ООП. Вопросы

1. **Объектная модель. Объекты, классы, Жизненный цикл объектов, примеры**
   Объект (object), или объект класса, или экземпляр класса – это адресуемая одним адресом область памяти для хранения значений всех полей класса.
   Любой объект в программе:
   1) создается;
   2) «живет»;
   3) удаляется.
   Согласно приведенному определению, создание объекта – это выделение и инициализация памяти для хранения значений полей объекта, а удаление объекта – это освобождение памяти, используемой для хранения значений полей объекта.
   Если говорить точно, объект является “живым” пока на него есть ссылки. Как только ссылок не остается — объект “умирает”.

   Жизненный цикл объектов класса
   Создание в памяти нового, реального объекта, который создается на базе класса, например, так:
   new Human().
   Но лучше сразу объявить ссылку и передать ей этот объект:
   Human human = new Human();.

   Жизнь /использование.
   Любые манипуляции с объектом, например:
   human.age = 12;
   System.out.println(human.age);

```
trololoshko.name = "Trololoshko";
trololoshko.surname = "SomeSurname";
trololoshko.age = 150;
trololoshko.sex = true;
// обратите внимание на способ доступа к полям!
```

   Сборщик мусора — внутренний механизм Java, который отвечает за освобождение памяти, то есть удаление из нее ненужных объектов. Его работа — удалять объекты, которые уже не используются в программе. Таким образом он освобождает в компьютере память для других объектов.
   Garbage Collector запускается многократно в течение работы твоей программы: его не надо вызывать специально и отдавать команды, хотя технически это возможно.

2. **Основные принципы ООП, примеры**
   Наследование — это передача всех свойств и поведения от одного класса другому, более конкретному. У карася и ерша, как и у всех рыб, есть плавники, хвосты, жабры и чешуя, они живут в воде и плавают.
   Абстракция — это сокрытие подробностей и предоставление пользователю лишь самых важных характеристик объекта. Например, в адресе здания важны такие данные, как почтовый индекс, страна, населенный пункт, улица и номер дома. Его этажность и материал стен в таком случае не имеют значения.
   Инкапсуляция — это размещение данных и методов для их обработки в одном объекте, а также сокрытие деталей его реализации. Мы знаем, как включать и выключать телевизор, переключать программы и регулировать громкость. Для этого не обязательно знать, как он устроен.
   Полиморфизм — это проявление одного поведения разными способами. Животные могут издавать звуки, при этом кошка мяукает, а собака лает.

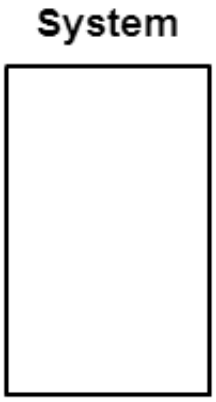
3. **UML: Актеры, прецеденты и сценарии, примеры**

Диаграмма вариантов прецедентов – это тип поведенческой диаграммы UML, который часто используется для анализа различных систем. Они позволяют визуализировать различные типы ролей в системе и то, как эти роли взаимодействуют с системой.
Актер использует диаграмму прецедентов – это любая сущность, которая выполняет роль в одной данной системе. Это может быть человек, организация или внешняя система и обычно рисуется как скелет, показанный ниже.

Actor

Случай использования представляет собой функцию или действие внутри системы. Она нарисована как овал и названа функцией.

Система используется для определения сферы применения и нарисована в виде прямоугольника. Это необязательный элемент, но полезный при визуализации больших систем.



Сценарий - описание поведения системы, когда она взаимодействует с кем-то (или чем-то) из внешней среды.

4. **Классы и взаимодействия, сущности и связи, примеры**

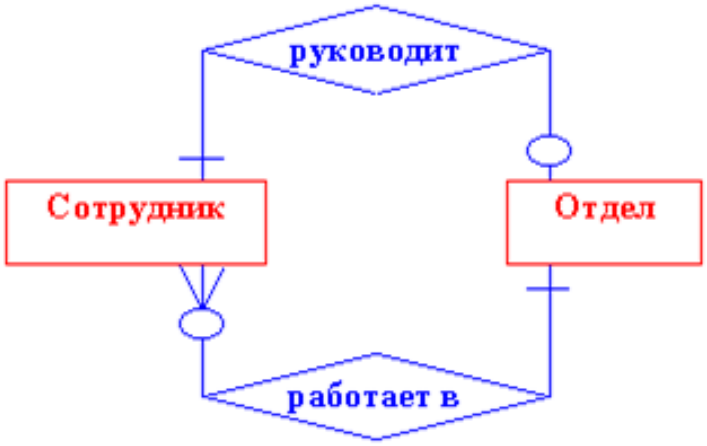
Сущность в базе данных – это любой объект в базе данных, который можно выделить исходя из сути предметной области для которой разрабатывается эта база данных. Разработчик базы данных должен уметь правильно определять сущности.
Связь (relationship) - это ассоциация, установленная между несколькими сущностями.
Набор связей (relationship set) - это отношение между n (причем n не меньше 2) сущностями, каждая из которых относится к некоторому набору сущностей.
Пример:

сущности		наборы сущностей
e1	принадлежит	E1
e2	принадлежит	E2
. . .		
en	принадлежит	En

тогда [e1,e2,...,en] - набор связей R
В случае n=2, т.е. когда связь объединяет две сущности, она называется бинарной.
Степень связи – то число сущностей, которое может быть ассоциировано через набор связей с другой сущностью.

Виды степеней связи:
1. один к одному (1 : 1). Это означает, что в такой связи сущности с одной ролью всегда соответствует не более одной сущности с другой ролью.

2. один ко многим (1 : n). Это означает что сущности с одной ролью может соответствовать любое число сущностей с другой ролью.



3. много к одному (n:1). Этот тип связи аналогичен связям один ко многим.
4. много ко многим (n:n). В этом случае каждая из ассоциированных сущностей может быть представлена любым количеством экземпляров

5. **Объекты контейнеры: назначение, виды, преимущества.**

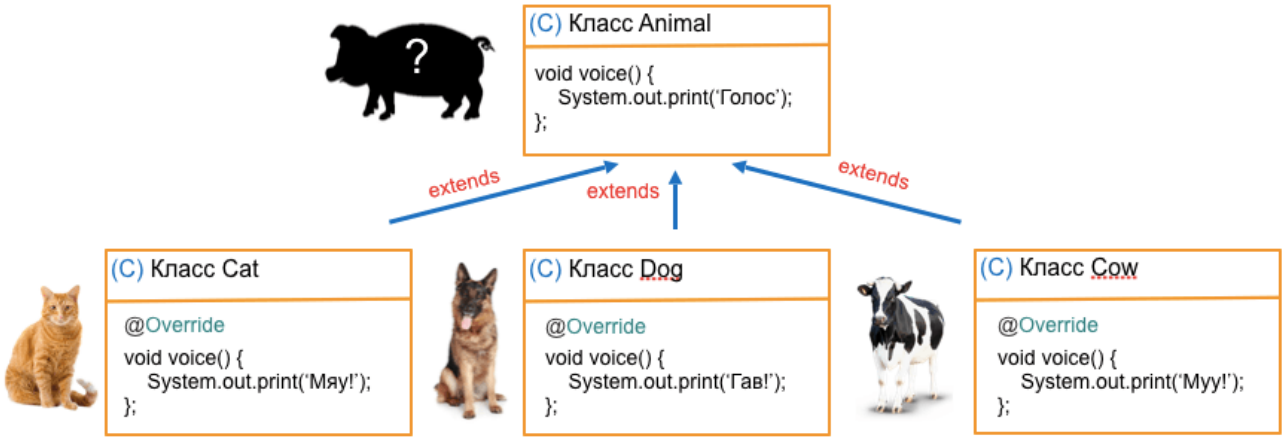
Контейнерами в Java принято называть классы, основная цель которых – хранить набор других элементов.
Контейнеры можно рассматривать как канонические классы для коллекций различных типов.
Два больших вида контейнеров: последовательные и ассоциативные
Последовательные контейнеры:
– очереди, двусторонние очереди (деки)
– списки (простые), двусторонние списки
– стеки
Особенность последовательных контейнеров – быстрое добавление и удаление, но последовательный поиск.
Ассоциативные контейнеры:
– Словари
– Множества
– Бинарные деревья
– Списковые массивы
Особенность - обеспечивают быстрый доступ к данным по ключу, но операции включения и исключения – медленнее, чем в последовательных списках.
Преимущества использования контейнеров:
– Снижают количество написанного кода.
– Повышают быстродействие и качество программы.
– Позволяет взаимодействовать не связанным между собой API. Коллекции - общеупотребляемые типы данных, поэтому могут легко использоваться в качестве аргументов функций и их возвращаемых значений.
– Уменьшает количество усилий, приложенных для изучения новых API.
– Способствует повторному использованию программного обеспечения.

6. **Полиморфизм. Реализация в языках, примеры кода**

Полиморфизм - возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования.

Полиморфизм, если перевести, - это значит "много форм". Например, актер в театре может примерять на себя много ролей - или принимать "много форм".

Полиморфизм



Вверху иерархии классов стоит класс Animal. Его наследуют три класса - Cat, Dog и Cow.
У класса "Animal" есть метод "голос" (voice). Этот метод выводит на экран сообщение "Голос". Естественно, ни собака, ни кошка не говорят "Голос" Они гавкают и мяукают. Соответственно, Вам нужно задать другой метод для классов Cat, Dog и Cow - чтобы кошка мяукала, собака гавкала, а корова говорила "Муу".
Поэтому, в классах-наследниках мы переопределяем метод voice(), чтобы мы в консоли получали "Мяу", "Гав" и "Муу".

Обратите внимание: перед методом, который мы переопределяем, пишем "@Override". Это дает понять компилятору, что мы хотим переопределить метод.

7. **Инкапсуляции в программировании, реализация в Java и C#, примеры кода**
Инкапсуляция — это одна из четырёх фундаментальных концепций ООП. свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.
Инкапсуляция в Java является механизмом обёртывания данных (переменных) и кода, работающего с данными (методами), в одно целое. В инкапсуляции переменные класса будут скрыты от других классов и доступ к ним может быть получен только с помощью метода их текущего класса. По-другому это называется скрытием данных.
Для достижения инкапсуляции в Java:
- Объявите переменные класса как private.
- Предоставьте public к методам установки и получения (сеттеру и геттеру) для изменения и просмотра значений переменных.

Преимущества инкапсуляции:
- Поля класса можно сделать только для чтения или только для записи.
- Класс может иметь полный контроль над тем, что хранится в его полях.

Ниже приведён пример процесса инкапсуляции в Java:

```
/* File name : EncapTest.java */
public class EncapTest {
  private String name;
  private String idNum;
```

```
  private int age;

  public int getAge() {
    return age;
  }

  public String getName() {
    return name;
  }

  public String getIdNum() {
    return idNum;
  }

  public void setAge(int newAge) {
    age = newAge;
  }

  public void setName(String newName) {
    name = newName;
  }

  public void setIdNum(String newId) {
    idNum = newId;
  }
}
```

8. **Нисходящее и восходящее проектирование, сопоставление. Декомпозиция, обобщение, архитектура, примеры**

Декомпозиция – разбиение системы на все меньшие и меньшие взаимодействующие части, каждая из которых совершенствуется независимо

Обобщения или generics (обобщенные типы и методы) позволяют нам уйти от жесткого определения используемых типов. Обобщения позволяют не указывать конкретный тип, который будет использоваться.

Пример Обобщения

```
class Account<T>{
   private T id;
   private int sum;

   Account(T id, int sum){
      this.id = id;
      this.sum = sum;
   }

   public T getId() { return id; }
   public int getSum() { return sum; }
   public void setSum(int sum) { this.sum = sum; }
}
```

В настоящее время в разработке ПО достаточно часто применяется многоуровневая архитектура или многослойная архитектура (**n-tier architecture**), в рамках которой компоненты проекта разделяются на уровни (или слои). Классическое приложение с многоуровневой архитектурой, чаще всего, состоит из 3 или 4 уровней, хотя их может быть и больше, учитывая возможность разделения некоторых уровней на подуровни.

