

# Принципы повторного использования и полиморфизм

1. Способы повторного использования классов
2. Композиция и агрегация
3. Наследование/Обобщение. Виды наследования
4. Создание и уничтожение объекта при наследовании
5. Принцип полиморфизма
6. Перегрузка и переопределение методов
7. Виртуальные функции
8. Статическое и динамическое связывание
9. Повторное использование интерфейса и реализации

Преподаватель:

Ботов Дмитрий Сергеевич

# Всегда ли нужно создавать новый класс?

Задача

Требуется создать  
новый класс

Решение

Проектирование  
новых классов **«с нуля»**,  
с последующим кодированием  
и тестированием

Использование для создания  
новых классов уже  
**существующих классов**,  
хорошо зарекомендовавших  
себя

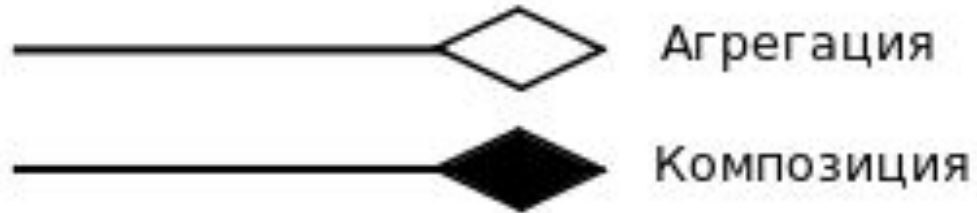
# Зачем использовать существующие классы?

---

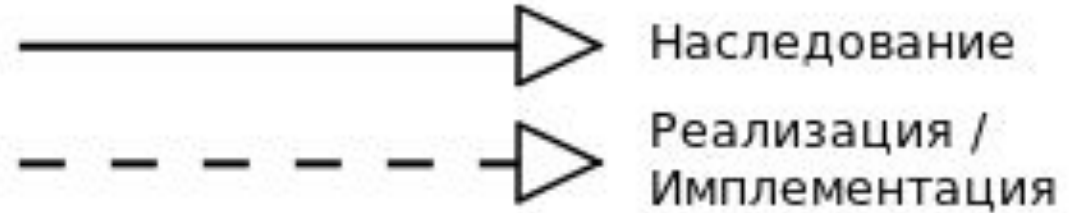
- Повторное использование ранее принятых решений
- Делает решение гибким и мобильным
- Существующие классы, как правило, хорошо отлажены и показали себя в работе

# Способы повторного использования классов

---



- «содержит»
- «является частью»
- «реализуется посредством»



- «является» («is a»)
- «частное / общее»
- «реализует интерфейс»

# Композиция и агрегация

---

**Композиция/агрегация** – это отношение между типами, которое возникает тогда, когда объект одного типа содержит в себе объекты других типов

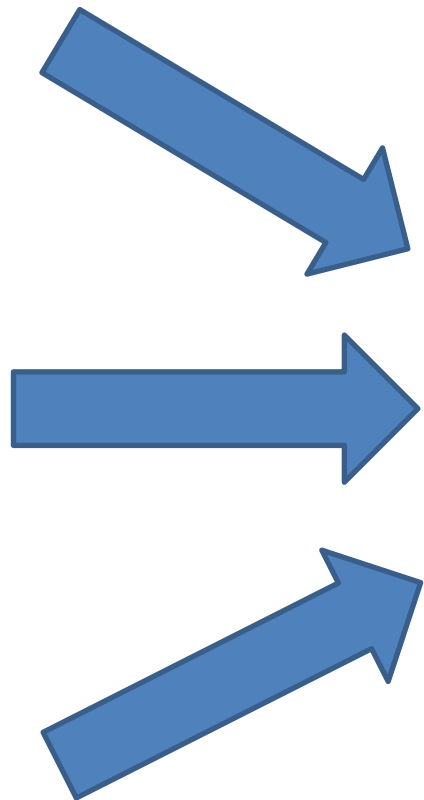
**Правило:** моделируйте отношение «содержит» или «реализуется посредством» с помощью композиции/агрегации

