# Лабораторная работа № 8

**Тема работы: «Разработка программ, реализующих и использующих**

**интерфейсы»**

**Цель работы:** закрепить реализации интерфейсов и абстрактных классов.

**Задание:** создать и реализовать интерфейсы, также использовать наследование и полиморфизм для указанных предметных областей.

В классе должны быть реализованы:

* поля (закрытые или защищённые), нужное количество;
* свойства (для каждого поля);
* конструкторы (не менее двух);
* перегруженные операции;
* методы (нужное количество);
* метод toString();
* статические поля, если они нужны;
* статические свойства (если есть статические поля). Номер варианта соответствует вашему номеру по списку.

**Задание 1**

1. interface Издание <- abstract class Книга <- class Справочник и Энциклопедия
2. interface Абитуриент <- abstract class Студент <- class Студент-Заочник.
3. interface Сотрудник <- class Инженер <- class Руководитель.
4. interface Здание <- abstract class Общественное Здание <- class Театр.
5. interface Mobile <- abstract class Siemens Mobile <- class Model.
6. interface Корабль <- abstract class Военный Корабль <- class Авианосец.
7. interface Врач <- class Хирург <- class Нейрохирург.
8. interface Корабль <- class Грузовой Корабль <- class Танкер.
9. interface Мебель <- abstract class Шкаф <- class Книжный Шкаф.
10. interface Фильм <- class Отечественный Фильм <- class Комедия.
11. interface Ткань <- abstract class Одежда <- class Костюм.
12. interface Техника <- abstract class Плеер <- class Видеоплеер.
13. interface Транспортное Средство <- abstract class Общественный Транспорт <- class Трамвай.
14. interface Устройство Печати <- class Принтер <- class Лазерный Принтер.
15. interface Бумага <- abstract class Тетрадь <- class Тетрадь Для Рисования.
16. interface Источник Света <- class Лампа <- class Настольная Лампа.

**Задание 2**

Необходимо в задание 1 лабораторной работы №7: для всех цепочек классов связных наследованием, добавить абстрактные классы и интерфейсы, необходимые по логике.

**Оснащение работы**

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

**Основные теоретические сведения Абстрактные классы и методы**

Класс, содержащий абстрактные методы, называется абстрактным классом. Такие классы помечаются ключевым словом **abstract**.

Абстрактный метод не завершѐн. Он состоит только из объявления и не имеет тела:

abstract void yourMethod();

По сути, мы создаём шаблон метода. Например, можно создать абстрактный метод для вычисления площади фигуры в абстрактном классе Фигура. А все другие производные классы от главного класса могут уже реализовать свой код для готового метода. Ведь площадь у прямоугольника и треугольника вычисляется по разным алгоритмам и универсального метода не существует.

Если вы объявляете класс, производный от абстрактного класса, но хотите иметь возможность создания объектов нового типа, вам придётся предоставить определения для всех абстрактных методов базового класса. Если этого не сделать, производный класс тоже останется абстрактным, и компилятор заставит пометить новый класс ключевым словом **abstract**.

Можно создавать класс с ключевым словом **abstract** даже, если в нем не имеется ни одного абстрактного метода. Это бывает полезным в ситуациях, где в классе абстрактные методы просто не нужны, но необходимо запретить создание экземпляров этого класса.

В тоже время абстрактный класс не обязательно должен иметь только абстрактные методы. Напомню ещё раз, что если класс содержит хотя бы один абстрактный метод, то он обязан быть сам абстрактным.

Создавать объект на основе абстрактного класса нельзя.

Дописываем в созданную заготовку свой код для метода.

@Override void neigh() {

System.out.println("");

}

Обратите внимание, что абстрактный класс может содержать не только абстрактные, но и обычные методы.

Раннее мы создавали класс Фигура, у которого был метод вычисления площади фигуры. Метод ничего не делал, так как невозможно вычислить площадь неизвестной фигуры. Поэтому, этот метод можно сделать абстрактным, а в классах, производных от Фигуры, переопределить данный метод.

**Интерфейсы**

Ключевое слово **interface** используется для создания полностью абстрактных классов. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не тела методов.

Наличие слова **interface** означает, что именно так должны выглядеть все классы, которые реализуют данный интерфейс. Таким образом, любой код, использующий конкретный интерфейс, знает только то, какие методы вызываются для этого интерфейса, но не более того.

Чтобы создать интерфейс, используйте ключевое слово **interface** вместо **class**. Как и в случае с классами, вы можете добавить перед словом **interface** спецификатор доступа **public** (но только если интерфейс определен в файле, имеющем то же имя) или оставить для него дружественный доступ, если он будет использоваться только в пределах своего пакета. Интерфейс может содержать поля, но они автоматически являются статическими (**static**) и неизменными (**final**). Все методы и переменные неявно объявляются как **public**.