9. **Базовые типы Java и C#, преобразования типов. Упаковка и распаковка, примеры кода.**

Типы данных в языке Java, делятся на три группы:
– примитивные типы (primitive types),
– ссылочные типы (reference types),
– тип – null. Переменную типа null создать невозможно, но можно присвоить значение null (только) ссылочному типу данных.
Примитивные типы данных Java. Существует восемь примитивных типов данных в Java:
 Числовые данные (numeric types):
Целые типы (integral types) – byte, short, int, long, char
Вещественные типы (floating-point types)– float, double
 Логический тип – Boolean
Преобразование типов. Преобразование типов в Java бывает двух видов: неявное и явное. Неявное преобразование типов выполняется в случае если выполняются условия:
– Оба типа совместимы
– Длина целевого типа больше или равна длине исходного типа
Во всех остальных случаях должно использоваться явное преобразование типов.
Ссылочные типы данных Java. Существует четыре типа ссылочных данных в Java:
– Классы (class types)
– Интерфейсы (interface types)
– Переменные типов (type variables)
– Массивы (array types)
Ссылочные типы хранят ссылку на объект, или же тип данных null, то есть нулевую (пустую) ссылку. В Java все является объектом, за исключением примитивных типов. Примитивные типы передаются по значению, а ссылочные по ссылке.
Создание объекта-оболочки из переменной примитивного типа называется упаковкой (boxing), а получение значения примитивного типа из объекта-оболочки -- распаковкой (unboxing). Объектам-оболочкам можно присваивать значения примитивных типов, а переменным примитивных типов - значения переменных-оболочек, при этом при необходимости автоматически создаются объекты-оболочки с соответствующими значениями (автоупаковка) или наоборот, примитивные значения извлекаются из оболочек (автораспаковка):

В приведенной далее строке кода значение 100 упаковывается вручную в объект типа Integer:
>Integer iOb = new Integer(l00);
В следующей строке кода значение из объекта iOb вручную распаковывается в переменную типа int:
int i = iOb.intValue();

10. Массивы. Объявления, инициализация. Многомерные массивы. Отличия массивов Java и C#, пример кода.

Массив – это контейнер, в котором содержится определённое количество значений конкретного типа данных.

```
int[] lotteryNumbers = {16,32,12,23,33,20};
```

Расценивайте массив как ряд коробок. Количество коробок в массиве изменить нельзя. В каждой коробке находится значение того же типа данных, что и в остальных коробках. Можно увидеть значение внутри или заменить содержимое коробки другим значением. В контексте массивов такие «*коробки*» называются элементами.

Оператор, отвечающий за объявление массива **Java,** похож на любую другую переменную. Он содержит тип данных, после которого следует название массива. Единственное отличие заключается в использовании квадратных скобок рядом с типом данных:

```
int[] intArray;
float[] floatArray;
char[] charArray;
```

Приведённые выше операторы объявления сообщают компилятору, что **intArrayvariable** – это массив целых чисел, **floatArrayis** – массив чисел с плавающей запятой, а **charArrayis** – массив символов.

Как и другие переменные, их нельзя использовать до инициализации и установки значения. В случае с массивом указание значения должно определять размер массива:

```
intArray = new int[10];
```

Число внутри скобок указывает, сколько элементов содержится в массиве. Приведённый выше код создает массив целых чисел, состоящий из 10 элементов.

Объявление и установку размера массива можно сделать в одном выражении:

```
float[] floatArray = new float[10];
```

Массивы не ограничиваются примитивными типами данных. Также можно создавать массив объектов Java (или строк):

```
String[] names = new String[5];
```

Многомерные массивы

До сих пор мы говорили лишь об одномерных массивах. Но у массивов может быть и больше одного измерения. Многомерные массивы представляют собой контейнеры, внутри которых находится сразу несколько массивов:

```
int[][] lotteryNumbers = {{16,32,12,23,33,20},{34,40,3,11,33,24}};
```

Индекс многомерных массивов содержит два числа:

```
System.out.println("The value of element 1,4 is " + lotteryNumbers[1][4]);
```

При этом длина массивов, находящихся внутри многомерного массива **Java**, не обаятельно должна быть одинаковой:

```
String[][] names = new String[5][7];
```

В Java могут быть объявлены, строго говоря, только одномерные массивы. Многомерный массив в Java — массив массивов. В C# есть как настоящие многомерные массивы, так и массивы массивов, которые в C# обычно называются «неровными», или «ступенчатыми» (jagged).

Массив массивов

От многомерных массивов надо отличать **массив массивов** или так называемый "зубчатый массив":

```
1    int[][] nums = new int[3][];
2    nums[0] = new int[2] { 1, 2 };         // выделяем память для первого подмассива
3    nums[1] = new int[3] { 1, 2, 3 };      // выделяем память для второго подмассива
4    nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для третьего подмассива
```

Здесь две группы квадратных скобок указывают, что это **массив массивов**, то есть такой массив, который в свою очередь содержит в себе другие массивы. Причем длина массива указывается только в первых квадратных скобках, все последующие квадратные скобки должны быть пусты: new int[3][]. В данном случае у нас массив nums содержит три массива. Причем размерность каждого из этих массивов может не совпадать.

11. Работа со строками. String, StringBuilder в Java и C#, примеры кода.

Класс String в Java предназначен для работы со строками в Java. Все строковые литералы, определенные в Java программе (например, "abc") — это экземпляры класса String.

Класс реализует интерфейсы Serializable и CharSequence. Поскольку он входит в пакет java.lang, его не нужно импортировать.Класс String в Java — это final класс, который не может иметь потомков.Класс String — immutable класс, то есть его объекты не могут быть изменены после создания. Любые операции над объектом String, результатом которых должен быть объект класса String, приведут к созданию нового объекта.Благодаря своей неизменности, объекты класса String являются потокобезопасными и могут быть использованы в многопоточной среде.

```
String example =  “Hello World!”; //Создание строк
String one = “Hello”;
String two = “world!”;
String three = one+” “+two //Сложение строк
If(three.equals(example)) System.out.println(example) //Сравнение строк
String[] splitstring = example.split(“ “); //Разбиение строки на массив строк
String subs = example.substring(5); //Извлечение подстроки из строки
Int indexH = example.indexOf(‘H’);//Определение позиции элемента в строке
example = example.toLowerCase(); //Перевод букв в нижний регистр
example = example.toUpperCase(); //Перевод букв в верхний регистр
```

12. Абстрактные классы и интерфейсы Java, инициализаторы, примеры кода.

Абстрактный класс abstract— это объявленный класс, который может включать или не включать абстрактные методы. Абстрактные классы не могут быть созданы, но они могут быть подклассами.(для описания сущностей, которые не имеют конкретного воплощения, предназначены абстрактные классы.)

Абстрактный метод - это метод, который объявляется без реализации (без фигурных скобок и с запятой), например:
abstract void moveTo (double deltaX, double deltaY);

Если класс включает абстрактные методы, то сам класс должен быть объявлен abstract, как в:

```
публичный абстрактный класс
 GraphicObject {
 // объявлять поля
 // объявить неабстрактные методы
abstract void draw();
}
```

Когда абстрактный класс является подклассом, подкласс обычно предоставляет реализации для всех абстрактных методов в своем родительском классе. Однако, если это не так, то подкласс также должен быть объявлен abstract.

13. Проблема ромба в наследовании. Методы по умолчанию в интерфейсах Java, пример кода.

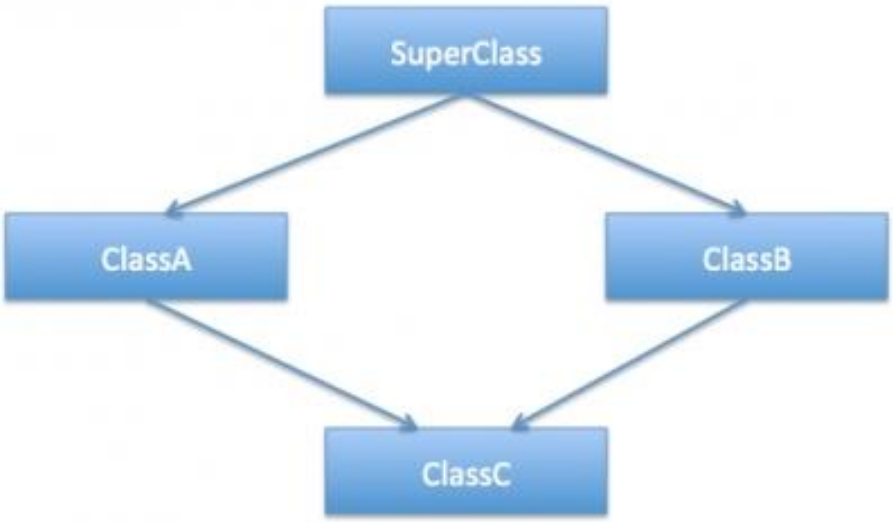
Множественное наследование в Java

Множественное наследование — возможность создания единого класса с несколькими суперклассами.

В отличие от некоторых других популярных объектно-ориентированных языках программирования, таких как C ++, Java не предоставляет поддержку множественного наследования в классах. Java не поддерживает множественное наследование классов, потому что это может привести к проблеме ромба (ромбовидное наследование) и вместо того, чтобы предоставлять сложный путь разрешения этой проблемы, придумали способ лучше.

Проблема ромба

Понять проблему с ромбами легко: давайте предположим, что множественное наследование было реализовано в Java. В этом случае, мы могли бы иметь иерархию классов, как на изображении ниже.



Давайте создадим абстрактный суперклас SuperClass с методом doSomething(), а также два класса ClassA, ClassB
SuperClass.java

```java
package ua.com.prologistic.inheritance;

public abstract class SuperClass {

    public abstract void doSomething();
}
```

ClassA.java

```java
package ua.com.prologistic.inheritance;

public class ClassA extends SuperClass{

    @Override
    public void doSomething(){
        System.out.println("doSomething реализуется в классе A");
    }

    //Собственный метод класса ClassA
    public void methodA(){

    }
}
```

ClassB.java

```java
package ua.com.prologistic.inheritance;

public class ClassB extends SuperClass{

    @Override
    public void doSomething(){
        System.out.println("doSomething реализуется классом B");
    }

    //Свой метод класса ClassB
    public void methodB(){

    }
}
```

А теперь давайте создадим класс ClassC, который наследует классы ClassA и ClassB

```java
package ua.com.prologistic.inheritance;

public class ClassC extends ClassA, ClassB{

    public void test(){
        //вызываем метод суперкласса
        doSomething();
    }

}
```

Обратите внимание, что метод test() вызывает метод суперкласса doSomething()

Это приводит к неопределенности: компилятор не знает, какой метод суперкласса выполнить из-за ромбовидной формы (выше на диаграмме классов). Это называют проблемой ромба — и это основная причина почему Java не поддерживает множественное наследование классов.

Методы по умолчанию

Ранее до JDK 8 при реализации интерфейса мы должны были обязательно реализовать все его методы в классе. А сам интерфейс мог содержать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как методы по умолчанию. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе Printable:

```
interface Printable {

    default void print(){

        System.out.println("Undefined printable");
    }
}
```

Метод по умолчанию - это обычный метод без модификаторов, который помечается ключевым словом default. Затем в классе Journal нам необязательно этот метод реализовать, хотя мы можем его и переопределить:

```
class Journal implements Printable {

    private String name;

    String getName(){
        return name;
    }
    Journal(String name){

        this.name = name;
    }
}
```

14. **Java: модификаторы доступа, ключевые слова this, super, abstract, примеры кода.**

Модификаторы — ключевые слова, которые Вы добавляете при инициализации для изменения значений.

this

Каждый раз при вызове обычного (не статического) метода, кроме явных аргументов, если они имеются, туда еще передается и не явный аргумент – this. this хранит ссылку на объект из которого был вызван метод this может использоваться внутри любого метода для ссылки на текущий объект. То есть this всегда служит ссылкой на объект, из которого был вызван метод. Ключевое слово this можно использовать везде, где допускается ссылка на текущий объект. Поскольку this указывает на текущий объект, то его можно использовать для явного указания обращений к полям текущего экземпляра класса, например:

this.age = 5;

this можно использовать для вызова одних конструкторов класса из других. Вызвать один метод класса из другого метода очень просто – по имени. И даже если имена одинаковые, то работает правило перегрузки, а вот как вызывать конструкторы? У них же имя совпадает с именем класса. Один метод из другого можно вызывать в любом месте метода, но вызов одного конструктора из другого может быть только в первой строке вызывающего конструктора.

super

В Java существует ключевое слово *super*, которое обозначает **суперкласс**, т.е. класс, производным от которого является текущий класс. В данном случае, супер не означает превосходство, скорее даже наоборот, дочерний класс имеет больше методов, чем родительский.

