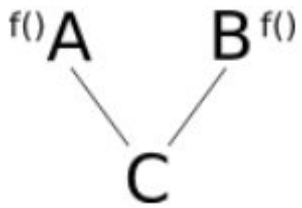
Лабораторная работа №5. Наследование и полиморфизм.

Наследование.

Наследование – один из трех важнейших механизмов объектно ориентированного программирования (наряду с инкапсуляцией и полиморфизмом), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом. Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса. Класс, от которого произошло наследование, называется базовым или родительским (англ. base class). Классы, которые произошли от базового, называются потомками, наследниками или производными классами (англ. derived class). В некоторых языках используются абстрактные классы. Абстрактный класс – это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного класса. Например, абстрактным классом может быть базовый класс «сотрудник вуза», от которого наследуются классы «аспирант», «профессор» и т. д. Так как производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза»

Множественное наследование. При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости. Множественное наследование реализовано в C++. Из других языков, предоставляющих эту возможность, можно отметить Python и Эйфель. Множественное наследование поддерживается в языке UML. Множественное наследование – потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках (рис. 1).



В языке Java поддерживается только простое наследование: любой подкласс является производным только от одного непосредственного суперкласса. При этом любой класс может наследоваться от нескольких интерфейсов. Наследование интерфейсов реализует некоторую замену множественному наследованию, когда вместо того чтобы один класс имел несколько непосредственных суперклассов, этот класс наследует несколько интерфейсов. Интерфейс представляет собой класс, в котором все свойства – константы (т.е. статические – static и неизменяемые – final), а все методы абстрактные, т.е. интерфейс позволяет определить некоторый шаблон класса: описание свойств и методов без их реализации.

Язык Java разрешает несколько уровней наследования, определяемых непосредственным суперклассом и косвенными суперклассами. Наследование можно использовать для создания иерархии классов.

При создании подкласса на основе одного или нескольких суперклассов возможны следующие способы изменения поведения и структуры класса: – расширение суперкласса путем добавления новых данных и методов; – замена методов суперкласса путем их переопределения; – слияние методов из суперклассов вызовом одноименных методов из соответствующих суперклассов

Все классы в Java неявно наследуют свойства и поведения класса Object, т.е он является предком всех классов, которые есть в JDK (Java Development Kit – комплект разработки на языке Java) и котороые будут создаваться

пользователем. Например: класс Person в листинге 1 неявно наследует свойства Object. Так как это предполагается для каждого класса, не нужно вводить фразу extends Object для каждого определяемого класса. Таким образом, класс Person имеет доступ к представленным переменным и методам своего суперкласса (Object). В данном случае Person может «видеть» и использовать общедоступные методы и переменные объекта Object, а также его защищенные методы и переменные.

Определим новый класс Employee, который наследует свойства Person. Его определение класса (или граф наследования) будет выглядеть следующим образом:

```
public class Employee extends Person {
        private String taxpayerIdentificationNumber;
        private String employeeNumber;
        private int salary;
// . . . }
```

Граф наследования для Employee указывает на то, что Employee имеет доступ ко всем общедоступным и защищенным переменным и методам Person (так как он непосредственно расширяет его), а также Object (так как он фактически расширяет и этот класс, хотя и опосредованно).

Для углубления в иерархию классов еще на один шаг, можно создать третий класс, который расширяет Employee:

```
public class Manager extends Employee {}
```

В языке Java любой класс может иметь не более одного суперкласса, но любое количество подклассов. Это самая важная особенность иерархии наследования языка Java, о которой надо помнить.

В Java все методы конструктора используют уточнения при переопределении методов по схеме сцепления конструкторов. В частности, выполнение конструктора начинается с обращения к конструктору суперкласса, которое может быть явным или неявным. Для явного обращения к конструктору используется оператор super, который указывает на вызов суперкласса (например, super() вызывает конструктор суперкласса без аргументов). Если же в теле конструктора явное обращение отсутствует, компилятор автоматически помещает в первой строке конструктора обращение к методу

super(). То есть в языке Java в конструкторах используется уточнение при переопределении методов, а в обычных методах используется замещение.

```
public class Person {
 private String name;
 public Person() {}
        public Person(String name) {
                this.name = name;
 }
}
public class Employee extends Person {
  public Employee(String name) {
 super(name);
 }
}
```

Иногда в классе Java необходимо обратиться к текущему экземпляру данного класса, который в данный момент обрабатывается методом. Для такого обращения используется ключевое слово this. Применение этой конструкции удобно в случае необходимости обращения к полю текущего объекта, имя которого совпадает с именем переменной, описанной в данном блоке

Если например: в базовом и дочернем классах есть методы с одинаковым именем и одинаковыми параметрами. В таком случае уместно говорить о переопределении методов, т.е. в дочернем классе изменяется реализация уже существовавшего в базовом классе метода. Если пометить метод модификатором final, то метод не может быть переопределен. Поля нельзя переопределить, их можно только скрыть.

