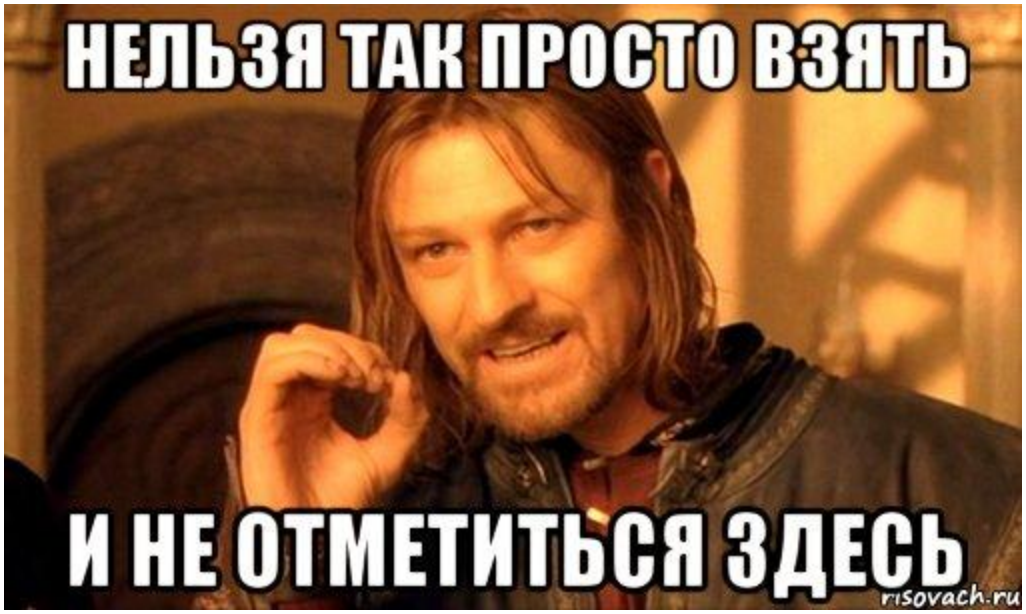
ТЕХНОТРЕК

Объектно-ориентированный дизайн

Архангельский Дмитрий
Галаганов Максим


НЕЛЬЗЯ ТАК ПРОСТО ВЗЯТЬ
И НЕ ОТМЕТИТЬСЯ ЗДЕСЬ
risovach.ru

Обратная связь

обратную связь можно оставить на портале к каждому занятию;

обратная связь позволяет оценить лекцию с другой стороны и что-то поменять;

нам хочется сделать лекции лучше;

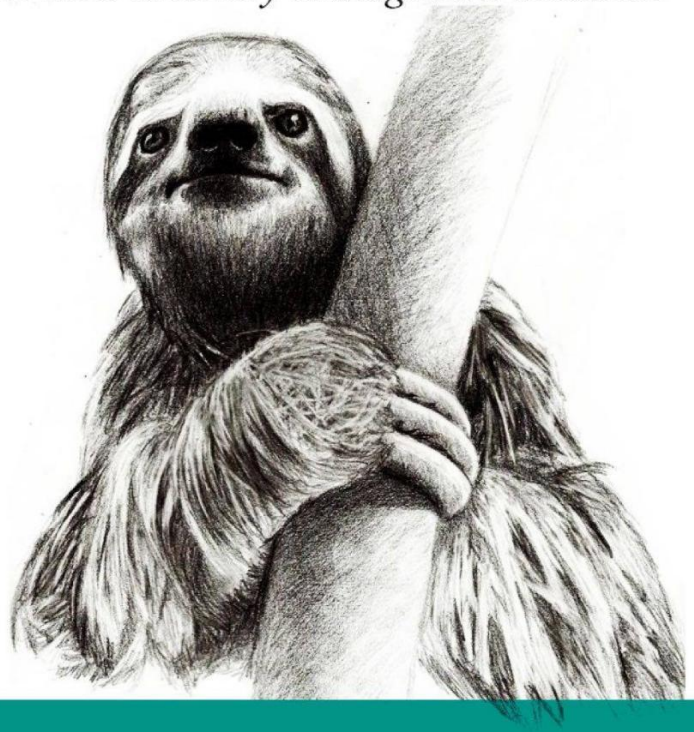
Вопросы задавайте сразу!

