OOP:

---

1. Определение инкапсуляции, наследования, полиморфизма.

Инкапсуляция, наследование и полиморфизм (и абстракция) - это основные принципы парадигмы объектно-ориентированного программирования.

Инкапсуляция - это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе, и скрыть детали реализации от пользователя. Иными словами, инкапсуляция - это договор для объекта, что он должен скрыть, а что открыть для доступа другим объектам. Это свойство позволяет защитить важные данные для компонента, кроме того, оно дает возможность пользователю не задумываться о сложности реализации используемого компонента, а взаимодействовать с ним посредством предоставляемого интерфейса - публичных данных и методов.

Данный механизм реализуется с помощью модификаторов доступа private, package-private, protected, public:

- public - доступ к компоненту из экземпляра любого класса и любого пакета;
- protected - доступ к компоненту из экземпляра родного класса или классов-потомков, а также внутри пакета;
- package-private - доступ к компоненту только внутри пакета;
- private - доступ к компоненту только внутри класса.

```
public class Car {
    private double engineLiters = 1.5;
    private String color = "orange";

    public void changeColor(String newColor) {
        color = newColor;
    }

    private void disassembleEngine() {
        System.out.println("Engine is disassembled");
    }
}
```

Пользователь класса Car может перекрасить автомобиль, но разобрать двигатель он не может - эта функциональность от него скрыта.

Плюсы:

- Полный контроль над данными класса
- Достижение модульного построения кода
- Упрощение поддержки программы

Простой пример из жизни - современный автомобиль. Водителю совсем не обязательно знать, какие процессы происходят в двигателе при движении, коробке передач при переключении скоростей, и в рулевой тяге при повороте руля. Более того, владельцам автомобиля с коробкой автомат даже не нужно переключать передачи! То есть, все сложное устройство (реализация) системы "автомобиль" инкапсулировано в интерфейс, состоящий из двух педалей, руля и регулятора громкости на магнитоле. А этим простым интерфейсом может пользоваться любой человек, не боясь сломать сложную внутреннюю систему.

Наследование - это свойство системы, позволяющее описать новый класс на основе уже существующего класса, при этом функциональность может быть заимствована частично или полностью. Класс, от которого производится наследование, называется базовым или суперклассом, а новый класс называется потомком, дочерним или производным классом. Производный класс **расширяет** функциональность базового класса благодаря добавлению новой функциональности, специфической только для производного класса.

Классический пример наследования это иерархия геометрических фигур:

Класс Shape является базовым классом, а классы Circle и Square - производными классами. Они наследуют функциональность базового класса, и добавляют свою собственную:

```
public class Shape {
    protected String color = "white";

    public void setColor(String color) {
        this.color = color;
    }

    public void printMe() {
        System.out.println("I'm a Shape, color is " + color);
    }
}


public class Circle extends Shape {
    private double radius;

    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override
    public void printMe() {
        System.out.println("I'm a Circle, color is " + color + ", radius is " + radius);
    }
}


public class Square extends Shape {
    private int sideLength;

    public void setSideLength(int sideLength) {
        this.sideLength = sideLength;
    }

    @Override
    public void printMe() {
        System.out.println("I'm a Square, color is " + color + ", side is " + sideLength);
    }
}


public class Main {
    public static void main(String[] args) {
        Shape shape = new Shape();
        shape.printMe();

        Circle circle = new Circle();
        circle.setColor("red");
        circle.setRadius(4.2);
        circle.printMe();
```

```
        Square square = new Square();
        square.setSideLength(5);
        square.setColor("green");
        square.printMe();
    }
}
```

/*Output:
I'm a Shape, color is white
I'm a Circle, color is red, radius is 4.2
I'm a Square, color is green, side is 5

Методы setRadius и setSideLength принадлежат сугубо производным классам, так как базовый класс не может решить соответствующих задач. Метод *setColor* реализован только в базовом классе, и эта функциональность полностью наследуется производными классами. А вот метод printMe реализован в базовом классе и переопределен в производных классах - таким образом мы получим разное поведение для каждого объекта.

