

Тест по прошлой лекции



Санкт-Петербургский
государственный
университет



<https://forms.gle/LH8DoMGNxCMYXHK69>





Парадигма ООП первая часть



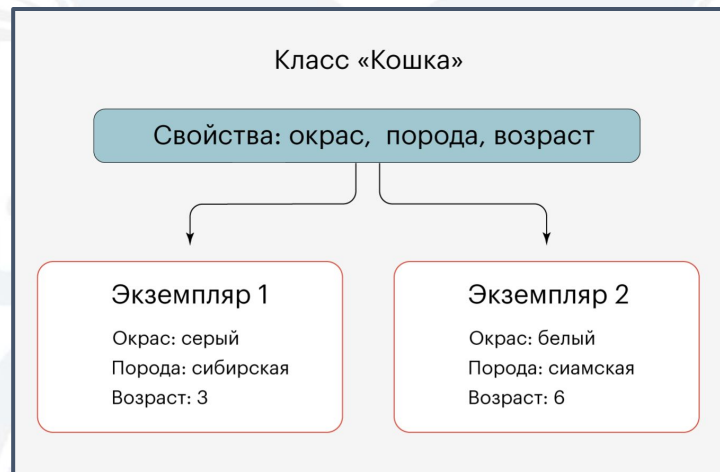
Наследование и инкапсуляция

Направление «Искусственный интеллект и наука о данных», 23.Б16-мм, 23.Б18-мм

17.11.2023



- **Объектно-ориентированное программирование** (сокращённо ООП) — это парадигма разработки программного обеспечения, согласно которой **приложения состоят из объектов**
- **Класс** — это тип данных, созданный пользователем. Он содержит свойства и методы. Можно воспринимать его как чертёж
- **Объект** — это экземпляр класса, то есть созданный по чертежу (классу) *предмет* с определёнными свойствами и методами





ООП базируется на нескольких принципах:

1. **Всё** является **объектом**
2. **Вычисления осуществляются путем взаимодействия** (обмена данными) **между объектами**, при котором один объект требует, чтобы другой объект выполнил некоторое действие. Объекты **взаимодействуют, посылая и получая сообщения**. Сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия
3. Каждый объект имеет **независимую память**, которая состоит из других объектов

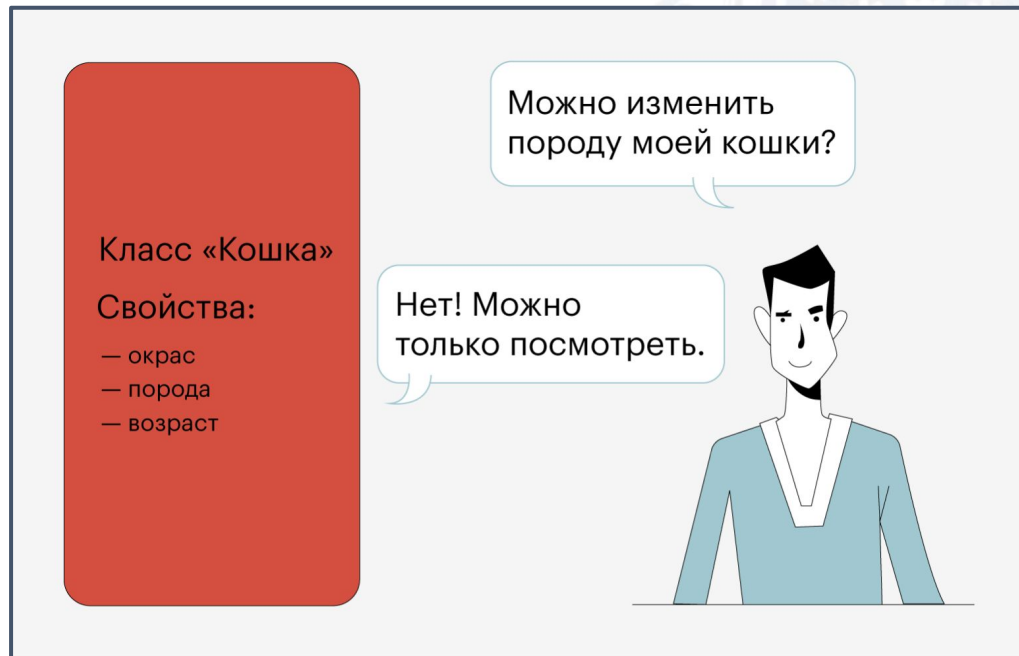




ООП базируется на нескольких принципах:

4. Каждый **объект является представителем** (экземпляром) **класса**, который выражает общие свойства объектов
5. **В классе задается поведение** (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия
6. **Классы организованы в единую древовидную структуру с общим корнем**, называемую **иерархией наследования**. Память и поведение, связанное с экземплярами определённого класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.