Если подкласс переопределяет метод из суперкласса, этот метод, по существу, скрыт, потому что вызов этого метода с помощью ссылки на подкласс вызывает версии метода подкласса, а не версию суперкласса. Это не означает, что метод суперкласса становится недоступным. Подкласс может вызвать метод суперкласса, добавив перед именем метода ключевое слово super (и в отличие от правил для конструкторов, это можно сделать в любой строке метода подкласса или даже совсем другого метода). По умолчанию Java-программа вызывает метод подкласса, если он вызывается с помощью ссылки на подкласс .

В контексте ООП абстрагирование означает обобщение данных и поведения до типа, более высокого по иерархии наследования, чем текущий класс. При перемещении переменных или методов из подкласса в суперкласс говорят, что эти члены абстрагируются. Основной причиной этого является возможность многократного использования общего кода путем продвижения его как можно выше по иерархии. Когда общий код собран в одном месте, облегчается его обслуживание.

Бывают моменты, когда нужно создавать классы, которые служат только как абстракции, и создавать их экземпляры никогда не придется. Такие классы называются абстрактными классами. К тому же бывают моменты, когда определенные методы должны быть реализованы по-разному для каждого подкласса, реализуемого суперклассом. Это абстрактные методы. Вот некоторые основные правила для абстрактных классов и методов:

– любой класс может быть объявлен абстрактным;

– абстрактные классы не допускают создания своих экземпляров;

– абстрактный метод не может содержать тела метода;

– класс, содержащий абстрактный метод, должен объявляться как абстрактный.

Например: Необходим метод для изучения состояния объекта Employee и проверки его правомерности. Это требование может показаться общим для всех объектов Employee, но между всеми потенциальными подклассами поведение будет существенно различаться, что дает нулевой потенциал для повторного использования. В этом случае вы объявляете метод validate() как абстрактный (заставляя все подклассы реализовывать его)

Теперь каждый прямой подкласс Employee (такой как Manager) должен реализовать метод validate(). При этом, как только подкласс реализовал метод validate(), ни одному из его подклассов не придется этого делать.

Полиморфизм

Полиморфизм в языках программирования – это возможность объектов с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь раз- личную реализацию – например, реализация класса может быть изменена в процессе наследования. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

В java существуют несколько способов организации полиморфизма:

1) Полиморфизм интерфейсов. Интерфейсы описывают методы, которые должны быть реализованы в классе, и типы параметров, которые должен получать и возвращать каждый член класса, но не содержат определенной реализации методов, оставляя это реализующему интерфейс классу. В этом и заключается полиморфизм интерфейсов. Несколько классов могут реализовывать один и тот же интерфейс, в то же время один класс может реализовывать один или больше интерфейсов. Интерфейсы находятся вне иерархии наследования классов, поэтому они исключают определение метода или набора методов из иерархии наследования.

```
interface Shape {
void draw();
void erase();
}
class Circle  implements Shape {
public void draw() { System.out.println("Circle.draw()");}
public void erase() { System.out.println("Circle.erase()");}
}
class Square implements Shape {
        public void draw() { System.out.println("Square.draw()");}
        public void erase() { System.out.println("Square.erase()");}
}
class Triangle implements Shape {
         public void draw() { System.out.println("Triangle.draw()");}
        public void erase() { System.out.println("Triangle.erase()");}
}
public class Shapes {
        public static Shape randShape() {
        switch((int)(Math.random() * 3)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
          }
        }
        public static void main(String[] args) {
          Shape[] s = new Shape[9];
          for(int i = 0; i < s.length; i++)
            s[i] = randShape();
          for(int i = 0; i < s.length; i++)
            s[i].draw();
        }
}
```

