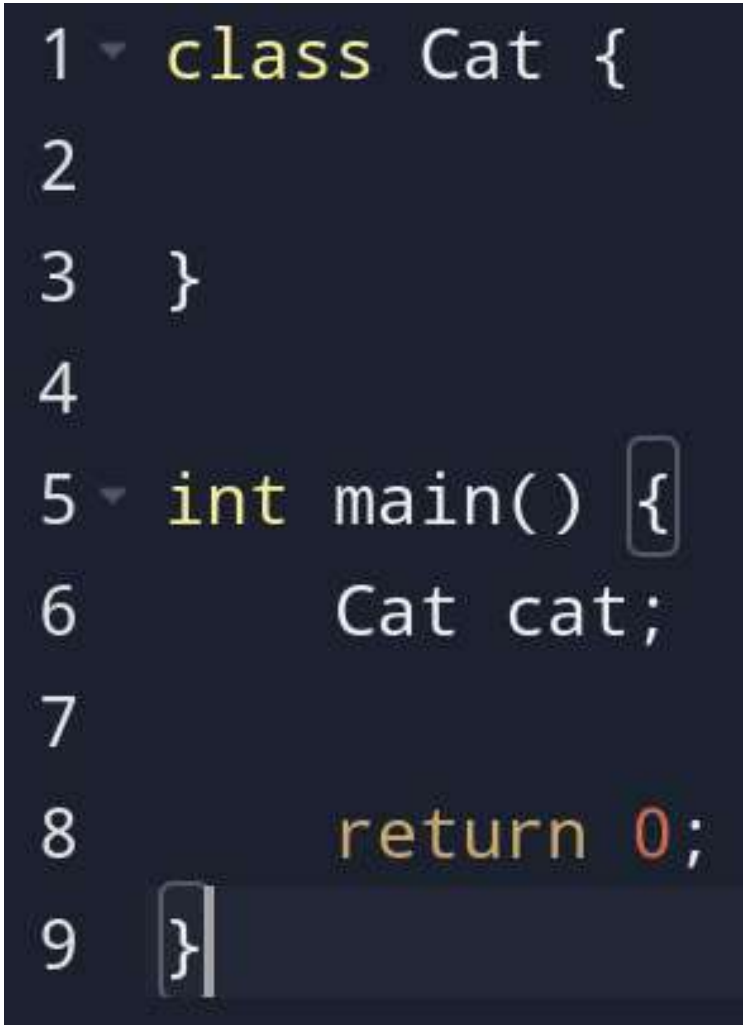
Объявление методов класса

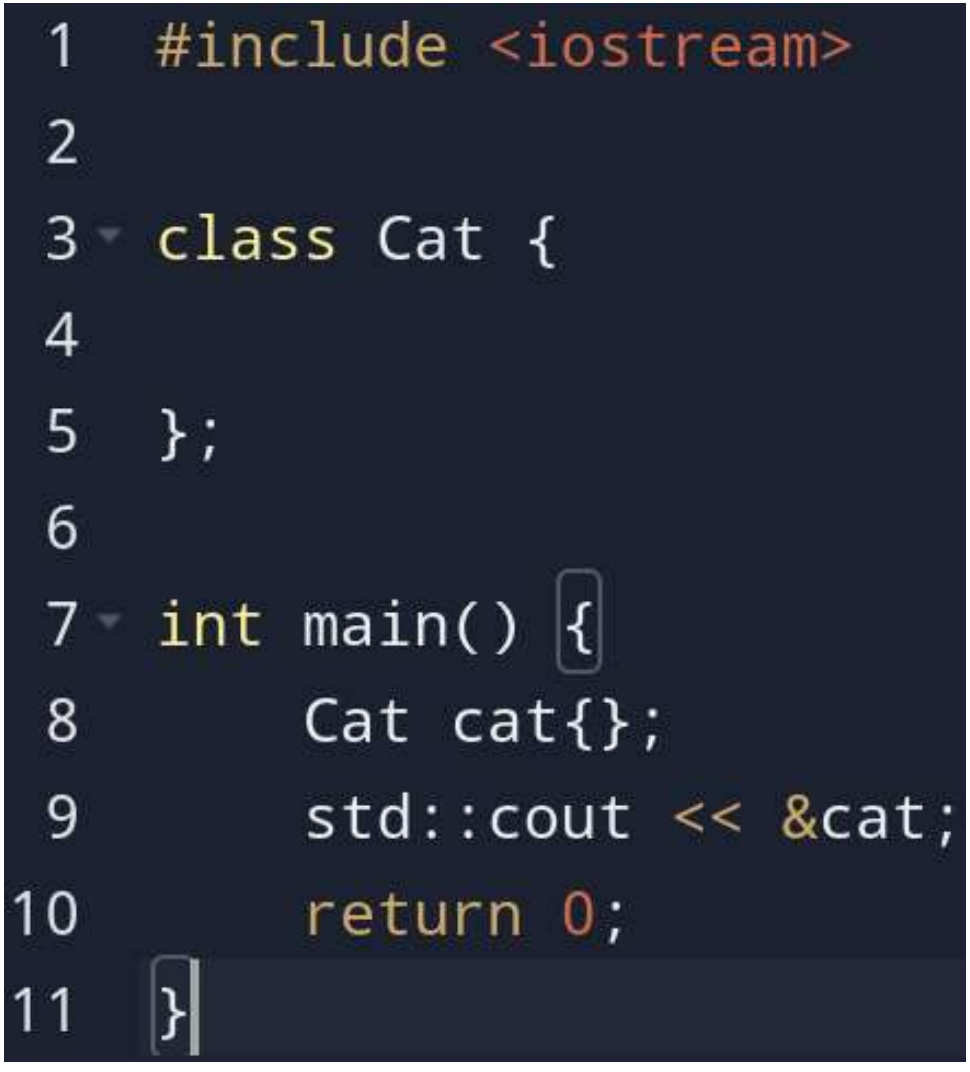
- разделение декларирование (.h/.hpp) и описания (.cpp/.cxx)
- объявления внутри класса
- объявления абстрактного метода -> абстрактный класс
- друзья класса

Пример

```
1   class Point {
2   public:
3       Point() {
4           x_ = .0f;
5           y_ = .0f;
6       }
7       Point(float x, float y);
8
9   private:
10      float x_;
11      float y_;
12  };
13
14  Point::Point(float x, float y) {
15      x_ = x;
16      y_ = y;
17  }
```

Вопросы


1 class Cat {
2
3 }
4
5 int main() {
6 Cat cat;
7
8 return 0;
9 }


1 #include <iostream>
2
3 class Cat {
4
5 };
6
7 int main() {
8 Cat cat{};
9 std::cout << &cat;
10 return 0;
11 }

```cpp
#include <iostream>

class A {
public:
    A() { std::cout << "A"; }
    ~A() { std::cout << "a"; }
};

class B {
public:
    B() { std::cout << "B"; }
    ~B() { std::cout << "b"; }
};

int main() {
    A* obj1 = new A();
    B* obj2 = new B();
    delete obj1;
    delete obj2;
    return 0;
}
```

```cpp
#include <iostream>
#include <string>

class Cat {
public:
    Cat(std::string name) : name_(name){}

    std::string getName() const {
        return name_;
    }
private:
    std::string name_;
};

int main() {
    Cat cat("Васька");
    std::cout << cat.getName();
    return 0;
}
```