Ключевое слово **super** можно использовать для вызова конструктора суперкласса и для обращения к члену суперкласса, скрытому членом подкласса.

Пример

```
class HeavyBox extends Box {
   int weight; // вес коробки
```

```
    // конструктор
    // инициализируем переменные с помощью ключевого слова super
    HeavyBox(int w, int h, int d, int m) {
        super(w, h, d); // вызов конструктора суперкласса
        weight = m; // масса
    }
}
```

abstract

Кроме обычных классов в Java есть **абстрактные классы**. Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, но в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово **abstract**:

```
public abstract class Human{

    private String name;
    public String getName() { return name; }
}
```

15. **Параметрические типы Java и C#, примеры кода.**

Дженерики **Java**. Дженерики (или обобщения) - это параметризованные **типы**.

Параметризованные **типы** позволяют объявлять классы, интерфейсы и методы, где **тип** данных, которыми они оперируют, указан в виде параметра. Используя дженерики, можно создать единственный класс, например, который будет автоматически работать с разными **типами** данных.

4. Как работают Generics

На самом деле Generics работают до ужаса примитивно.

Компилятор просто заменяет тип с параметром на него же, только без параметра. А при взаимодействии с его методами добавляет операцию приведения типа к типу-параметру:

Код	Что сделает компилятор
`ArrayList<Integer> list = new ArrayList<Integer>();`	`ArrayList list = new ArrayList();`
`list.add(1);`	`list.add( (Integer) 1 );`
`int x = list.get(0);`	`int x = (Integer) list.get(0);`
`list.set(0, 10);`	`list.set(0, (Integer) 10);`

16. **Интерфейсы и реализации коллекций в Java: List (Vector, Stack, ArrayList), примеры кода.**

List - **это** абстрактный тип данных. ArrayList - конкретная реализация **этого** типа на базе массива. **Vector** - **это** коллекция из древних времён **Java**, сохранённая в стандартной библиотеке для того, чтобы старый код мог работать на новых JVM.

Vector **синхронизируется** , что означает, что только один поток за раз может получить доступ к коду, в то время как arrayList **не синхронизирован** , что означает, что несколько потоков могут работать с arrayList одновременно.

Класс **Java Stack**, **java**.util.**Stack**, представляет собой классическую структуру данных стека. Вы можете поместить элементы на вершину стека и снова извлечь их, что означает чтение и удаление элементов с его вершины. Класс фактически реализует интерфейс **List**, но вы редко используете его в качестве списка – за исключением, когда нужно проверить все элементы, хранящиеся в данный момент в стеке.

Пример

```
Vector v = new Vector(3, 2); System.out.println("Начальный размер: " + v.size()); System.out.println("Начальная ёмкость: " + v.capacity());
```

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
```

```
Stack st = new Stack();
```

```
System.out.println("Empty stack: " + st);
```

17. **Интерфейсы и реализации коллекций в Java: Queue (ArrauDeque, LinkedList)**

Интерфейс Queue

Интерфейс Queue расширяет Collection и объявляет поведение очередей, которые представляют собой список с дисциплиной "первый вошел, первый вышел" (FIFO). Существуют разные типы очередей, в которых порядок основан на некотором критерии. Очереди не могут хранить значения null.

Класс ArrayDeque

ArrayDeque создает двунаправленную очередь.

Конструкторы класса ArrayDeque:

- ArrayDeque() - создает пустую двунаправленную очередь с вместимостью 16 элементов.
- ArrayDeque(Collection<? extends E> c) - создает двунаправленную очередь из элементов коллекции с в том порядке, в котором они возвращаются итератором коллекции с.
- ArrayDeque(int numElements) - создает пустую двунаправленную очередь с вместимостью numElements.

Класс LinkedList

Класс LinkedList реализует интерфейсы сразу два интерфейса - List, Deque. LinkedList – это двусвязный список.

Конструкторы класса LinkedList:

- LinkedList()
- LinkedList(Collection<? extends E> c)

Внутри класса LinkedList существует static inner класс Entry, с помощью которого создаются новые элементы:

```
private static class Entry<E>
{
   E element;
   Entry<E> next;
   Entry<E> prev;

   Entry(E element, Entry<E> next, Entry<E> prev)
   {
      this.element = element;
      this.next = next;
      this.prev = prev;
   }
}
```

Из LinkedList можно организовать стек, очередь, или двойную очередь, со временем доступа O(1). На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n). Однако, на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется O(1). Позволяет добавлять любые значения, в том числе и null.

18. **Интерфейсы и реализации коллекций в Java: Map (HashMap, TreeMap, HashSet, TreeSet)**

Интерфейс Queue

Интерфейс Queue расширяет Collection и объявляет поведение очередей, которые представляют собой список с дисциплиной "первый вошел, первый вышел" (FIFO). Существуют разные типы очередей, в которых порядок основан на некотором критерии. Очереди не могут хранить значения null.

Класс ArrayDeque

ArrayDeque создает двунаправленную очередь.

Конструкторы класса ArrayDeque:

- ArrayDeque() - создает пустую двунаправленную очередь с вместимостью 16 элементов.
- ArrayDeque(Collection<? extends E> c) - создает двунаправленную очередь из элементов коллекции с в том порядке, в котором они возвращаются итератором коллекции с.
- ArrayDeque(int numElements) - создает пустую двунаправленную очередь с вместимостью numElements.

Класс LinkedList

Класс LinkedList реализует интерфейсы сразу два интерфейса - List, Deque. LinkedList – это двусвязный список.

Конструкторы класса LinkedList:

- LinkedList()
- LinkedList(Collection<? extends E> c)

Внутри класса LinkedList существует static inner класс Entry, с помощью которого создаются новые элементы:

```
private static class Entry<E>
{
   E element;
   Entry<E> next;
   Entry<E> prev;

   Entry(E element, Entry<E> next, Entry<E> prev)
   {
      this.element = element;
      this.next = next;
      this.prev = prev;
   }
}
```

Из LinkedList можно организовать стек, очередь, или двойную очередь, со временем доступа O(1). На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n). Однако, на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется O(1). Позволяет добавлять любые значения, в том числе и null.

19. Постоянство объектов. Потоки ввода/вывода. Сохранение в «плоских» файлах, примеры кода

Постоянство объектов - это свойство объектов существовать вне зависимости от программы. Стандартное средство решения вопросов постоянства объектов, которое называется механизмом сериализации.

Потоки ввода вывода:

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса:

InputStream (представляющий потоки ввода)

OutputStream (представляющий потоки вывода)

Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы

Reader (для чтения потоков символов)

Writer (для записи потоков символов)

Код

```
static void Test01W(){
try(ObjectOutputStream oos = new
ObjectOutputStream(
new FileOutputStream("d:/-/user.dat")))
{ User1 p = new User1("Петр","Петров",25);
oos.writeObject(p);
catch(Exception ex){
System.out.println(ex.getMessage());
}
}
```

КОД СЧИТЫВАНИЯ

```
static void Test01R(){
try(ObjectInputStream ois = new ObjectInputStream(
new FileInputStream("d:/-/user.dat")))
{
User1 p = (User1)ois.readObject();
System.out.printf("Имя: %s \t Возраст: %d \n", p.Name, p.Age);
}
catch(Exception ex){
```

System.out.println(ex.getMessage());

20. Сохранение объектов в потоках. Бинарные потоки, пример кода.

Класс OutputStream является базовым классом для всех классов, которые работают с бинарными потоками записи. Свою функциональность он реализует через следующие методы:

void close(): закрывает поток

void flush(): очищает буфер вывода, записывая все его содержимое

void write(int b): записывает в выходной поток один байт, который представлен целочисленным параметром b

void write(byte[] buffer): записывает в выходной поток массив байтов buffer.

void write(byte[] buffer, int offset, int length): записывает в выходной поток некоторое число байтов, равное length, из массива buffer, начиная со смещения offset, то есть с элемента buffer[offset].

21. Сохранение объектов в потоках. Формат хранения JSON, примеры кода.

простой документ JSON

```
{
  "name": "Benjamin Watson",
  "age": 31,
  "isMarried": true,
  "hobbies": ["Football", "Swimming"],
  "kids": [
    {
      "name": "Billy",
      "age": 5
    },
   {
      "name": "Milly",
      "age": 3
    }
  ]
}
```

Сохранение

```
public void Save(Book[] array) throws IOException {
    JSONArray listMain = new JSONArray();

    FileWriter file = null;
    try {
        file = new FileWriter("E:\\java\\lab\\src\\main\\java\\Lab4_6\\inf.json");
    } catch (IOException e) {
        e.printStackTrace();
    }

    Arrays.stream(array).forEach(x ->
    {
        JSONObject obj = new JSONObject();
        obj.put("udc", x.getUdc());
        obj.put("familynameauthor", x.getFamilynameauthor());
        obj.put("nameauthor", x.getNameauthor());
        obj.put("namebook", x.getNamebook());
        obj.put("price", x.getPrice());
        obj.put("coll", x.getColl());
        listMain.add(obj);
```

```
    });
    try {
        file.write(listMain.toJSONString());
        file.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

22. Чтение данных из базы данных Access, примеры кода.

Для чтения базы данных Access в Java используется стандарт Java Database Connectivity (JDBC). Платформенно независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета, входящего в состав Java SE. JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает. Существуют следующие виды типов Statement, различающихся по назначению:

- java.sql.Statement — Statement общего назначения;
- java.sql.PreparedStatement — Statement, служащий для выполнения запросов, содержащих подставляемые параметры (обозначаются символом '?' в теле запроса);
- java.sql.CallableStatement — Statement, предназначенный для вызова хранимых процедур.

```
import java.sql.*;
//подключение библиотек
private static final String AccessURL = "jdbc:ucanaccess:Database.accdb";
// адрес к базе данных
Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
//указание драйвера
Connection connect = DriverManager.getConnection(AccessURL);
//подключение к базе данных
Statement stat = connect.createStatement();
//Создание инструкции
Resultset rs = stat.executeQuery(“SELECT * FROM Test”);
//Выполнение тестового запроса и получение данных
While(rs.next())
System.out.pintln(rs.getNString(1));
//Вывод данных
```

23. Чтение данных из базы данных SqLite, примеры кода.

Для чтения базы данных SQLite в Java используется стандарт Java Database Connectivity (JDBC). Платформенно независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета, входящего в состав Java SE. JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает. Существуют следующие виды типов Statement, различающихся по назначению:

- java.sql.Statement — Statement общего назначения;
- java.sql.PreparedStatement — Statement, служащий для выполнения запросов, содержащих подставляемые параметры (обозначаются символом '?' в теле запроса);
- java.sql.CallableStatement — Statement, предназначенный для вызова хранимых процедур.