2) Полиморфизм наследования. При наследовании класс получает все методы, свойства и события базового класса такими, какими они реализованы в базовом классе. При необходимости в наследуемых классах можно определять дополнительные члены или переопределять члены, доставшиеся от базового класса, чтобы реализовать их иначе (листинг 6). Наследуемый класс также может реализовывать интерфейсы. В данном случае полиморфизм проявляется в том, что функционал базового класса присутствует в наследуемых классах неявно. Функционал может быть дополнен и переопределен. А наследуемые классы, несущие в себе этот функционал выступают в роли многих форм.

```
class Shape {
void draw() {}
void erase() {}
}
class Circle extends Shape {
void draw() { System.out.println("Circle.draw()");}
void erase() { System.out.println("Circle.erase()");}
}
class Square extends Shape {
void draw() { System.out.println("Square.draw()");}
void erase() { System.out.println("Square.erase()");}
}
class Triangle extends Shape {
void draw() { System.out.println("Triangle.draw()");}
void erase() { System.out.println("Triangle.erase()");}
}
public class Shapes {
public static Shape randShape() {
switch((int)(Math.random() * 3)) {
default:
case 0: return new Circle();
                case 1: return new Square();
                case 2: return new Triangle();
                }
        }
public static void main(String[] args) {
        Shape[] s = new Shape[9];
        for(int i = 0; i < s.length; i++)
                s[i] = randShape();
        for(int i = 0; i < s.length; i++)
                s[i].draw();
        }
}
}
```

3) Полиморфизм при помощи абстрактных классов. Абстрактные классы поддерживают как наследование, так и возможности интерфейсов (листинг 5.4). При построении сложной иерархии, для обеспечения полиморфизма программисты часто вынуждены вводить методы в классы верхнего уровня при том, что эти методы ещё не определены. Абстрактный класс – это класс,

экземпляр которого невозможно создать; этот класс может лишь служить базовым классом при наследовании. Нельзя объявлять абстрактные конструкторы или абстрактные статические методы. Некоторые или все члены этого класса могут оставаться нереализованными, их реализацию должен обеспечить наследующий класс. Производные классы, которые не переопределяют все абстрактные методы, должны быть отмечены как абстрактные. Порожденный класс может реализовывать также дополнительные интерфейсы.

4) Полиморфизм методов. Способность классов поддерживать различные реализации методов с одинаковыми именами – один из способов реализации полиморфизма. Различные реализации методов с одинаковыми именами в Java называется перегрузкой методов (листинг 7). На практике часто приходится реализовывать один и тот же метод для различных типов данных. Право выбора специфической версии метода предоставлено компилятору

Отдельным вариантом полиморфизма методов является полиморфизм методов с переменным числом аргументов, введенный в версии Java 2 5.0. Перегрузка методов здесь предусмотрена неявно, т.е. перегруженный метод может вызываться с разным числом аргументов, а в некоторых случаях даже без параметров. Перегрузка методов как правило делается для тесно связанных по смыслу операций. Ответственность за построение перегруженных методов и выполнения ими однородных по смыслу операций лежит на разработчике.

```
    public class PrintStream extends FilterOutputStream
implements Appendable , Closeable  {
    /*...*/
    public void print(boolean b) {
          write(b ? "true" : "false");
    }
    public void print(char c) {
          write(String.valueOf(c));
    }
    public void print(int i) {
          write(String.valueOf(i));
    }
    public void print(long l) {
          write(String.valueOf(l));
    }
    /*...*/
    public void write(int b) {
    }
    }
```

5) Полиморфизм через переопределение методов. Если перегруженные методы с одинаковыми именами находятся в одном классе, списки параметров должны отличаться. Но если метод подкласса совпадает с методом суперкласса, то метод подкласса переопределяет метод суперкласса (листинг 3). Совпадать при этом должны и имена методов и типы входных параметров. В данном случае переопределение методов является основой концепции динамического связывания (или позднее связывание),

реализующей полиморфизм. Суть динамической диспетчеризации методов состоит в том, что решение на вызов переопределенного метода принимается во время выполнения, а не во время компиляции. Однако final-методы не являются пе- реопределяемыми, их вызов может быть организован во время компиляции и называется ранним связыванием.

В данной лабораторной работе вам необходимо не только разработать программу, но и изобразить её в виде UML диаграммы.

Пример выполнения работы:

Вариант 0.