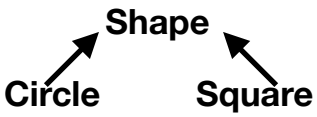
Плюсы:
- Отсутствие дублирования кода
- Достижение модульного построения кода
- Иерархическая структура программы

Минусы:
- Сильная связанность: производный класс зависит от реализации базового класса.

Простой пример из жизни - новая модификация автомобиля. Если в предыдущей модификации были доступны опции поворота руля и торможения, то в новой версии эти опции тоже будут доступны, только к ним добавятся еще противотуманные фары и магнитола. То есть, новая модификация получит функциональность предыдущей и свою собственную. Старая модификация же функциональностью новой обладать не будет.

Полиморфизм - это свойство системы, позволяющее использовать один и тот же интерфейс для общего класса действий. При этом каждое действие зависит от конкретного объекта. То есть полиморфизм - это способность выбирать более конкретные методы для исполнения в зависимости от типа объекта.

Рассмотрим предыдущий пример:




```
public class Shape {
    protected String color = "white";

    public void printMe() {
        System.out.println("I'm a Shape");
    }
}


public class Circle extends Shape {
    private double radius;

    @Override
    public void printMe() {
        System.out.println("I'm a Circle");
    }
}
```

```
public class Square extends Shape {
    private int sideLength;

    @Override
    public void printMe() {
        System.out.println("I'm a Square");
    }
}
```

Один и тот же метод *printMe* будет выполнять разный код в зависимости от того, для какого объекта этот метод вызывается:

```
public class Main {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        for (int i = 0; i < shapes.length; i++) {
            shapes[i] = nextShape();
        }
        for (Shape shape : shapes) {
            shape.printMe();
        }
    }

    private static Shape nextShape() {
        Random random = new Random();
        switch (random.nextInt(3)) {
            case 0:
                return new Circle();
            case 1:
                return new Square();
            default:
                return new Shape();
        }
    }
}
```

```
/* Output:
I'm a Circle
I'm a Square
I'm a Shape
I'm a Circle
I'm a Square
```

В методе main мы создали массив ссылок типа Shape, а затем заполнили его ссылками на случайно созданные объекты разных типов. После этого последовательно вызвали метод printShape, в который передали ссылки из массива shapes. На момент написания программы и на момент компиляции неизвестно, на объект какого типа будет указывать очередная ссылка в массиве. Однако при вызове метода printMe для каждого элемента массива будет вызван метод того объекта, на который реально указывает ссылка. Это и есть пример динамического полиморфизма, реализованный с помощью механизма позднего или динамического связывания.

Удобство полиморфизма заключается в том, что код, работая с различными классами, не должен знать, какой именно класс он использует, так как все они работают по одному принципу.

Плюсы:

- Обеспечение слабосвязанного кода
- Удобство реализации и ускорение разработки
- Отсутствие дублирования кода

Простой пример из жизни - обучаясь вождению, человек может и не знать, каким именно автомобилем ему придется управлять после обучения. Однако это совершенно не важно, потому что основные и доступные для пользователя части автомобиля (интерфейс) устроены по одному и тому же принципу: педаль газа находится справа от педали тормоза, руль имеет форму круга и используется для поворота автомобиля.

---

2. Определение раннего и позднего связывания, механизма переопределения методов.

Присоединение метода к телу метода называется связыванием.

Если связывание производится перед запуском программы (например, компилятором), то оно называется ранним связыванием. Тело методы выбирается в зависимости от типа ссылки. Оно применяется при вызове static и final методов (private методы по умолчанию является final): статические методы существуют на уровне класса, а не на уровне экземпляра класса, а final методы нельзя переопределять.

Если связывание производится во время работы программы, то оно называется поздним или динамическим связыванием. Тело метода выбирается в зависимости от фактического типа объекта.

```
public class Shape {
    protected String color = "white";

    public void printMe() {
        System.out.println("I'm a Shape");
    }

    public static void testStatic() {
        System.out.println("Shape.testStatic");
    }
}


public class Circle extends Shape {
    private double radius;

    @Override
    public void printMe() {
        System.out.println("I'm a Circle");
    }

    public static void testStatic() {
        System.out.println("Circle.testStatic");
    }
}


public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.printMe();
        shape.testStatic();
    }
}
```

/* Output:
I'm a Circle

Shape.testStatic

При вызове метода testStatic был задействован код класса Shape - того класса, какого типа ссылка использовалась для вызова. Это ранее связывание.