Класс «Кошка»

Свойства:

- окрас
- порода
- возраст

Можно изменить породу моей кошки?

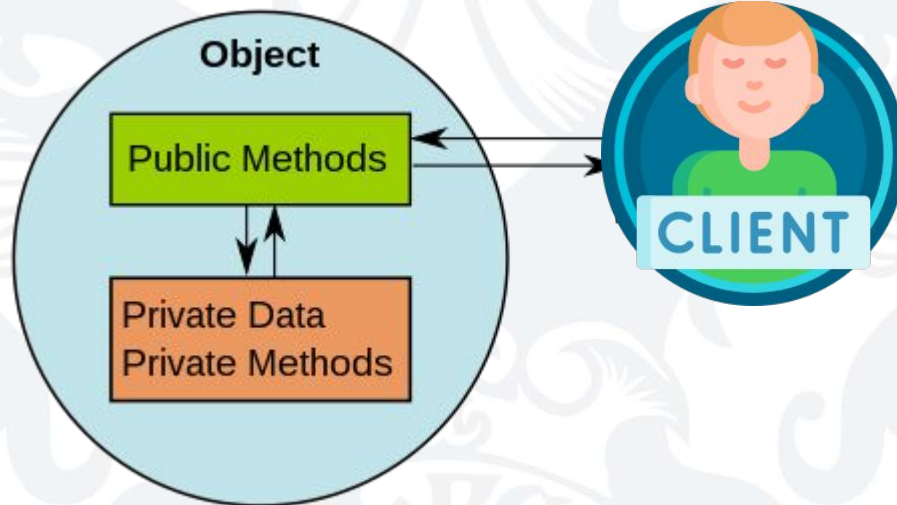
Нет! Можно только посмотреть.

The diagram shows a red box representing a class 'Cat' with its properties: color, breed, and age. A person is asking if the breed can be changed, and the response is that only the breed can be viewed, not changed, illustrating the principle of encapsulation.

- **Инкапсуляция** обеспечивает **сокрытие деталей реализации** (наружу видно только интерфейс объекта, или набор интерфейсов), и **сбор всех данных и методов их обработки в одном месте** — в самом объекте
- **Объект** можно представлять себе как **черный ящик**, у которого есть какие-то входы, куда можно послать какие-то данные и получить ответ

