ООП

Самая важная лекция

Содержание

1. Что такое ООП

В предыдущей лекции мы писали код в котором создавали разные классы. Класс для треугольника, класс для точки. Класс для прямоугольника и т.д. Можем ли мы писать тот же код без создания классов? Конечно. Тогда у нас просто будет простыня (длинный код) и мы не сможем понять ничего почти никогда. Классы помогают нам структурировать наш код, переиспользовать логику и т.д. Вообще говоря есть несколько видов языков программирования и джава относится к обьекто-ориентированным. Что это такое? ООП. Обьектно-ориентированное программирование. Это когда вы пишете классы проще говоря. Т.е. вы не пишете переменная сторона треугольника 1, переменная сторона треугольника 2 и т.д. а пишете класс Треугольник у которого 3 стороны. Мы в ООП пишем код так, как будто переносим нашу реальную жизнь в мир программирования. Ведь что такое треугольник? Это же геометрическая фигура, верно? Обьект. Скажем так. Что такое карандаш? Обьект. Что такое доска? Тоже обьект. Мы считаем, что все вокруг это обьекты. Мы же в реальной жизни разделяем и разграничиваем обьекты, верно? Если нам нужно написать что-то, то мы берем пишущий обьект (карандаш, ручку, фломастер, мелки) и пишем на обьекте, на котором это возможно (лист бумаги, доска, плакат, асфальт в конце концов). Если бы мы писали код по аналогии с этой ситуацией, то мы бы брали наш пишущий обьект и вызывали бы у него метод письма, которому передавали бы в аргумент целевой обьект, где нужно вывести данные. Ну или наоборот, как у нас в System.out.println.

И у ООП конечно же есть какие-то припципы, которые собственно и делают ООП тем, что оно есть.

2. Инкапсуляция

```java
public class Triangle {

    private final int sideA;
    private final int sideB;
    private final int sideC;

    public Triangle(Point a, Point b, Point c) {
        this(a.getDistanceTo(b), a.getDistanceTo(c), b.getDistanceTo(c));
    }
```

На самом деле мы уже применяли этот принцип в наших лекциях. Помните класс треугольник? Тогда, когда мы закрываем доступ к полям класса и не даем их менять после создания обьекта это и есть инкапсуляция. Когда извне нельзя напрямую иметь доступ к полям класса. Можно только получить информацию по виду треугольника или получить площадь треугольника, а как оно там все рассчитывается тому, кто вызывает этот метод знать не нужно.

Инкапсуляция в реальной жизни это когда вы нажимаете на кнопку на принтере и ваш документ выходит из него. Как там все устроено внутри вы не знаете и не нужно знать. Вам доступны лишь некоторые вещи, которые уже внутри сделают всю работу. От вас нужно лишь подключить в розетку принтер и положить бумагу. Все остальное скрыто за корпусом и недоступно для вас как для конечного юзера.

В нашем случае даже лучше, если мы просто передаем 3 точки и все там внутри проверяется и создается. Нам для того чтобы знать например прямоугольный ли треугольник или нет не нужно ничего делать, все должно быть сделано внутри уже к тому моменту, когда мы лишь вызываем метод например getDescription. Я как юзер (тот кто вызывает код из мейна) не хочу знать ни теорему Пифагора, ни правила существования треугольника. И тут кто-то скажет – но ведь мы сами написали это все. Да, потому что мы пишем с нуля то же что и юзаем. А в реальной разработке зачастую вы используете те классы, которые написали ранее другие люди. И вот вам не нужно знать как оно все там работает. Вы хотели бы просто вызвать один метод и чтобы все там само собой работало. Вот суть инкапсуляции.

Вы можете услышать простую формулировку – когда поля класса private но это лишь упрощение. Потому что многие делают ошибку и даже обьявив поле private отдают их же в паблик методах и называют их геттерами. Запомните, геттеры это зло. И суть инкапсуляции нарушится. Зачем вам обьявлять поле класса закрытым и открывать доступ к нему через метод? Хотя согласен, в некоторых редких случаях нужно дать информацию по полям класса, но это скорей плохой код, чем исключение из правил. Та же штука про сеттеры. Делать поле класса private но не final и назначать значение через метод называя его setter. Не надо так делать.