Напоминалка

Друзья, наш курс не про синтаксис, а о практических вещах. Нужно писать код и много учить самостоятельно. А для этого:

- не стесняйтесь почитать что-то дома;
- не стесняйтесь написать код;
- есть много книг и сайтов где можно посмотреть примеры;
- есть даже крутая официальная документация по java ;)

Cutting corners to meet arbitrary management deadlines



Essential

Copying and Pasting from Stack Overflow

O'REILLY®

The Practical Developer
@ThePracticalDev

Back to school

```
typedef struct _user {
   long id;
   char* name;
   short age;
   ...
} user;
```

```
typedef struct _admin {
   long id;
   char* name;
   short age;
   permission* permission;
   ...
} admin;
```

Back to school

```
typedef struct _user {
   long id;
   char* name;
   short age;
   ...
} user;
```

```
typedef struct _admin {
   user user;
   permission* permission;
   ...
} admin;
```

```
// лайкнуть
void (*like) (user* user, photo* photo);

// забанить
void (*ban) (admin* admin, user* user);
```

Back to school

```
typedef struct _user {
   long id;
   char* name;
   short age;
   …
  void (*like) (
      user* user,
      photo* photo);
} user;
```

```
typedef struct _admin {
   user user;
   permission* permission;
   ...
} admin;
```

```
admin* admin = …;
((user *) admin)->id = 42;
((user *) admin)->like(other, some_photo);
```

Back to school

```
struct user {
   long id;
   char* name;
   short age;
   …
   void (*like) (user* user, photo* photo);
}
```

Это свойства “реальной” сущности - пользователя. Они есть у каждого пользователя.

А это действия, характерные для пользователя.

Классы и объекты

- класс представляет объект “реального” мира
- класс описывает свойства объекта (поля) и действия над ним (методы)
- объект - это конкретный экземпляр класса; то есть набор свойств общий, а значения у каждого экземпляра свое.

Back to school

```
typedef struct _user {
   long id;
   char* name;
   short age;
   ...
} user;
```

```
typedef struct _admin {
   user user;
   permission* permission;
   ...
} admin;
```

admin IS user

Java class

```
class User {
    String name;
    int age;
    public String getName() { return name;}
}
// Конструктор по умолчанию - без аргументов
User user = new User();

// Доступ к полю объекта
user.name = “Ваня”;
user.age = 22;

// Использование метода объекта
print(user.getName());
```

Java class

```
class User {
    String name;
    int age;

    // вспомогательный метод - конструктор
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

User user = new User(“Ваня”, 22);
```

Модификатор static

Класс - это описание свойств и методов некоторого объекта
Объект - экземпляр (инстанс) класса.

Поля и методы объекта существуют только, когда объект создан.

static - означает, что поле или метод принадлежит классу, как таковому, а не конкретному объекту. Обращаться к такому полю (методу) можно через имя класса.

static

- почему main() - static?
- можно ли переопределить статический метод в дочернем классе?
- плюсы/минусы статической переменной