# Пример композиции

```
// Колесо  
class CWheel  
{  
...  
};
```

```
// Двигатель  
class CEngine  
{  
...  
};
```

```
// Кузов  
class CBody  
{  
...  
};
```



```
// Автомобиль  
class CAutomobile  
{  
public:  
...  
private:  
CBody m_body;  
CEngine m_engine;  
CWheel m_wheels[4];  
};
```

# Пример агрегации

```
class Professor;

class UniversityDepartment {
    ...
private:
    Professor** professors; // массив указателей на объекты, живущие своей жизнью
public:
    /* constructor */
    UniversityDepartment (int profAmount) {
        professors = new Professor[ profAmount ];
        for (int pi = 0; pi < profAmount; pi++)
            professors[ pi ] = NULL;
    } // constructor
    /* destructor */
    ~UniversityDepartment() {
        delete[] professors; // удаляем массив, но не сами объекты в нём
    } // destructor
    ...
}; // class UniversityDepartment ...
```

# Пример композиции

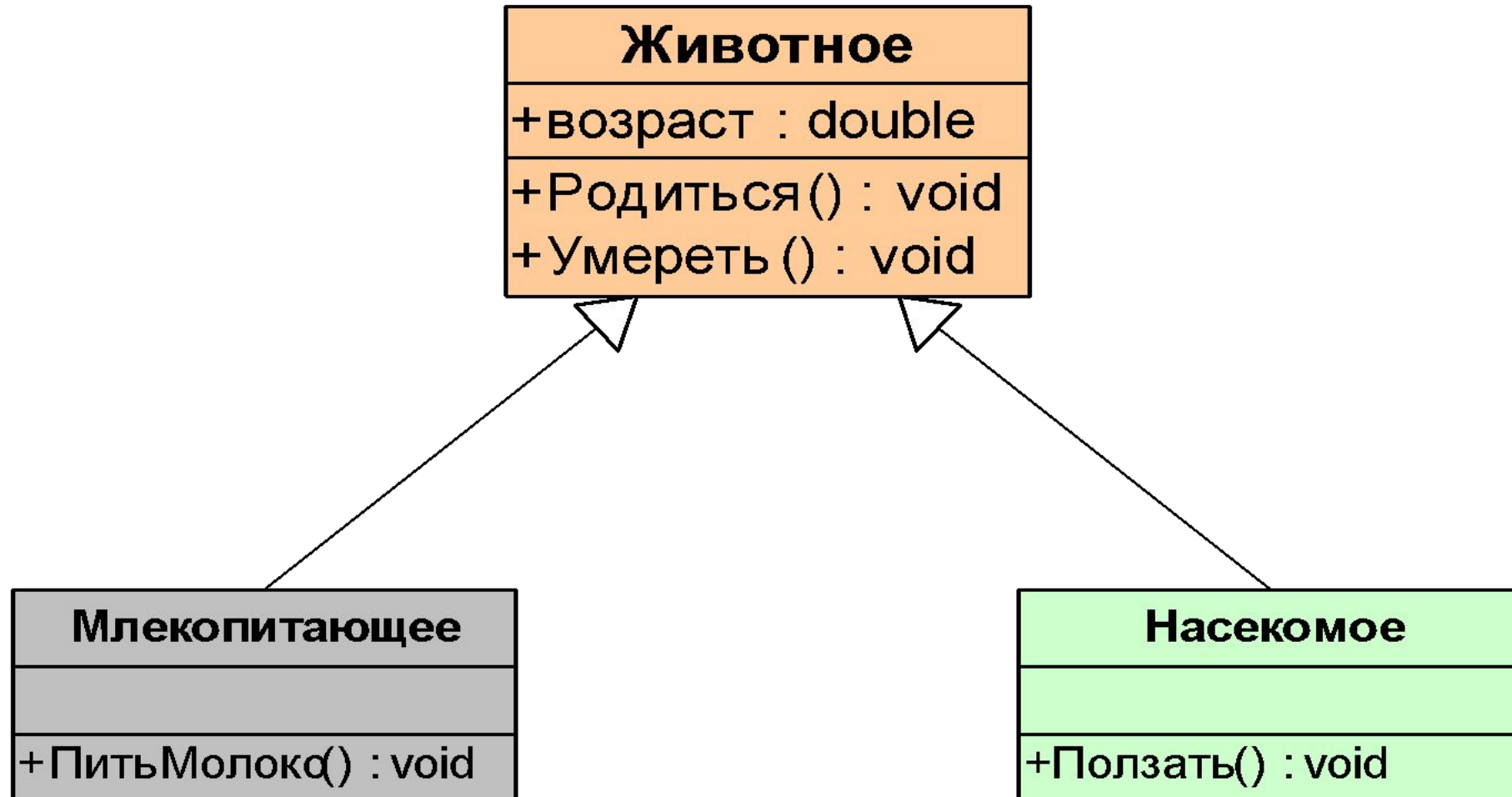
```
class Employee;
class UniversityDepartment;

class University {
    ...
private:
    Employee rector; // объект создается автоматически
    UniversityDepartment* departments; // массив экземпляров, а не указателей на них
public:
    /* constructor */
    University (int depAmount) {
        departments = new UniversityDepartment [ depAmount ];
    }// constructor
    /* destructor */
    ~University () {
        delete[] departments; // все департаменты уничтожаются
        // объект rector уничтожается
    }// destructor
    ...
};// class University ...
```