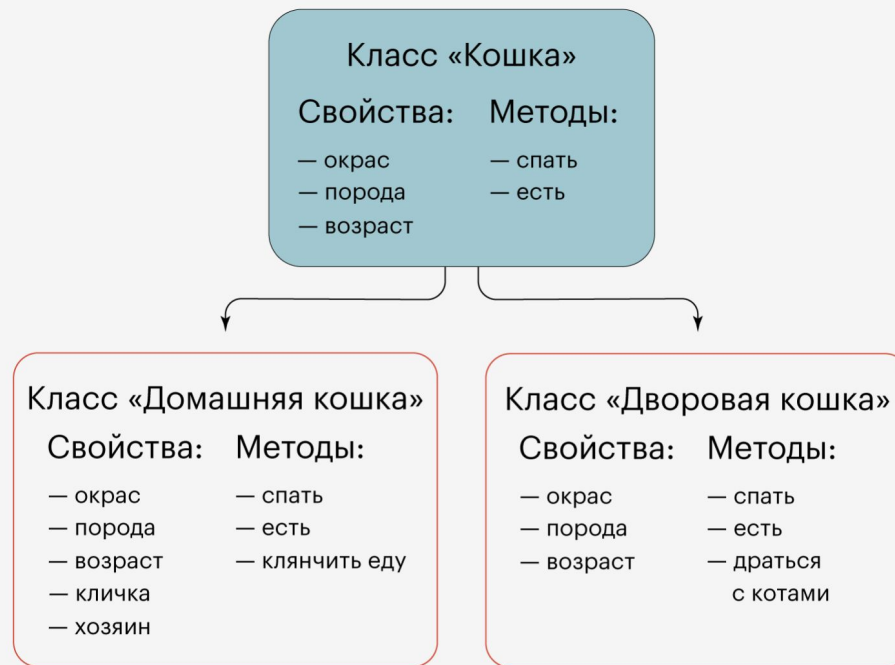


- Слово «**инкапсуляция**» происходит от латинского *in capsula* — «**размещение в оболочке**»
- Инкапсуляция – **заключение данных и функциональности в оболочку**. В ООП в роли оболочки выступают классы
- Распространенные **публичные метод** класса — *getter* и *setter*



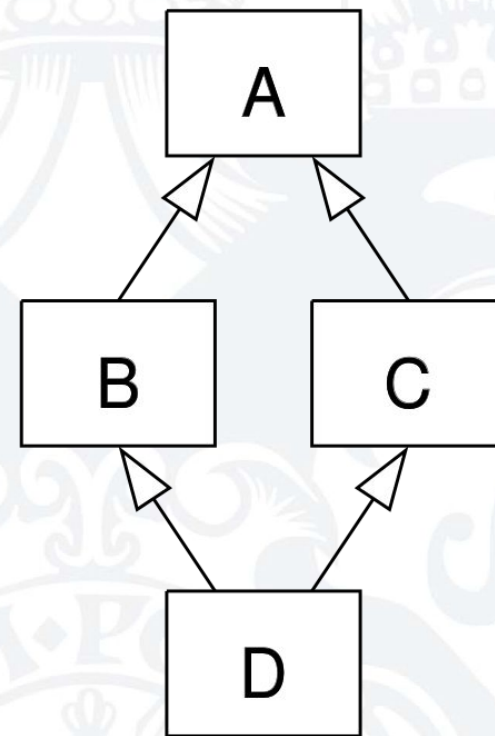


- Наследование — свойство **свойств и методов класса**
- При наследовании родительского класса в дочерние **передаются** определенные в родительском методы и свойства
- При этом, дочерние классы могут иметь **свои собственные** методы и свойства, а также **переопределять родительские**





- **А что если родительских классов много?** Причем они еще и находятся на разных уровнях иерархии наследования
- **Множественное наследование** — свойство, поддерживаемое частью объектно-ориентированных языков программирования, когда **класс может иметь более одного суперкласса** (непосредственного класса-родителя)
- **C++** требует, чтобы программист **явно указал**, элемент (свойство или метод) какого из родительских классов должен использоваться при отсутствии реализации
- А вот **Python** позволяет избежать такой записи. Он **использует алгоритм C3-линеаризации**





- **Интерфейс** - это **контракт**: человек, пишущий интерфейс, говорит: "Я принимаю всё, что выглядит именно так", а человек, использующий интерфейс, говорит: "Хорошо, класс, который я пишу, выглядит именно так"
- Интерфейс - это пустая оболочка. Есть только сигнатуры методов, что подразумевает, что у методов нет тела. Это просто шаблон.

```
interface MotorVehicle
{
    void run();
    int getFuel();
}

class Car implements MotorVehicle
{
    int fuel;
    void run() { print("Wrrroooooooooom"); }
    int getFuel() { return this.fuel; }
}
```

- **Абстрактные классы**, в отличие от интерфейсов, являются классами. Они во многом похожи на интерфейсы, но у них есть нечто большее: вы можете **определить для них часть поведения**
- Это больше похоже на то, как человек говорит: "эти классы должны выглядеть так, и у них есть вот эти общие методы, так что заполните пробелы (абстрактные функции и свойства)"

```
abstract class MotorVehicle
{
    int fuel;
    int getFuel() { return this.fuel; }
    abstract void run();
}

class Car extends MotorVehicle
{
    void run() { print("Wrrroooooooooom"); }
}
```