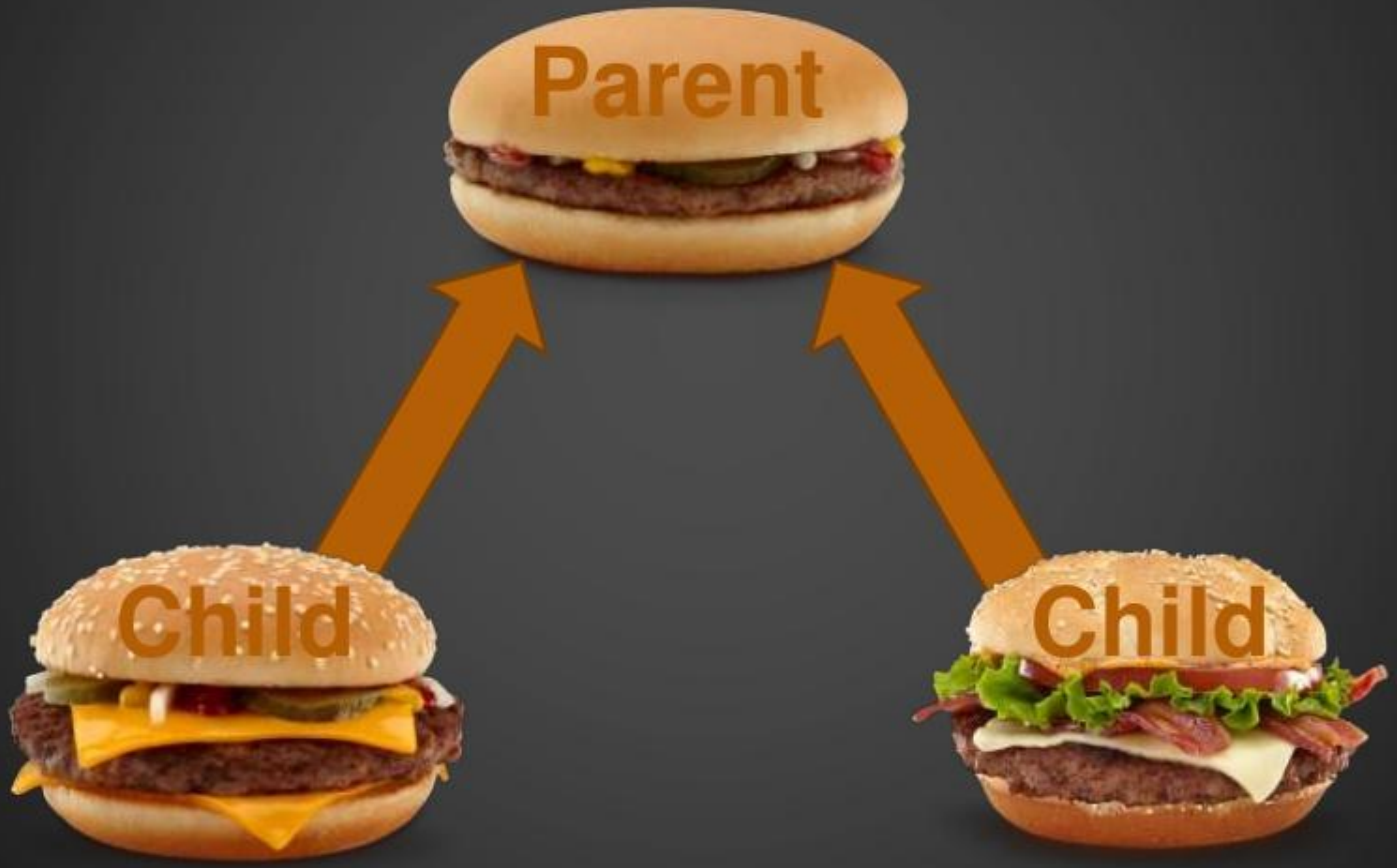
Наследование

Наследование

Дочерний класс является тем же, что и класс родитель (IS-A). Базовый класс определяет поведение дочернего.

4 Major Principles for OOP

Inheritance



Наследование

- Переиспользовал часть кода
- Выделил общее поведение
- Класс-потомок можно использовать вместо класса-родителя в коде
- Класс-потомок может переопределить поведение родителя
- Нет множественного наследования