Вы можете представить такую ситуацию – инициализурем треугольник посредством 2 точек, а третью типа дописываем позже. Нет конечно же, зачем так делать. Каждый обьект должен требовать все нужные ему данные сразу же при инициализации. Эти поля и есть строго говоря суть этого обьекта. Вообще изменяемых полей в классе не должно быть.

Вот рассмотрим модель данных для выбора варианта типа один из многих.

```java
public class Option {

    private final String description;
    private final boolean isChosen;

    public Option(String description, boolean isChosen) {
        this.description = description;
        this.isChosen = isChosen;
    }
}
```

У этой модели данных должно быть условно 2 поля – текст и boolean флаг выбрано или нет.

И если вы делаете хорошо и правильно, то оба поля у вас будут private final. Ведь обьект этого класса описывает состояние выбора. Предположим у вас 2 обьекта и пока ничто из них не выбрано. И после уже юзер выбирает. И нужно поменять флаг на true. И вы бы могли написать код типа такого

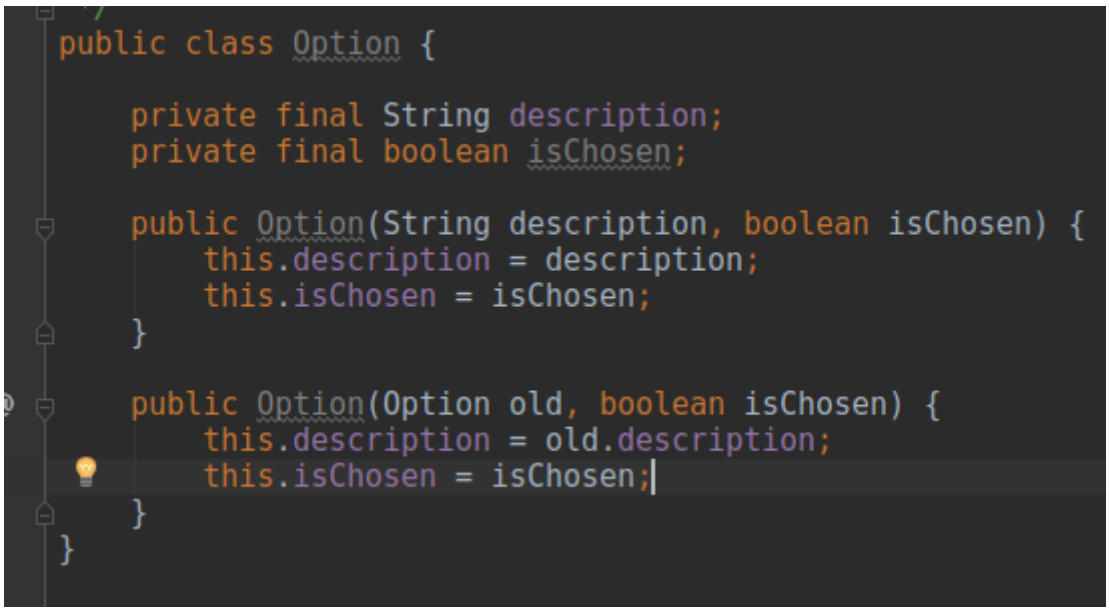
```java
public class Option {

    private final String description;
    private boolean isChosen;

    public Option(String description, boolean isChosen) {
        this.description = description;
        this.isChosen = isChosen;
    }

    public void setChosen(boolean chosen) {
        isChosen = chosen;
    }
}
```

Да, можно взять один из существующих обьектов и у него поменять поле. Но я не рекомендую так делать. А как тогда? Нужно породить новый обьект и положить в конструктор нужные значения. Для этого можно написать удобный конструктор, который примет в аргумент обьект этого же класса. Типа.