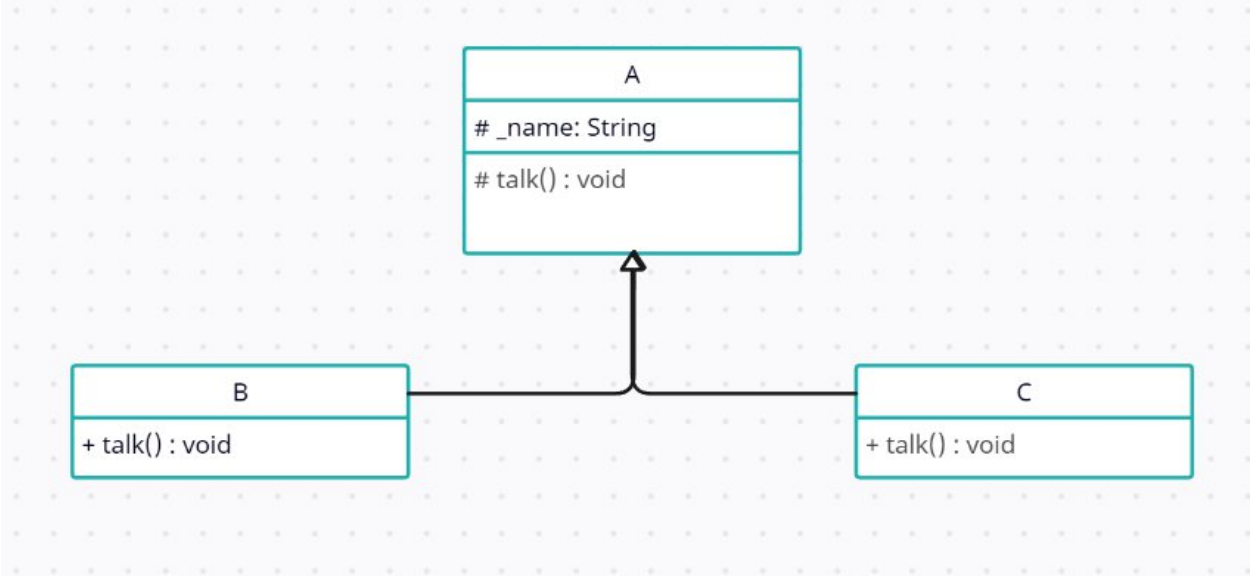
Задание 1.

Вам дан класс A. В нём присутствуют конструктор, абстрактный метод talk() и поле name. Наследуйте классы B, C, в котором переопределите методы.

Решение:

```
public class Main
{
	public static void main(String[] args) {
		B b = new B("Barash");
		C c = new C("Crosh");
		b.talk();
		c.talk();
	}
}

abstract class A {
   protected String _name;
   abstract void talk();
}

class B extends A {
   public B(String name) {
      _name = name;
   }

   @Override
```

```
    void talk() {
        System.out.println("I am B and my name is " + _name);
    }
}

class C extends A {
    public C(String name) {
        _name = name;
    }

    @Override
    void talk() {
        System.out.println("I am C and my name is " + _name);
    }
}
```



Советую чекнуть статью на хабре: https://habr.com/ru/articles/150041/

Задания по вариантам

Задание 1.

Вариант 1: Животные

- **Классы**: Animal (базовый), Dog, Cat, Bird (наследники).
- **Метод**: makeSound(), полиморфно переопределяемый в каждом классе.
- **Диаграмма**: базовый класс Animal с полиморфными методами.

Вариант 2: Геометрические фигуры

- **Классы**: Shape (базовый), Circle, Rectangle, Triangle (наследники).
- **Метод**: calculateArea(), полиморфное вычисление площади для каждой фигуры.
- **Диаграмма**: интерфейс или базовый класс Shape, от которого наследуются конкретные фигуры.

Вариант 3: Транспорт

- **Классы**: Vehicle (базовый), Car, Bicycle, Boat (наследники).
- **Метод**: move(), реализующий различное поведение для каждого вида транспорта.
- **Диаграмма**: абстрактный класс Vehicle, от которого наследуются конкретные типы транспорта.

Вариант 4: Электронные устройства

- **Классы**: Device (базовый), Phone, Laptop, Tablet (наследники).
- **Метод**: turnOn(), полиморфно реализуемый в каждом классе.
- **Диаграмма**: интерфейс Device, с реализацией методов у разных устройств.