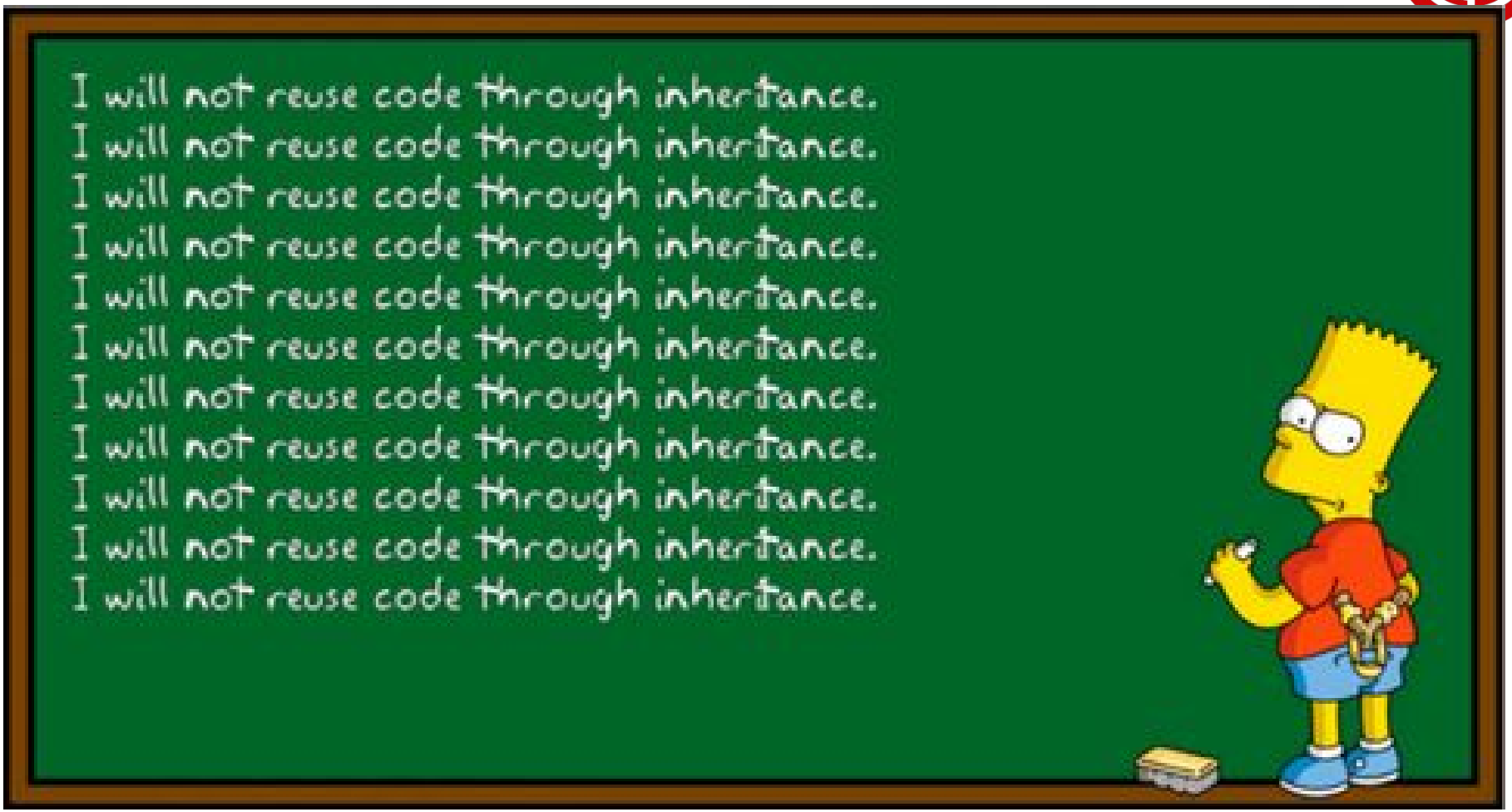
Наследование (extends)

```
class User {
   String sayHello() {
      return “Hello!”;
   }
}
```

```
class Wizard extends Unit {
   @Override
   String sayHello() {
      return “Abrakadabra!”;
   }
}
```

```
Unit unit = new Unit();
unit.sayHello(); // Hello!
Wizard wiz = new Wizard(); // Можно также Unit wiz =...
wiz.sayHello(); // Abrakadabra!
```

Недостатки наследования



```
public abstract class User extends SocketException { }
```

#9674

Композиция

Композиция

Отношение HAS-A. Класс обладает свойствами своих составных частей. Можем динамически менять поведение класса.

http://tproger.ru/translations/inheritance-and-composition-in-java/

Композиция

```
class Auto {
   Engine engine;   // компоненты, которые можно
   Gear gear;       // изменять, чтобы поменялось
                    // поведение

   public Auto(Engine engine, Gear gear) {
      this.engine = engine;
      this.gear = gear;
   }
}
```

Quiz!

- Фотография и фотоальбом
- Видеоролик и рекламный ролик
- Чат и мультичат
- Медиа-пост в ленте и текстовый пост

Абстрактный класс

Определяет “каркас” поведения.

Детали отданы дочерним классам на переопределение, а общее поведение вынесено в родительский абстрактный класс.

Создать экземпляр такого класса нельзя, так как его описание неполно.

Абстрактный класс

```
abstract class Message {
   long ownerId;
   long timestamp;

   //метка абстрактного метода
   abstract Type getType();

}

// compile time error!
Message msg = new Message();
```

```
class TextMessage extends
Message {


@Override
   Type getType() {
      return Type.TEXT;
   }

}

TextMessage msg = new
TextMessage();
```

Класс Object

В java всё - объект.