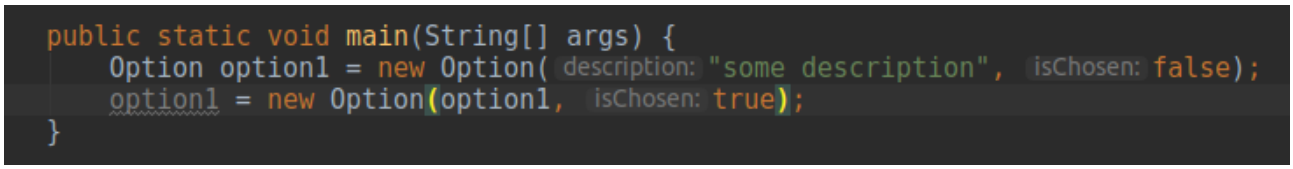


```
public class Option {

    private final String description;
    private final boolean isChosen;

    public Option(String description, boolean isChosen) {
        this.description = description;
        this.isChosen = isChosen;
    }

    public Option(Option old, boolean isChosen) {
        this.description = old.description;
        this.isChosen = isChosen;
    }
}
```



```
public static void main(String[] args) {
    Option option1 = new Option( description: "some description",  isChosen: false);
    option1 = new Option(option1,  isChosen: true);
}
```

Зачем же так делать? Можно было просто вызвать обычный конструктор и передать ему…. Описание? А откуда вы его возьмете? А если у класса завтра будет не 2 поля, а например 5. А вам условно менять только этот флаг выбран ли обьект или нет.

Запомните этот случай. Мы к нему вернемся позже.

3. Наследование

Давайте на секунду вернемся к нашим геометрическим обьектам. Рассмотрим 3 типа – круг, прямоугольник и треугольник. Давайте предположим что мы написали некий класс, который определяет валидность данных и мы порождаем уже готовые обьекты. Т.е.писать проверку аргументов не нужно.

Теперь, наша задача например передать стороны в аргумент класса и рассчитать площадь фигуры и ее периметр. Как бы мы это сделали? Давайте посмотрим.

Для круга мы передаем лишь 1 сторону – радиус. И рассчитывем площадь и периметр.

```java
public class Circle {

    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getPerimeter() {
        return 2 * 3.14 * radius;
    }

    public double getArea() {
        return 3.14 * radius * radius;
    }
}
```

В прямоугольник мы передадим 2 стороны и высчитаем площадь и периметр какими-то своими способами.

```java
public class Rectangle {

    private final double a;
    private final double b;

    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }

    public double getPerimeter() {
        return a + a + b + b;
    }

    public double getArea() {
        return a * b;
    }
}
```

Треугольнику мы передадим уже 3 стороны и высчитаем площадь универсальным способом.

Хорошо, мы написали три класса. Давайте посмотрим на них внимательно. Что у них общего? Во-первых они все геометрические фигуры и у всех у них есть площадь и периметр. Хорошо, мы помним про истинное предназначение методов – упростить нам жизнь и переиспользовать их. В чем же тогда истинное предназначение классов? В том же – упростить нам жизнь и вынести повторяющийся код в него. Но погодите. Мы написали 3 класса и у них у всех очень много общего. Во-первых все конструкторы принимают аргументом стороны – да, где-то 1, где-то 2, а где-то и 3, но по крайней мере они все одного типа. А еще они все по-своему рассчитывают площадь и периметр. Но они все же делают это и там и там и там, так?

```java
public class Triangle {

    private final double a;
    private final double b;
    private final double c;

    public Triangle(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public double getPerimeter() {
        return a + b + c;
    }

    public double getArea() {
        double p = (a + b + c) / 2;
        return Math.sqrt(p * (p - a) * (p - b) * (p - c));
    }
}
```

Было бы классно иметь какой-нибудь общий класс аля Геометрическая фигура, который бы хранил стороны и имел методы для расчета периметра и площади. Но при этом мы не можем породить обьект этого класса. В языке Java это называется абстрактный класс. Единственное его отличие от обычного класса именно в том, что нельзя создать обьект этого класса. Давайте посмотрим.

```java
public class Main {

    public static void main(String[] args) {
        Figure figure = new Figure();
    }
}
```

