

08.



*курс лекций по информатике и программированию
для студентов первого курса ИТИС КФУ (java-поток)
2023/2024*

М.М. Абрамский

кандидат технических наук, доцент кафедры программной инженерии

**СНАЧАЛА – ЗАДУМАЕМСЯ О ДАННЫХ,
КОТОРЫЕ ПРИНАДЛЕЖАТ КЛАССУ В
ЦЕЛОМ**

Статические поля и методы

- У каждого объекта – свой набор значений полей:
 - У каждого договора – свои поля
 - У каждого игрока – свои hp, клич
- Но иногда есть необходимость в атрибутах класса, общих на всех
 - Что если хочу сделать сквозную нумерацию договоров IndividualContract?

Static

```
public class IndividualContract {  
  
    private static int numberOfContracts = 0;  
}
```

модификатор для таких данных

static данные общие для всех объектов, они должны существовать, даже если ни одного объекта не создано (ни одного конструктора не вызвано). Поэтому инициализируем вот так.

Работа с этими данными обычна

```
public IndividualContract(String subject,  
                           Date dueTo, double cost) {  
    this.subject = subject;  
    this.dueTo = dueTo;  
    this.cost = cost;  
    numberOfContracts++;  
}
```

Стоп-стоп

Но если есть данные, привязанные к классам, а не к объектам, то наверное, есть и методы, привязанные только классам.

- Не имеющие смысла для отдельных объектов

Примеры

- `Math.cos`, `Math.sin`, ...
- `Integer.parseInt`, `Double.parseDouble`
- А какой еще метод должен (ОБЯЗАН) работать, даже тогда, когда еще ни одного объекта не создано?

Разгадка main!

```
public static void main(String[] args)
```

Точка запуска программы

- **public** – чтобы ее могли запустить извне
- **static** – потому что main не принадлежит конкретному объекту. Метод main должен запуститься, когда еще ни один объект не создан.

В статических методах

- Могут быть вызваны только другие статические методы/использованы статические атрибуты класса
 - Опять, потому что должны работать, когда ни одного объекта не создано.
 - Поэтому все методы рядом с `main` писались тоже как `public static`.

ДОСТУП К СОСТОЯНИЮ ОБЪЕКТА

Доступ к полям

Вернемся к договорам. Нам сказали, что у договоров «можно менять сроки и сумму». Казалось бы, бери и меняй (это же переменные)

```
IndividualContract ic1 =  
    new IndividualContract("Development",  
        new Date(2016, 3, 15), 100000);  
...  
ic1.cost = 200000;
```

НО ЭТО ЖЕ ПРОСТО ПЕРЕМЕННЫЕ, ЗНАЧИТ МОЖНО...

```
ic1.subject = "Фигня всякая";
```

Можно в смысле «программист, использующий экземпляры моего класса, может сделать так».

Модификаторы доступа

- Определяют возможность прямого доступа к членам класса и к самим классам
 - **public** – доступ всем отовсюду
 - **private** – прямой доступ только внутри класса, в котором находится данный атрибут/метод
 - ...
 - ...



Инкапсуляция

- Скрытие реализации объекта (*атрибутов и тел его методов*)
- «Мне не нужно знать устройство машины, чтобы ее водить»

ЭТО СТАРЫЙ ПЛОХОЙ ПРИМЕР

IndividualContract.java

```
public class IndividualContract {  
    private String subject;  
    private Date dueTo;  
    private double cost;  
    private Individual individual;  
    private Employee responsible;  
  
    public IndividualContract(String subject,  
                               Date dueTo, double cost) {  
        this.subject = subject;  
        this.dueTo = dueTo;  
        this.cost = cost;  
    }  
}
```