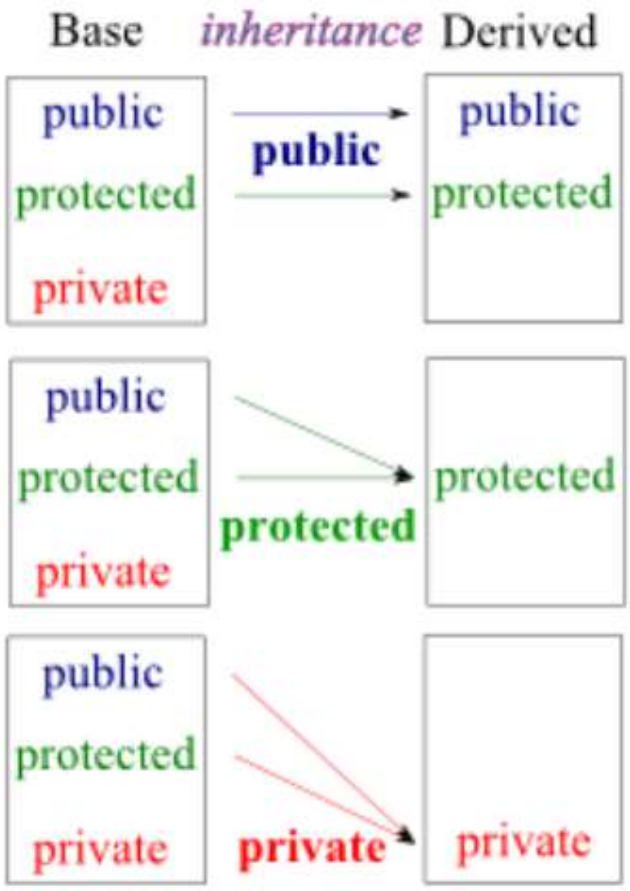
```cpp
#include <iostream>
#include <string>

class Cat {
public:
    Cat() = default;
    Cat(std::string name = "Васька") : name_(name){}

    std::string getName() const {
        return name_;
    }
private:
    std::string name_;
};

int main() {
    Cat cat;
    std::cout << cat.getName();
    return 0;
}
```

Наследование

- Одиночное, множественное
- Модификаторы доступа
- Использование базового класса
- Запрет на наследование, ключевое слово final

Пример 1

```
struct Base {
    int a, b, c;
};
// every object of type Derived includes Base as a subobject
struct Derived : Base {
    int b;
};
// every object of type Derived2 includes Derived and Base as subobjects
struct Derived2 : Derived {
    int c;
};
```


Base
inheritance
Derived
public
protected
private
public
public
protected
public
protected
private
protected
protected
public
protected
private
private
private

Пример 2

```
struct B { int n; };
class X : public virtual B {};
class Y : virtual public B {};
class Z : public B {};
// every object of type AA has one X, one Y, one Z, and two B's:
// one that is the base of Z and one that is shared by X and Y
struct AA : X, Y, Z {
    AA() {
        X::n = 1; // modifies the virtual B subobject's member
        Y::n = 2; // modifies the same virtual B subobject's member
        Z::n = 3; // modifies the non-virtual B subobject's member

        std::cout << X::n << Y::n << Z::n << '\n'; // prints 223
    }
};
```

Пример 3

```
struct B {
    int n;
    B(int x) : n(x) {}
};

struct X : virtual B {
    X() : B(1) {}
};

struct Y : virtual B { Y() : B(2) {} };

struct AA : X, Y {
    AA() : B(3), X(), Y() {}
};
```

Ключевое слово final

```
struct Base
{
    virtual void foo();
};

struct A : Base
{
    void foo() final; // Base::foo is overridden and A::foo is the final override
    void bar() final; // Error: bar cannot be final as it is non-virtual
};

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden as it is final in A
};

struct C : B // Error: B is final
{
};
```

Диаграмма Классов

Язык UML

Язык UML

Unified Modeling Language

Графический язык моделирования

Актуальная версия 2.5

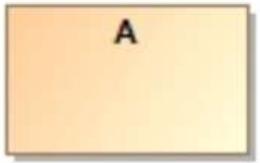
Диаграмма классов

Диаграмма классов (class diagram) — диаграмма языка UML, на которой представлена совокупность декларативных или статических элементов модели, таких как **классы** с атрибутами и операциями, а также связывающие их отношения.

Класс

Абстрактное описание множества однородных объектов, имеющих одинаковые атрибуты, операции и отношения с объектами других классов

```
class A {
};
```



Имя класса

- Должно быть уникальным в пределах пакета, который может содержать одну или несколько диаграмм классов
- Должно начинаться с заглавной буквы
- Абстрактный класс (abstract class) – используется курсив
- Если нужно указать пакет, то <Имя пакета>::<Имя класса>
- Стереотип или ссылка на стандартный шаблон указывается как <<singleton>>

Атрибуты класса

Атрибут (attribute) — содержательная характеристика класса, описывающая множество значений, которые могут принимать отдельные объекты этого класса.