При вызове метода printMe был задействован код класса Circle, не смотря на то, что он был вызван для ссылки типа Shape. Это позднее связывание.

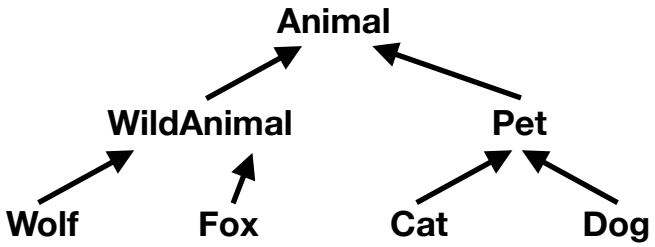
Переопределение метода (Overriding) - это создание в производном класса метода, который полностью совпадает по имени и сигнатуре с методом базового класса. Переопределение используется для изменения поведения метода базового класса в производных классах. Для переопределения методов используется аннотация @Override, которая сообщает компилятору о попытке переопределись метод, а компилятор, в свою очередь, проверяет, правильно ли метод переопределен. Когда предопределенный метод вызывается из своего производного класса, он всегда ссылается на свой вариант. Если же требуется получить доступ к методу базового класса, то используется ключевое слово super.

---

3. Приведение типов при наследовании.

Приведение типов - это преобразование значения переменной одного типа в значение другого типа.

Рассмотрим следующую иерархию:



Класс Cat является производным от класса Pet, который, в свою очередь является производным от класса Animal. Таким образом, такая запись будет верна:

```
Animal animalCat = new Cat();
Animal animalDog = new Dog();
```

Пример выше - это расширяющее (или неявное) приведение: ссылки animalCat и animalDog указывают на объекты Cat и Dog. Через эти ссылки можно получить доступ к любым методам, которые есть в классе Animal, но нельзя получить доступ к специфическим методам классов Cat и Dog.

Сужающее (явное) приведение происходит в другую сторону:

```
Cat cat = (Cat) animalCat;
Dog dog = (Dog) animalDog;
```

Сужающее приведение не всегда возможно, при этом компилятор не укажет на ошибку, но в RunTime будет сгенерирован ClassCastException. Такая ситуация возникнет, например, при преобразовании Cat в Dog:

```
Dog dog = (Dog) animalCat;
```

Для того, чтобы избежать exception, необходимо использовать instanceof:

```
if (animalCat instance Dog) {
        Dog dog = (Dog) animalCat;
}
```

Если объект, на который указывает ссылка animalCat, не является объектом класса Dog, то преобразование не произойдет и exception сгенерирован не будет.

Расширяющее преобразование используется вместе с динамическим связыванием - мы можем создать массив ссылок на Animal, которые будут указывать на разных животных в иерархии, а затем для каждой ссылки вызовем метод sleep, который определен в базовом классе. При этом, мы не сможем для такой ссылки вызвать метод eatRabbit, определенный в классе Fox, чтобы не заставить кота есть кроликов.

4. Наследование: преимущества и недостатки, альтернатива.

Наследование - это свойство системы, позволяющее описать новый класс на основе уже существующего класса, при этом функциональность может быть заимствована частично или полностью. Класс, от которого производится наследование, называется базовым или суперклассом, а новый класс называется потомком, дочерним или производным классом. Производный класс **расширяет** функциональность базового класса благодаря добавлению новой функциональности, специфической только для производного класса. Когда один класс наследует другой, то производный класс расширяет базовый.

Главное преимущество наследования - это предотвращение дублирования кода. Общее поведение для группы классов абстрагируется и помещается в один базовый класс. Если необходимо будет внести какие-то изменения в функциональность всей иерархии, то придется это сделать только один раз - в базовом классе. Таким образом облегчается сопровождение программы и увеличивается скорость разработки. Благодаря наследованию мы получаем преимущества полиморфизма, а это значит, что появляется возможность писать универсальный и чистый код, который не только быстро писать, но и легко масштабировать.