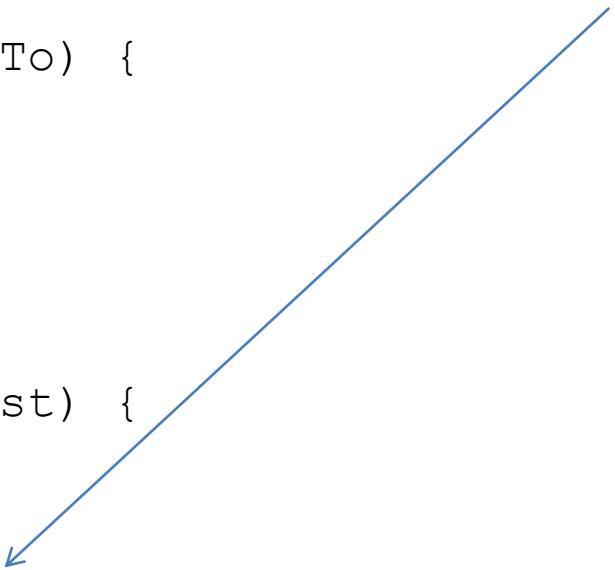
Set-, get- методы

- Методы, с помощью которых мы получаем доступ к атрибутам (get) и изменяем их (set).
- В средах разработки могут быть сгенерированы автоматически.

set, get в Contract

```
private String subject;  
private Date dueTo;  
private double cost;  
  
public Date getDueTo() {  
    return dueTo;  
}  
public void setDueTo(Date dueTo) {  
    this.dueTo = dueTo;  
}  
public double getCost() {  
    return cost;  
}  
public void setCost(double cost) {  
    this.cost = cost;  
}  
public String getSubject() {  
    return subject;  
}
```

На Subject только get



заказчик пришел, сказал,
так и быть, штрафовать не будет

Тук-тук-тук!

Заказчик:

«... а, увидел недавно в одной игре какой-то – там игроки не только ударяли, но и могли себя исцелять себя. Добавьте таких игроков, но обычных тоже оставьте..»

Окей!

Прошлись граблями по требованиям заказчика

- Есть обычный игрок
 - атрибуты `hp`, `name`, `battleCry`, методы `kick`, `shoutBattleCry`
- А есть продвинутый игрок
 - атрибуты `hp`, `name`, `battleCry`, методы `kick`, `battleCry`
 - новый атрибут `healPoints` – на сколько он может заживлять максимум. При каждом хиле уменьшается на значение.
 - новый метод `heal(p)` – *исцеление* – увеличение **hp** на **p** очков.

Player

```
public class Player {  
    private int hp;  
    private String name;  
    private String battleCry;  
    public Player(String name, String battleCry) {  
        hp = 100;  
        this.name = name;  
        this.battleCry = battleCry;  
    }  
    ...  
    public void shoutBattleCry() {  
        ...  
    }  
    public void kick(Player p) {  
        ...  
    }  
}
```

HealerPlayer

```
public class HealerPlayer {
    private int hp;
    private int healPoints;
    private String name;
    private String battleCry;
    public HealerPlayer(String name, String battleCry) {
        hp = 100;
        healPoints = 20;
        this.name = name;
        this.battleCry = battleCry;
    }
    ...
    public void shoutBattleCry() { ... }
    public void kick(Player p) { ... }
    public void heal(int p) {
        if (p <= healPoints) {
            hp += p;
            healPoints -= p;
        }
    }
}
```

Вместе в одном проекте

```
public class Player {  
    private int hp;  
  
    private String name;  
    private String battleCry;  
    public Player(String name,  
                   String battleCry) {  
        hp = 100;  
  
        this.name = name;  
        this.battleCry = battleCry;  
    }  
    ...  
    public void shoutBattleCry() { ... }  
    public void kick(Player p) { ... }  
}
```

```
public class HealerPlayer {  
    private int hp;  
    private int healPoints;  
    private String name;  
    private String battleCry;  
    public HealerPlayer(String name,  
                         String battleCry) {  
        hp = 100;  
        healPoints = 20;  
        this.name = name;  
        this.battleCry = battleCry;  
    }  
    ...  
    public void shoutBattleCry() { ... }  
    public void kick(Player p) { ... }  
    public void heal(int p) {  
        if (p <= healPoints) {  
            hp += p;  
            healPoints -= p;  
        }  
    }  
}
```

Что-то смущает, правда?