Вариант 5: Работники компании

- **Классы**: Employee (базовый), Manager, Developer, Designer (наследники).
- **Метод**: getSalary(), полиморфное вычисление зарплаты для каждого типа сотрудника.
- **Диаграмма**: базовый класс Employee и классы-наследники.

Вариант 6: Учебные курсы

- **Классы**: Course (базовый), MathCourse, ScienceCourse, LiteratureCourse (наследники).

- **Метод**: getCourseInfo(), предоставляющий различную информацию о каждом курсе.
- **Диаграмма**: базовый класс Course, расширяемый специфическими курсами.

Вариант 7: Банк

- **Классы**: Account (базовый), CheckingAccount, SavingsAccount, CreditAccount (наследники).
- **Метод**: calculateInterest(), полиморфное вычисление процентов для каждого типа счета.
- **Диаграмма**: абстрактный класс Account.

Вариант 8: Инструменты

- **Классы**: Tool (базовый), Hammer, Screwdriver, Wrench (наследники).
- **Метод**: use(), различное поведение для каждого инструмента.
- **Диаграмма**: базовый класс Tool.

Вариант 9: Музыкальные инструменты

- **Классы**: Instrument (базовый), Piano, Guitar, Drums (наследники).
- **Метод**: play(), полиморфное поведение для каждого инструмента.
- **Диаграмма**: базовый класс Instrument.

Вариант 10: Обработка файлов

- **Классы**: File (базовый), TextFile, ImageFile, AudioFile (наследники).
- **Метод**: open(), полиморфное открытие файлов разных типов.
- **Диаграмма**: базовый класс File.

Вариант 11: Платежные системы

- **Классы**: Payment (базовый), CreditCardPayment, PayPalPayment, BankTransferPayment (наследники).
- **Метод**: processPayment(), различная реализация обработки платежей.
- **Диаграмма**: интерфейс Payment.

Вариант 12: Игровые персонажи

- **Классы**: GameCharacter (базовый), Warrior, Mage, Archer (наследники).
- **Метод**: attack(), полиморфные действия для атаки у разных классов.
- **Диаграмма**: абстрактный класс GameCharacter.

Вариант 13: Автомобили

- **Классы**: Car (базовый), Sedan, SUV, Coupe (наследники).

- **Метод**: drive(), полиморфное поведение при вождении.
- **Диаграмма**: базовый класс Car.

Вариант 14: Домашние устройства

- **Классы**: HomeDevice (базовый), TV, Refrigerator, WashingMachine (наследники).
- **Метод**: operate(), полиморфное управление для каждого устройства.
- **Диаграмма**: интерфейс HomeDevice.

Вариант 15: Журналы подписки

- **Классы**: Subscription (базовый), DailySubscription, WeeklySubscription, MonthlySubscription (наследники).
- **Метод**: getCost(), полиморфное вычисление стоимости.
- **Диаграмма**: базовый класс Subscription.

Вариант 16: Услуги доставки

- **Классы**: Delivery (базовый), FoodDelivery, PackageDelivery, CourierDelivery (наследники).
- **Метод**: deliver(), полиморфное выполнение доставки.
- **Диаграмма**: базовый класс Delivery.

Задание 2

Вариант 1: Животные

Усложнение: Добавить интерфейс Swimmable, который реализуют классы Dog и Bird, но не Cat. Реализовать метод swim(), полиморфно определяющий способность плавать.

- **Классы**: Animal, Dog, Cat, Bird, Swimmable (интерфейс).
- **Методы**: makeSound(), swim().

Вариант 2: Геометрические фигуры

Усложнение: Добавить интерфейс Resizable, который могут реализовать фигуры с возможностью изменения размера, например Rectangle. В классе Resizable добавить метод resize(), который изменяет размер фигуры.

- **Классы**: Shape, Circle, Rectangle, Triangle, Resizable (интерфейс).

- **Методы**: calculateArea(), resize().

Вариант 3: Транспорт

Усложнение: Ввести абстрактный класс ElectricVehicle, который наследует Vehicle, и добавить классы ElectricCar и ElectricBicycle. Реализовать метод chargeBattery().

- **Классы**: Vehicle, Car, Bicycle, Boat, ElectricVehicle, ElectricCar, ElectricBicycle.
- **Методы**: move(), chargeBattery().

