

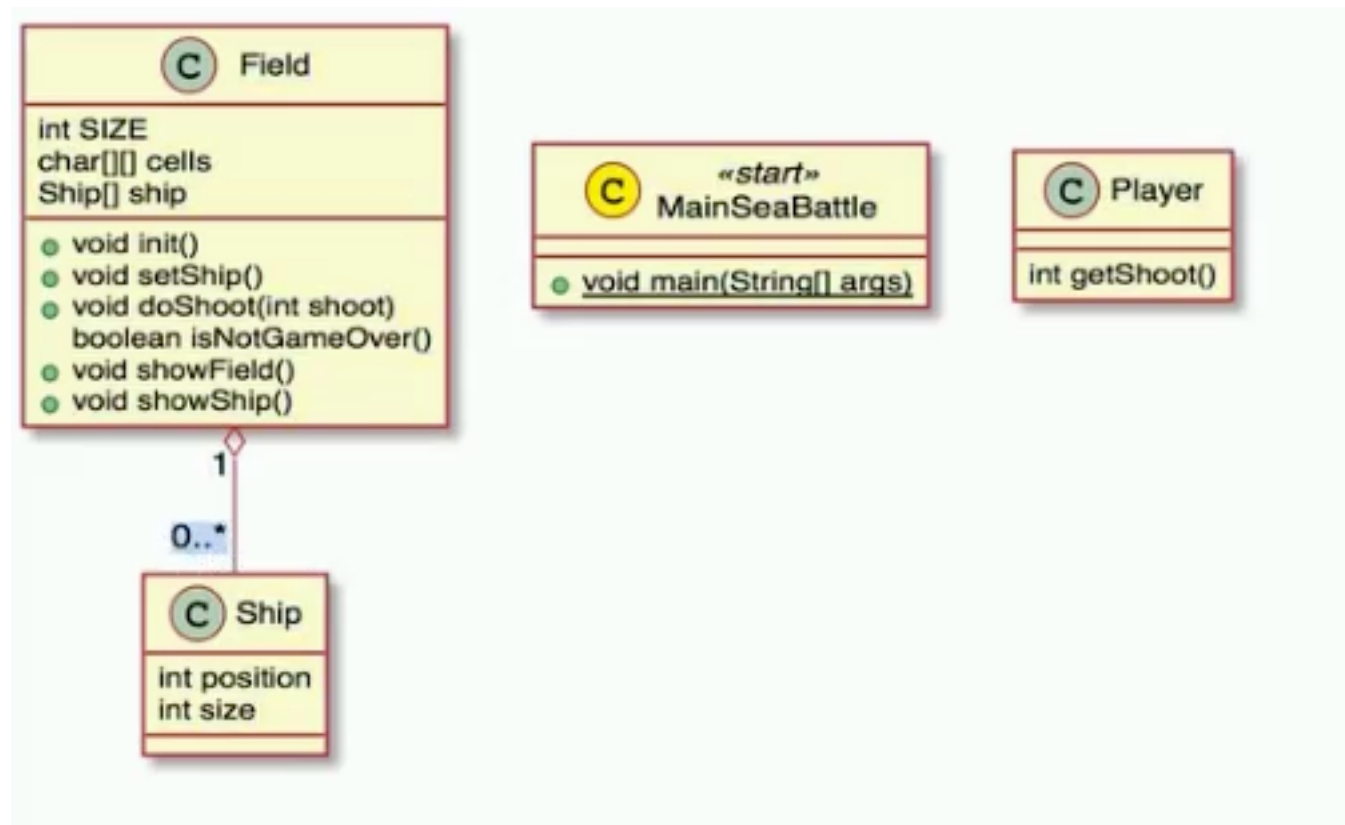
Java 0

JavaDive
day two

Первая часть

UML

- Основная диаграмма в UML — диаграмма классов. Это просто визуальное отображение классов в виде прямоугольников и связей между ними
- Преобразователь кода в UML диаграмму можно скачать здесь: java2uml.ru



The screenshot shows an IDE with the following code in `MainSeaBattle.java`:

```
/**
 * Created by 000 on 13.04.2015.
 */
public class MainSeaBattle {

    public static void main(String[] args) {

        Field field = new Field();
        Player player = new Player();

        field.init();
        field.setShip();

        System.out.println("Начало игры!");

        do {
            field.showField();
            field.doShoot(player.getShoot());
        } while (field.isNotGameOver());
    }
}
```

The project structure on the left includes `test4SeaBattle`, `.idea`, `src` (containing `Field`, `MainSeaBattle`, `Player`, and `Ship`), `classes.plantuml`, `diagram.png`, and `test4SeaBattle.iml`.

