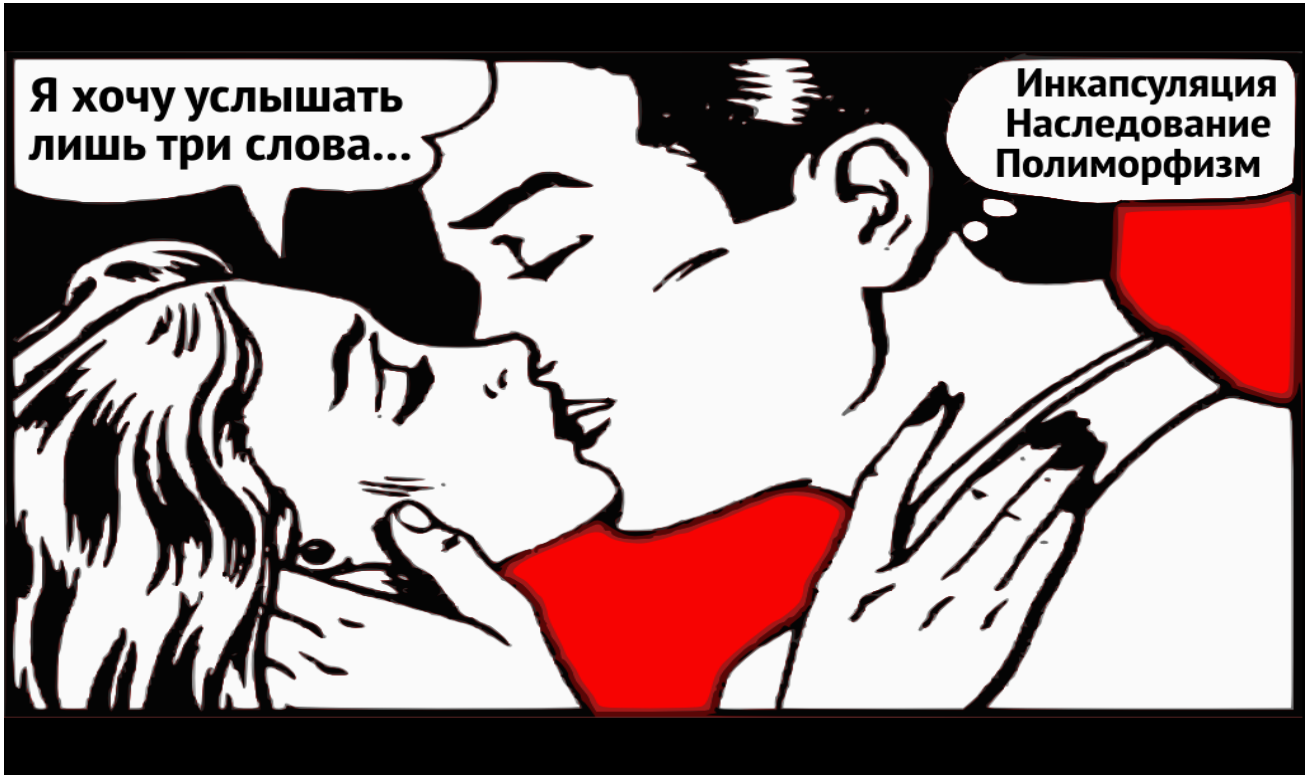
06. ООП2



Лекции по информатике для студентов первого курса Высшей школы ИТИС
2019 год

МИХАИЛ АБРАМСКИЙ
старший преподаватель
Высшая школа ИТИС КФУ

ЗАКАЗЧИК ВЕРНУЛСЯ ИЗ ОТПУСКА… ТАК, А ЧТО ОН ТАМ ХОТЕЛ??

Ах, да. Описание продукта #1

Требуется разработать систему управления договорами компании. Договор может быть с физическим лицом (человеком), может быть с юридическим лицом (другой компанией). У каждого договора есть предмет, сумма и сроки. Сроки и сумма могут быть изменены. У договора должен быть статус, а также должна быть возможность узнать, кто из сотрудников компании является ответственным за договор с ФИЗ.лицом. У физического лица должны быть известны ФИО, паспортные данные, адрес прописки, у юридического – наименование, адрес, банковские реквизиты, директор. Все договора хранятся в некоем хранилище, должна быть возможность искать в нем по физ.лицу или юр.лицу.