HealerPlayer и Player

- *Сильное* дублирование кода
- Любое изменение Player влечет изменение HealerPlayer
- HealerPlayer может вести себя как Player, но не наоборот.
 - Т.е. в HealerPlayer имеет весь функционал Player, в обратную сторону это не верно.

Принцип (кит) ООП #2

НАСЛЕДОВАНИЕ



- Классы могут использовать **готовую реализацию** других классов, добавляя лишь то, чего не хватает в исходном (базовом, супер, над-, родительском классе)
 - Концепция «повторного использования компонентов»
- По-английски – **Inheritance**
 - Хотя есть понятия «родительских» и «дочерних» классов, понимать наследование нужно скорее как «расширение» или «уточнение»

Пример наследования #1

Родительский класс – **Человек**

Дочерний класс – **Студент**

- *Студент* является *Человеком* (в реальности на планете Земля)
 - может все то же, что может человек
 - имеет все атрибуты человека
- *Человек* не обязательно является *Студентом*
 - у студента есть атрибуты (зачетка, студенческий) и методы (сдать экзамен, посетить лекцию), которых нет у произвольного человека.

Плохой пример наследования

Родительский класс – **Вид спорта**

Дочерний класс – **Футбол**

Это не «родитель-наследник», а, скорее «класс-объект»

Игроки

- Очевидно, в нашем примере HealerPlayer – потомок Player
 - Может все то же, что и Player, но добавляет в Player новый атрибут и новый метод, а также *уточняет* конструктор.
 - Поехали кодить. Player пока не меняется. А вот HealerPlayer

Расширение

```
public class HealerPlayer extends Player {
```

- `extends` – «расширяет»
- Что пишем внутри? Того, что нет в `Player`.

Атрибуты

- Пишем только те, которые новые для HealerPlayer, остальные есть в Player
 - С ними будет небольшое веселье, но позже.

```
public class HealerPlayer extends Player {  
    private int healPoints;  
}
```

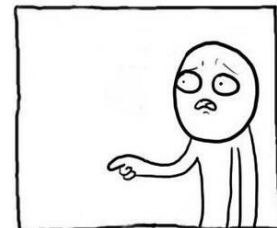
Добавим метод heal

```
public class HealerPlayer extends Player {  
    private int healPoints;  
  
    public void heal(int p) {  
        if (p <= healPoints) {  
            hp += p;  
            healPoints -= p;  
        }  
    }  
}
```

Добавим метод heal

```
public class HealerPlayer extends Player {  
    private int healPoints;  
  
    public void heal(int p) {  
        if (p <= healPoints) {  
            hp += p;  
            healPoints -= p;  
        }  
    }  
}
```

- Не скомпилируется с ошибкой: **hp has private access in Player.**
 - Что? Почему? Я же в этом же классе работаю?



Правда о **private** и наследовании

private разрешает прямой доступ только в базовом классе.

На наследниках это не работает!

«И что делать?»

2 способа

Способ #1: использовать public set- и get- методы для hp, объявив их в Player

- public – он и везде public, смогу вместо ***hp -= p*** вызвать ***setHP(getHP() - p)***

Способ #2: изменить модификатор у hp

- Чтобы потомкам можно было напрямую обращаться
- Но только потомкам! Извне – нет!

Модификатор доступа `protected`

Прямое обращение к членам класса из базового класса и всех его потомков.