ArrayList

- ArrayList — класс в Java, который является, как бы «оберткой» над обычным массивом. С ним ваши массивы становятся динамическими!
- Можно менять их размер, произвольно добавляя элементы с помощью метода `add()`.
- Можно удалять элементы с помощью метода `remove()`
- Или пробегать по всему ArrayList в цикле, как по обычному массиву

Пример использования

- В обычный ArrayList можно помещать разные объекты вперемешку

```
public class Main2 {  
    public static void main(String[] args) {  
        ArrayList list = new ArrayList();  
  
        list.add(123);  
        list.add("hello");  
        list.add("hello");  
        list.add(333);  
        list.add(333);  
  
        System.out.println(list);  
  
        list.remove("hello");  
        list.remove(new Integer(333));  
        // list.remove(1);  
  
        for (Object o : list) {  
            System.out.println(o);  
        }  
    }  
}
```

Generics и ArrayList

- Дженирики — это способ указать, что в данный ArrayList можно поместить только определенный тип данных, для этого указываем в угловых скобках ЭТОТ ТИП:

```
ArrayList<String> list = new ArrayList<>(); // Generics
```

- *Начиная с седьмой версии Java можно справа указывать только <>, а вот в шестой придется повторить <String>*

Вторая часть

O static

- Модификатор `static` перед именем поля класса или метода означает, что эта переменная или метод теперь принадлежит самому классу, то есть является общей для всех его экземпляров.
- Наглядно это видно, если хотим посчитать, сколько экземпляров кошки мы сделали. Для этого переменную `amount` придется описать как `static`, иначе она все время будет обнуляться для каждой следующей вновь созданной кошки

Пример с static

- Чтобы посчитать кошек мы не только переменную `catsAmount` сделали `static`, но еще и метод `showCatsAmount`, который с ней работает, тогда к нему можно обращаться без создания экземпляра класса:

```
public class Cat {
    static int catsAmount;

    String name; // null
    int id;

    public Cat(String name) {
        this.name = name;
        catsAmount++;
        id = catsAmount;
    }

    static void showCatsAmount() {
        System.out.println("Кошек: " + catsAmount);
    }
}
```

[illegible]

ВЫЗОВ МЕТОДОВ

- Из статического метода можно вызвать только другие статические методы и работать только со статическими переменными (Цари кушают отдельно)
- Из обычного же метода можно вызвать любой метод, хоть статический, хоть не статический. И можно работать и со статическими и не статическими переменными

```
public class Main2 {  
    public static void main(String[] args) {  
        // notAStatic(); // так нельзя  
        staticMethod();  
    }  
  
    void notAStatic() {  
  
    }  
  
    static void staticMethod() {  
  
    }  
}
```

О equals и ==

- Примитивные переменные == сравнивает без всяких проблем и неожиданностей
- А вот объектные переменные, ссылочные, те, что с большой буквы принято писать, сравнивает не по значению, а по тому, одинаковый ли у них адрес в памяти
- Это может привести к проблеме, если сравним две строки, с одинаковым содержимым, но случайно оказавшихся в разных частях памяти (такое иногда бывает)
- Поэтому строки всегда нужно сравнивать с помощью метода .equals()

Пример

- Сначала будет выведено «не равны», а затем «равны»

```
public class Main {  
    public static void main(String[] args) {  
        String string1 = "hello"; // строковый пул  
        String string2 = new String("hello");  
  
        if (string1 == string2) {  
            System.out.println("равны");  
        } else {  
            System.out.println("не равны");  
        }  
  
        // int a = 10; // у примитивных типов equals нет  
  
        if (string1.equals(string2)) {  
            System.out.println("равны");  
        } else {  
            System.out.println("не равны");  
        }  
    }  
}
```