- **Класс** — это тип данных (чертеж), созданный пользователем, а объект — созданный по чертежу предмет с определенными свойствами и методами. **Приложения состоят из объектов**
- **Инкапсуляция** обеспечивает **сокрытие деталей реализации и сбор всех данных и методов их обработки в одном месте** — в самом объекте
- При **наследовании** от родительского класса **в дочерний передаются методы и свойства**. Дочерние классы в праве переопределить эти методы (а могут просто не использовать часть из них) и иметь свои собственные
- **Интерфейс** можно понимать как **контракт**. **Абстрактный класс** — это класс с хотя бы одним абстрактным методом. **Из него нельзя сделать экземпляр**, но можно унаследовать, заполнить пробелы и использовать как родительский





Парадигма ООП первая часть



Типизация и полиморфизм

Направление «Искусственный интеллект и наука о данных», 23.Б16-мм, 23.Б18-мм

17.11.2023



```
string plus(string a, string b)
{
    return a + b;
}

plus("some text ", "another text");
```

Статическая типизация

C#, Java, C++

```
function plus(a, b) {
    return a + b;
}

plus('some text ', 'another text');
```

Динамическая типизация

JavaScript, Python



```
string plus(string a, string b)
{
    return a + b;
}

plus("some text ", "another text");
plus(12, 34); /// нельзя даже скомпилировать
```

Статическая типизация

C#, Java, C++

```
function plus(a, b) {
    return a + b;
}

plus('some text ', 'another text');
plus(12, 34); /// 46
```

Динамическая типизация

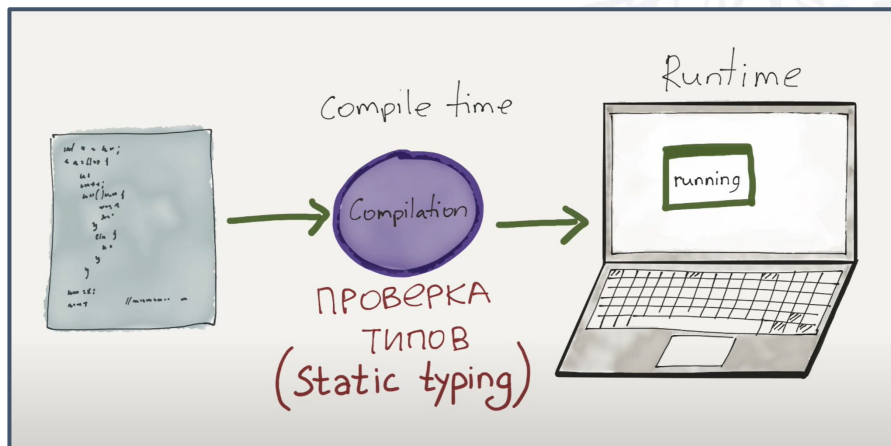
JavaScript, Python



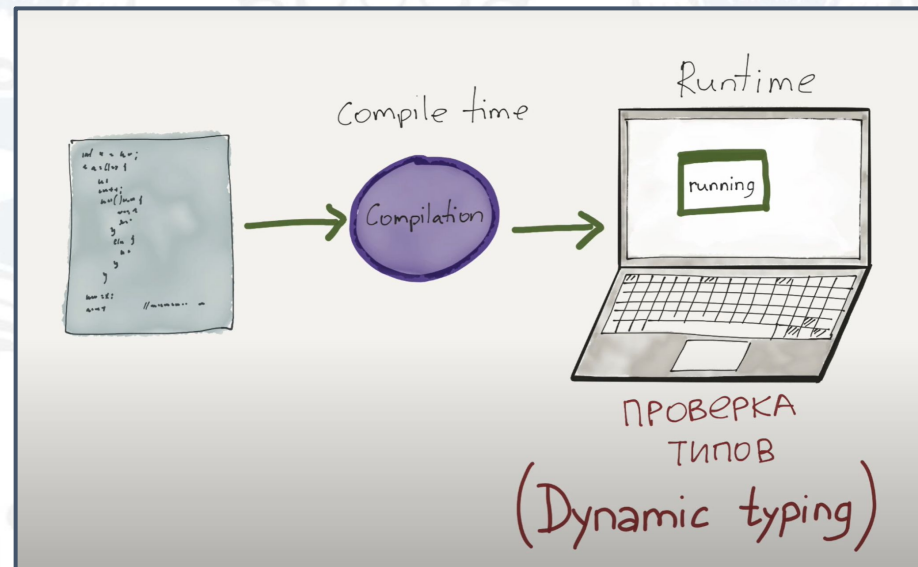
```
dynamic plus(dynamic a, dynamic b)
{
    return a + b;
}

plus("some text ", "another text"); /// some text another text
plus(12, 34) /// 46
```