Все объекты неявно наследуются от **java.lang.Object**

Класс Object

- **`toString(): String`**
- **`hashCode(): int`**
- **`equals(): boolean`**
- **`getClass(): Class`**
- `wait()`
- `notify()`
- `clone()`

equals & hashCode

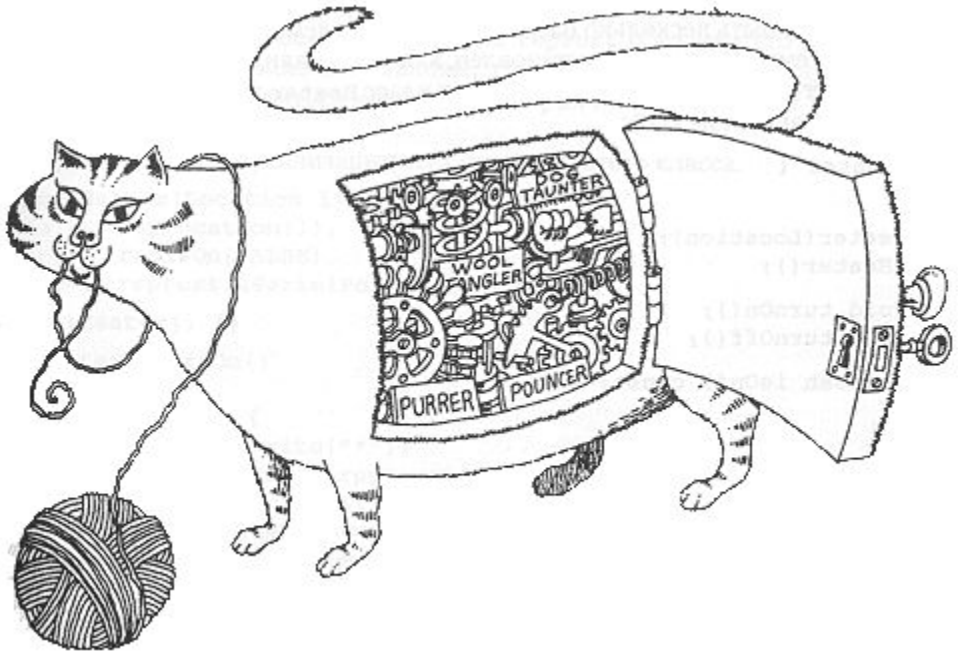
- Если переопределили equals(), то необходимо переопределить hashCode() и наоборот

- Контракт hashCode - одинаковые объекты имеют одинаковый hashCode(), разные могут иметь разный.

Quiz!

- А что если не переопределить?
- А как бы вы реализовали hashCode() для строк?

Инкапсуляция



Инкапсуляция

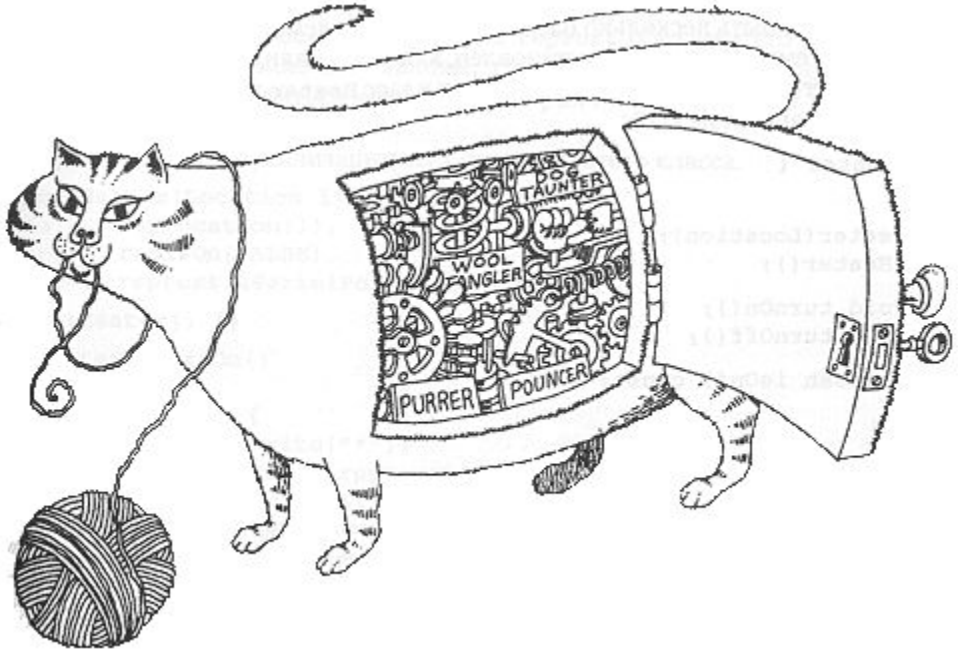
- контроль доступа - это поле можно только читать, или его вообще нельзя видеть.