Описание продукта #2

Требуется разработать текстовую игру, где есть два игрока, которые наносят друг другу удары по очереди. Игроки вводят силу удара от 1 до 9, с увеличением силы возрастает вероятность промахнуться. При успешном ударе у противника уменьшаются очки здоровья (health points, hp). Когда hp одного из игроков становится <= 0, этот игрок проиграл.

В предыдущих сериях

Классы, объекты, атрибуты, методы, конструкторы, this, модификаторы, private, public, абстракция, инкапсуляция, static.

Мы даже разрабатывали код #1 но теперь поправим

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

```
public class Employee {
    private String fio;
    private Department department;
    private Employee chief;
}
```

```
public class Individual {
    private String fio;
    private PassportInfo passportInfo;
    private Address address;
}
```

- 1 public class – 1 одноименный java-файл, иначе не скомпилируется
- модификаторы доступа обязательны!

#2

```
public class Player {
    private int hp;
    private String name;
    private String battleCry;
    public Player(String name) {
        this(name, "Leroooooy Jenkinsss");
    }
    public Player(String name, String battleCry) {
        hp = 100;
        this.name = name;
        this.battleCry = battleCry;
    }
    public void shoutBattleCry() {
        System.out.println(name + ": " + battleCry);
    }
}
```

Заказчик пришел, опять поменял требования

- Это, ребят, мне нужно хранить инфу о том, когда какой договор с физ.лицом был заведен. Ну как журнал такой.

… через 5 минут …

- Да, и еще мне надо, чтобы результаты игр между игроками сохранялись. Имена игроков, дата, и кто победил. Ну в принципе тоже журнал.

Ваши действия как менеджера проекта

- *Разработчик А, журнал для договоров на тебе!*
- *Разработчик Б, журнал для записи результатов игр на тебе!*
- *Вам обоим 3 дня на разработку, потом ваш код присоединяем к проекту.*

Логика разработчика А

- **Журнал договоров** – это **набор записей**, в каждой хранятся ссылка на договор с физ.лицом и дата, когда он был заключен.
  - **Запись в журнале** – сущность с двумя атрибутами (договор, дата)
  - **Журнал** – набор (массив, список) записей
    - Пока мы знаем только массив, но при этом мы инкапсулируем его в отдельный класс.

» ~~*java.util.Collection 2 sem*~~

Журнал договоров от А

```
public class JournalEntry {
    private IndividualContract individualContract;
    private Date date;
    //...
}
```

```
public class Journal {
    private static final int JOURNAL_CAPACITY = 1000;
    private JournalEntry [] journal =
            new JournalEntry[JOURNAL_CAPACITY];
    private int journalSize = 0;
    //...
}
```

! Кстати, как быть, если превысим JOURNAL_CAPACITY?

Логика разработчика Б

- **Журнал результатов игр** – это **набор записей**, в каждой хранятся имена обоих игроков, дата игры, и номер победителя (1 или 2)

  – **Запись в журнале** – сущность с четырьмя атрибутами (имена игроков, дата игры и результат)

  – **Журнал** – набор (массив, список) записей

Журнал договоров от Б

```
public class JournalEntry {
    private String player1, player2;
    private Date date;
    private byte result;
    //...
}

public class Journal {
    private static final int journalCapacity = 3000;
    private JournalEntry[] journal =
            new JournalEntry[journalCapacity];
    private int journalSize = 0;
    //...
}
```

Вы вернулись

- «А, Б, давайте код!»
- Вам дают код, и тут…
  - **У обоих разработчиков разработаны по два разных класса с одинаковыми названиями!**
    - » При этом классы под именем Journal почти совпадают
    - » Но два других – одинаковые по названию JournalEntry – разные по содержимому.