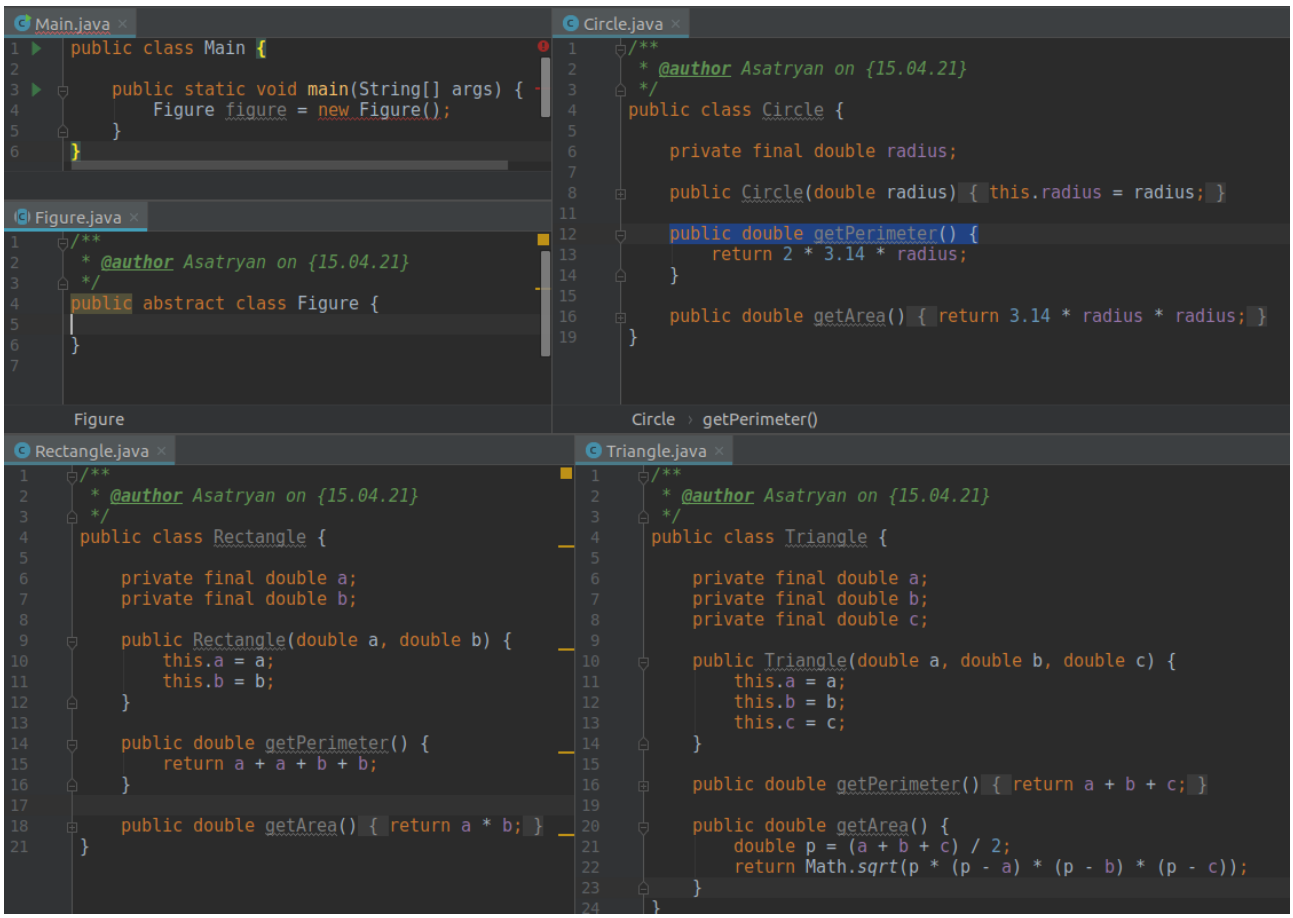
Main › main()

Figure.java

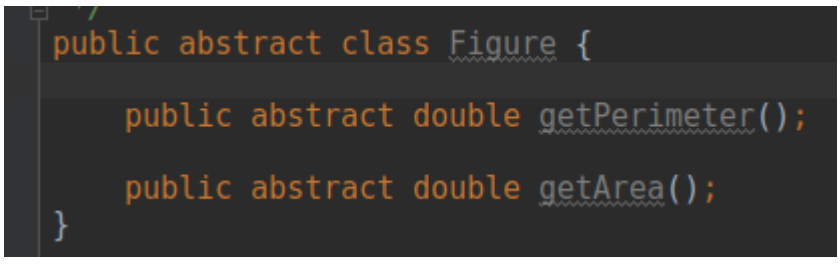
```java
/**
 * @author Asatryan on {15.04.21}
 */
public abstract class Figure {

}
```

Кстати, вы можете перенести отображение класса в идее нажав на правый клик по табу и Move Down. С помощью этого можно добиться того, что сразу видеть несколько классов.