- Ключевое слово *dynamic* обеспечивает **полный уход от статической типизации** в C#
- При использовании в объявлении переменной оно указывает компилятору вообще **не обрабатывать тип переменной**: тип должен быть таким, каким он окажется в период выполнения (*runtime*)



При **статической** типизации проверка типов происходит **на уровне компиляции**



При **динамической** типизации проверка типов происходит **во время работы программы**

Статическая

- Статическая типизация **предотвращает ошибки типов на стадии компиляции** и не дает проникнуть багам в продакшен



- С динамической типизацией **легче и быстрее писать код** для определенных задач: обучить небольшую модель данных, написать небольшой парсер



Динамическая

Сильная и слабая типизации



Санкт-Петербургский
государственный
университет

```
4 + '7';      // '47'  
4 * '7';      // 28  
2 + true;     // 3  
false - 3;    // -3
```

Неявные преобразования типов
в JavaScript

4 → '4'
'4' + '7' → '47'

JavaScript имеет представление о типах
данных при несовпадении из всех сил
старается как-то их сконвертировать

Сильная и слабая типизации



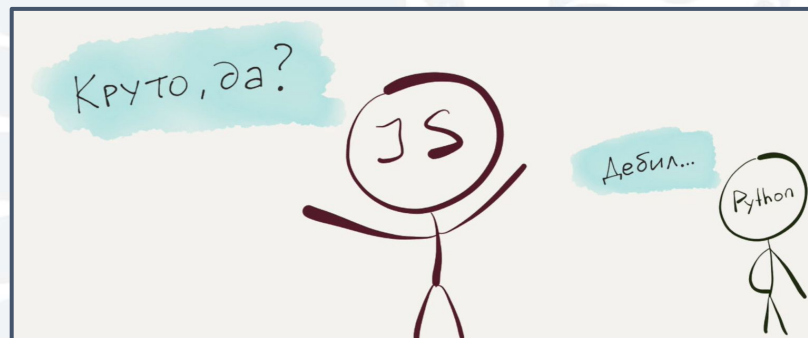
Санкт-Петербургский
государственный
университет

```
4 + '7';           // '47'  
4 * '7';           // 28  
2 + true;          // 3  
false - 3;         // -3
```

Неявные преобразования типов
в JavaScript

4 → '4'
'4' + '7' → '47'

JavaScript имеет представление о типах
данных при несовпадении из-за всех сил
старается как-то их сконвертировать

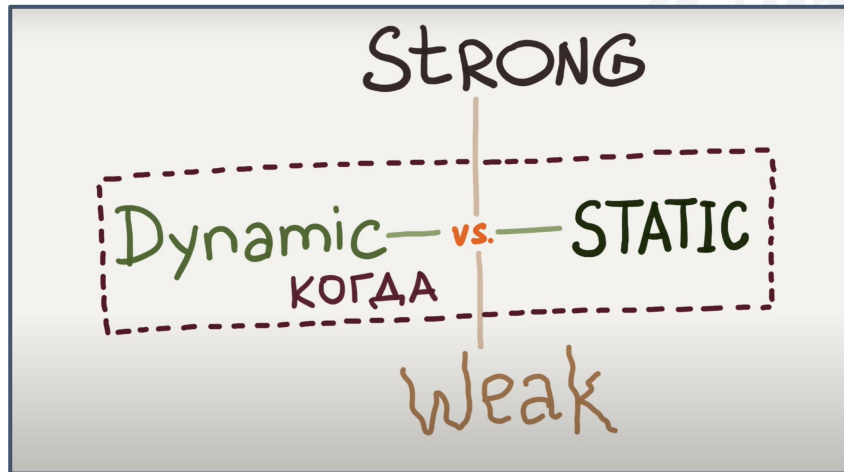


- **Надежность:** программист получит исключение или ошибку компиляции вместо неправильного поведения
- **Определенность:** у программиста появляется понимание, что сделанные явные преобразования могут замедлить код