# Понятие обобщения/наследования

Связь типа «является» («is a») или «частное/общее»



# Пример наследования

---

```
class Animal {
private:
    int age;           // возраст
    ...
};

class Mammal : public Animal {    // млекопитающее
public:
    int GetAge() { return age; }  // можно вынести функцию в родителя
    void SuckMilk() { ... }
    ...
};

class Insect : public Animal {    // насекомое
public:
    int GetAge() { return age; }
    void Crawl() { ... }
    ...
};
```

# Виды наследования

---

Объекты дочернего класса (подкласса) наследуют **все свойства и поведение** родительского класса (суперкласса)

## Типы наследования:

- **public** – **открытое наследование**  
все модификаторы доступа полей и методов суперкласса в подклассе остаются без изменений
- **protected** – **защищенное наследование**  
*public* поля и методы суперкласса в подклассе становятся *protected*
- **private** – **закрытое наследование**  
все поля и методы суперкласса в подклассе становятся *private*

```
class Pub : public A {};
```

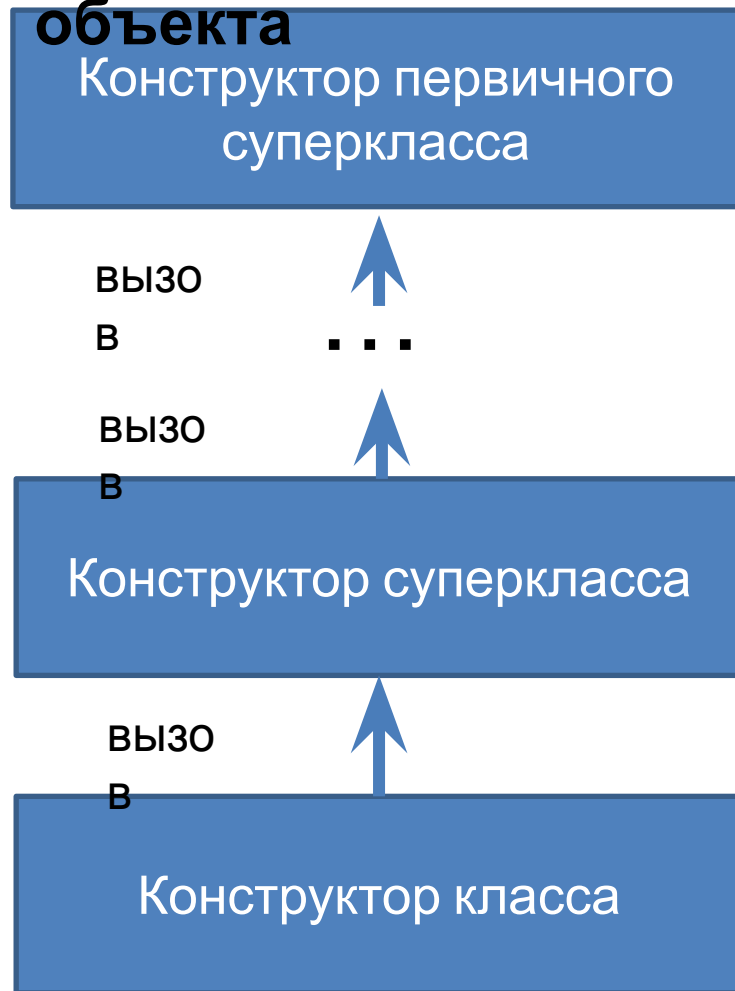
```
class Pro : protected A {};
```

```
class Pri : private A {};
```