Свободнее `private`, жестче чем `public`.

```
public class Player {  
    protected int hp;  
}
```

- Теперь можно у потомка `HealerPlayer` вызывать `hp -= p`

Продолжаем. Конструктор HealerPlayer

```
public HealerPlayer(String name, String battleCry) {  
    hp = 100;  
    healPoints = 20;  
    this.name = name;  
    this.battleCry = battleCry;  
}
```

- Решили вопрос с прямым доступом, но:
 - Конструктор почти полностью дублирует аналогичный конструктор в Player
 - Если честно, код вообще не скомпилируется.
 - даст ошибку «no default constructor available in Player»
» *Кто такой default конструктор*

Правда об объектах классов-наследников

Представим на секундочку, что все получилось, и мы создаем объект HealerPlayer:

- `HealerPlayer sp = new HealerPlayer(...);`



При создании объекта дочернего класса сначала неявно создается объект родительского класса, а потом уже выполняется все, что связано со HealerPlayer.

Но раз создается (пусть неявно) объект суперкласса, то значит, вызывается его конструктор.

Так вот

```
public HealerPlayer(String name, String battleCry) {  
    hp = 100;  
    healPoints = 20;  
    this.name = name;  
    this.battleCry = battleCry;  
}
```

- В дочернем конструкторе всегда вызывается родительский, первым же оператором
 - Даже если явно это не указана, происходит попытка вызвать конструктор по умолчанию – а его у Player нет, поэтому ошибка.

Решение

super – обращение к конструктору родительского класса

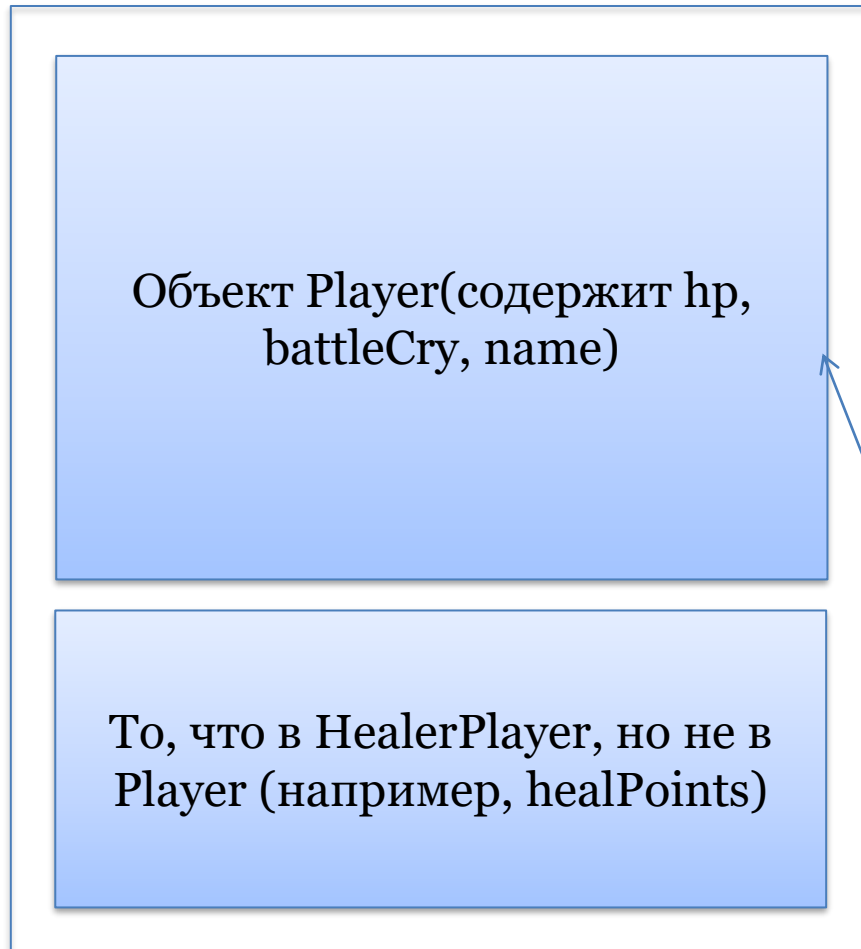
```
public HealerPlayer(String name, String battleCry)
{
    super(name, battleCry);
    healPoints = 20;
}
```

super обязан быть самым первым
в дочернем конструкторе

Делаю все, что нужно,
на уровне Player,
Затем делаю то, что нужно сделать
именно в HealerPlayer