Вариант 4: Электронные устройства

Усложнение: Добавить интерфейс Connectable, который реализуют устройства с возможностью подключения к интернету, такие как Phone и Laptop. Реализовать метод connectToInternet().

- **Классы**: Device, Phone, Laptop, Tablet, Connectable (интерфейс).
- **Методы**: turnOn(), connectToInternet().

Вариант 5: Работники компании

Усложнение: Добавить интерфейс ProjectManager, который реализует только класс Manager. Реализовать метод assignTask(), который будет использоваться для делегирования задач.

- **Классы**: Employee, Manager, Developer, Designer, ProjectManager (интерфейс).
- **Методы**: getSalary(), assignTask().

Вариант 6: Учебные курсы

Усложнение: Добавить абстрактный класс OnlineCourse, от которого наследуют MathCourse и ScienceCourse. Ввести метод joinOnline() для регистрации студентов на онлайн-курсы.

- **Классы**: Course, MathCourse, ScienceCourse, LiteratureCourse, OnlineCourse.
- **Методы**: getCourseInfo(), joinOnline().

Вариант 7: Банк

Усложнение: Добавить интерфейс Loanable, который реализуют классы CheckingAccount и CreditAccount. Реализовать метод applyForLoan().

- **Классы**: Account, CheckingAccount, SavingsAccount, CreditAccount, Loanable (интерфейс).
- **Методы**: calculateInterest(), applyForLoan().

Вариант 8: Инструменты

Усложнение: Добавить интерфейс ElectricTool, который реализуют электрические инструменты, такие как Screwdriver, с методом charge().

- **Классы**: Tool, Hammer, Screwdriver, Wrench, ElectricTool (интерфейс).
- **Методы**: use(), charge().

Вариант 9: Музыкальные инструменты

Усложнение: Добавить класс ElectronicInstrument, который наследует Instrument, и добавить Synthesizer. Реализовать метод adjustVolume().

- **Классы**: Instrument, Piano, Guitar, Drums, ElectronicInstrument, Synthesizer.
- **Методы**: play(), adjustVolume().

Вариант 10: Обработка файлов

Усложнение: Добавить интерфейс Compressible, который реализуют классы TextFile и ImageFile. Реализовать метод compress().

- **Классы**: File, TextFile, ImageFile, AudioFile, Compressible (интерфейс).
- **Методы**: open(), compress().

Вариант 11: Платежные системы

Усложнение: Добавить интерфейс Refundable, который реализуют только классы CreditCardPayment и PayPalPayment. Реализовать метод refund().

- **Классы**: Payment, CreditCardPayment, PayPalPayment, BankTransferPayment, Refundable (интерфейс).
- **Методы**: processPayment(), refund().

Вариант 12: Игровые персонажи

Усложнение: Добавить интерфейс Healable, который реализует только класс Mage. Реализовать метод heal().

- **Классы**: GameCharacter, Warrior, Mage, Archer, Healable (интерфейс).
- **Методы**: attack(), heal().

Вариант 13: Автомобили

Усложнение: Добавить интерфейс SelfDriving, который реализует класс Sedan. Реализовать метод autoPilot().

- **Классы**: Car, Sedan, SUV, Coupe, SelfDriving (интерфейс).
- **Методы**: drive(), autoPilot().

Вариант 14: Домашние устройства

Усложнение: Добавить интерфейс SmartDevice, который реализуют TV и Refrigerator, с методом connectToSmartHome().

- **Классы**: HomeDevice, TV, Refrigerator, WashingMachine, SmartDevice (интерфейс).
- **Методы**: operate(), connectToSmartHome().

Вариант 15: Журналы подписки

Усложнение: Добавить интерфейс Renewable, который реализуют DailySubscription и WeeklySubscription. Реализовать метод renewSubscription().

- **Классы**: Subscription, DailySubscription, WeeklySubscription, MonthlySubscription, Renewable (интерфейс).
- **Методы**: getCost(), renewSubscription().

Вариант 16: Услуги доставки

Усложнение: Добавить интерфейс Trackable, который реализует класс PackageDelivery. Реализовать метод track().

- **Классы**: Delivery, FoodDelivery, PackageDelivery, CourierDelivery, Trackable (интерфейс).
- **Методы**: deliver(), track().