# Создание и уничтожение объекта при наследовании

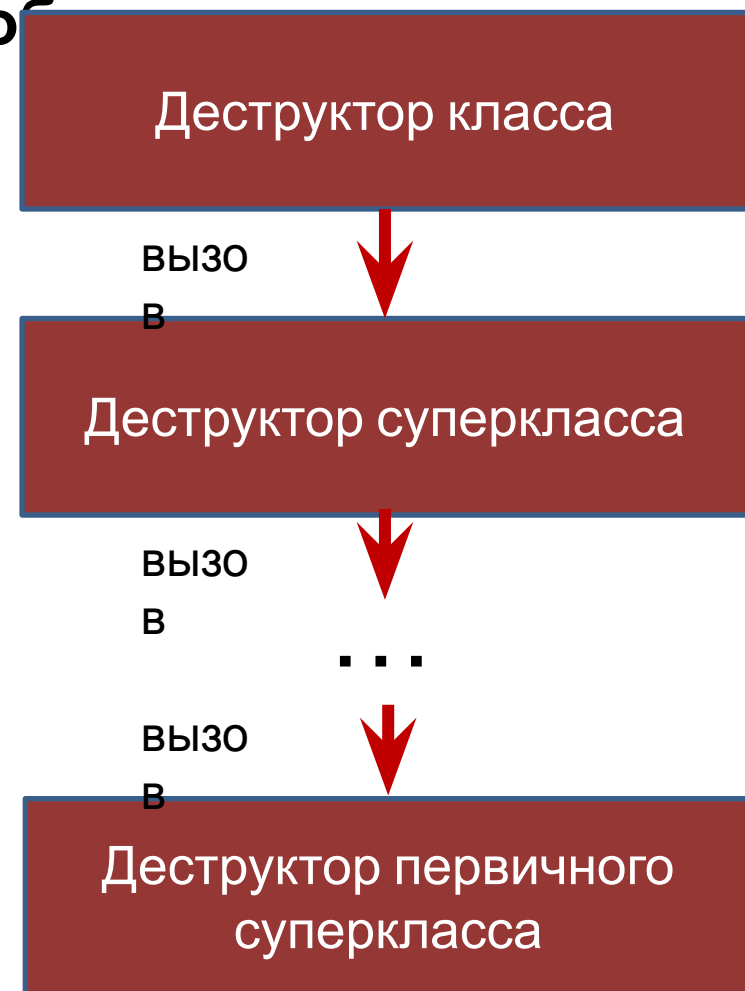
## Создание

### объекта

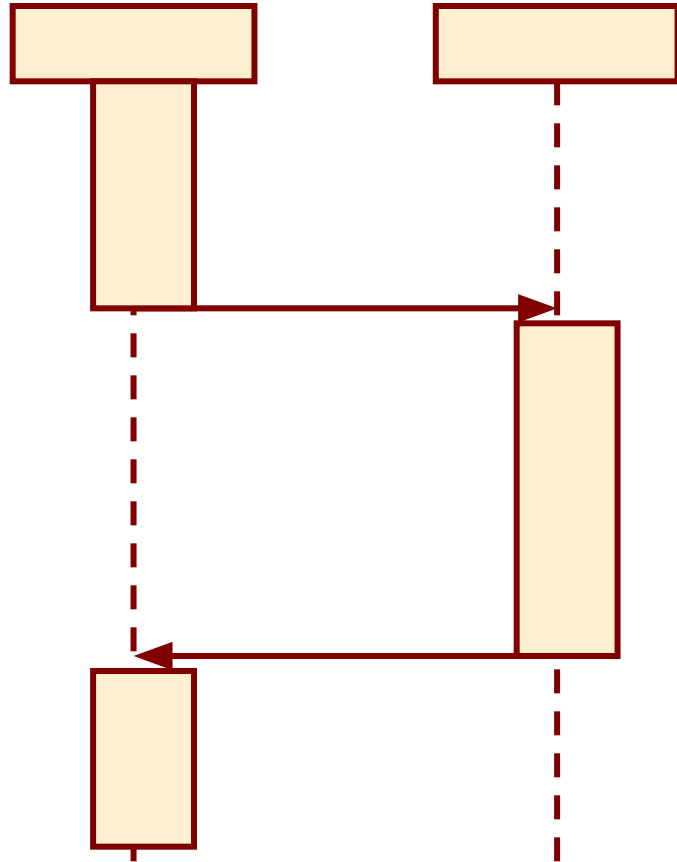