Главный, но не единственный, недостаток наследования - это сильная связанность кода: каждый производный класс зависит от реализации базового класса, а каждое изменение базового класса затронет любой производный класс. Более того, часто требуется совмещать в объекте поведение, характерное для двух и более независимых иерархий. В некоторых языках программирования эта возможность реализована за счет множественного наследования, однако в Java множественное наследование запрещено по разным причинам, например, для избежания так называемого ромбовидного наследования. Альтернативой множественного наследования в Java являются интерфейсы. Интерфейсы - это классы, в которых реализация методов не представлена, то есть все методы абстрактные. Класс в Java может реализовывать (implements) произвольное число интерфейсов, а проблема дублирования одноименных методов (по одному от каждого родителя) отсутствует, так как в интерфейсах методы не реализованы.

Другая особенность наследования заключается в том, что оно относится к поведению объектов класса - наследники должны быть устроены так, чтобы отличия в их внутреннем устройстве никак не влияло на абстракцию их поведения. Внутренне устройство подчеркивается в таком механизме, как композиция. Композиция (или агрегация) - это описание объекта как состоящего из других объектов. И если наследование характеризуется отношением "is-a", то композиция - "has-a". Композиция позволяется объединить отдельные части в единую, более сложную систему. Композиция во многих случаях может служить альтернативой множественному наследованию, причем тогда, когда необходимо унаследовать от двух и более классов их поля и методы. Кроме того, композиция позволяет повторно использовать код даже из final класса.

5. Инкапсуляция.

Инкапсуляция - это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе, и скрыть детали реализации от пользователя. Иными словами, инкапсуляция - это договор для объекта, что он должен скрыть, а что открыть для доступа другим объектам. Это свойство позволяет защитить важные данные для компонента, кроме того, оно дает возможность пользователю не задумываться о сложности реализации используемого компонента, а взаимодействовать с ним посредством предоставляемого интерфейса - публичных данных и методов.

Данный механизм реализуется с помощью модификаторов доступа private, package-private, protected, public:

- public - доступ к компоненту из экземпляра любого класса и любого пакета;
- protected - доступ к компоненту из экземпляра родного класса или классов-потомков, а также внутри пакета;
- package-private - доступ к компоненту только внутри пакета;
- private - доступ к компоненту только внутри класса.

```
public class Car {
    private double engineLiters = 1.5;
    private String color = "orange";
```

```
    public void changeColor(String newColor) {
        color = newColor;
    }

    private void disassembleEngine() {
        System.out.println("Engine is disassembled");
    }
}
```

Пользователь класса Car может перекрасить автомобиль, но разобрать двигатель он не может - эта функциональность от него скрыта.

Плюсы:
- Полный контроль над данными класса
- Достижение модульного построения кода
- Упрощение поддержки программы

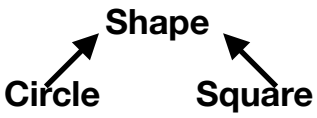
Простой пример из жизни - современный автомобиль. Водителю совсем не обязательно знать, какие процессы происходят в двигателе при движении, коробке передач при переключении скоростей, и в рулевой тяге при повороте руля. Более того, владельцам автомобиля с коробкой автомат даже не нужно переключать передачи! То есть, все сложное устройство (реализация) системы "автомобиль" инкапсулировано в интерфейс, состоящий из двух педалей, руля и регулятора громкости на магнитоле. А этим простым интерфейсом может пользоваться любой человек, не боясь сломать сложную внутреннюю систему.

---

6. Полиморфизм, механизмы Java, его обеспечивающие.

Полиморфизм - это свойство системы, позволяющее использовать один и тот же интерфейс для общего класса действий. При этом каждое действие зависит от конкретного объекта. То есть полиморфизм - это способность выбирать более конкретные методы для исполнения в зависимости от типа объекта.