- контроль целостности/валидности данных - это внутреннее поле класса, только владелец знает, как его корректно изменять. Например некий curPos внутри вашей структуры данных

- возможность изменения реализации - если есть метод get()/set(), то есть возможность изменить его логику

Инкапсуляция

- контроль доступа
- контроль целостности/валидности данных
- возможность изменения реализации
- **контракт для разработчика на get/set**



Инкапсуляция. Ограничение доступа

Access Modifiers	*Same Class*	*Same Package*	*Subclass*	*Other packages*
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

Полиморфизм

- Дочерний класс может быть использован везде, где используется родительский
- Если дочерний класс приведен к родительскому, то доступны только методы родительского класса (по типу ссылки)
- Вызывается реализация по реальному типу объекта (@Override)

Принцип подстановки

- Дочерний класс может быть использован везде, где используется родительский

```
class Admin extends User;

1. User admin = new Admin(); // ссылка имеет родительский тип

// можно передать дочерний тип, там где ожидается родительский
2. foo(User u) { return u.getName();}
3. Admin admin = new Admin();
4. foo(admin);
```

Доступны методы по типу ссылки

Во время компиляции проверяется, что такой метод есть у объекта заданного типа.

```
class User {
   getName() {return “U”;};
}

class Admin extends User {
   void ban(User user) {...};
}
```

```
User u = new Admin();
u.getName();
u.ban(); // Compile-time err

Admin a = new Admin();
a.getName();
a.ban(); // ok
```

Вызывается перегруженный метод

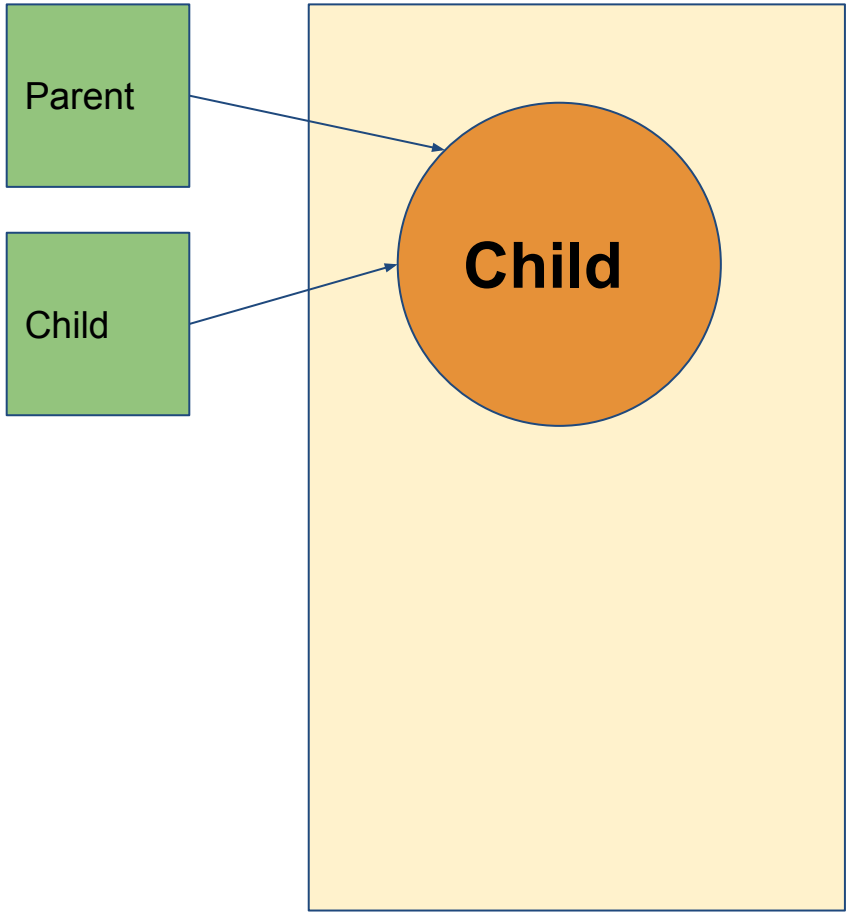
Вызывается реализация по реальному типу объекта (@Override)

```
class User {
   getName() {return “U”;};
}

class Admin extends User {
   void ban(User user) {...};
   getName() {return “A”;};
}
```

```
Admin a = new Admin();
a.getName();// ?

User u = new User();
u.getName();// ?

User u = new Admin();
u.getName();// ?
```

Полиморфизм

```
class Parent {
   test(){print(“P”);}
}
class Child ext. Parent {
   @Override
   test(){print(“C”);}
}

Child child = new Child();
Parent pChild = child;
child.test();
pChild.test();
```



Quiz!