В каждой шутке доля…

- Разумеется, у нас 2 разных проекта.
  - И конечно на самом деле конфликта имен можно избежать.
- Разумеется, если даже они были в одном проекте, можно было решить проблему, переименовав классы
  - GameJournalEntry, GameJournal
  - ContractsJournalEntry, ContractsJournal

Но.

- Масштабный Java-проект – несколько тысяч классов, которые писали разработчики различных компаний всего мира.

- Разработчиков много, а названий сущностей мало:
  - List, Connection, Entry, Scanner, TextField, Printer, Reader, Writer …

    » *сущность называют так, как она себя ведет. Но некоторые разные сущности ведут себя похоже.*

List

- **List** в случае работы с коллекциями – интерфейс, означающий список элементов, в котором у каждого элемента есть номер.

- **List** в случае работы с графическими приложениями – класс для виджета «выпадающий список».

И они оба входят в Java SE! А если подключили бы сторонние библиотеки, могло бы набраться до 6ти различных List-ов.

КАК НЕ ПЕРЕПУТАТЬ ДВЕ ВНЕШНЕ ОДИНАКОВЫЕ ВЕЩИ?

Концепция пакета

- Пакет (package) – набор классов и вложенных пакетов
  - ну почти как папка
  - C++, C# - namespaces.
- У пакета есть уникальное имя
- Полное имя класса:
  – имяпакета.названиеКласса

В случае с List

- List,который коллекция – лежит в пакете **java.util**
- List, который виджет – лежит в **java.awt**
- Полные имена классов:
  - java.util.List
  - java.awt.List
    - не перепутать.

Полные имена классов

- Абсолютно спокойно используются в тексте программ:

```
java.util.Scanner scanner = new java.util.Scanner(System.in);
```

- просто лень постоянно писать полное имя, поэтому были введены **import**

import

- Пишем до объявление класса import:
  - Либо полное имя класса: import java.util.Scanner;
  - Либо разрешаем импорт всего содержимого пакета: java.util.*;
- Теперь Scanner можно использовать без java.util.
- Если есть совпадения в коротких именах классов, import не делаем и используем полное имя:
  - Как в случае с java.util.List и java.awt.List

Об import #1

import – не рекурсивная операция

- В пакете java.util есть пакет java.util.regex, а в нем класс Pattern
- Если написано java.util.*, то это **не** означает, что java.util.regex.* был вызван.
  - Т.е. Pattern нельзя использовать по короткому имени. И даже regex.Pattern нельзя.

Об import #2 – import static

- Соскучились по вызовам функций (не методов, а функций)?
- Можно импортировать у класса его статические методы, чтобы вызывать их без его имени:
  - `import static java.lang.Math.cos;`
  - …
  - `System.out.println(cos(0));`

*Можно сделать import static вместе со *, чтобы импортировать все статические методы класса*

Ваши мысли сейчас

- Ну мы поняли, как это выглядит и используется!

- Но не понятно, что «физически» есть пакет, как его *создавать*, и как туда ~~класть, ложить,~~ помещать классы!»

ОКЕЙ!

Поместить класс в пакет в коде

- Написать САМОЙ ПЕРВОЙ строчкой:

```
package имяпакета;
```

- Пример для Player.java (где объявлен Player):

```
package kickgame;
```

или

```
package work.projects.kickgame;
```

Если в имени пакета через точку написаны несколько слов, то для каждого слова создается пакет, а следующий в него вложен:

- в примере пакет **work** содержит пакет **projects**, а пакет **projects** содержит пакет **kickgame**, а в пакете **kickgame** лежит класс **Player**.
- тогда полное имя класса **work.projects.kickgame.Player**

Визуально для Player.java

```
package work.projects.kickgame;

public class Player {
    private int hp;
    private String name;
    private String battleCry;

    …
}
```

Использование

- При использовании класса в другом классе ДРУГОГО пакета его надо импортировать:

```
import work.projects.kickgame.Player;

public class Game {
    public void go(){
        Player p = new Player("Denis Popov");
    }
}
```

У Game не указан пакет – Game лежит в пакете «по умолчанию»

Использование

- Но если предположить, что Game лежит в том же пакете, что и Player, import для Player не требуется:

```
package work.projects.kickgame;

public class Game {
    public void go(){
        Player p = new Player("Denis Popov");
    }
}
```

Единица компиляции (Compilation Unit)

Понятие в Java, означающее базовую конструкцию, которая может быть скомпилирована. Включает в себя:

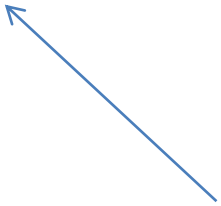
- Пакет
- Импорты
- Объявление сущности (пока для нас это всегда класс)
  - В перспективе разумеется не только.