Рассмотрим предыдущий пример:




```
public class Shape {
    protected String color = "white";

    public void printMe() {
        System.out.println("I'm a Shape");
    }
}


public class Circle extends Shape {
    private double radius;

    @Override
    public void printMe() {
        System.out.println("I'm a Circle");
    }
}


public class Square extends Shape {
    private int sideLength;
```

```
    @Override
    public void printMe() {
        System.out.println("I'm a Square");
    }
}
```

Один и тот же метод *printMe* будет выполнять разный код в зависимости от того, для какого объекта этот метод вызывается:

```
public class Main {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        for (int i = 0; i < shapes.length; i++) {
            shapes[i] = nextShape();
        }
        for (Shape shape : shapes) {
            shape.printMe();
        }
    }

    private static Shape nextShape() {
        Random random = new Random();
        switch (random.nextInt(3)) {
            case 0:
                return new Circle();
            case 1:
                return new Square();
            default:
                return new Shape();
        }
    }
}
```

```
/* Output:
I'm a Circle
I'm a Square
I'm a Shape
I'm a Circle
I'm a Square
```

В методе main мы создали массив ссылок типа Shape, а затем заполнили его ссылками на случайно созданные объекты разных типов. После этого последовательно вызвали метод printShape, в который передали ссылки из массива shapes. На момент написания программы и на момент компиляции неизвестно, на объект какого типа будет указывать очередная ссылка в массиве. Однако при вызове метода printMe для каждого элемента массива будет вызван метод того объекта, на который реально указывает ссылка. Это и есть пример динамического полиморфизма, реализованный с помощью механизма позднего или динамического связывания (связывание производится во время работы программы). Тело метода выбирается в зависимости от фактического типа объекта.

Если связывание производится перед запуском программы (например, компилятором), то оно называется ранним связыванием. Тело методы выбирается в зависимости от типа ссылки. Оно применяется при вызове static и final методов (private методы по умолчанию является final): статические методы существуют на уровне класса, а не на уровне экземпляра класса, а final методы нельзя переопределять.

```
public class Shape {
    protected String color = "white";
```

```
    public static void testStatic() {
        System.out.println("Shape.testStatic");
    }
}


public class Circle extends Shape {
    private double radius;

    public static void testStatic() {
        System.out.println("Circle.testStatic");
    }
}


public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.testStatic();
    }
}
```

/* Output:
Shape.testStatic

При вызове метода testStatic был задействован код класса Shape - того класса, какого типа ссылка использовалась для вызова. Это ранее связывание.

С понятием полиморфизма тесно связано переопределение метода (Overriding) - это создание в производном класса метода, который полностью совпадает по имени и сигнатуре с методом базового класса. Переопределение необходимо для работы механизма позднего связывания и используется для изменения поведения метода базового класса в производных классах. Для переопределения методов используется аннотация @Override, которая сообщает компилятору о попытке переопределись метод, а компилятор, в свою очередь, проверяет, правильно ли метод переопределен. Когда предопределенный метод вызывается из своего производного класса, он всегда ссылается на свой вариант. Если же требуется получить доступ к методу базового класса, то используется ключевое слово super:

Удобство полиморфизма заключается в том, что код, работая с различными классами, не должен знать, какой именно класс он использует, так как все они работают по одному принципу.

Плюсы:
- Обеспечение слабосвязанного кода
- Удобство реализации и ускорение разработки
- Отсутствие дублирования кода

Простой пример из жизни - обучаясь вождению, человек может и не знать, каким именно автомобилем ему придется управлять после обучения. Однако это совершенно не важно, потому что основные и доступные для пользователя части автомобиля (интерфейс) устроены по одному и тому же принципу: педаль газа находится справа от педали тормоза, руль имеет форму круга и используется для поворота автомобиля.

---

7. Модификаторы доступа.

Java обеспечивает контроль доступа через модификаторы доступа, таким образом реализуется принцип инкапсуляции:
- public - доступ к компоненту из экземпляра любого класса и любого пакета;

- protected - доступ к компоненту из экземпляра родного класса или классов-потомков, а также внутри пакета;
- default (package-private) - доступ к компоненту только внутри пакета;
- private - доступ к компоненту только внутри класса.