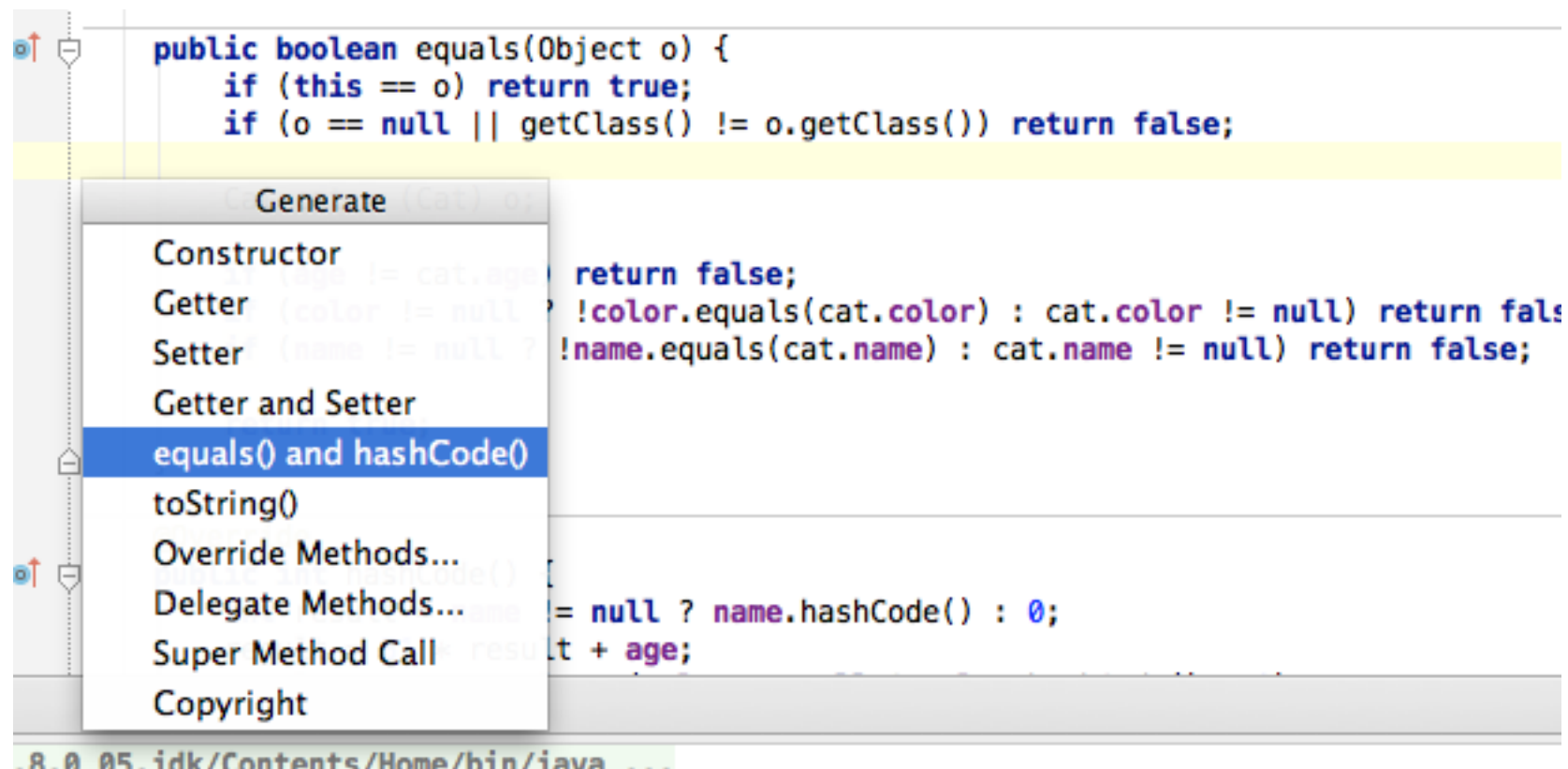
equals и новые классы

- Для тех классов, которые мы создаем сами, equals по умолчанию работает так же, как и ==
- Поэтому в этом примере будет дважды «не равно»

```
public class Main2 {  
    public static void main(String[] args) {  
        Cat cat1 = new Cat("Kitty");  
        Cat cat2 = new Cat("Kitty");  
  
        if (cat1 == cat2) {  
            System.out.println("равны");  
        } else {  
            System.out.println("не равны");  
        }  
  
        if (cat1.equals(cat2)) { // equal по умолчанию работает как == (кроме String)  
            System.out.println("равны");  
        } else {  
            System.out.println("не равны");  
        }  
    }  
}
```

Сравнение с помощью equals

- Чтобы можно было сравнивать новые объекты с помощью equals, нам нужно в своем классе «переопределить» этот метод. То есть создать такой же, но с нужным нам содержимым.
- Можно упростив задачу через меню генерации (Alt-Ins на Windows, Ctrl-Enter на Mac)