Представим, что мы дописали игру

```
package work.projects.kickgame;

public class Game {
    public void go(){
        Player p = new Player("Denis Popov");
        //..
    }

    public static void main(String[] args) {
        (new Game()).go();
    }
}
```



Объект без ссылки (используется ровно 1 раз)

- ГО КОМПИЛИТЬ И ГАМАТЬ!

Переходим в папку, где лежат Player и Game

- Компилировать надо все, но можно сразу вызвать Game (в нем main) – он инициирует компиляцию всего необходимого (связанных с ним классов – в нашем случае Player)

- Выполняем

```
javac Game.java
```

  - ай-ай (подробности на следующем слайде)

Что видит студент

```
F:\projects\kickgame>javac Game.java
Game.java:5: error: cannot find symbol
        Player p = new Player("Denis Popov");
        ^
  symbol:   class Player
  location: class Game
Game.java:5: error: cannot find symbol
        Player p = new Player("Denis Popov");
                       ^
  symbol:   class Player
  location: class Game
2 errors
```

Какая-то ошибка... ((

Что вывелось на экран

```
F:\projects\kickgame>javac Game.java
Game.java:5: error: cannot find symbol
        Player p = new Player("Denis Popov");
        ^
  symbol:   class Player
  location: class Game
Game.java:5: error: cannot find symbol
        Player p = new Player("Denis Popov");
                       ^
  symbol:   class Player
  location: class Game
2 errors
```

Если вкратце: Game ~~под носом у себя~~ не видит Player.

Чтобы заработало, нужно

1. создать иерархию папок, соответствующую иерархии пакетов
2. поместить исходные файлы в нужные места
3. поместить все полученное в специальную папку для исходников
   - исходники по-английски – **source**
4. компилировать из корня проекта (та папка, которая содержит sources).

Иерархия для Player и Game

```
Project
    src
        work
            projects
                kickgame
                    Game.java
                    Player.java
```

папка - корень проекта

папка с исходниками проекта

Вот так должно работать

```
javac src\work\projects\kickgame\Game.java
```

Вот так должно работать

```
javac src\work\projects\kickgame\Game.java
```

```
Game.java:5: error: cannot find symbol
        Player p = new Player("Denis Popov");
        ^
  symbol:   class Player
  location: class Game
Game.java:5: error: cannot find symbol
        Player p = new Player("Denis Popov");
                       ^
  symbol:   class Player
  location: class Game
2 errors
```