## Уничтожение

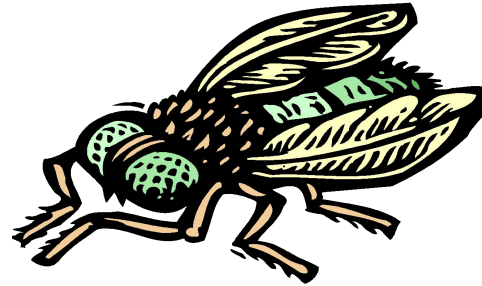
### об



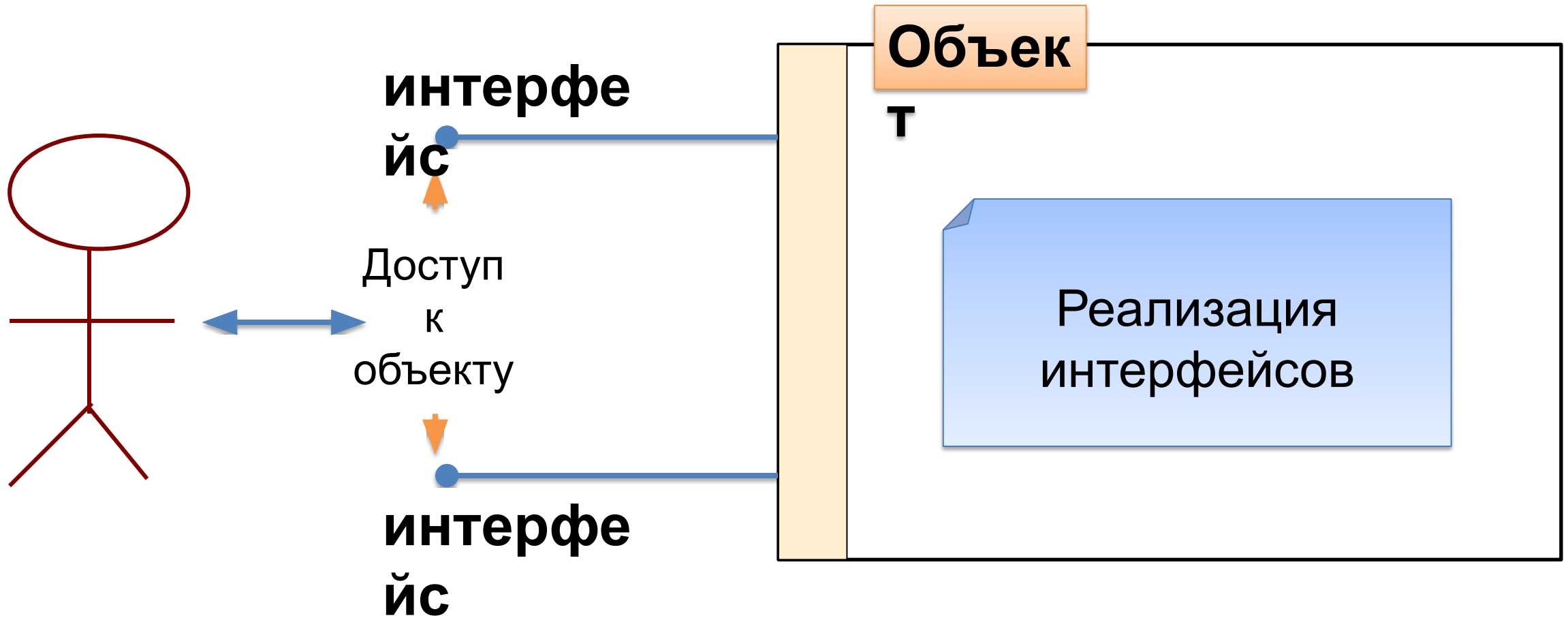
# Дискуссионный вопрос



Что общего у этих объектов?