Пример

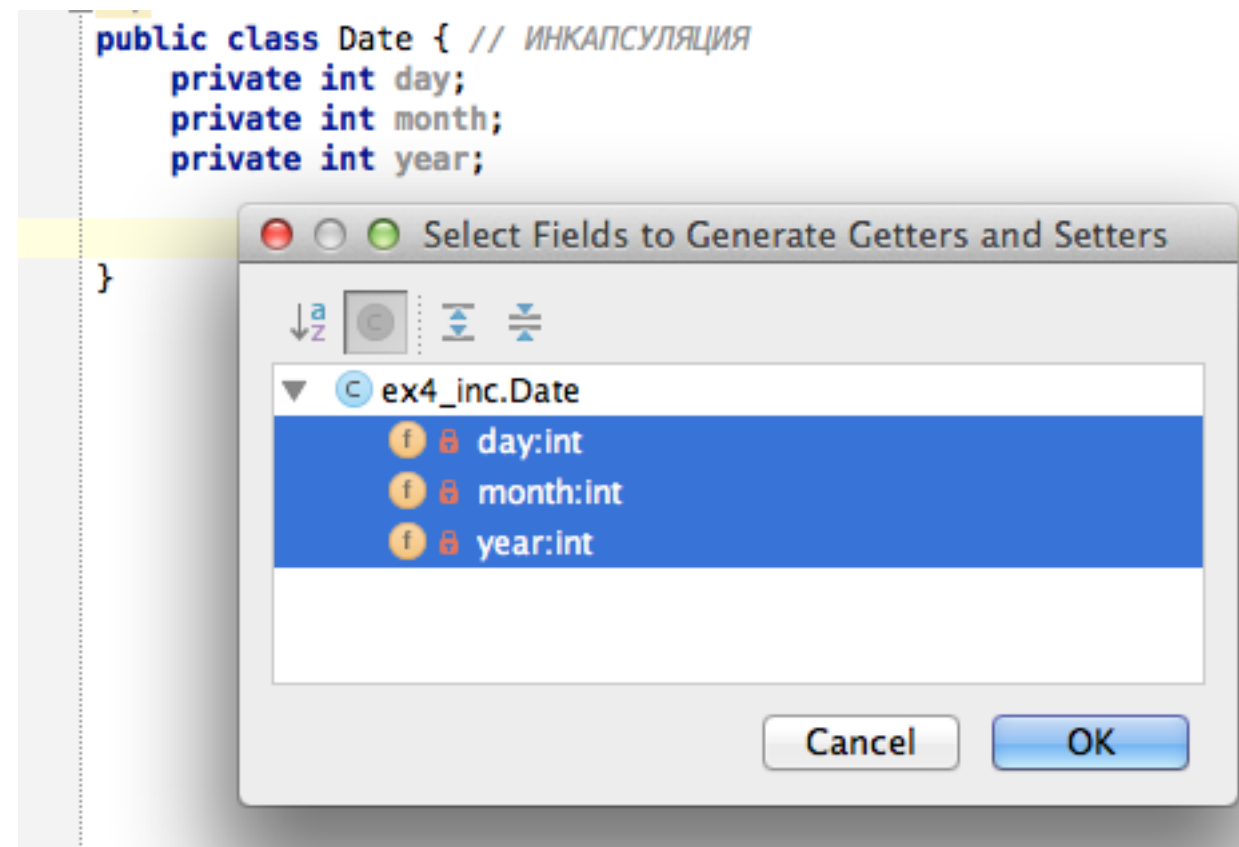
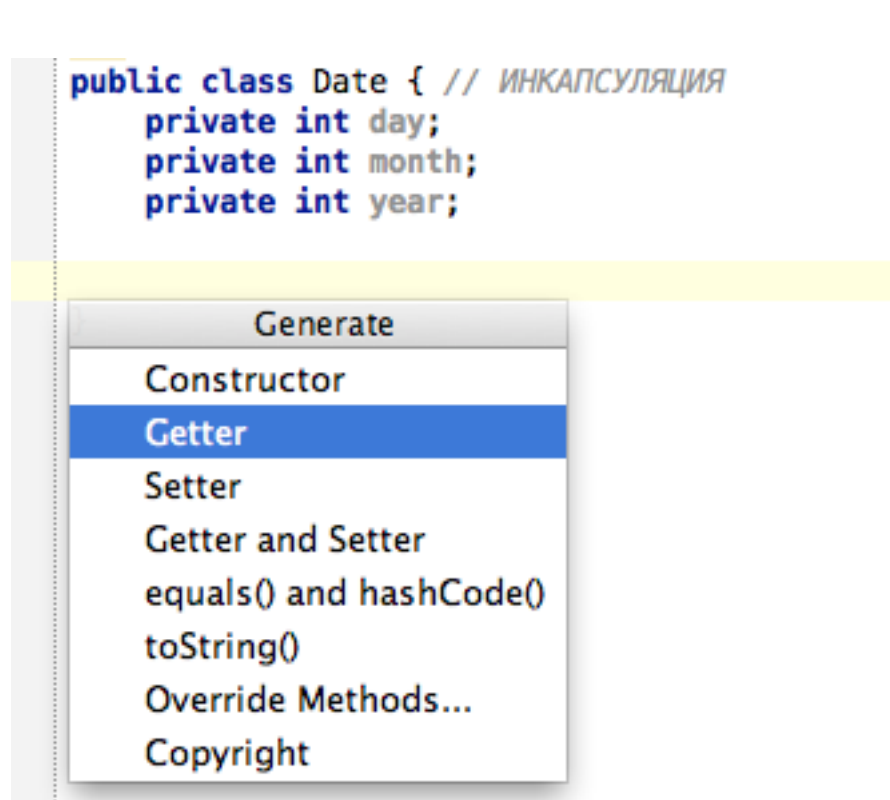
- Пример автоматически сгенерированного equals

```
public class Cat {  
    String name;  
    int age;  
    String color;  
  
    public Cat(String name) {  
        this.name = name;  
    }  
  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
  
        Cat cat = (Cat) o;  
  
        if (age != cat.age) return false;  
        if (color != null ? !color.equals(cat.color) : cat.color != null) return false;  
        if (name != null ? !name.equals(cat.name) : cat.name != null) return false;  
  
        return true;  
    }  
}
```

Третья часть

ООП: инкапсуляция

- Все поля класса принято помечать как `private`. Но чтобы к ним получить доступ создавать т.н. «геттеры» и «сеттеры»
- Задачу можно упростить с помощью того же меню генерации



Пример геттеров и сеттеров

```
public class Date { // ИНКАПСУЛЯЦИЯ
    private int day;
    private int month;
    private int year;

    public int getDay() { // Геттеры и Сеттеры
        return day;
    }

    public void setDay(int day) {
        if (day <= 31) {
            this.day = day;
        }
    }

    public int getMonth() {
        return month;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }
}
```