Вот так должно работать

```
javac src\work\projects\kickgame\Game.java
```

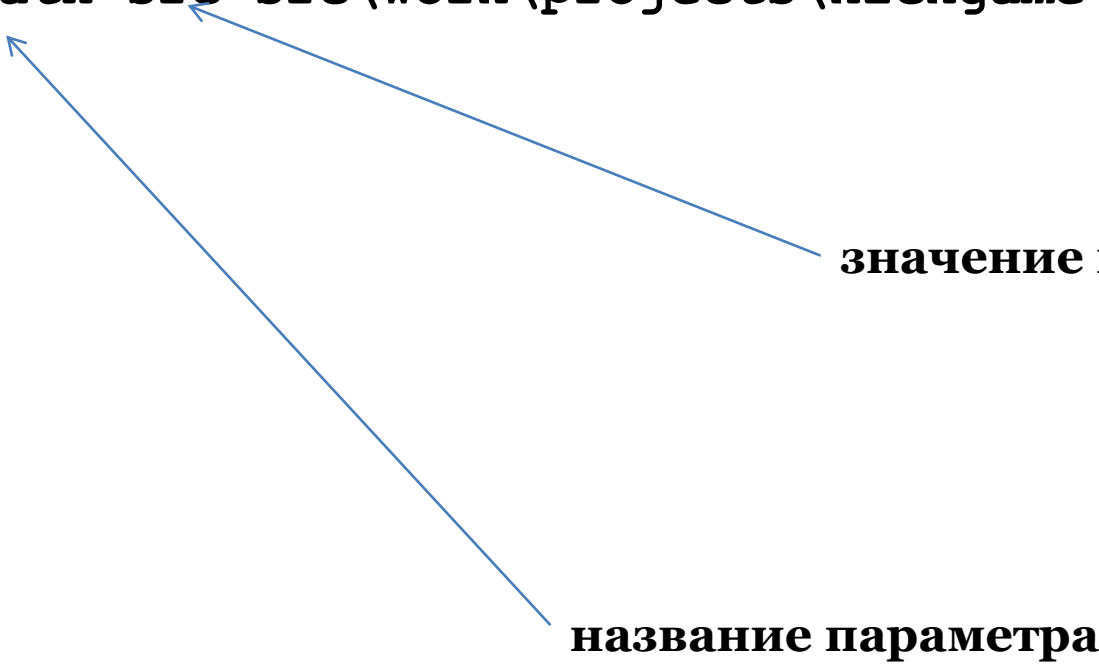
```
Game.java:5: error: cannot find symbol
        Player p = new Player("Denis Popov");
        ^
  symbol:   class Player
  location: class Game
Game.java:5: error: cannot find symbol
        Player p = new Player("Denis Popov");
                   ^
  symbol:   class Player
  location: class Game
2 errors
```

Ну вот... ((

Разыскивается Player

- Надо при компилировании Game сказать, где лежит Player.java
- Параметр/ключ для команды в командной строке под названием **sourcepath**

```
javac -sourcepath src src\work\projects\kickgame\Game.java
```



Запускаем Game

- А нет, рано. Посмотрите на содержимое папок:

```
Project
    src
        work
            projects
                kickgame
                    Game.class
                    Game.java
                    Player.class
                    Player.java
```

Все исходники перемешаны с .class-файлами. Это плохо.
Попросят скинуть исходники/бинарники – будете ходить по всем папкам и удалять лишнее – муторно, долго, неэффективно.

Разделение при компиляции

- Параметр компиляции -d
  - Указывает, куда поместить соответствующие скомпилированные файлы?
- Указываем как значение папку рядом с src
  - Т.к. будет аналогичная иерархия папок для пакетов
- Название этой папки варьируется:
  - bin, out, classes. Мы возьмем bin.

Выполняем

```
javac -sourcepath src -d bin src\work\projects\kickgame\Game.java
```

Получаем:

```
Project
    bin
        work
            projects
                kickgame
                    Game.class
                    Player.class
    src
        work
            projects
                kickgame
                    Game.java
                    Player.java
```

Но бывает и такое

Может случиться, что мы используем стороннюю библиотеку или, например, у нас нет Player.java, а есть только Player.class

При запуске необходимо указывать, откуда брать .class-файлы, необходимые для запуска. Этот параметр называется **classpath (при компиляции можно писать cp)**

Classpath

- Важнейшее понятие в проектах на Java
  - Которое прячут среды разработки
    - » Поэтому некоторые из вас все еще на Sublime и Notepad++
- Переменная, содержащая путь (или пути через ;) к необходимым бинарникам:
  - Путь к корню иерархии пакетов проекта
  - Пути к библиотекам
- В нашем случае classpath – папка bin.

Запуск Game!

```
java -classpath bin work.projects.kickgame.Game
```

Указываем полное имя класса

Запускаем, находясь в корневой папке проекта.

ТУТ CLASSPATH ОБЯЗАТЕЛЕН!

Ура!

- Мы научились создавать пакеты, компилировать и запускать классы, находящиеся в разных пакетах!

- Вот только мы кое-что упустили…

Названия пакетов

- Разработчиков много, названий пакетов мало!
  - Та же проблема, что и с классами.
    - В Индии, в Кении, в США, в России и т.д. слова *work, project, game* имеют одинаковый смысл – есть риск совпадения не только имен классов, но имен пакетов.