[visibility] [/] name [: type] [multiplicity] [= default] [{property-string}]

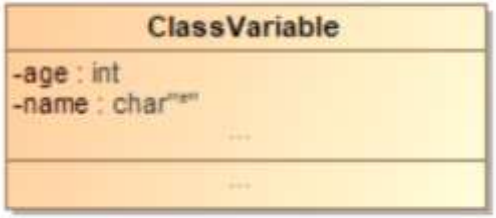
Символ "+" обозначает атрибут с областью видимости типа общедоступный (public),

Символ "#" - защищенный (protected),

Символ "-" - закрытый (private)

Пример

```
class ClassVariable {
 int age;
 char* name;
};
```




```
class CVModifiers {
const int* const constAttribute;
}
```

Операции класса

Операция (operation) - это сервис, предоставляемый каждым экземпляром или объектом класса по требованию своих клиентов, в качестве которых могут выступать другие объекты, в том числе и экземпляры данного класса.

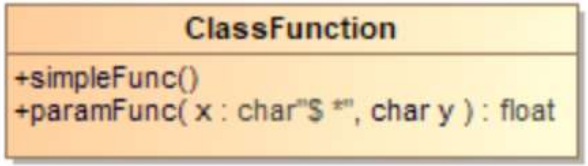
[visibility] name [(parameter-list)] [{property-string}]

Для параметров:

[direction] name : type [multiplicity] [= default-value]

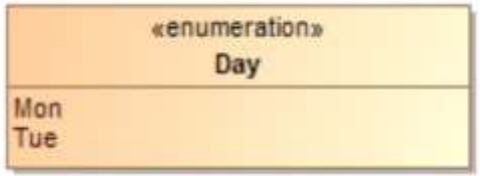
Пример

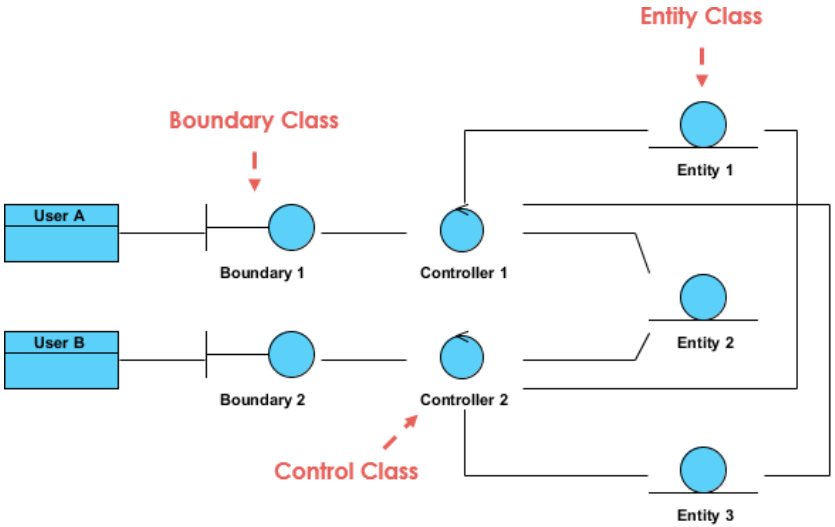
```
class ClassFunction {
public:
 void simpleFunc();
 float paramFunc(int x,char y);
};
```