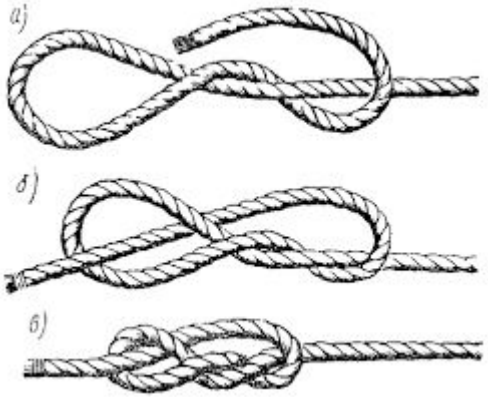
polimorf

https://github.com/tehnotrack/track17-autumn/blob/master/L3-oop/src/main/java/ru/track/demos/Polimorf.java

Связывание

Раннее связывание (compile time)

- Overloading - поиск подходящей сигнатуры в зависимости от списка параметров



Позднее связывание (runtime)

- Overriding - поиск подходящей реализации (по реальному типу объекта)

https://github.com/tehnotrack/track17-spring/blob/master/src/main/java/track/lections/lection3oop/LinkageTest.java

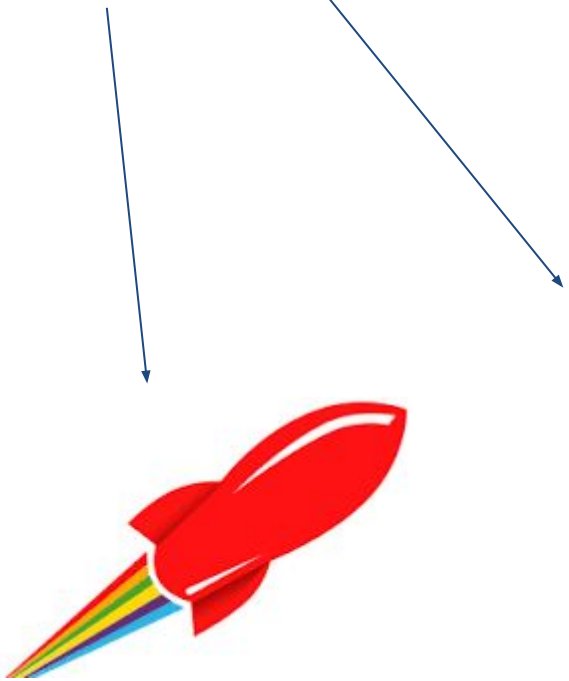
Чего не хватает?

- наследование
- инкапсуляция
- полиморфизм
- ...