Правда об объектах классов-наследников в памяти

Объект HealerPlayer



← Объект подкласса всегда может «прикинуться» объектом родительского класса – так устроены подобные объекты (в объекте `HealerPlayer` есть не просто атрибуты `Player`, но спрятанная сущность `Player`)

← `super` – как и `this` – ссылка, вот сюда.
`super` – и есть скрытая сущность родительского класса у дочернего объекта

На самом деле

- Player – тоже наследник.
- И любой класс, объявляемый в java – наследник класса Object.
- Object – корень иерархии классов. В нем даже есть свои собственные методы
 - Попробуйте вывести Player с помощью `System.out.println`

Выведется

Player@15db9742

Название класса + значение метода hashCode, который объявлен и реализован в Object и унаследовался в Player (возвращает число, уникальное для каждого объекта).

Но как получилась эта строка?

- В **Object** есть метод **toString**, который возвращает строковое представление объекта, который тоже есть у любого класса, а значит и у Player.
- Соответственно, println неявно вызывает метод **toString**, но реализация этого метода абсолютно нас не устраивает:
 - Player@15db9742, как информативно!

Переопределение

- Изменение потомком реализации родительского метода.
- *Если я вывожу на экран игрока, что я хочу увидеть? Да пусть хотя бы имя!*
- В Player добавим метод:

```
public String toString() {  
    return name;  
}
```

и все будет хорошо

Разница между перегрузкой и переопределением

- **Перегрузка** – совпадают названия методов, а наборы параметров разные
 - Существует и без наследования – мы перегружали конструкторы в Player
- **Переопределение** – совпадают заголовки (сигнатуры) методов
 - Существует только в контексте наследования – в одном классе не может быть двух методов с одинаковыми заголовками

Поиск реализации метода при вызове у объекта

1. Сначала ищется реализация метода в классе
2. Если метод с нужной сигнатурой не найден, проводится поиск у родителя.
 - *Старательно умалчиваю тот факт, что в Java у класса может быть только один родитель.*
3. Если и там не найден, то у родителя родителя, и т.д. до Object
4. Если метод не найден у Object, то выведется ошибка «Symbol not Found»
 - Где Symbol – это не символ, а class-member.

Переопределим что-нибудь еще

- Пусть HealerPlayer при выкрике боевого клича еще и хвастается, что он умеет хилить!
- Значит метод shoutBattleCry должен отличаться от Player
 - Но при этом возможно вызывает его (клич-то тоже надо выкликивать!)

HealerPlayer

```
public void shoutBattleCry() {  
    super.shoutBattleCry();  
    System.out.println("And I can heal!");  
}
```

- Обращаемся к родителю (super), к его реализации shoutBattleCry.
- Затем добавляем то, что характерно только для HealerPlayer.

Техлид:

- Наш HealerPlayer настолько идеален, что лучше его мы не напишем. И никто не сможет написать!

Запретите наследовать HealerPlayer!

- Не дадим переопределить поведение метода heal в наследниках!

Модификатор `final`

- У **атрибута** – как у переменной – константа.

- У **класса**:

```
public final class HealerPlayer
```

– теперь **HealerPlayer** запрещено наследовать.

– Очень многие классы в Java `final` – напр., `String`, `Scanner`

- У **метода**:

```
public final void f () { ... }
```

– теперь `f()` нельзя переопределять в потомках

Посмотрим на код еще раз

```
public class Player {
    protected int hp;

    protected String name;
    protected String battleCry;

    ...
    public void shoutBattleCry() { ... }
    public void kick(Player p) { ... }

}

public class HealerPlayer extends Player {
    private int healPoints;

    ...
    public void heal(int p) {
        ...
    }

}
```

Временные соглашения о названиях

Есть объект (экземпляр класса)

- Все его public-методы (их заголовки) назовем **интерфейсом** объекта (класса)
- То, как именно эти метод работают, назовем конкретной **реализацией/поведением** объекта (класса)

Отцы и дети

- Интерфейс любого потомка включает в себя интерфейс родителя:
 - Т.к. любой объект подкласса может быть объектом суперкласса.
- Но при одинаковых методах интерфейса в них может быть разная реализация.