# Интерфейс и его реализация



# Понятие интерфейса в ООП

---



**Интерфейс** — это конструкция в коде программы, используемая для специфицирования услуг, предоставляемых классом или компонентом

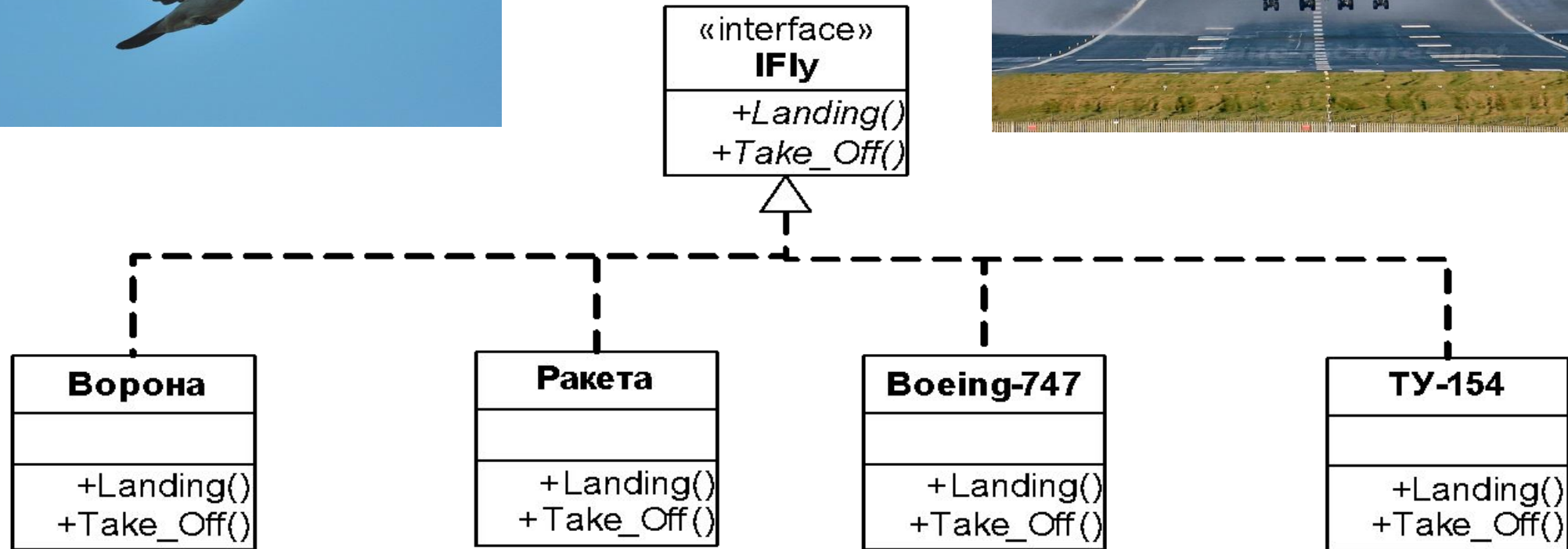
# Понятие интерфейса в ООП

---

- Интерфейс определяет **границу взаимодействия** между классами или компонентами, специфицируя определенную **абстракцию**
- Операции интерфейса **реализуются** (алгоритмизируются) конкретными классами, которые поддерживают интерфейс
- Класс (и все его объекты) могут иметь один или несколько интерфейсов



# Пример применения интерфейса



# Полиморфизм

---

## Полиморфизм

(от др.греч. — *многообразный*) — свойство, которое позволяет использовать один и тот же **интерфейс** для общего класса действий