- *Как решить проблему уникальности имен пакетов?*

Решение

- У каждой серьезной компании, занимающейся разработкой, есть **сайт**.
- Берем **его** название, **разворачиваем** – получаем начало названия пакетов проектов данной компании.
- Дальше можно идти по иерархии отделов, проектов в самой компании, пока это требуется для уникальности.

Мы

- ***itis.kpfu.ru***

- Разворачиваем, получаем: ***ru.kpfu.itis***
  - Достаточно ли поместить Player и Game туда? Вряд ли, ага.

- Данный код может быть неуникален на уровне:
  - Преподавателя (не только Абрамский на Java читает)
  - Предмета (не только на информатике можем писать java код)
  - Курса (не только на 1 курсе может быть Player)
  - Года (такой код мог писать 1 курс в 2012, 2014, 2015 и т.п.)

- Подбираем имя, учитывающее все эти условия:
  - ***ru.kpfu.itis.abramskiy.informatics.year2017.course01***
  - и туда помещаем Player, Game

  - разумеется можно упросить имя пакета для своих нужд – подумайте самостоятельно.

Официальное заявление

- Если вы поняли пакеты и можете легко скомпилировать классы с учетом пакетов, sourcepath и classpath, можете переходить на среду разработки:
  - Intellij Idea
  - Eclipse
  - NetBeanse
  - …

Новый модификатор доступа

```
class Player {
    int hp;
    String battleCry;

    Player() {
        hp = 100;
    }

    void shoutBattleCry() {
        System.out.println(battleCry);
    }
}
```

Нет модификатора? Нет! Есть!
Модификатор по умолчанию (default)
«Прямой доступ извне разрешен всем «однопакетникам» (классам, объявленным в том же пакете)»

В Game могу спокойно:

```
Player p = new Player("Denis Popov");
p.hp = 100500;
```

Насколько это адекватно – другой разговор.

Внимание

- Дальше снова примеры без пакетов, чтобы не нагружать восприятие лекции, но вы-то теперь знаете, как делать правильно.

Тук-тук-тук!

Заказчик: *«Вы чего так долго с журналом возились! Сроки срываете! Быстрее дописывайте!...*

... а, кстати, 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 заказчику~~ по требованиям заказчика

- Есть обычный игрок
  - атрибуты hp, name, battleCry, методы kick, battleCry
- А есть продвинутый игрок
  - атрибуты hp, name, battleCry, метод 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

- Жуткое 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.

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.

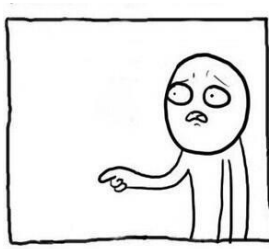
```
public class HealerPlayer extends Player {
    private int healPoints;
```

Добавим метод heal

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

    public void heal(int p) {
        if (p <= healPoints) {
            hp += p;
            healPoints -= p;
        }
    }
}
```

- 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
- *Вот вы и выучили 4 основных модификатора доступа в java.*

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

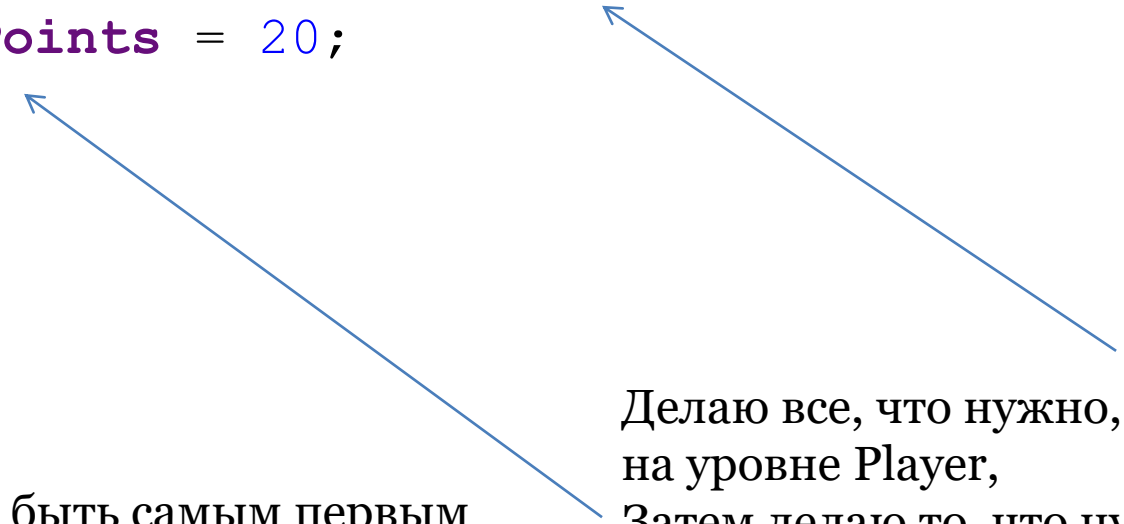
```
public HealerPlayer(String name, String battleCry) {
    hp = 100;
    healPoints = 20;
    this.name = name;
    this.battleCry = battleCry;
}
```

- 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;
}
```