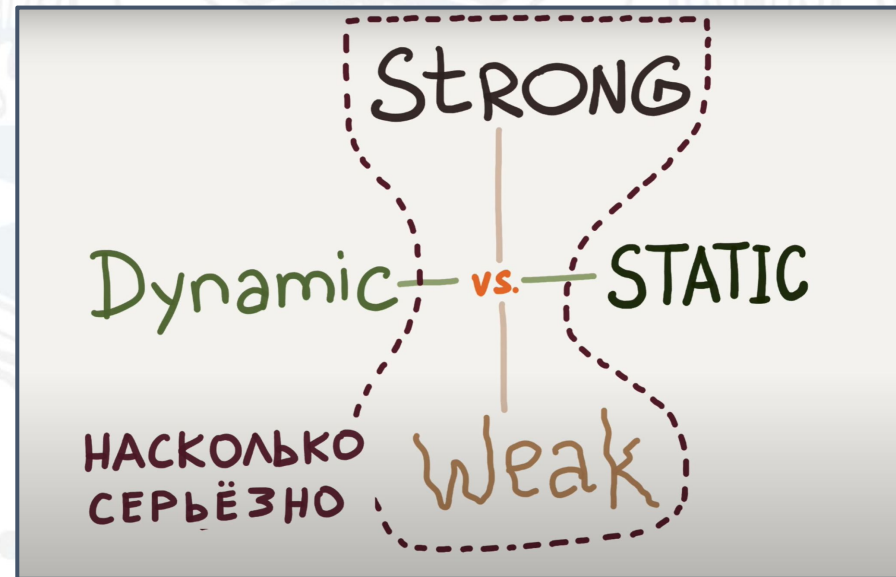


- **Удобство использования смешанных выражений:** программисту не надо приводить *int* к *float*, чтобы сложить два числа
- **Краткость записи:** программист пишет код короче, все преобразования за него делает компилятор

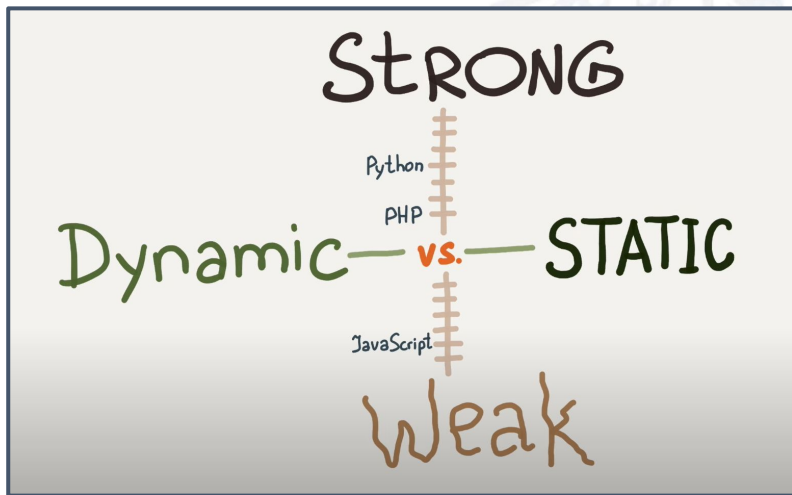




Динамическая и статическая типизации отвечают за то, **когда** будет выполнена проверка типов



Сильная и слабая типизации отвечают за то, **насколько серьёзно** будет выполнена проверка типов



- «Сила» типизации – это **мера**, в отличие от статического/динамического разделения
- Например, типизация в PHP сильнее типизации в JavaScript, но слабее, чем в Python
- При этом все три языка являются **динамически типизированными**



Видео на тему типизаций



```
class Map {  
    public Map(Radiant radiant, Dire dire) {  
        // конструктор карты  
    }  
}  
  
class Radiant {  
    public Radiant(Magnus m, Pudge p, Windranger w, Sven s, Luna l) {  
        // конструктор команды света  
    }  
}  
  
class Dire {  
    public Dire(Weaver w, Treant Protector tp, Puck p, Tusk t, Beastmaster b) {  
        // конструктор команды тьмы  
    }  
}
```

- Пишем код на каком-нибудь **статическом языке**
- Представим, что мы хотим написать аналог одной известной игры, где две команды выбирают себе по 5 игроков
- Пользуясь принципами ООП, создаем **класс Карты**, в конструктор которого передаем **команду Света** и **команду Тьмы**
- Команды включают в себя 5 случайных игровых персонажей