И такое возможно

```
Player p = new HealerPlayer();  
p.shoutBattleCry();
```

- Интерфейс `p` определяется `Player` (левая часть, ссылка, интерфейс)
 - `p` не может вызвать `heal` (т.к. у `Player` его нет)
- Но `p.shoutBattleCry()` выведется в реализации `HealerPlayer` (правая часть, реализация)

Смысл

- В предыдущем примере нет никакой необходимости создавать дочерние классы с интерфейсом родителя
- Но иногда в этом есть необходимость!

Телефон и Смартфон

- Класс **Phone**, метод **call**
- Класс **SmartPhone** наследует **Phone**, переопределенный метод **call**, метод **takePhoto()**

Восходящее преобразование

- Сужение интерфейса потомка до интерфейса родителя.
- `Phone p = new SmartPhone () ;`

Телефон и Смартфон, восходящее преобразование

// Код написан в 2005 году

```
class Human {  
    public void callWith(Phone p) {  
        p.call();  
    }  
}
```

// Код написан в 2015 году

```
SmartPhone p = new SmartPhone();  
Human h = new Human();  
h.callWith(p);
```

- `p.call()` – работает, т.к. есть у `Phone`, но с реализацией `SmartPhone`
- `p.takePhoto()` – не работает

Много детей

У одного родителя может быть **несколько наследников**.

- У нас есть договоры с физ.лицами и есть с юр.лицами.
- Очевидно, что они оба могут быть наследниками общего класса Contract.

```
public class CompanyContract {  
  
    private String subject;  
    private Date dueTo;  
    private double cost;  
    private Company company;  
}  
  
public class IndividualContract {  
  
    private String subject;  
    private Date dueTo;  
    private double cost;  
    private Individual individual;  
    private Employee responsible;  
}
```

Наводим иерархию

```
public class Contract {  
    protected String number;  
    protected String subject;  
    protected Date dueTo;  
    protected double cost;  
}
```

```
public class IndividualContract extends Contract {  
    private Individual individual;  
    private Employee responsible;  
}
```

```
public class CompanyContract extends Contract {  
    private Company company;  
}
```

new Требование()

Заказчик:

«Слушай, нужно сделать хранилище договоров, общее и для физ, и для юр.лиц.


И еще нужно, чтобы по договору можно было видеть вывод всей информации о нем:

- предмет,
- сумма,
- срок другая нужная для инфа физ. или юр.лица.»

Storage

жестко привязать нельзя

```
public class Storage {  
    private static final int CAPACITY = 1000;  
  
    private IndividualContract [] journal =  
new IndividualContract[CAPACITY];  
  
    private CompanyContract [] journal =  
new CompanyContract[CAPACITY];  
  
    private int storageSize = 0;  
    // ...  
}
```



Вспоминаем про объекты в памяти

- Объект подкласса хранит экземпляр родительского класса (super). И может подыграть в его качестве.
 - Т.е. – IndividualContract – это Contract
 - И CompanyContract – это Contract

Используем восходящее преобразование

```
public class Storage {  
    private static final int CAPACITY = 1000;  
    private Contract [] journal =  
        new Contract[CAPACITY];  
    private int storageSize = 0;  
    ///...  
}
```

Кстати!

```
private Contract [] contracts  
    = new Contract [CAPACITY] ;
```

Напомню: ни одного объекта Contract кстати не создано!
Создан массив ссылок на экземпляры Contract, но ни одного объекта.

А что тогда с объектами?