Так вот. Мы создали абстрактный класс, но не можем создать обьект этого класса. Так зачем же мы его написали? Не забываем – для того, чтобы вынести общий повторяющийся код в него из классов Circle, Rectangle и Triangle. Начнем с простого – методов периметра и площади. Они разнятся по логике в каждом классе и потому мы не можем выделить какой-то уникальный код для всех случаев. Значит метод в абстрактном классе тоже будет абстрактный для обоих случаев. Давайте попробуем сделать это.



Так, если у нас есть абстрактный класс и нельзя создать обьект, то по той же логике и будут абстрактные методы в этом классе которые нельзя вызвать? Ну тип того. Так как разные классы имеют разную реализацию общих методов, то в абстрактном классе пишем абстрактные методы, т.е. у них отсутствует реализация (нет фигурных скобок, метод кончается ;). Ладно, с методами разобрались. А как быть с полями? Они разные в отношении количества. И здесь мы можем использовать наши массивы. Пусть будет конструктор с массивом чисел. Так, стоп. Конструктор для класса, обьект которого нельзя создать? Ну да. Что такого. Наша цель хранить данные по сторонам. Давайте напишем массив.

```
public abstract class Figure {

    private final double[] sides;

    protected Figure(double[] sides) {
        this.sides = sides;
    }

    public abstract double getPerimeter();

    public abstract double getArea();
}
```

После того как мы написали private final double[] sides у нас высвечивается ошибка от идеи и мы жмем Alt+Enter и генерируем конструктор. Обратите внимание, что у конструктора модификатор доступа (да, все эти public private protected называются модификаторами доступа) не public. Ведь тогда бы он был доступен отовсюду. Но и не private. Ведь тогда бы он был бы виден изнутри этого класса. А что-то новое – protected. Чтобы понять что это значит и в чем отличие от других нам нужно найти применение нашему классу.

Итак, что мы имеем на данный момент – мы написали абстрактный класс для фигуры, у него написали 2 метода для вычисления периметра и площади. Хотя стоп. Мы же знаем как наш метод высчитывает периметр у фигуры, верно? Он просто складывает все стороны. Да, но для круга это не так. Ну и пусть. Мы напишем общую логику. А там где будет отличаться мы перезапишем, ровно так же как повторно инициализировали переменные. Да, для методов повторная инициализация тоже есть, но она называется иначе – переопределение. Вы поймете почему так. Давайте уберем слово abstract у метода getPerimeter и напишем дефолтную реализацию.

```
public abstract class Figure {

    private final double[] sides;

    protected Figure(double[] sides) {
        this.sides = sides;
    }

    public abstract double getArea();

    public double getPerimeter() {
        double perimeter = 0;
        for (double side : sides) {
            perimeter += side;
        }
        return perimeter;
    }
}
```

Так, стоп. Мы написали метод у класса, обьект которого нельзя создать? Ну да. А как же вызвать его если он не статик? Действительно, никак.

И теперь мы перейдем к самой теме нашей лекции – наследование. В реальной жизни мы встречаем это слово в контексте – унаследовать от предков что-то. Любой человек по умолчанию наследует от родителей их фамилию, группу крови (не уверен) и еще какие-то вещи. Когда человек умирает то нажитое добро переходит по наследству детям или родственникам. То же самое и в программировании.

У нас есть родительский абстрактный класс геометрической фигуры. У нее есть поле сторон, сколько неважно и методы расчета периметра и площади. У наследников этого класса тоже будут стороны, у кого-то 1, у кого-то 2, у кого-то 3 и т.д. и у всех конечно же будет метод расчета периметра, который в общем случае находится через сумму этих сторон и площадь, которую рассчитывать в каждом случае по-своему.

Ладно, мы написали наши классы круга, прямоугольника и треугольника и абстрактный класс для геометрической фигуры. Как в итоге указать наследование? Для этого в языке Java есть ключевое слово extends которое нужно писать после имени класса. Важный пункт нужно сразу сказать – у любого класса может быть лишь 1 родительский класс. Ровно так же как у любого человека 1 мать.