Класс, который собирается использовать определѐнный интерфейс, использует ключевое слово **implements**. Оно указывает, что интерфейс лишь определяет форму, а вам нужно наполнить кодом. Методы, которые реализуют интерфейс, должны быть объявлены как **public**.

Интерфейсов у класса может быть несколько, тогда они перечисляются за ключевым словом **implements** и разделяются запятыми.

Интерфейсы могут вкладываться в классы и в другие интерфейсы.

Если класс содержит интерфейс, но не полностью реализует определѐнные им методы, он должен быть объявлен как**abstract**.

Интерфейсы — это не классы. С помощью ключевого слова **new** нельзя создать экземпляр интерфейса:

х = new List(...); // Нельзя!

Но можно объявлять интерфейсные переменные:

List<String> catNames; // Можно!

При этом интерфейсная переменная должна ссылаться на объект класса, реализующего данный интерфейс.

List<String> catNames = new ArrayList<>();

Рассмотрим быстрый пример создания интерфейса. Выберите в меню **File**

**| New | Interface** и придумайте имя для нового интерфейса. В полученной заготовке добавьте два имени метода (только имена, без кода).

public interface SimpleInterface {

String getClassName(); int getAge();

}

Создайте или откройте какой-нибудь класс, к которому нужно применить интерфейс, и добавьте к нему **implements SimpleInterface**. Среда разработки подчеркнёт красной линией имя класса и предложит добавить методы, которые требуются интерфейсом. Соглашаемся и получаем результат:

public class Cat extends Animal implements SimpleInterface {

@Override

public String getClassName() { return null;

}

@Override

public int getAge() { return 0;

}

}

Среда разработки сгенерировала два метода и использовала в качестве возвращаемых результатов значения по умолчанию. Это могут быть и нулевые значения и **null**. Осталось подправить шаблоны созданных методов под свои задачи. Например, так:

@Override

public String getClassName() { return "Cat";

}

@Override

public int getAge() { return 5;

}

Первый метод возвращает имя класса, а второй - возраст кота (странно, что всем котам будет по пять лет, но это лишь пример).

Здесь важно понять роль интерфейса. Мы лишь придумываем имена, а класс уже реализует нужную задачу. Для примера можно создать в интерфейсе метод **play()** для класса Пианино и класса Гитара, так как играть можно на обеих инструментах. Но код в методах будет отличаться, так как принцип игры на инструментах совершенно разный.

**Константы в интерфейсах**

Интерфейсы можно использовать для импорта констант в несколько классов. Вы просто объявляете интерфейс, содержащий переменные с нужными значениями. При реализации интерфейса в классе имена переменных будут помещены в область констант. Поля для констант становятся открытыми и являются статическими и конечными (модификаторы **public static final**). При этом, если интерфейс не будет содержать никаких методов, то класс не будет ничего реализовывать. Хотя данный подход не рекомендуют использовать.

**Расширение интерфейсов**

Интерфейс может наследоваться от другого интерфейса через ключевое слово **extends**.

**Методы обратного вызова**

Интерфейсы часто используются для создания методов обратного вызова (callback). Рассмотрим такой пример. Создадим новый класс **SubClass** с интерфейсом **MyCallback**:

public class SubClass {

interface MyCallback{

void callBackReturn();

}

MyCallback myCallback;

void registerCallBack(MyCallback callback){ this.myCallback = callback;

}

void doSomething(){

// вызываем метод обратного вызова

myCallback.callBackReturn();

}

}

У интерфейса мы определили один метод **callBackReturn()**. Далее в классе мы создали объект интерфейса и инициализировали его в конструкторе класса. В классе также был создан метод **doSomething()**, в котором может содержаться какой-то сложный код. В конце метода вызывается метод интерфейса. В данном случае мы сами создали метод и знаем его код. Но во многих случаях, вы будете использовать готовый метод какого-то класса и вы не будете знать, что именно содержится в этом методе. Вам надо только знать, что такой метод существует, например, из документации и он выполняет конкретную задачу.

Переходим в код активности и подключаем интерфейс через ключевое слово **implements**:

public class MainActivity extends ActionBarActivity implements SubClass.MyCallback {}

Среда разработки поможет вставить шаблон метода интерфейса.

@Override

public void callBackReturn() {

}

Теперь мы можем использовать метод обратного вызова **callBackReturn()** для решения своих задач. Допустим у нас есть текстовая метка и кнопка. При щелчке выполняется какой-то сложный код из класса **SubClass**. Когда он закончит работу, то сработает метод обратного вызова **callBackReturn()**, в котором пропишем нужные действия.

Очень часто для интерфейса используют слово **Listener**.