**"один интерфейс, несколько методов"**

- Можно спроектировать общий интерфейс для группы связанных между собой действий
- Конкретное действие определяется конкретным характером ситуации

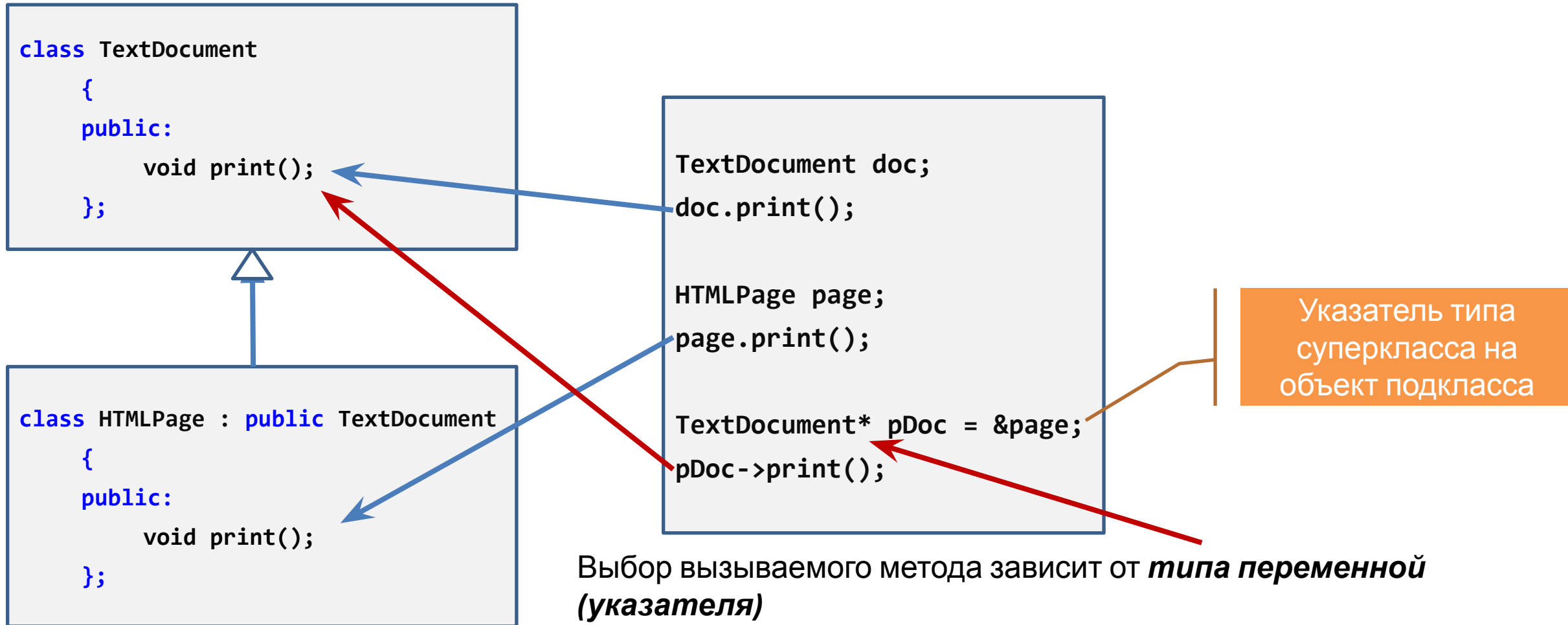
# Перегрузка (overloading) функций (методов)

---

- Определение функций (методов) с одинаковым именем, но с разным списком параметров (разные типы и/или количество параметров)
- Способ борьбы со сложностью кода

```
class Printer
{
public:
    void print(string str);
    void print(Document doc);
    void print(HTMLpage page);
};
```

# Наследование обычных функций (методов) с перекрытием имен



# Переопределение (overriding) виртуальных функций (методов)

```
class TextDocument
{
public:
    virtual void print();
};
```

```
class HTMLPage : public TextDocument
{
public:
    void print();
};
```

## Переопределение (overriding)

действует для *виртуальных функций*  
(методов)