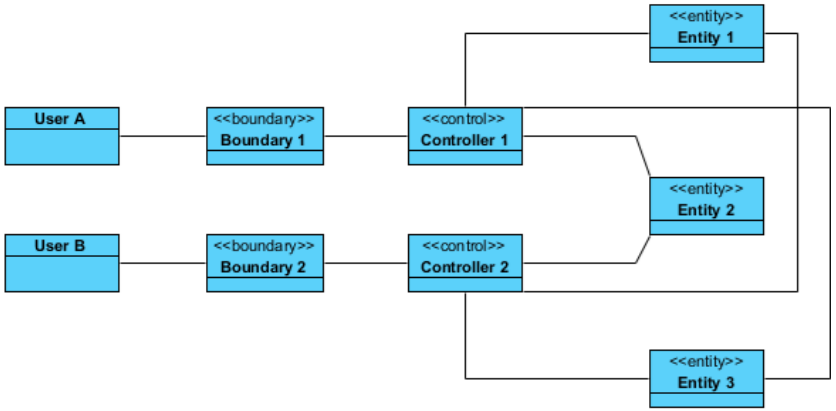


Стереотипы

```
enum Day {
      Mon,
      Tue=2
};
```

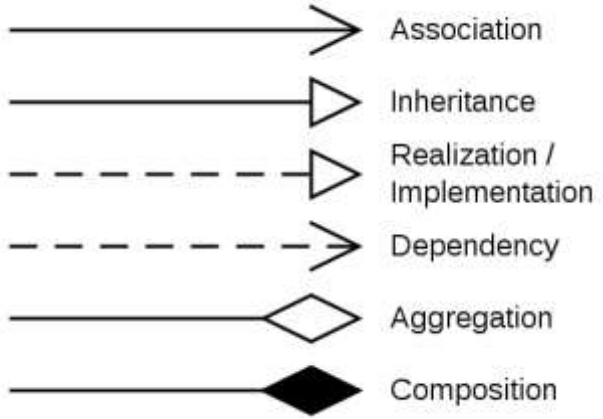






Отношения между классами

- Отношение зависимости (dependency relationship)
- Отношение ассоциации (association relationship)
- Отношение обобщения (generalization relationship)
- Отношение реализации (realization relationship)



Отношение зависимости

Отношение зависимости в общем случае указывает некоторое семантическое отношение между двумя элементами модели или двумя множествами таких элементов, которое не является отношением ассоциации, обобщения или реализации.

Отношение зависимости используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели

Отношение зависимости

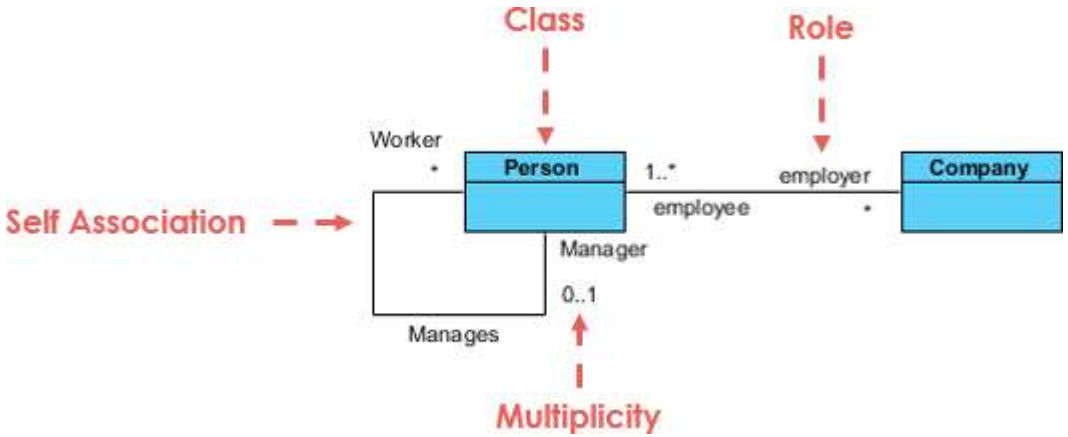
- "access" - служит для обозначения доступности открытых атрибутов и операций класса-источника для классов-клиентов;
- "bind" - класс-клиент может использовать некоторый шаблон для своей последующей параметризации;
- "derive" - атрибуты класса-клиента могут быть вычислены по атрибутам класса-источника;
- "import" - открытые атрибуты и операции класса-источника становятся частью класса-клиента, как если бы они были объявлены непосредственно в нем;
- "refine" - указывает, что класс-клиент служит уточнением класса-источника в силу причин исторического характера, когда появляется дополнительная информация в ходе работы над проектом.