- Сразу возникает вопрос: если мы **заранее** не знаем, каких персонажей (из 123) выберут команды, что **передавать в конструктор команд Света и Тьмы?**
- Неужели придется **для каждой комбинации прописывать свой конструктор?** Это же ~216 миллионов комбинаций!

```
class Radiant {  
    public Radiant(Magnus m, Pudge p, Windranger w, Sven s, Luna l) {  
        // конструктор команды света  
    }  
    public Radiant(Magnus m, Pudge p, Windranger w, Sven s, Weaver w) {  
        // конструктор команды света  
    }  
    public Radiant(Magnus m, Pudge p, Windranger w, Sven s, Beastmaster b) {  
        // конструктор команды света  
    }  
    public Radiant(Magnus m, Pudge p, Windranger w, Sven s, Tusk t) {  
        // конструктор команды света  
    }  
}
```



```
class Radiant {  
    public Radiant(dynamic h1, dynamic h2, dynamic h3, dynamic h4, dynamic h5) {  
        // конструктор команды света  
    }  
}
```

- Может откажемся от безопасности и **не будем пользоваться статической типизацией?**
- Тогда в конструктор **может попасть все что угодно** и мы не сможем проинициализировать класс
- Более того, а что если *dynamic* **не поддерживается** в нашем языке?



```
abstract class Hero {  
    abstract firstSkill();  
    abstract thirdSkill();  
}  
  
class Pudge: Hero {  
    firstSkill() { /* код первого скилла у Pudge */ }  
    thirdSkill() { /* код третьего скилла у Pudge */ }  
}  
  
class Luna: Hero {  
    firstSkill() { /* код первого скилла у Luna */ }  
    thirdSkill() { /* код третьего скилла у Luna */ }  
}
```

```
class Radiant {  
    public Radiant(Hero h1, Hero h2, Hero h3, Hero h4, Hero h5) {  
        // конструктор команды света  
    }  
  
    private FirstAndThird() {  
        this.h2.firstSkill();  
        this.h2.thirdSkill();  
    }  
}
```

- Вспоминаем, что у нас есть **наследование** и **инкапсуляция**
- Создаем абстрактный класс Персонажа, наследуемся от него, переопределяем нужные методы в наследниках
- В конструктор команды передаются экземпляры наследников класса Hero, а в работе класса Radiant **используются методы абстрактного класса** (вернее, их различные реализации в дочерних классах)



- **Полиморфизм обеспечивает гибкость:** он позволил нам с легкостью переопределить поведение базового абстрактного класса
- Без полиморфизма **языки со статической типизацией** были бы «деревянными»
- Полиморфизм дает возможность использовать одни и те же методы для объектов разных классов
- Неважно, как эти объекты устроены, — в ООП можно сказать самолёту и квадрокоптеру: «лети», и они будут делать это как умеют: квадрокоптер закрутит лопастями, а самолет начнет разгон по взлетно-посадочной полосе





- Рассмотрели два вида **типизации языков**: статическую/динамическую и сильную/слабую. Есть еще явная и неявная типизации
- **Статическая/динамическая типизации** отвечают за то, **когда** типы будут проверены
- **Сильная/слабая типизации** отвечают за то, **насколько серьезно** стоит подойти к типизации
- **Полиморфизм** (с греческого «многоформенность») — взаимозаменяемость объектов с одинаковым интерфейсом.
- Он не только необходим для статически типизированных языков, но и предоставляет удобную конструкцию для гибкого «переопределения» функциональности





Парадигма ООП первая часть



SOLID-принципы

Направление «Искусственный интеллект и наука о данных», 23.Б16-мм, 23.Б18-мм

17.11.2023



1. **Single responsibility principle.** Принцип единственной обязанности. Каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс
2. **Open/closed principle.** Принцип открытости/закрытости: программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения
3. **Liskov substitution principle.** Принцип подстановки Барбары Лисков. Должна быть возможность вместо базового (родительского) типа (класса) подставить любой его подтип (класс-наследник), при этом работа программы не должна измениться
4. **Interface segregation principle.** Принцип разделения интерфейсов. Данный принцип означает, что не нужно заставлять клиента (класс) реализовывать интерфейс, который не имеет к нему отношения
5. **Dependency inversion principle.** Принцип инверсии зависимостей. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракции. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.