Давайте посмотрим на наши классы после наследования. Я сразу напишу итог, но вы можете сами редактировать их.

```
public class Circle extends Figure {

    public Circle(double radius) {
        super(new double[]{radius});
    }

    public double getPerimeter() {
        return 2 * 3.14 * radius;
    }

    public double getArea() {
        return 3.14 * radius * radius;
    }
}
```

Стоп, мы же сказали, что у нас поля класса будут храниться в родительском абстрактном классе, а как же мне тогда их использовать в наследнике? Вспомним, что мы их по привычке обьявили private, что означало видимость внутри этого класса. А конструктор был обьявлен protected и как видим мы его вызываем в конструкторе круга через ключевое слово super(). Но я могу в конструктор моего круга получать по-прежнему 1 число и передать в родительский класс массив, состоящий из 1 элемента.

Запомните, если вы отнаследовались от абстрактного класса и у него есть конструктор, то в конструкторе вашего наследника нужно вызвать конструктор родителя в первой же строке. Вы можете это проверить написав код в первой линии после обьявления конструктора. Итак, у нас не видна переменная массива, что же делать? Поменять ей модификатор доступа.

```java
public abstract class Figure {

    protected final double[] sides;

    protected Figure(double[] sides) {
        this.sides = sides;
    }
```

И теперь в классе круга мы можем использовать переменную для сторон для вычисления наших методов.

```java
public class Circle extends Figure {

    public Circle(double radius) {
        super(new double[]{radius});
    }

    @Override
    public double getPerimeter() {
        return 2 * 3.14 * sides[0];
    }

    @Override
    public double getArea() {
        return 3.14 * sides[0] * sides[0];
    }
}
```

Итак, мы используем массив для хранения сторон как будто он есть в классе круга. Это и есть наследование. Переменная хранится в абстрактном классе, но видна в наследнике.

И у нас появилось нечто новое – видите? @Override это что такое? Мы говорили о том, что у нас есть метод для расчета периметра, но он не годится для всех наследников и было бы неплохо повторно инициализировать его, т.е. переопределить. Это аннотация, она указывает что метод был переопределен. Т.е. он обьявлен в родительском классе, но был переопределен в наследнике. И когда будет вызываться метод, то при переопределении будет вызываться именно реализация в наследнике. Мы сможем это проверить простым дебагером.

Давайте теперь посмотрим что произойдет с классом прямоугольника при наследовании.

```java
public class Rectangle extends Figure {

    public Rectangle(double a, double b) {
        super(new double[]{a, b});
    }

    @Override
    public double getArea() {
        return sides[0] * sides[1];
    }

    @Override
    public double getPerimeter() {
        return 2 * super.getPerimeter();
    }
}
```

Так, мы в конструкторе получаем 2 стороны и передаем массив родительскому классу через super. Для площади берем все так же эти 2 стороны и перемножаем друг на друга. Ок. А что с периметром? Мы помним, что там написали метод, который просто суммирует все стороны. Но мы не хотим передавать в массив 4 числа, ведь они попарно равны. Мы передали 2 числа и метод родителя будет суммировать а и б, но периметр же в 2 раза больше. Поэтому я бы мог рассчитать периметр сначала в родительском классе (super.getPerimeter()) и после умножить на 2. Или же самый простой способ в конструкторе передать массив super(new double[]{a, b, a, b}) и не переопределять метод расчета периметра. Попробуйте сами и увидите что разницы нет. На вкус и цвет конечно. Единственная разница это хранения массива на 4 числа или на 2.

Ладно, а что же с треугольником тогда?

```
public class Triangle extends Figure {

    public Triangle(double a, double b, double c) {
        super(new double[]{a, b, c});
    }

    @Override
    public double getArea() {
        double p = (sides[0] + sides[1] + sides[2]) / 2;
        return Math.sqrt(p * (p - sides[0]) * (p - sides[1]) * (p - sides[2]));
    }
}
```

Так, мы получаем в конструкторе все так же 3 числа и отдаем их родительскому классу. Рассчитываем площадь по нашей формуле. А где же метод расчета периметра? Он в родительском классе и он нас устраивает, его не нужно переопределять. Когда вы вызовете метод расчета периметра у обьекта треугольник, то компилятор посмотрит нет ли его в классе треугольника и пойдет в родительский класс. Можно проверить дебагером.