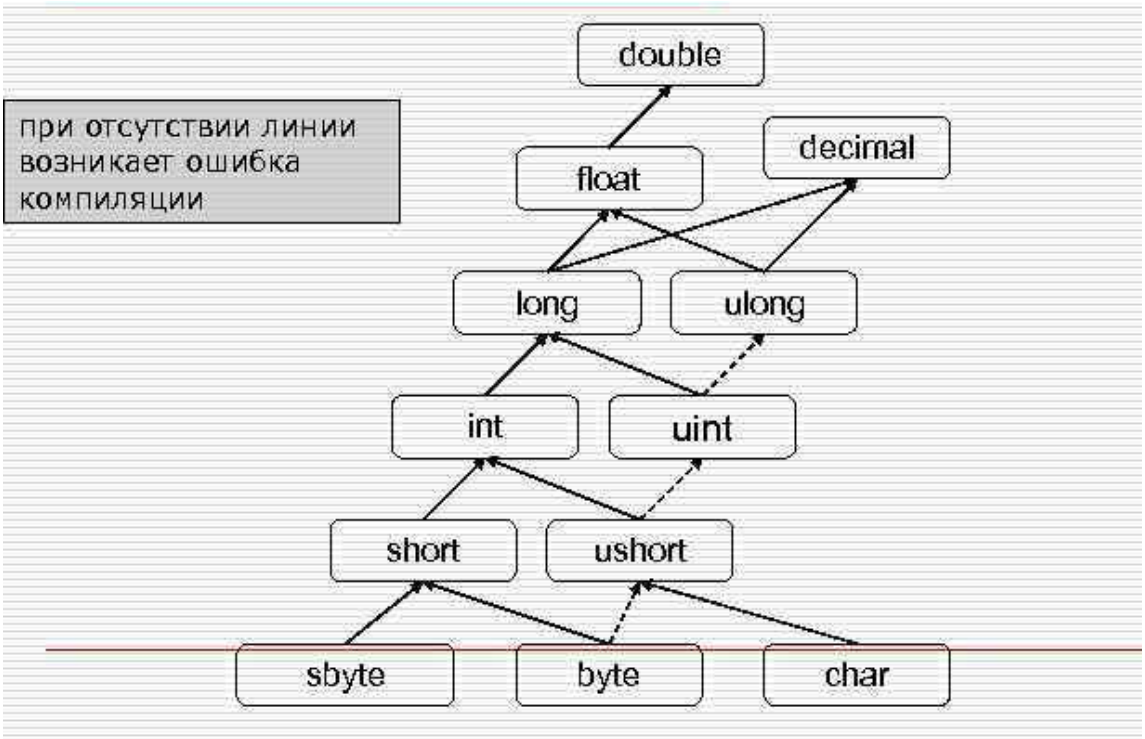
```
import java.sql.*;
//подключение библиотек
```

```
private static final String AccessURL = "jdbc:sqlite:Database.db";
// адрес к базе данных
Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
//указание драйвера
Connection connect = DriverManager.getConnection(AccessURL);
//подключение к базе данных
Statement stat = connect.createStatement();
//Создание инструкции
Resultset rs = stat.executeQuery(“SELECT * FROM Test”);
//Выполнение тестового запроса и получение данных
While(rs.next())
System.out.pintln(rs.getNString(1));
//Вывод данных
```

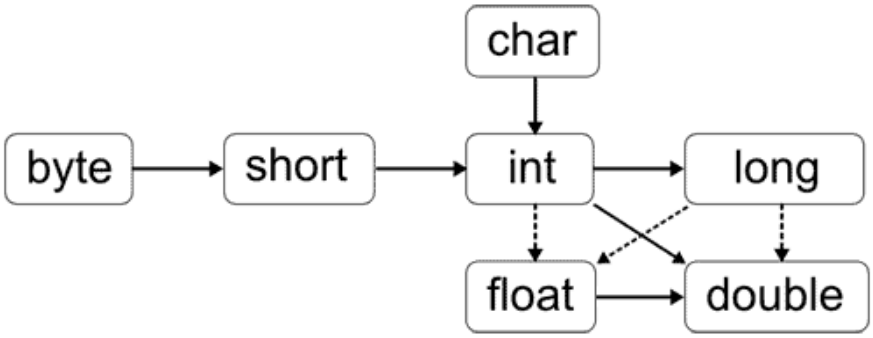
24. **Типизация. Сопоставление статической и динамический типизации, Классификация языков по типизации. Приведение типов в Java, C#.**

Языки программирования делятся на статически и динамически типизированные языки.
Статическая типизация — приём, широко используемый в языках программирования, при котором переменная, параметр подпрограммы, возвращаемое значение функции связывается с типом в момент объявления и тип не может быть изменён позже.
Динамиическая типизация — приём, используемый в языках программирования и языках спецификации, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.
Как правило, динамическая типизация используется в интерпретируемых языках, так как это добавляет гибкости.
Статическая типизация используется в компилируемых языках для максимальной производительности и скорости работы.
Java является строго типизированным языком. Это означает, что любая переменная и любое выражение имеют известный тип еще на момент компиляции.
Языки программирования по признакам типизации делятся на две категории — нетипизированные (untyped) и типизированные (typed). К нетипизированным можно отнести например Assembler или Forth, а к типизированным Java, C#, JavaScript или Python.

Приведение типов в c#:



Приведение типов в java:



25. Рефлексия, динамическая идентификация типов в языках C# и Java, примеры кода.
Рефлексия *(от позднелат. reflexio — обращение назад)* — это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов. Сам же механизм рефлексии позволяет обрабатывать типы, отсутствующие при компиляции, но появившиеся во время выполнения программы. Рефлексия и наличие логически целостной модели выдачи информации об ошибках дает возможность создавать корректный динамический код. Иначе говоря, понимание принципов работы рефлексии в java открывает перед вами ряд удивительных возможностей. Вы буквально можете жонглировать классами и их составляющими.
Вот основной список того, что позволяет рефлексия:

- Узнать/определить класс объекта;
- Получить информацию о модификаторах класса, полях, методах, константах, конструкторах и суперклассах;
- Выяснить, какие методы принадлежат реализуемому интерфейсу/интерфейсам;
- Создать экземпляр класса, причем имя класса неизвестно до момента выполнения программы;
- Получить и установить значение поля объекта по имени;

Вызвать метод объекта по имени.

26. Пространство имен java.util.stream: Потоки, отличие от коллекций Java, виды потоков, примеры кода.

Начиная с версии JDK 8 в языке Java были введены средства работы с потоками данных, которые получили название прикладной потоковый интерфейс API (Stream Application Programming Interface). Работа со средствами Stream API базируется на использовании лямбда-выражений.
Характерными особенностями этого интерфейса является использование различных операций над потоками данных. К этим операциям можно отнести:

- поиск данных;
- модификация данных;
- фильтрование потока данных для получения нового потока;
- сортировка данных;
- другие разнообразные манипуляции над данными.

Разница между коллекцией(Collection) данных и потоком (Stream) в том
что коллекции позволяют работать с элементами по-отдельности, тогда как
поток(Stream) не позволяет. Например, с использованием коллекций, вы можете
добавлять элементы, удалять, и вставлять в середину. Поток(Stream) не позволяет
манипулировать отдельными элементами из набора данных, но вместо этого позволяет
выполнять функции над данными как одним целом.

```
List<String> myList =
   Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
   .stream()
   .filter(s -> s.startsWith("c"))
   .map(String::toUpperCase)
   .sorted()
   .forEach(System.out::println);
```

Методы потоков бывают промежуточными (intermediate) и терминальными (terminal). Промежуточные методы возвращают поток, что позволяет последовательно вызывать множество таких методов. Терминальные методы либо не возвращают значения (void) либо возвращают результат типа отличного от потока. В вышеприведенном примере методы filter, map и sorted являются промежуточными, а forEach — терминальным.

Помимо обычных потоков объектов Java 8 располагает специальными типами потоков для работы с примитивными типами: int, long, double. Как можно догадаться это IntStream, LongStream, DoubleStream.
Иногда полезно превратить поток объектов в поток примитивов или наоборот. Для этой цели потоки объектов поддерживают специальные методы: mapToInt(), mapToLong(), mapToDouble()

27. **Функциональные интерфейсы Java, сопоставление с делегатами C#, примеры кода.**

В Java лямбда-выражение можно просто рассматривать как анонимную функцию, реализующую интерфейс только с одним методом. Этот тип интерфейса называется функциональным интерфейсом и может быть снабжен аннотацией @FunctionalInterface , которая указывает компилятору применять правило только одного метода:

```
@FunctionalInterface
interface Printer {
  void print(String message);
}
Printer standardPrinter = message -> System.out.println(message);

// The print method of standardPrinter can be invoked
standardPrinter.print("Hello World!");
```

28. **Лямбда выражения Java и C#, ссылка на методы, примеры кода.**

Лямбда-выражения используют преимущества параллельных процессов в многоядерных средах, что видно при поддержке операций с конвейерами данных в Stream API.
Это анонимные методы (методы без имени), используемые для реализации метода, определенного функциональным интерфейсом. Важно знать, что такое функциональный интерфейс, прежде чем вы начнете использовать Лямбда-выражения.

Оператор Стрелка
Лямбда-выражения вводят новый оператор стрелка -> в Java. Он разделяет лямбда-выражение на 2 части:
(n) -> n*n

Ссылки на методы бывают четырех видов:

Тип	Пример	
Ссылка на статический метод	***ContainingClass::staticMethodName***	Function<String, Boolean> function = Boolean::valueOf; System.out.println(function.apply("TRUE"));
Ссылка на нестатический метод конкретного объекта	***containingObject::instanceMethodName***	Consumer<String> consumer = System.out::println; consumer.accept("OCPJP 8");
Ссылка на нестатический метод любого объекта конкретного типа	***ContainingType::methodName***	Function<String, String> function = String::toLowerCase; System.out.println(function.apply("OCPJP 8"));
Ссылка на конструктор	***ClassName::new***	Function<String, Integer> function = Integer::new; System.out.println(function.apply("4"));

29. **Пространство имен java.util.stream: Создание потоков, примеры кода.**
Начиная с JDK 8 в Java появился новый API - Stream API. Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream.
Ключевым понятием в Stream API является поток данных. Вообще сам термин "поток" довольно перегружен в программировании в целом и в Java в частности. В одной из предыдущих глав рассматривалась работа с символьными и байтовыми потоками при чтении-записи файлов. Применительно к Stream API поток представляет канал передачи данных из источника данных. Причем в качестве источника могут выступать как файлы, так и массивы и коллекции.
Одной из отличительных черт Stream API является применение лямбда-выражений, которые позволяют значительно сократить запись выполняемых действий.
При ближайшем рассмотрении мы можем найти в других технологиях программирования аналоги подобного API. В частности, в языке C# некоторым аналогом Stream API будет технология LINQ.
Рассмотрим простейший пример. Допустим, у нас есть задача: найти в массиве количество всех чисел, которые больше 0. До JDK 8 мы бы могли написать что-то наподобие следующего:
int[] numbers = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};

```
int count=0;
for(int i:numbers){

   if(i > 0) count++;
}
System.out.println(count);
```

Теперь применим Stream API:

```
import java.util.stream.*;
//.....................
long count = IntStream.of(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5).filter(w -> w > 0).count();
System.out.println(count);
```

Теперь вместо цикла и кучи условных конструкций, которые мы бы использовали до JDK 8, мы можем записать цепочку методов, которые будут выполнять те же действия.

При работе со Stream API важно понимать, что все операции с потоками бывают либо терминальными (terminal), либо промежуточными (intermediate). Промежуточные операции возвращают трансформированный поток. Например, выше в примере метод filter принимал поток чисел и возвращал уже преобразованный поток, в котором только числа больше 0. К возвращенному потоку также можно применить ряд промежуточных операций.

Конечные или терминальные операции возвращают конкретный результат. Например, в примере выше метод count() представляет терминальную операцию и возвращает число. После этого никаких промежуточных операций естественно применять нельзя.

Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция. То есть в данном случае применяется отложенное выполнение.

В основе Stream API лежит интерфейс BaseStream. Его полное определение:

```
1    interface BaseStream<T , S extends BaseStream<T , S>>
```

Здесь параметр T означает тип данных в потоке, а S - тип потока, который наследуется от интерфейса BaseStream.

30. Пространство имен java.util.stream: Терминальные операции, примеры кода.

Конечные или терминальные операции возвращают конкретный результат. Например, в примере выше метод count() представляет терминальную операцию и возвращает число. После этого никаких промежуточных операций естественно применять нельзя.

31. Пространство имен java.util.stream: Промежуточные операции, примеры кода.

Промежуточные операции возвращают трансформированный поток. Например, выше в примере метод filter принимал поток чисел и возвращал уже преобразованный поток, в котором только числа больше 0. К возвращенному потоку также можно применить ряд промежуточных операций.

32. Признаки плохого проекта.

Закрепощенность

Закрепощенность проявляется в том случае, когда программа с трудом подда-ется изменениям, производимым с помощью простых методов.