Отношение ассоциации

Отношение ассоциации соответствует наличию некоторого отношения между классами

- Навигация
- Видимость
- Квалификатор
- Классы-ассоциации

Пример

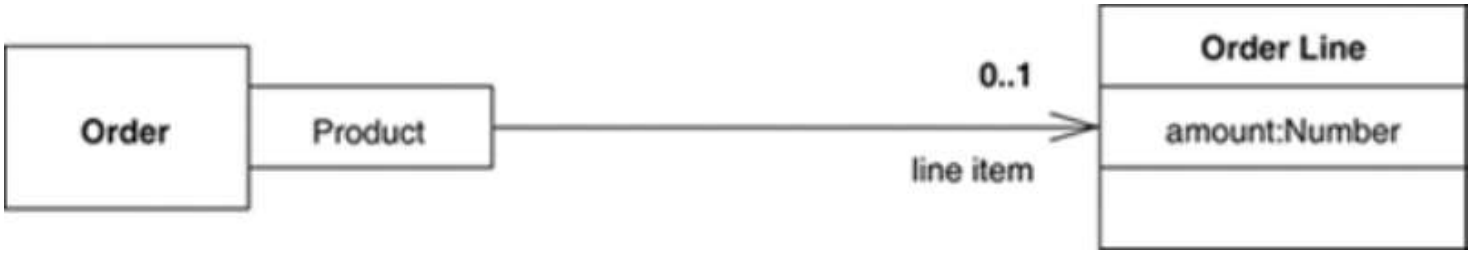

Class
Role
Worker
*
Person
1..*
employer
Company
employee
*
Self Association
Manager
0..1
Manages
Multiplicity

Квалификатор

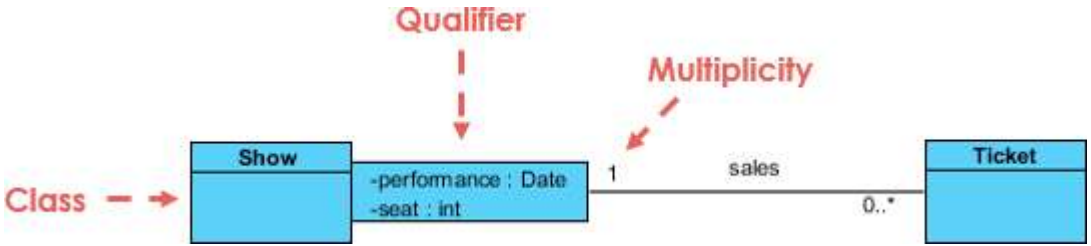
Как, зная объект на одном конце ассоциации, можно определить объект или группу объектов на другом ее конце?

Если на одном конце ассоциации можно поместить поисковую структуру данных (например, хэш-таблицу или в-дерево), то объявляйте индекс, по которому производится поиск, как квалификатор.

Пример квалификатора

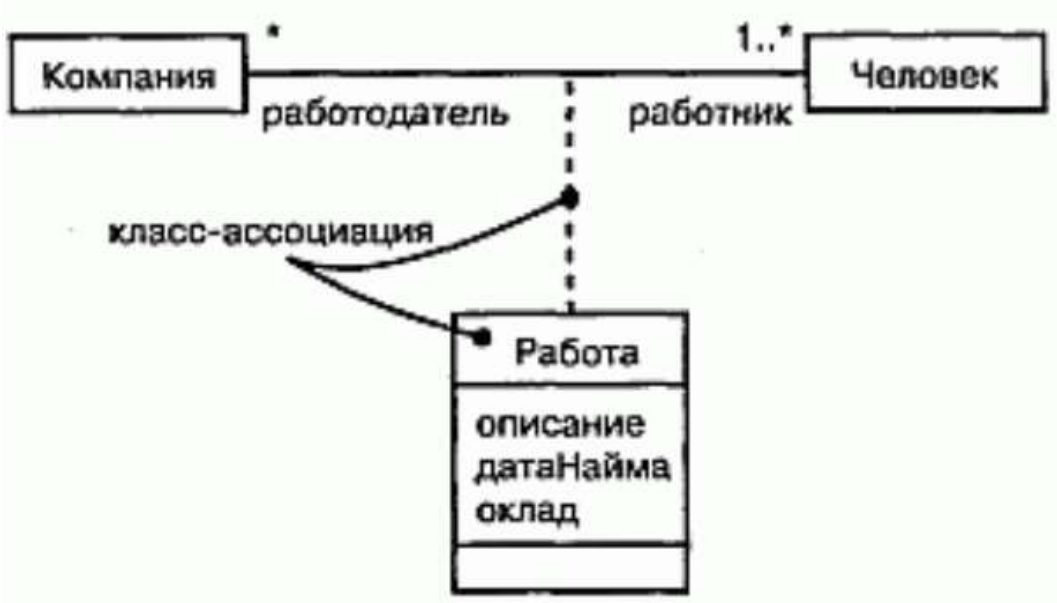



```
class Order ...
public OrderLine getLineItem(Product aProduct);
public void addLineItem(Number amount, Product forProduct);
```