Ладно, мы отнаследовали 3 класса от другого класса. И где же профит? По крайней мере в том, что для 2 классов мы не написали одинаковый метод для расчета периметра (если вы все же решили передать в классе прямоугольника 4 стороны). И это не все. Давайте посмотрим на деле как можно использовать это все.

4. Полиморфизм

В методе мейн я напишу массив из … фигур. Стоп, что? Да, мы не могли создать обьект класса фигура, но мы можем создать массив из фигур. Как? Легко. Посмотрим на то, чем мы заполняем их. В первый элемент я запишу круг, во второй прямоугольник, а в третий треугольник. Классно, да? Но не это самое крутое. Далее я беру массив фигур и циклом прохожу по каждому и вызываю у них методы расчета площади и периметра. Это называется полиморфизм. Так как все эти классы наследовались от одного, то я могу вызвать у всех у них одни и те же методы не зная совсем какие конкретно реализации у них. т.е. я бы мог создать массив кругов и вызывать у них метод расчета периметра. А теперь у меня массив как бы неодинаковых обьектов. Ведь вспомните, массив данных был конкретного одного типа. Так? Мы же писали int[] array = new int[5]; и тем самым обьявляли массив чисел. А здесь у нас в одном массиве 3 разных обьекта. Как же так? А все потому что у них у всех один общий родитель. И так как методы периметра и площади были обьявлены в классе

фигуры мы можем их вызвать. Единственное что не нравится так это вывод в консоль. Я бы хотел улучшить это. Давайте напишем нечто вроде типа у класса фигуры в которую просто запишем название фигуры. Это нам поможет выводить в консоль более ясную информацию.

```
    public static void main(String[] args) {
        Figure[] figures = new Figure[3];
        figures[0] = new Circle( radius: 2);
        figures[1] = new Rectangle( a: 5, b: 6);
        figures[2] = new Triangle( a: 3, b: 4, c: 5);

        for (Figure figure : figures) {
            print("area: " + figure.getArea() + ", perimeter: " + figure.getPerimeter());
        }
    }

    private static void print(String text) {
        System.out.println(text);
    }
}
```

Main › print()

un: Main ×

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
area: 12.56, perimeter: 12.56
area: 30.0, perimeter: 22.0
area: 6.0, perimeter: 12.0

Process finished with exit code 0
```

```
public abstract class Figure {

    final String type;

    protected final double[] sides;

    protected Figure(String type, double[] sides) {
        this.type = type;
        this.sides = sides;
    }
```

Я добавил просто поле строкового типа и требую инициализации у наследников в конструкторе.

```
public Circle(double radius) {
    super( type: "Circle", new double[]{radius});
}
```

Для круга передам просто название этого же класса. То же самое для прямоугольника и треугольника.

```
public Rectangle(double a, double b) {
    super( type: "Rectangle", new double[]{a, b});
}
```

```
public Triangle(double a, double b, double c) {
    super( type: "Triangle", new double[]{a, b, c});
}
```

И поменяем вызов в мейне так

```
    public static void main(String[] args) {
        Figure[] figures = new Figure[3];
        figures[0] = new Circle( radius: 2);
        figures[1] = new Rectangle( a: 5, b: 6);
        figures[2] = new Triangle( a: 3, b: 4, c: 5);

        for (Figure figure : figures) {
            print(figure.type + " - area: " + figure.getArea() +
                    ", perimeter: " + figure.getPerimeter());
        }
    }

    private static void print(String text) {
        System.out.println(text);
    }
}
```

Main › main()

Main ×

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
Circle - area: 12.56, perimeter: 12.56
Rectangle - area: 30.0, perimeter: 22.0
Triangle - area: 6.0, perimeter: 12.0

Process finished with exit code 0
```

Теперь мы сразу видим какая фигура и значения ее площади и периметра.

В заключении скажу – постарайтесь понять эти принципы ООП ибо мы их будем применять каждый день. Если у вас возникают вопросы пишите в чаты, ищите статьи в гугле, старайтесь реально понять что к чему.

В следующей лекции мы будем иметь дело с этим же кодом, улучшим его и немного поменяем.

Удачи!