Неустойчивость

Неустойчивость при внесении одного изменения программа "разрушается" во многих местах. Очень часто новые проблемы возникают в тех областях, которые не связаны с изменяемым компонентом. В процессе исправления этих ошибок возникают новые ошибки. По мере возрастания неустойчивости программного модуля вероятность появ-ления непредвиденных проблем приближается к 100%. Несмотря на всю абсурд-ность подобного утверждения, подобные модули встречаются довольно часто. Неподвижность Проект считается неподвижным, если он содержит компоненты, которые мо-гут применяться в других системах, однако усилия и степень риска, связанные с изоляцией этих компонентов от первоначальной системы, слишком велики. К со-жалению, эта тенденция проявляется весьма часто.

Вязкость

Вязкость может проявляться в двух формах: по отношению к ПО и к среде. В случае необходимости внесения изменений разработчики, как правило, при-меняют различные методы. Некоторые из них способствуют сохранению исход-ного проекта, а другие — нет (поскольку относятся к разряду хакерских приемов). Если предлагаемые проектом методы сложнее в применении, чем хакерские прие-мы, то говорят, что вязкость проекта чрезвычайно высока. В этом случае легко до-пустить ошибку, а нужные и корректные действия выполнить не так уж и просто. В идеале требуется разработка программы, допускающих внесение изменений, сохраняющих структуру проекта. Симптом вязкости наблюдается в случае, если среда разработки характери-зуется словами "медленный" и "неэффективный".

Неоправданная сложность

Проект имеет неоправданную сложность, если содержит элементы, не исполь-зующиеся в настоящий момент времени. Это часто происходит в том случае, когда разработчики предвидят изменения в требованиях и проводят мероприятия, на-правленные на то, чтобы справиться с этими потенциальными изменениями.

Неоправданные повторения

Операции "вырезки" и "вставки" могут быть полезными при редактирова-нии текста, но в то же время они могут быть опасными операциями в случае редактирования кода. Слишком часто программные системы выстраиваются на десятках или сотнях повторяющихся элементов кода.

Неопределенность

Неопределенность программного модуля проявляется в сложности его пони-мания. Код может быть написан либо в четкой и выразительной манере, либо быть неопределенным и трудным для понимания.

33. SOLID: Принцип открытости/закрытости, примеры кода

Модули, отвечающие принципу открытости-закрытости, имеют два главных признака:

Открыты для расширения. Это означает, что поведение модуля может быть расширено. То есть мы можем добавить модулю новое поведение в соответствии с изменившимися требованиями к приложению или для удовлетворения нужд новых приложений.

Закрыты для изменений. Исходный код такого модуля неприкасаем. Никто не вправе вносить в него изменения.

Ключ к решению — абстракция

Абстрактный класс

```
class Shape
{
public:
   virtual void Draw() const = 0;
};
class Square : public Shape
{
public:
   virtual void Draw() const;
};
class Circle : public Shape
{
public:
   virtual void Draw() const;
};
void DrawAllShapes(Set<Shape*>& list)
{
   for (Iterator<Shape*>i(list); i; i++)
      (*i)->Draw();
}
```

34. SOLID: Принцип персональной ответственности, примеры кода

Принцип единственной ответственности (англ. single-responsibility principle, SRP) — принцип ООП, обозначающий, что каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.

```
class Book {

   function getTitle() {
      return "A Great Book";
   }

   function getAuthor() {
      return "John Doe";
   }

   function turnPage() {
      // pointer to next page
   }

   function getCurrentPage() {
      return "current page content";
   }

}

interface Printer {

   function printPage($page);
}

class PlainTextPrinter implements Printer {

   function printPage($page) {
      echo $page;
   }

}

class HtmlPrinter implements Printer {

   function printPage($page) {
      echo '<div style="single-page">' . $page . '</div>';
   }
}
```

35. **SOLID: Принцип подстановки Лисков, примеры кода**

Принцип подстановки Лисков (Liskov Substitution Principle) представляет собой некоторое руководство по созданию иерархий наследования.
Класс S может считаться подклассом T, если замена объектов T на объекты S не приведет к изменению работы программы.

Классический пример: класс квадрат и класс прямоугольник.

```
class Rectangle {
   public virtual int Width { get; set; }
```

```
    public virtual int Height { get; set; }
    public int GetArea() {
      return Width * Height;
    }
  }

  class Square : Rectangle {
    public override int Width {
      get {
        return base.Width;
      }
      set {
        base.Width = value;
        base.Height = value;
      }
    }
    public override int Height {
      get {
        return base.Height;
      }
      Set {
        base.Height = value;
        base.Width = value;
      }
    }
  }
```

Производный класс, который может делать меньше, чем базовый, обычно нельзя подставить вместо базового, и поэтому он нарушает принцип подстановки Лисков.

Принцип подстановки (замещения) Лисков имеет близкое отношение к методологии контрактного программирования, и ведёт к некоторым ограничениям на то, как контракты могут взаимодействовать с наследованием:

- Предусловия не могут быть усилены в подклассе.
- Постусловия не могут быть ослаблены в подклассе.
- Исторические ограничения («правило истории») — подкласс не должен создавать новых мутаторов свойств базового класса. Если базовый класс не предусматривал методов для изменения определенных в нем свойств, подтип этого класса так же не должен создавать таких методов.

36. SOLID: Принцип разделения интерфейса, примеры кода

Роберт С. Мартин определил[1] этот принцип так:

Программные сущности не должны зависеть от методов, которые они не используют.

Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы программные сущности маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться программные сущности, которые этот метод не используют.

```
interface IMessage
{
  void Send();
  string Text { get; set;}
  string Subject { get; set;}
  string ToAddress { get; set; }
  string FromAddress { get; set; }
}
class EmailMessage : IMessage
```

```
{
   public string Subject { get; set; } = "";
   public string Text { get; set; } = "";
   public string FromAddress { get; set; } = "";
   public string ToAddress { get; set; } = "";

   public void Send() => Console.WriteLine($"Отправляем Email-сообщение: {Text}");
}
class SmsMessage : IMessage
{
   public string Text { get; set; } = "";
   public string FromAddress { get; set; } = "";
   public string ToAddress { get; set; } = "";

   public string Subject
// Свойство Subject, которое определяет тему сообщения, при отправке смс не //указывается, поэтому оно в данном классе не нужно. В классе SmsMessage появляется //избыточная функциональность, от которой класс SmsMessage начинает зависеть.

   {
      get
      {
         throw new NotImplementedException();
      }

      set
      {
         throw new NotImplementedException();
      }
   }

   public void Send() => Console.WriteLine($"Отправляем Sms-сообщение: {Text}");
}
```

37. SOLID: Принцип инверсии зависимостей, примеры кода

Принцип инверсии зависимостей (Dependency Inversion Principle) служит для создания слабосвязанных сущностей, которые легко тестировать, модифицировать и обновлять. Этот принцип можно сформулировать следующим образом:

Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

```
interface IPrinter {
   void Print(string text);
}
class Book
{
   public string Text { get; set; }
   public IPrinter Printer { get; set; }

   public Book(IPrinter printer) {
      this.Printer = printer;
   }
```

```
        public void Print() {
            Printer.Print(Text);
        }
    }
    class ConsolePrinter : IPrinter {
        public void Print(string text) {
            Console.WriteLine("Печать на консоли");
        }
    }
    class HtmlPrinter : IPrinter {
        public void Print(string text) {
            Console.WriteLine("Печать в html");
        }
    }
```

В итоге и класс Book и класс ConsolePrinter зависят от абстракции IPrinter. Кроме того, теперь мы также можем создать дополнительные низкоуровневые реализации абстракции IPrinter и динамически применять их в программе.

38. Паттерн «Наблюдатель», примеры кода

Паттерн Наблюдатель – поведенческий шаблон проектирования, определяет отношение «один ко многим» между субъектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых компонентов.

```
import java.util.ArrayList;
import java.util.List;

interface Notifier{
   public void addObserver(Observer obs);
   public void removeObserver(Observer obs);
   public void notifyObserver();
}

class CentralComp implements Notifier{
   private List observers;    // список наблюдателей
   private int speed;      // скорость
   private int rpm;       // обороты двигателя
   private int oilPressure;    // давление масла

   public CentralComp(){
      observers = new ArrayList();
   }

   // добавить слушателя
   public void addObserver(Observer obs) {
      observers.add(obs);
   }

   // удалить слушателя
   public void removeObserver(Observer obs) {
      int i = observers.indexOf(obs);
      if (i >= 0){
         observers.remove(i);
      }
   }
```

```
    // уведомить слушателей
    public void notifyObserver() {
        for (int i = 0; i < observers.size(); i++){
            Observer obs = (Observer)observers.get(i);
            obs.update(speed, rpm, oilPressure);
        }
    }

    public void changeData(int speed, int rpm, int oilPressure){    // метод для изменения характеристик при движении
автомобиля
        this.speed = speed;
        this.rpm = rpm;
        this.oilPressure = oilPressure;
        notifyObserver();       // уведомляем слушателей об изменениях
    }

}
interface Observer{
    public void update(int speed, int rpm, int oilPressure);
}
class Dashboard implements Observer{
    private Notifier notifier;
    private int speed;      // скорость
    private int rpm;        // обороты двигателя
    private int oilPressure;    // давление масла

    public Dashboard(Notifier notifier){
        this.notifier = notifier;
        notifier.addObserver(this); // регистрируем приборную панель в качестве наблюдателя
    }

    public void update(int speed, int rpm, int oilPressure) {
        this.speed = speed;
        this.rpm = rpm;
        this.oilPressure = oilPressure;
        show();
    }

    // отображение на приборной панели информации
    public void show(){
        System.out.println("Speed = " + speed + ", RPM = " + rpm +
                ", Oil pressure = " + oilPressure);
    }


}

public class Example {
    public static void main(String[] args) {
        CentralComp cp = new CentralComp(); //создаем центральный процессор
        Dashboard db = new Dashboard(cp);   //создаем приборную панель
```

```
        cp.changeData(10, 2000, 30);
        cp.changeData(20, 2500, 40);
        cp.changeData(60, 5000, 80);
    }

}
```

39. Паттерн «Одиночка», примеры кода

Singleton относится к порождающим паттернам. Его дословный перевод – одиночка. Этот паттерн гарантирует, что у класса есть только один объект (один экземпляр класса) и к этому объекту предоставляется глобальная точка доступа. Из описания должно быть понятно, что этот паттерн должен применяться в двух случаях:
когда в вашей программе должно быть создано не более одного объекта какого-либо класса. Например, в компьютерной игре у вас есть класс «Персонаж», и у этого класса должен быть только один объект описывающий самого персонажа.

когда требуется предоставить глобальную точку доступа к объекту класса. Другими словами, нужно сделать так, чтобы объект вызывался из любого места программы. И, увы, для этого не достаточно просто создать глобальную переменную, ведь она не защищена от записи и кто угодно может изменить значение этой переменной и глобальная точка доступа к объекту будет потеряна. Это свойства Singleton'а нужно, например, когда у вас есть объект класса, который работает с базой данных, и вам нужно чтобы к базе данных был доступ из разных частей программы. А Singleton будет гарантировать, что никакой другой код не заменил созданный ранее экземпляр класса.
Поведение Одиночки на Java невозможно реализовать с помощью обычного конструктора, потому что конструктор всегда возвращает новый объект. Поэтому все реализации Singleton’а сводятся к тому, чтобы скрыть конструктор и создать публичный статический метод, который будет управлять существованием объекта-одиночки и «уничтожать» всех вновь-появляющихся объектов. В случае вызова Singleton’а он должен либо создать новый объект (если его еще нет в программе), либо вернуть уже созданный.
Для этого: #1. – Нужно добавить в класс приватное статическое поле, содержащее одиночный объект:

```
public class LazyInitializedSingleton {
      private static LazyInitializedSingleton instance; //#1
}
```

#2. – Сделать конструктор класса (конструктор по-умолчанию) приватным (чтобы доступ к нему был закрыть за пределами класса, тогда он не сможет возвращать новые объекты):

```
public class LazyInitializedSingleton {
      private static LazyInitializedSingleton instance;
private LazyInitializedSingleton(){} // #2
}
```

#3. – Объявить статический создающий метод, который будет использоваться для получения одиночки:

```
public class LazyInitializedSingleton {
   private static LazyInitializedSingleton instance;
      private LazyInitializedSingleton(){}
      public static LazyInitializedSingleton getInstance(){ // #3
      if(instance == null){                  //если объект еще не создан
         instance = new LazyInitializedSingleton();       //создать новый объект
      }
      return instance;                       // вернуть ранее созданный объект
   }
}
```

40. Паттерн «Фасад», примеры кода

Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.

Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.

К примеру, программа, заливающая видео котиков в социальные сети, может использовать профессиональную библиотеку сжатия видео. Но все, что нужно клиентскому коду этой программы — простой метод `encode(filename, format)`. Создав класс с таким методом, вы реализуете свой первый фасад.

Паттерн Фасад — структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы. Несмотря на мудреное определение, наверное, самый простой шаблон проектирования и очень скоро вы в этом сами сможете убедиться.
Реализация: давайте попробуем разработать навигационную систему автомобиля. Перед поездкой водитель будет включать GPS, загружать данные о пробках, прокладывать путь и выключать после поездки. За каждое описанное действие будет отвечать отдельный класс, кроме включения и выключения. И так, класс управления электропитанием *GPSPower*:

```
1              class GPSPower{
2
3                public void powerOn(){
4                   System.out.println("Power ON");
5                }
6
7                public void powerOff(){
8                   System.out.println("Power OFF");
9                }
10             }
```

Класс содержит два метода, один включает GPS другой выключает.
Следующий наш класс будет отвечать за получение информации о пробках на дорогах *GPSNotifier*:

```
             class GPSNotifier{
2
3               public void downloadRoadInfo(){
4                  System.out.println("Downloading road information...");
5                  System.out.println("Download complete!");
6               }
7            }
```

Осталось только обработать полученную информацию о ситуации на дорогах и проложить оптимальный маршрут, этим будет заниматься класс *RoadAdvisor*:

```
Class RoadAdvisor{
public void route(){
      System.out.println("Create a route");
   }

}
```

Настал волнующий момент посмотреть, как это все работает:

```
   public static void main(String... args){
```

```
        GPSPower power = new GPSPower();
        GPSNotifier notifier = new GPSNotifier();
        RoadAdvisor advisor = new RoadAdvisor();

        //Водитель включает навигационную систему
        power.powerOn();
        //Водитель нажимает кнопку загрузки информации о дорогах
        notifier.downloadRoadInfo();
        //Водитель нажимает кнопку прокладки маршрута
        advisor.route();
        //Водитель выключает навигационную систему
        power.powerOff();
    }
```

Работает неплохо, но как вы можете заметить водитель приходится слишком много взаимодействовать с навигационной системой, если мы ничего не предпримем конкуренты нас смогут обойти. В этом нам и поможет паттерн Фасад, в качестве фасада, у нас будет выступать класс *GPSInterface*, который будет за водителя выполнять однотипные действия:

```
class GPSInterface{
    private GPSPower power;
    private GPSNotifier notifier;
    private RoadAdvisor advisor;

    public GPSInterface(GPSPower power, GPSNotifier notifier, RoadAdvisor advisor){
        this.power = power;
        this.notifier = notifier;
        this.advisor = advisor;
    }

    public void activate(){
        power.powerOn();
        notifier.downloadRoadInfo();
        advisor.route();
    }
}
```

В конструктор класса передаются все элементы, управление которыми мы хотим скрыть «за фасадом».
Метод *activate()* будет выполнять всю рутинную работу за водителя – включать систему, загружать информацию о ситуации на дорогах и прокладывать оптимальный маршрут и это все по нажатию одной кнопки! Давайте проверим как работает наш новый класс:

```
public static void main(String... args){
        GPSPower power = new GPSPower();
        GPSNotifier notifier = new GPSNotifier();
        RoadAdvisor advisor = new RoadAdvisor();

        GPSInterface gps = new GPSInterface(power, notifier, advisor);

        //Водитель включает навигационную систему
        gps.activate();
        //Водитель выключает навигационную систему
        power.powerOff();
    }
```

41. Паттерн «Стратегия», примеры кода

это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

```
package refactoring_guru.strategy.example.strategies;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;

/**
 * Конкретная стратегия. Реализует оплату корзины интернет магазина через
 * платежную систему PayPal.
 */
public class PayByPayPal implements PayStrategy {
    private static final Map<String, String> DATA_BASE = new HashMap<>();
    private final BufferedReader READER = new BufferedReader(new InputStreamReader(System.in));
    private String email;
    private String password;
    private boolean signedIn;

    static {
        DATA_BASE.put("amanda1985", "amanda@ya.com");
        DATA_BASE.put("qwerty", "john@amazon.eu");
    }

    /**
     * Собираем данные от клиента.
     */
    @Override
    public void collectPaymentDetails() {
        try {
            while (!signedIn) {
                System.out.print("Enter the user's email: ");
                email = READER.readLine();
                System.out.print("Enter the password: ");
                password = READER.readLine();
                if (verify()) {
                    System.out.println("Data verification has been successful.");
                } else {
                    System.out.println("Wrong email or password!");
                }
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    private boolean verify() {
        setSignedIn(email.equals(DATA_BASE.get(password)));
        return signedIn;
    }
```

```
    /**
     * Если клиент уже вошел в систему, то для следующей оплаты данные вводить
     * не придется.
     */
    @Override
    public boolean pay(int paymentAmount) {
        if (signedIn) {
            System.out.println("Paying " + paymentAmount + " using PayPal.");
            return true;
        } else {
            return false;
        }
    }

    private void setSignedIn(boolean signedIn) {
        this.signedIn = signedIn;
    }
}
```

strategies/PayByCreditCard.java: Оплата кредиткой

```
package refactoring_guru.strategy.example.strategies;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * Конкретная стратегия. Реализует оплату корзины интернет магазина кредитной
 * картой клиента.
 */
public class PayByCreditCard implements PayStrategy {
    private final BufferedReader READER = new BufferedReader(new InputStreamReader(System.in));
    private CreditCard card;

    /**
     * Собираем данные карты клиента.
     */
    @Override
    public void collectPaymentDetails() {
        try {
            System.out.print("Enter the card number: ");
            String number = READER.readLine();
            System.out.print("Enter the card expiration date 'mm/yy': ");
            String date = READER.readLine();
            System.out.print("Enter the CVV code: ");
            String cvv = READER.readLine();
            card = new CreditCard(number, date, cvv);

            // Валидируем номер карты...

        } catch (IOException ex) {
            ex.printStackTrace();
        }
```

```
    }

    /**
     * После проверки карты мы можем совершить оплату. Если клиент продолжает
     * покупки, мы не запрашиваем карту заново.
     */
    @Override
    public boolean pay(int paymentAmount) {
        if (cardIsPresent()) {
            System.out.println("Paying " + paymentAmount + " using Credit Card.");
            card.setAmount(card.getAmount() - paymentAmount);
            return true;
        } else {
            return false;
        }
    }

    private boolean cardIsPresent() {
        return card != null;
    }
}
```

42. Паттерн «Абстрактная фабрика», примеры кода

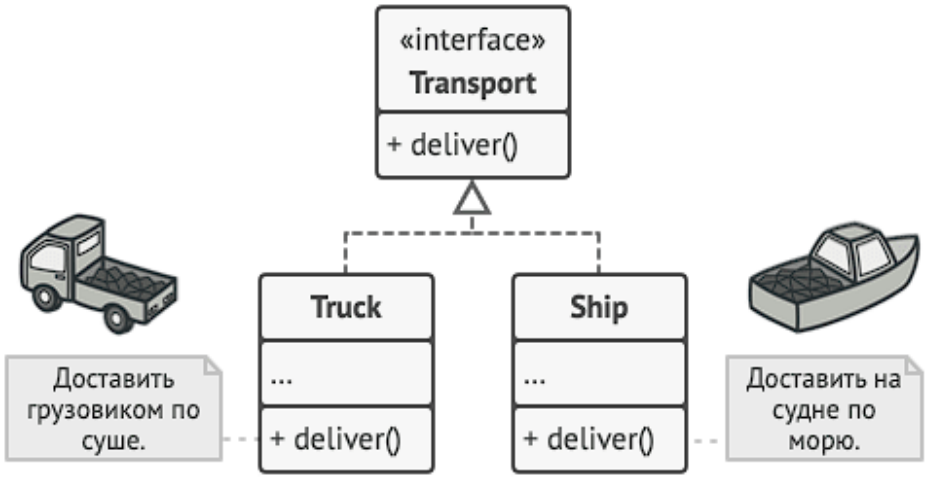
    Абстрактная фабрика задаёт интерфейс создания всех доступных типов продуктов, а каждая конкретная реализация фабрики порождает продукты одной из вариаций. Клиентский код вызывает методы фабрики для получения продуктов, вместо самостоятельного создания с помощью оператора `new`. При этом фабрика сама следит за тем, чтобы создать продукт нужной вариации.

```
package refactoring_guru.abstract_factory.example.factories;

import refactoring_guru.abstract_factory.example.buttons.Button;
import refactoring_guru.abstract_factory.example.checkboxes.Checkbox;

/**
 * Абстрактная фабрика знает обо всех (абстрактных) типах продуктов.
 */
public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}
```

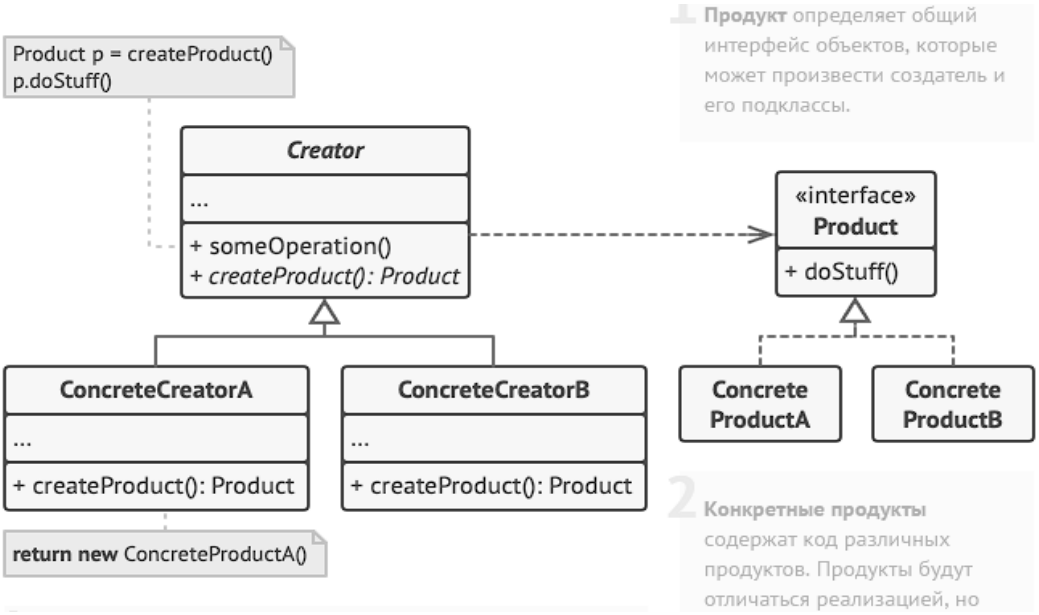
43. Паттерн «Абстрактный метод», примеры кода

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.
Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов особого фабричного метода.

Теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта. Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.