Классы-ассоциации

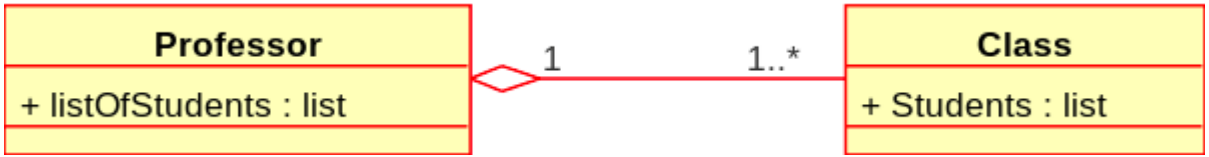
В ассоциации между двумя классами сама ассоциация также может иметь свойства



Отношение агрегации

Отношение агрегации имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности.

Отношение типа "часть-целое"



Отношение композиции

Частный случай отношения агрегации

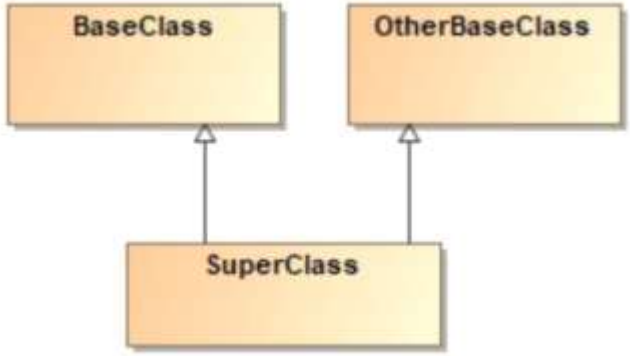
Составляющие части в «некотором смысле» находятся внутри целого

Части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части

Отношение обобщения

Отношение обобщения – отношение между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком)

```
class BaseClass {};
class OtherBaseClass {};
class SuperClass :
 public BaseClass,
 protected virtual OtherBaseClass {
};
```



Интерфейс

Интерфейсом (Interface) называется набор операций, используемый для специфицирования услуг, предоставляемых классом или компонентом

Отношение реализации

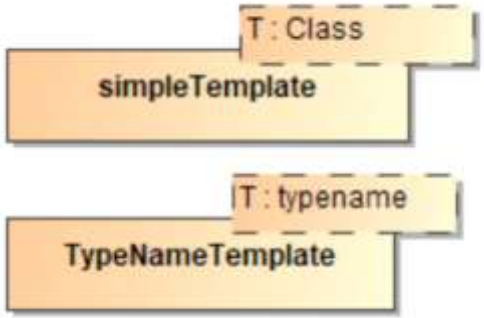
Реализацией (Realization) называется семантическое отношение между классификаторами, при котором один из них описывает контракт, а другой гарантирует его выполнение

Семантически реализация - это нечто среднее между обобщением и зависимостью

Параметризованный класс

Параметризованный класс (parametrized class) предназначен для обозначения такого класса, который имеет один (или более) нефиксированный формальный параметр.

```
template <class T>
class simpleTemplate {
};
template <typename T>
class TypeNameTemplate {
};
```



Объект

Объект (object) является отдельным экземпляром класса

Он имеет свое собственное имя и конкретные значения атрибутов.