с одинаковым именем и списком параметров

```
TextDocument* pDoc = new TextDocument();
pDoc->print();
```

```
TextDocument* pDoc = new HTMLPage();
pDoc->print();
```

Выбор вызываемого метода зависит от *типа созданного объекта*

# Связывание (binding)

## Статическое связывание:

случай **overloading**

```
MyClass* obj = new MyClass;  
obj->method(5);
```

```
void MyClass::method(char c) {...}
```

```
void MyClass::method(int k) {...}
```

Выбор реализации метода зависит от **списка фактических параметров**

## Динамическое связывание:

случай **overriding**

```
SubClass obj = new SubClass;  
obj->method(5);
```

```
virtual void SuperClass::method(int k)  
{  
    // реализация «по умолчанию»  
}
```

```
void SubClass::method(int k)  
{  
    // переопределенная  
    // реализация  
}
```

Выбор реализации метода зависит от **типа созданного объекта**

# Method Overloading vs. Overriding

	Overloading	Overriding
Определение	Определение функций с одинаковым именем, но с разным списком параметров (разные типы и/или количество параметров)	Определение в подклассе метода с именем и списком параметров, совпадающим с методом суперкласса, но отличающимся реализацией в коде
Поведение	Добавляет/Расширяет поведение существующих методов для другого набора/типов данных	Изменяет существующее поведение метода
Сигнатуры методов	Сигнатуры должны отличаться списком параметров	Сигнатуры не должны отличаться
Наследование	Необязательно	Обязательно
Связывание	Статическое	Динамическое
Полиморфизм	Времени компиляции	Времени выполнения

# Полиморфизм: шаги реализации (1)

1. Определить *иерархию наследования* классов
2. *Переопределить* в подклассах *виртуальные функции*, унаследованные от суперкласса

```
class TextDocument
{
public:
    virtual void print();
};
```

```
class HTMLPage : public TextDocument
{
public:
    HTMLPage(string URL);
    virtual void print();
};
```

```
class PDFDocument : public TextDocument
{
public:
    PDFDocument(File* pFile);
    virtual void print();
};
```



# Полиморфизм: шаги реализации (2)

---

3. Объявить указатель (или массив указателей) **типа суперкласса**
4. Создать **объект нужного подкласса** (сохранив адрес в объявленном ранее указателе)
5. Вызвать **переопределенный метод** через указатель на

```
TextDocument* pDoc;  
  
if( /*условие*/ )  
    pDoc = new HTMLPage(url);  
else  
    pDoc = new PDFDocument(pFile);  
  
pDoc->print();
```

# Абстрактные классы - интерфейсы

```
class Printable
{
public:
    virtual void print() = 0;
};
```

Чисто виртуальная  
функция (абстрактная)

*Отношение реализации*

```
class String : public Printable
{
public:
    String(string str);
    virtual void print();
};
```

```
class Document : public Printable
{
public:
    Document(File* pFile);
    virtual void print();
};
```

```
class HTMLPage : public Printable
{
public:
    HTMLPage(string url);
    virtual void print();
};
```

# Повторное использование в C++

---

- **Правило 1:** Используйте открытое наследование для моделирования отношения «является»
- **Правило 2:** Моделируйте отношение «содержит» или «реализуется посредством» с помощью композиции (агрегации)
- **Правило 3:** Различайте наследование интерфейса от наследования реализации

# Повторное использование реализации и интерфейса

Способ	Реализация	Интерфейс
Агрегация/ композиция	+	-
Наследование не виртуальных функций	+(обязательная)	+
Наследование виртуальных функций	+(по умолчанию)	+
Наследование чисто виртуальных функций	-	+
Закрытое наследование	+	-

# Резюме: рассмотренные вопросы

---

- Зачем использовать существующие классы?
- Какие существуют способы повторного использования?
- В каком случае стоит применять наследование, а в каком – композицию или агрегацию?
- Какие существуют виды наследования?
- В чем заключается принцип полиморфизма?
- Что такое перегрузка и переопределение методов?
- Чем отличаются виртуальные и неvirtуальные функции при наследовании?
- В чем отличие использования статического и динамического связывания?
- Каковы шаги реализации полиморфизма в программном коде?
- Как обеспечить повторное использование реализации и интерфейса?