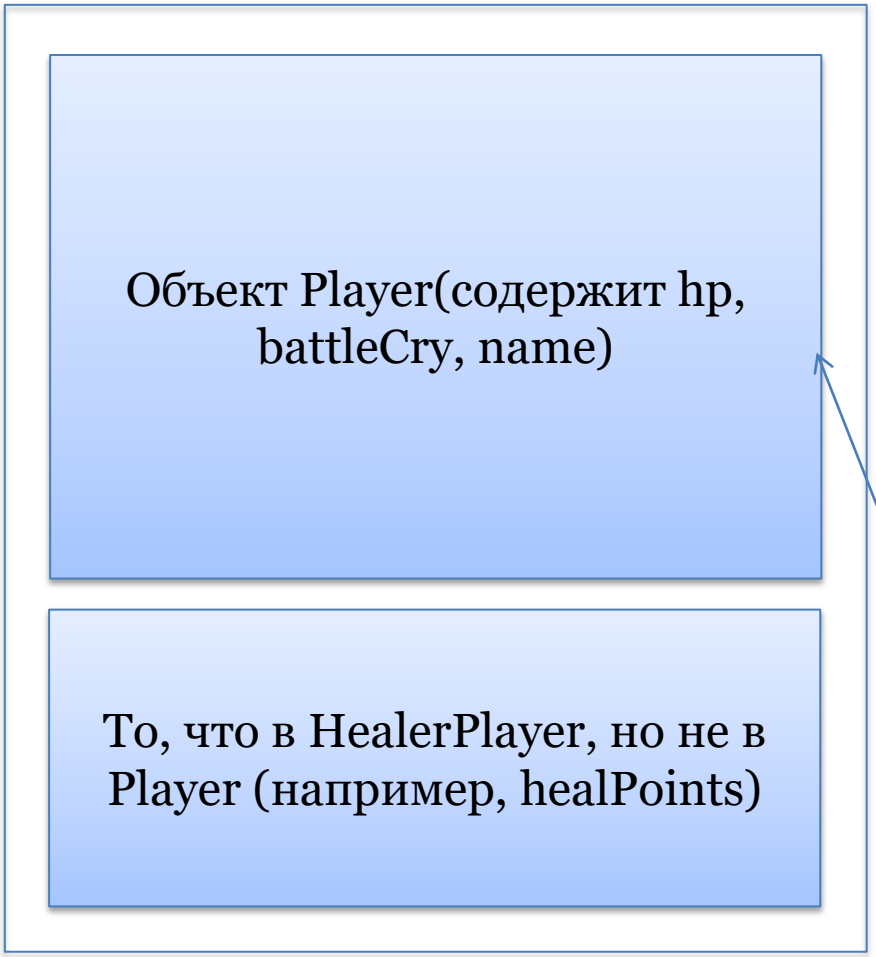


super обязан быть самым первым в дочернем конструкторе

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)

Нам это еще пригодится.

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