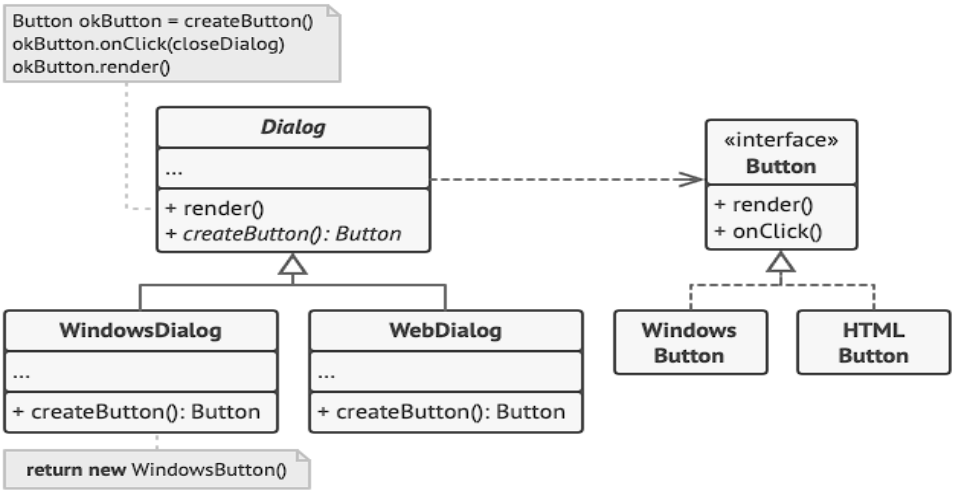
Все объекты-продукты должны иметь общий интерфейс.



Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.
Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.
Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.
Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.
Несмотря на название, важно понимать, что создание продуктов не является единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.
Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты.
Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

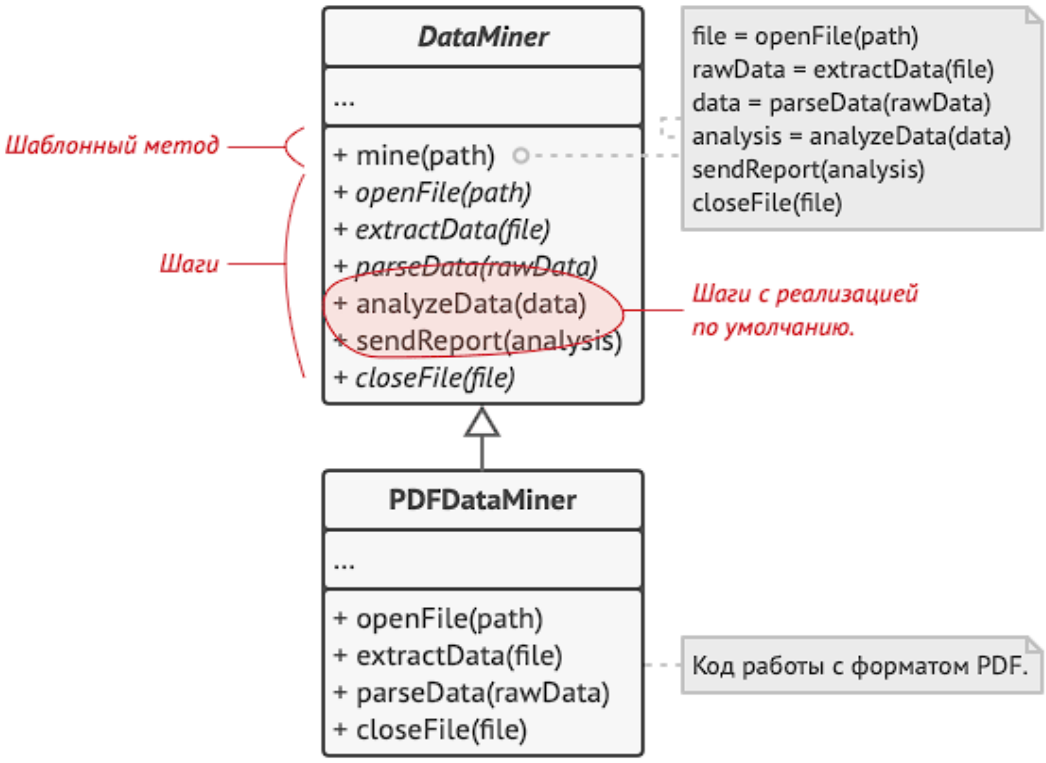
Псевдокод

В этом примере **Фабричный метод** помогает создавать кросс-платформенные элементы интерфейса, не привязывая основной код программы к конкретным классам элементов.



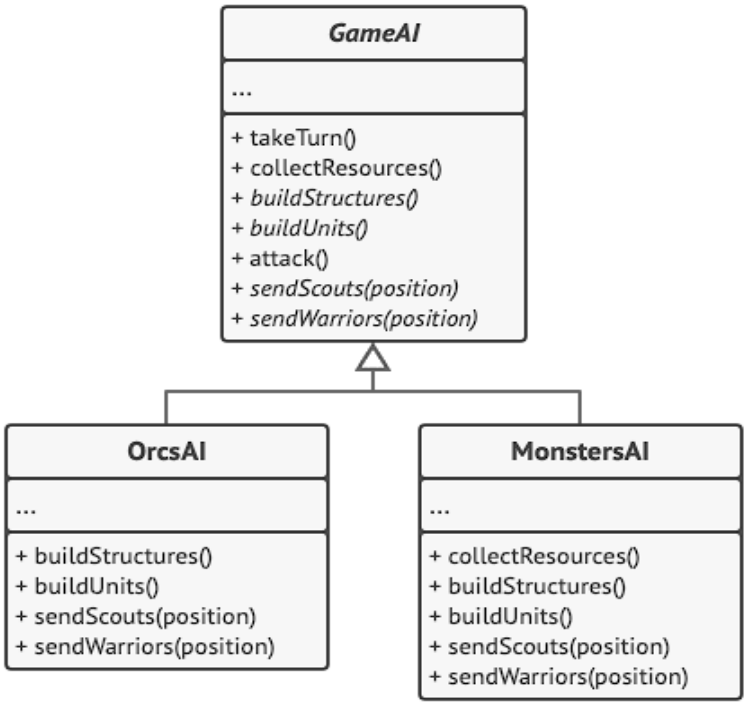
Пример кросс-платформенного диалога.

44. **Паттерн «Шаблонный метод», примеры кода**
    Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.
    Паттерн Шаблонный метод предлагает разбить алгоритм на последовательность шагов, описать эти шаги в отдельных методах и вызывать их в одном шаблонном методе друг за другом.
    Это позволит подклассам переопределять некоторые шаги алгоритма, оставляя без изменений его структуру и остальные шаги, которые для этого подкласса не так важны.


DataMiner
...
+ mine(path)
+ openFile(path)
+ extractData(file)
+ parseData(rawData)
+ analyzeData(data)
+ sendReport(analysis)
+ closeFile(file)
file = openFile(path)
rawData = extractData(file)
data = parseData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
Шаблонный метод
Шаги
Шаги с реализацией по умолчанию.
PDFDataMiner
...
+ openFile(path)
+ extractData(file)
+ parseData(rawData)
+ closeFile(file)
Код работы с форматом PDF.

Псевдокод

В этом примере **Шаблонный метод** используется как заготовка для стандартного искусственного интеллекта в простой игре-стратегии. Для введения в игру новой расы достаточно создать подкласс и реализовать в нём недостающие методы.



Пример классов искусственного интеллекта для простой игры.

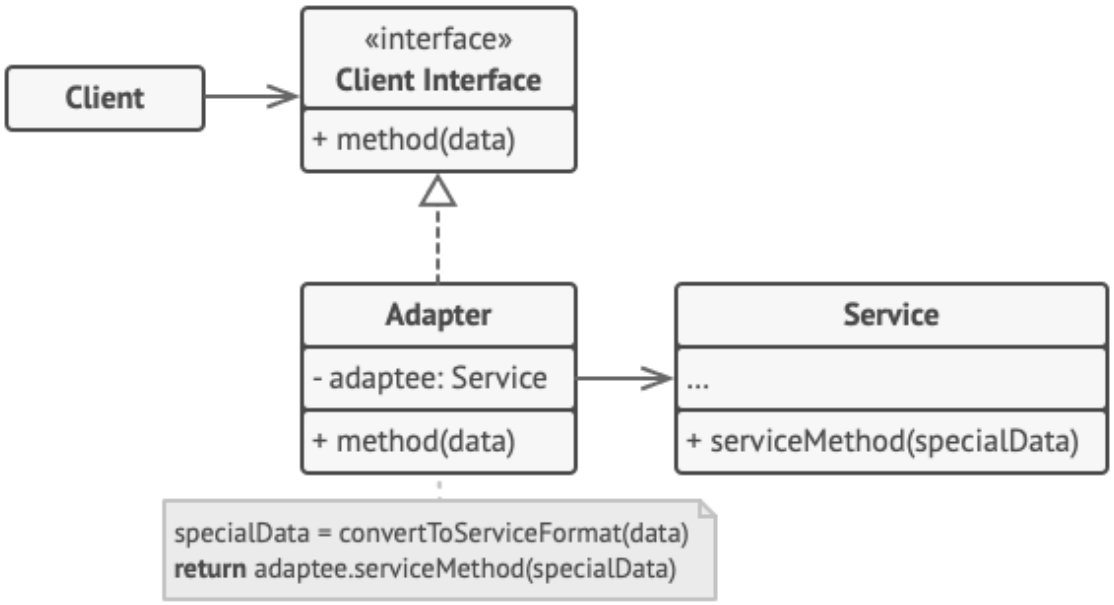
45. **Паттерн «Адаптер», примеры кода**

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.
Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.



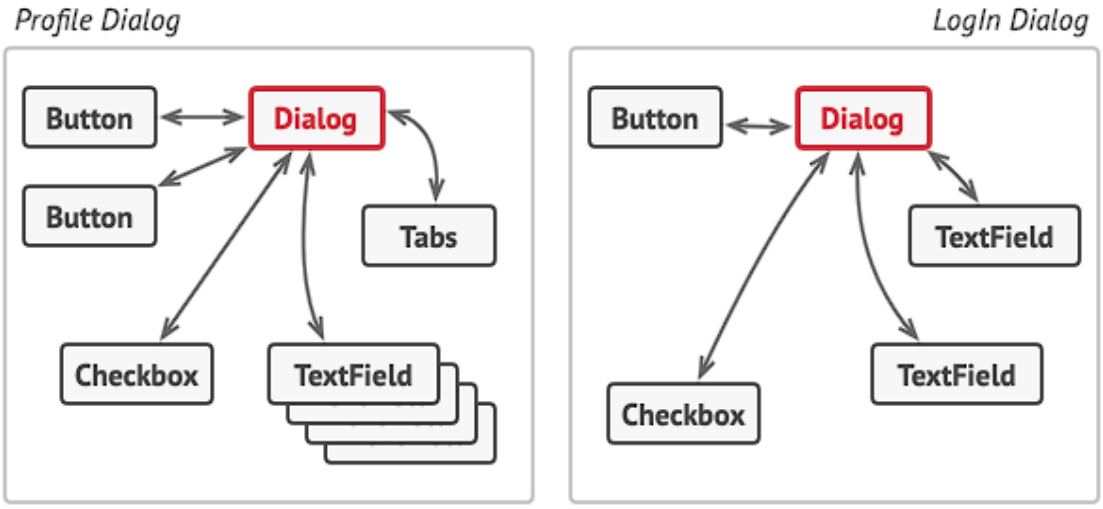
⚖ Преимущества и недостатки

✓ Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

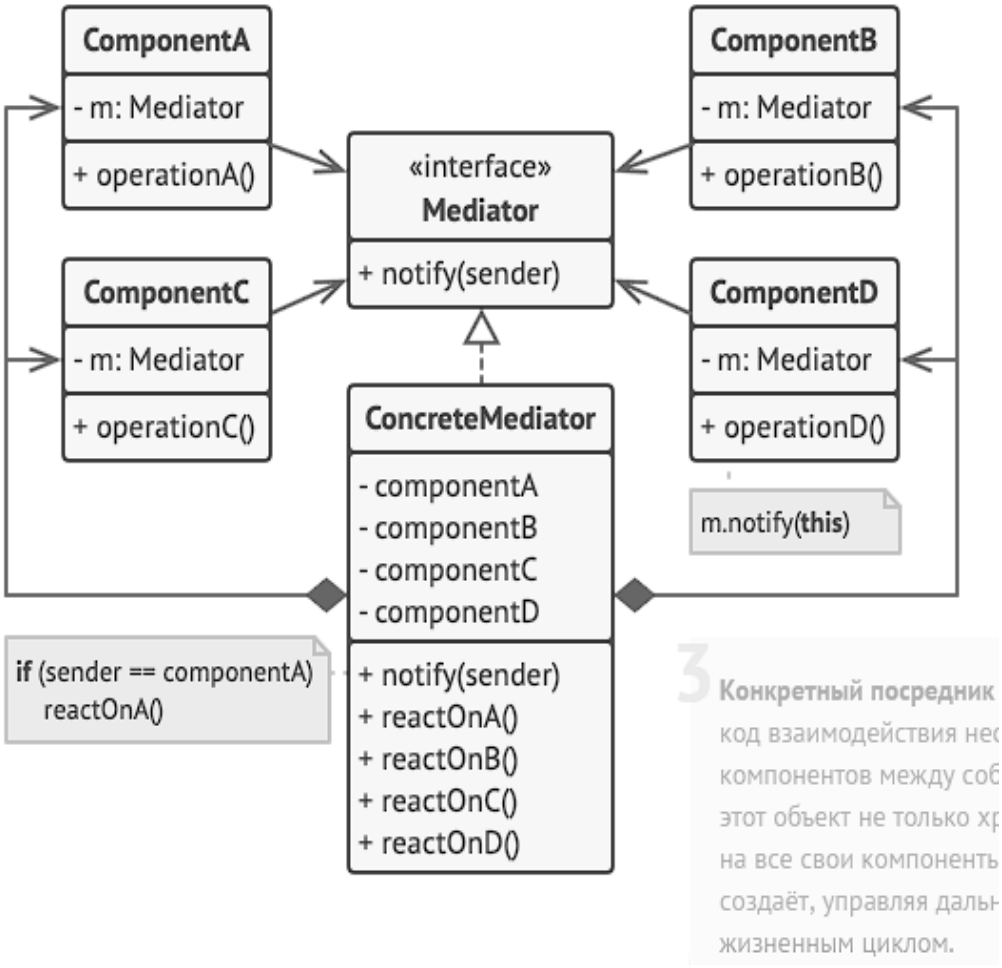
✕ Усложняет код программы из-за введения дополнительных классов.