Добавляем новый договор в хранилище

```
public class Storage {  
    //...  
    public void add(Contract contract) {  
        contracts[storageSize] = contract;  
        storageSize++;  
    }  
}
```


И в том месте, где реализовано использование Storage


```
Storage storage = new Storage();  
//...  
IndividualContract ic1 = new IndividualContract(  
    "Development", new Date(2016, 3, 15), 100000);  
  
CompanyContract cc1 = new CompanyContract(  
    "Awesome Development", new Date(2016, 3, 15), 500000);  
  
//В ic1 добавляют физ.лицо, ответственного, в cc1 юр.лицо  
//а потом ...  
storage.add(ic1);  
storage.add(cc1);
```

Что произошло

- Контракты **IndividualContract** и **CompanyContract** были переданы в метод `add` как **Contract**.
 - Что это? Чем является?

Да, восходящее преобразование

```
public class Storage {  
    //...  
    public void add(Contract contract) {  
        contracts[storageSize] = contract;  
        storageSize++;  
    }  
}
```



IndividualContract и CompanyContract
ограничены по родительскому интерфейсу в
Contract.

*Ну и что? Нам не нужна их специфика.
Мы используем их как контракты*

Что там с информацией о договоре

```
public class IndividualContract extends Contract {
    //...
    public void printInfo() {
        System.out.println(number);
        System.out.println(subject + ". Due to " +
                               dueTo + ". Money: " + cost);
        System.out.println(individual);
        System.out.println(responsible);
    }
}

public class CompanyContract extends Contract {
    //...
    public void printInfo() {
        System.out.println(number);
        System.out.println(subject + ". Due to " +
                               dueTo + ". Money: " + cost);
        System.out.println(company);
    }
}
```

Приводим в порядок

```
public class Contract {
    public void printInfo() {
        System.out.println(number);
        System.out.println(subject + ". Due to " +
                               dueTo + ". Money: " + cost);
    }
}

public class CompanyContract extends Contract {
    public void printInfo() {
        super.printInfo();
        System.out.println(company);
    }
}

public class IndividualContract extends Contract {
    public void printInfo() {
        super.printInfo();
        System.out.println(individual);
        System.out.println(responsible);
    }
}
```

Вывести всю инфу о всех договорах из storage

```
public class Storage {  
    //...  
  
    public void printAllInfo () {  
        for (Contract contract : contracts) {  
            contract.printInfo();  
        }  
    }  
}
```



Цикл, который мы в Java очень любим

Вывести всю инфу о всех договорах из storage

```
public class Storage {  
    //...  
  
    public void printAllInfo () {  
        for (Contract contract : contracts) {  
            contract.printInfo();  
        }  
    }  
}
```



У нас три printInfo, что вызовется??

Связывание (binding)

- Присоединение вызова метода к телу метода.
- `contract.printInfo()` – вызов
- методы `printInfo()` есть у нескольких классах.

в Java - позднее связывание

- Реализация вызываемого метода определяется в момент выполнения!
 - Компилятор не знает заранее, какая реализация `printInfo` будет использована для `contract.printInfo()`!
 - » Опять байка про то, что родитель написан в 2015 году в США, а наследник – в 2023 году в Китае
 - Но JVM разрулит все, опираясь на конкретные созданные объекты.
 - » !
- У договора тип – `Contract`, но в момент вызова:
 - у тех, кто по факту `IndividualContract`, вызовется их реализация;
 - у тех, кто по факту `CompanyContract`, вызовется их реализация;

Третий принцип (кит) Полиморфизм

Возможность реализовывать уникальное поведение у нескольких подклассов при едином интерфейсе суперкласса.



Третий принцип (кит) ООП

ПОЛИМОРФИЗМ

Разное понимание

- ad hoc полиморфизм – один интерфейс, множество реализаций
 - сюда иногда добавляют перегрузку
- Полиморфизм наследования
 - то, что было у нас
- Параметрический полиморфизм
 - обобщение, когда тип параметра – тоже параметр

Вот такой цикл – признак полиморфизма

```
for (Contract contract : contracts) {  
    contract.printInfo();  
}
```