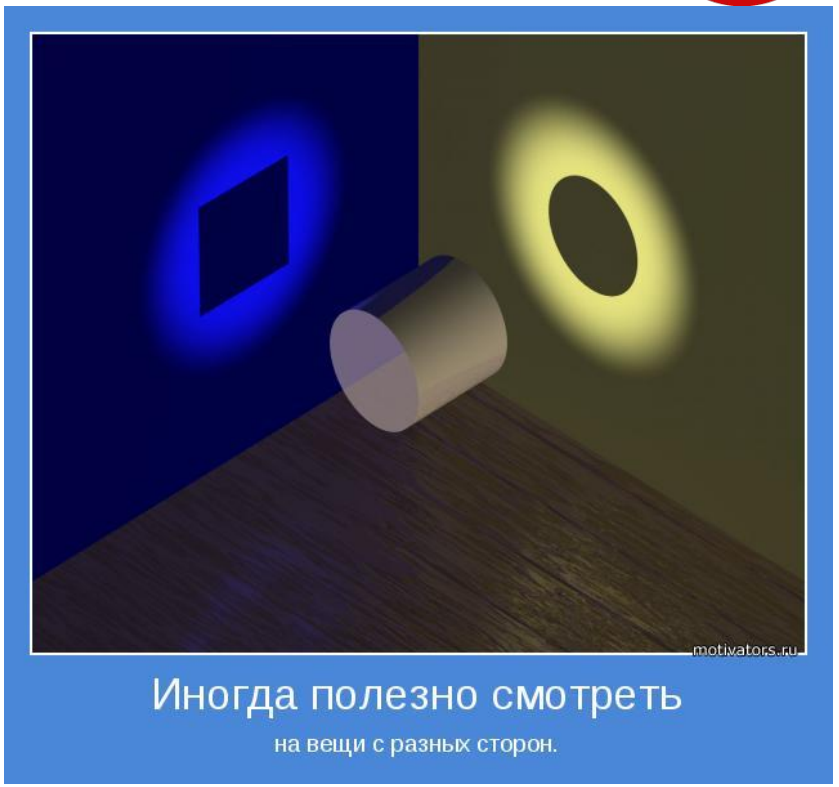
Интерфейс

- определяет, что можно сделать с классом;
- не определяет как это сделать;
- класс может удовлетворять нескольким интерфейсам (алиасы);
- абстракция от реализации;
- обобщение по свойству;

Шаблон Observer (Listener)



Интерфейс как тип




```
class User implements
      Comparable,
      Serializable {
   private long id;
   private String name;
}

User user = getUser();
Comparable c = (Comparable) user;
c.compare(other);

Serializable s = (Serializable) user;
s.write();
```

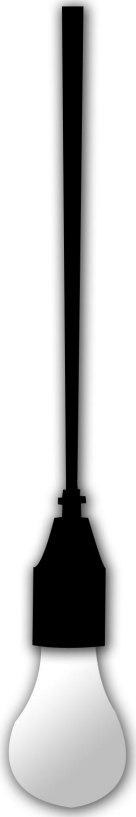
Принципы проектирования (SOLID)

- единственность ответственности (single responsibility)
- открытость для расширения (open closed principle)
- подстановка (Liskov substitution principle)
- разделение интерфейса (interface segregation)
- инверсия зависимостей (dependency inversion)

Single responsibility

- объект должен иметь одну обязанность
- она должна быть инкапсулирована внутри объекта

keep it **simple.**



Open-closed principle

Open - возможность изменить или расширить реализацию при изменении требований (например, через механизм наследования, переопределив поведение дочернего класса)

Closed - при этом мы не должны переписывать пол-проекта, если требования поменялись

https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D1%81%D1%82%D0%B8/%D0%B7%D0%B0%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D1%81%D1%82%D0%B8

http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/

Внедрение зависимостей

объекты высокого уровня не зависят от реализации объектов низкого уровня.

Dependency Injection

- через конструктор
- через метод set()

Внедрение зависимостей

```
class Server {

   private MessageStore messageStore;
   private CommandHandler handler;

   public Server (int concurrencyLevel,
          String pathTo,
          List<Command> commands, ...) {

      messageStore = new MessageStore(pathTo, concurrencyLevel);
      handler = new CommandHandler(commands);
      ...
      // создаем другие компоненты, используемые сервером
   }
```

Внедрение зависимостей

```
class Server {

    private MessageStore messageStore;
    private CommandHandler handler;

    public Server (MessageStore store, CommandHandler handler) {

        messageStore = store;
        this.handler = handler;
        ...
    }

    // or

    server.setMessageStore(store);
    server.setCommandHandler(handler);
```

Интерфейсы

- Интерфейсы позволяют строить гибкую архитектуру
- Реализацию всегда можно подменить
- Интерфейс - это контракт

Вопросы?

ТЕХНОТРЕК

Спасибо за внимание!

Time to hack.

https://github.com/tehnotrack/track17-autumn/wiki/3-object-oriented-design

equals

```
class User {
   private int id;
   private String name;
   public User(int id, String name) {
      this.id = id;
      this.name = name;
   }
}
```

```
User u1 = new User(1, “Bob”);
User u2 = new User(1, “Bob”);
User bob = u1;
```

```
assert(u1 == u2);
assert(u1.equals(u2));
assert(u1 == bob);
assert(u1.equals(bob));
```

equals

```
class User {
    public void equals(Object other) {
        if (other == this) return true;
        if (other == null) return false;
        if (!(other instanceof User)) return false;

        User otherUser = (User) other;
        return id == other.id
                && name != null && name.equals(other.name);
    }
}

User u1 = new User(1, “Bob”);
User u2 = new User(1, “Bob”);
User bob = u1;
```

```
assert(u1 == u2);
assert(u1.equals(u2));
assert(u1 == bob);
assert(u1.equals(bob));
```