46. Паттерн «Медиатор», примеры кода

Посредник(Mediator) — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.
Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.
В нашем примере посредником мог бы стать диалог. Скорее всего, класс диалога и так знает, из каких элементов состоит, поэтому никаких новых связей добавлять в него не придётся.



Элементы интерфейса общаются через посредника.



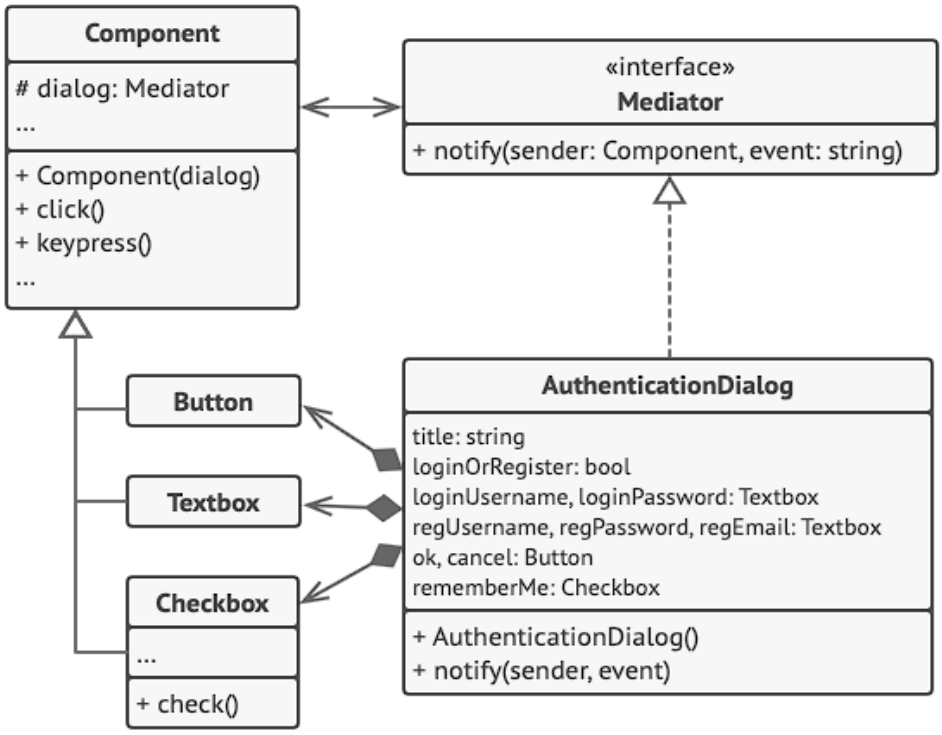
Компоненты — это разнородные объекты, содержащие бизнес-логику программы. Каждый компонент хранит ссылку на объект посредника, но работает с ним только через абстрактный интерфейс посредников. Благодаря этому, компоненты можно повторно использовать в другой программе, связав их с посредником другого типа.

Посредник определяет интерфейс для обмена информацией с компонентами. Обычно хватает одного метода, чтобы оповещать посредника о событиях, произошедших в компонентах. В параметрах этого метода можно передавать детали события: ссылку на компонент, в котором оно произошло, и любые другие данные.

Конкретный посредник содержит код взаимодействия нескольких компонентов между собой. Зачастую этот объект не только хранит ссылки на все свои компоненты, но и сам их создаёт, управляя дальнейшим жизненным циклом.

Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, он должен оповестить своего посредника, а тот сам решит — касается ли событие других компонентов, и стоит ли их оповещать. При этом компонент-отправитель не знает кто обработает его запрос, а компонент-получатель не знает кто его прислал.

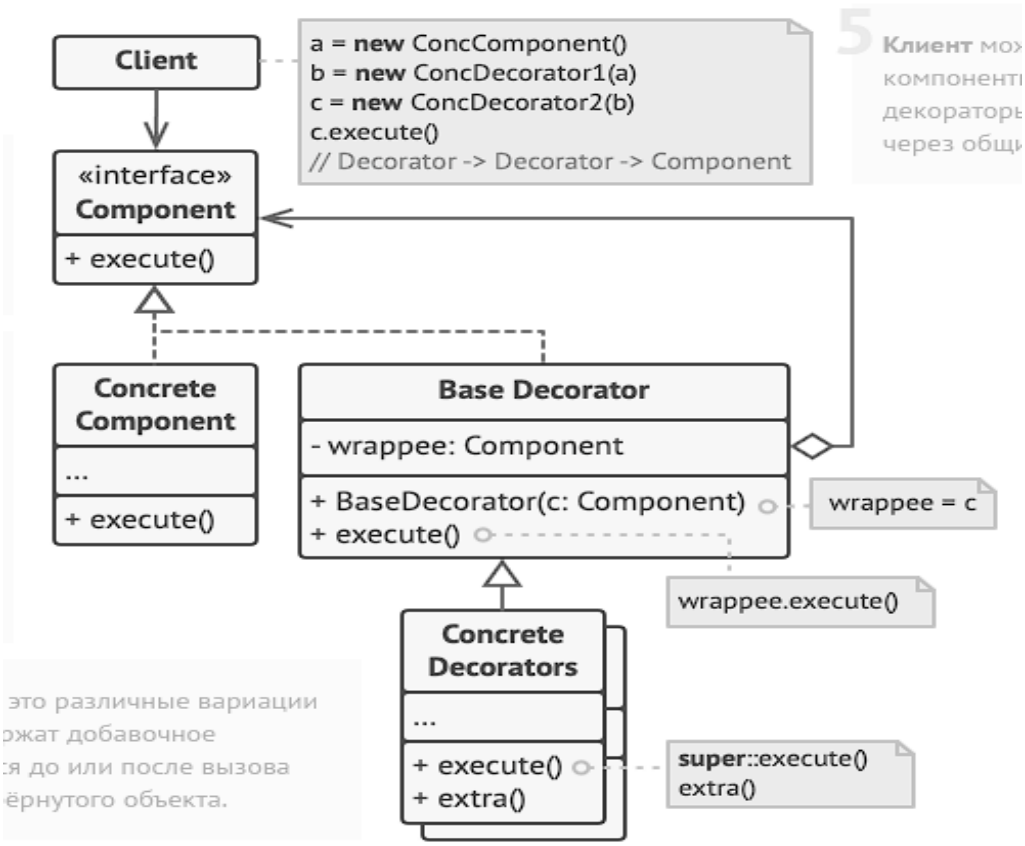
В этом примере **Посредник** помогает избавиться от зависимостей между классами различных элементов пользовательского интерфейса: кнопками, чекбоксами и надписями



Пример структурирования классов UI-диалогов.

47. **Паттерн «Декоратор», примеры кода**

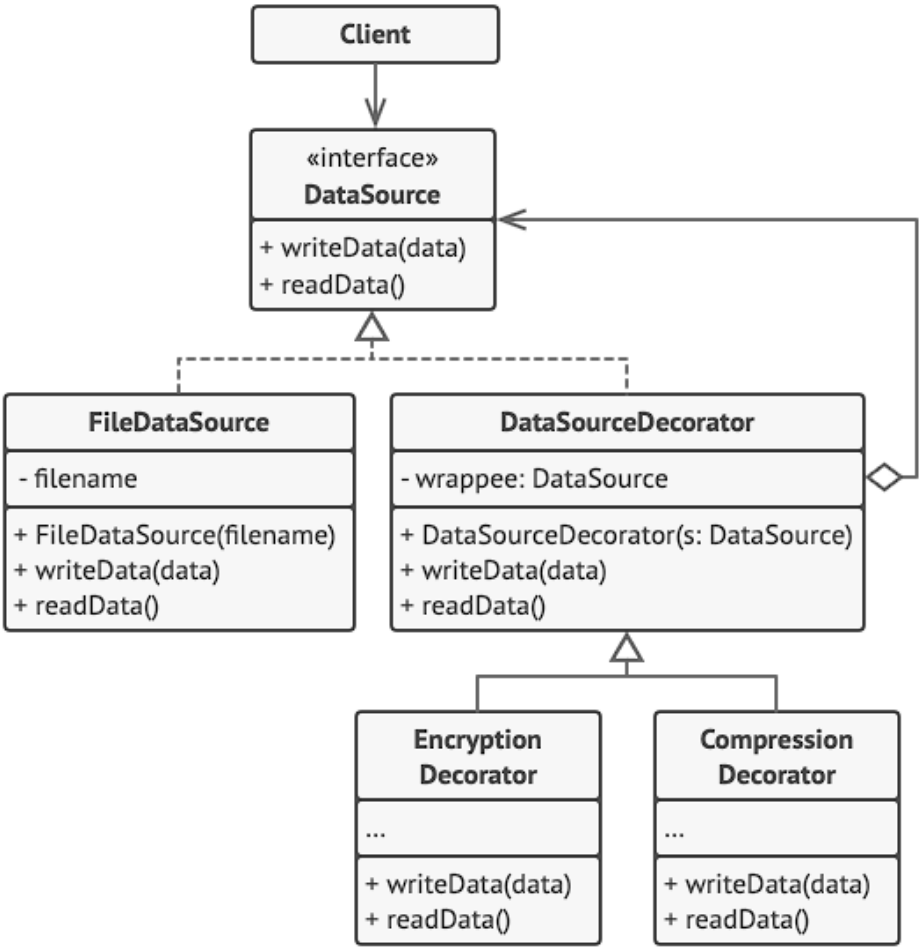
Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».
Декоратор основывается на механизме агрегации либо композиции . Это когда один объект содержит ссылку на другой и делегирует ему работу, вместо того чтобы самому наследовать его поведение. Декоратор имеет альтернативное название — обёртка. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё. Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу.



Компонент задаёт общий интерфейс обёрток и оборачиваемых объектов.
Конкретный компонент определяет класс оборачиваемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы.
Базовый декоратор хранит ссылку на вложенный объект-компонент. Им может быть как конкретный компонент, так и один из конкретных декораторов. Базовый декоратор делегирует все свои операции вложенному объекту.
Дополнительное поведение будет жить в конкретных декораторах.
Конкретные декораторы — это различные вариации декораторов, которые содержат добавочное поведение. Оно выполняется до или после вызова аналогичного поведения обёрнутого объекта.
Клиент может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

Псевдокод

В этом примере **Декоратор** защищает финансовые данные дополнительными уровнями безопасности прозрачно для кода, который их использует.



Пример шифрования и компрессии данных с помощью обёрток.