ООП: наследование

- Наследование — это способ построить «Матрешки» из классов. То есть сделать так, чтобы описав класс один раз, например `Animal`, все остальные производные от него классы, например `Cat`, `Dog` etc., получили бы в свое распоряжение все (почти) его переменные и методы.
- Все классы являются потомками от «первородной обезьяны», класса `Object`
- Чтобы указать, что наш класс это потомок другого класса используем слово `extends`

```
public class Cat extends Animal{
```

Класс предок Animal

```
public class Animal { // extends Object
    String name;
    int age;

    Animal() {
    }

    public Animal(int age) {
        this.age = age;
    }

    public void run() {
        System.out.println("Animal run");
    }

    void animalSuperMethod() {
    }
}
```

Потомок Cat

```
public class Cat extends Animal{ // принцип матрешки
    String home;
    String murrString;

    @Override
    public void run() {
        System.out.println("Кошка бежит за мышкой");
        // super.run();
    }
}
```

Или потомок Dog

```
public class Dog extends Animal {  
    // Animal super; // скрытая переменная (метафора)  
  
    Dog() {  
        super(4); // обращение к родительскому конструктору  
    }  
  
    @Override  
    public void run() {  
        System.out.println("Собака бежит за кошкой");  
        super.run();  
    }  
}
```

ООП: полиморфизм

- Полиморфизм — это возможность создать массив предков и поместить в него самых разных потомков.
- Так можно сделать, так как внутри каждого потомка всегда кроется предок! Помним принцип «Матрешки». То есть у каждого потомка будут все (почти) переменные и методы, которые есть у предка
- Обратите внимание, что в потомка предка нельзя поместить просто так! Так как нет гарантии, что у предка будут те же методы и переменные, что и у потомка.
- Смысл полиморфизма в переопределении методов предка каждым потомком по своему. Это дает нам возможность пробежать по массиву и у всех его элементов вызвать метод с одинаковым названием, а вот отработает он по разному! В зависимости от того, как именно он написан в потомках.

Пример полиморфизма

- Пробегаем по массиву животных и у всех вызываем метод run()

```
public static void main(String[] args) {  
    Cat cat = new Cat();  
    cat.name = "Tom";  
    System.out.println(cat.name);  
  
    Animal[] animals = new Animal[4];  
    animals[0] = new Cat();  
    animals[1] = new Dog();  
    animals[2] = new Crocodile();  
    animals[3] = cat;  
  
    for (Animal animal : animals) {  
        animal.run(); // ПОЛИМОРФИЗМ  
    }  
}
```


Бонус

маленький пример со Swing

Делаем простое графическое окно

- Применим знания о наследовании и просто отнаследуем наш класс от класса JFrame, который умеет открывать окна и помещать на них элементы

```
public class MyWindow extends JFrame {
```

```
    public MyWindow() {  
        setTitle("hello");  
        setSize(300, 400);  
        setVisible(true);  
    }  
}
```

- После чего можем в другом классе открыть новое окно:

```
public class Main {  
    public static void main(String[] args) {  
        new MyWindow();  
    }  
}
```

Улучшаем графическое ОКНО

- Лучше всего было бы вызывать еще один хитрый метод, чтобы наша программа закрывалась при закрытии окна:

```
public class MyWindow extends JFrame {  
    public MyWindow() throws HeadlessException {  
        setTitle("hello");  
        setSize(300, 400);  
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);  
        setVisible(true);  
    }  
}
```

- После чего можем в другом классе открыть новое окно:

```
public class Main {  
    public static void main(String[] args) {  
        new MyWindow();  
    }  
}
```

Просто пример

- Поместим в наше окно элемент ТекстовоеПоле и наполним его значением:

```
public class MyWindow extends JFrame {  
    static JTextArea textArea;  
  
    public MyWindow() throws HeadlessException {  
        setTitle("hello");  
        setSize(300, 400);  
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);  
        textArea = new JTextArea();  
        add(textArea);  
        textArea.setText("Hello, Java Dive!\n");  
        for (int i = 0; i < 20; i++) {  
            textArea.setText(textArea.getText() + "ho ho ho Super\n");  
        }  
  
        setVisible(true);  
    }  
}
```