Модификаторы доступа могут использоваться с классами, переменными и методами.

С классом могут быть использованы только модификаторы public и default: доступ к public классу можно получить из любого другого класса, в исходном файле может существовать только один public класс, а имя исходного файла должно совпадать с именем public класса.

Внутри класса, то есть с переменными-членами и методами класса могут использоваться все 4 модификатора, однако они не должны быть более доступны, чем сам класс.

ПРО ДРУГИЕ МОДИФИКАТОРЫ, ТИПА FINAL, STATIC и тд?
http://www.quizful.net/post/features-of-the-application-of-modifiers-in-java

---

8. Что такое объектно-ориентированный подход (принципы SOLID).

Объектно-ориентированный подход - это такой способ программирование, который напоминает процесс человеческого мышления. ООП более структурировано, чем другие способы программирования (например, процедурное) и позволяет создавать модульные программы с представление данных на определенном уровне абстракции. Основная цель ООП - это повышение эффективности разработки программ.

Весь окружающий мир состоит из объектов, которые представляются как единое целое, и такие объекты взаимодействуют друг с другом. Базом в ООП является понятие объекта, который имеет определенные свойства. Каждый объект знает как решать определенные задачи, то есть располагает методами решения. Программа, написанная с использованием ООП, состоит из объектов, которые могут взаимодействовать друг с другом. Все объекты с одинаковыми наборами атрибутов принадлежат к одному классу. Объединение объектов в классы определяется семантикой, то есть смыслом. Каждый класс имеет свои особенности поведения, которые определяют этот класс. Один класс отличается от другого именем и набором "сообщений", которые можно посылать объектам данных классов (интерфейсом).

Характеристики ООП:

- все является объектом
- объекты взаимодействуют между собой путем обмена сообщениями, при котором один объект требует, чтобы другой объект выполнил некоторое действие.
- каждый объект является представителем класса, который выражает общие свойства объектов.
- в классе задается функциональность (поведение) объекта, а все объекты одного класса могут выполнять одни и те же действия.
- классы организованы в иерархическую структуру.

ООП основывается на 4х принципах:

Абстракция - выделение некоторых существенных характеристик объекта, которые выделяют его из всех других объектов. Абстракция позволяет отделить существенные особенности поведения объекта от деталей их реализации. Например, главной характеристикой объекта "директор" будет то, что этот объект чем-то управляет, а чем и как именно (финансами или персоналом), то есть детали реализации, являются второстепенной информацией.

Инкапсуляция - это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе, и скрыть детали реализации от пользователя. Иными словами, инкапсуляция - это договор для объекта, что он должен скрыть, а что открыть для доступа другим объектам. Это свойство позволяет защитить важные данные для компонента, кроме того, оно дает возможность пользователю не задумываться о сложности реализации используемого компонента, а взаимодействовать с ним посредством предоставляемого интерфейса - публичных данных и методов.

Наследование - это свойство системы, позволяющее описать новый класс на основе уже существующего класса, при этом функциональность может быть заимствована частично

или полностью. Класс, от которого производится наследование, называется базовым или суперклассом, а новый класс называется потомком, дочерним или производным классом. Производный класс **расширяет** функциональность базового класса благодаря добавлению новой функциональности, специфической только для производного класса.
Полиморфизм - это свойство системы, позволяющее использовать один и тот же интерфейс для общего класса действий. При этом каждое действие зависит от конкретного объекта. То есть полиморфизм - это способность выбирать более конкретные методы для исполнения в зависимости от типа объекта.

При обсуждении принципов ООП следует упомянуть 5 принципов SOLID:

1. Единственность ответственности (Single Responsibility)
2. Принцип открытости и закрытости (Open/Close Principle)
3. Принцип замещения Лисков (The Liskov Substitution Principle)
4. Принцип разделения интерфейса (The Interface Segregation Principle)
5. Инверсия зависимости (The Dependency Inversion Principle)

http://info.javarush.ru/translation/2013/08/06/Пять-основных-принципов-дизайна-классов-S-O-L-I-D-в-Java.html