Лекция 10

Интерфейсы: сравнение с абстрактным классом, множественное наследование.

Причины существования интерфейсов

- В широком смысле интерфейс какой-нибудь вещи – это механизм взаимодействия этой вещи с другими предметами. Например, пульт от телевизора – это дистанционный интерфейс. Собака понимает и исполняет команды — это значит, что собака поддерживает голосовой интерфейс (управления). Если все это подытожить, то можно сказать, что интерфейс – это стандартизированный способ взаимодействия двух вещей, и этот стандарт известен двум сторонам. Когда человек говорит собаке «сидеть», он отдает команду в соответствии с «голосовым интерфейсом управления собакой», и если собака выполняет эту команду, то мы говорим, что собака поддерживает этот интерфейс.

- Так же и в программировании. Методы – это действия над объектом, над его данными. И если класс реализует определенные методы, то он «поддерживает исполнение» определенных команд. Какие же преимущества дает объединение методов в интерфейс?

- **1)** Каждый **interface**, как и **class**, имеет уникальное имя. Обе стороны могут быть на 100% уверены, что вторая сторона поддерживает именно нужный (известный им) интерфейс, а не похожий.

- **2)** Каждый интерфейс налагает определенные ограничения на тот класс, который собирается поддерживать его. Класс сам решает (его разработчик), что он будет делать в случае вызова его методов, которые он унаследовал от интерфейса, но результат должен находиться в пределах ожиданий. Если мы скомандовали собаке «сидеть», и она покрутилась 5 минут на месте и села, то это — поддержка интерфейса. А если она вместо этого вцепилась вам в ногу, то ни о какой поддержке тут не может быть и речи. Выполнение команды не привело к ожидаемым результатам.

- Допустим, вы с друзьями участвуете в написании компьютерной игры. И вам досталась работа запрограммировать поведение одного персонажа. Один ваш коллега уже написал код по отображению всех персонажей на экран. Второй, отвечающий за сохранение игры на диск, написал код по сохранению всех объектов игры в файл. Каждый из них написал много кода и сделал интерфейс для взаимодействия с ним. Например, это может выглядеть так:

	Код на Java	Описание

1	**interface** Saveable
	{
	void saveToMap(Map<String, Object> map);
	void loadFromMap(Map<String, Object> map);
	}
		интерфейс по сохранению/загрузке объекта из map'а.

2	**interface** Drawable
	{
	void draw(Screen screen);
	}
		интерфейс по отрисовке объекта внутри переданного объекта screen.

3	**class** PacMan **implements** Saveable, Drawable
	{
	...
	}
		Ваш класс, реализующий поддержку двух интерфейсов.

- Другими словами, чтобы поддержать реализацию какого-то интерфейса (группы интерфейсов) в своем классе нужно:
- **1)** Унаследоваться от них
- **2)** Реализовать объявленные в них методы
- **3)** Методы должны делать то, для чего они предназначены.
- Тогда остальной код программы, который ничего не знает о вашем классе и его объектах, сможет успешно работать с ним.

Как пользоваться абстрактными классами

- Сложно представить аналогию абстрактного класса в реальной жизни. Обычно класс является моделью какой-нибудь сущности. Но абстрактный класс содержит не только реализованные методы, но и не реализованные. Что же это значит? Аналогом чего является абстрактный класс и есть ли у него аналоги в реальном мире?

- На самом деле есть. Представьте себе почти законченный кузов машины на конвейере. Туда могут поставить как спортивный двигатель, так и экономичный. Как кожаный салон, так и матерчатый. Конкретная реализация машины еще не определена. Более того, таких конкретных реализаций на основе этого кузова предполагается несколько. Но в таком виде машина никому не нужна. Это — **классический абстрактный класс**: его объекты не имеют смысла, поэтому их создание запрещено, класс имеет смысл, но только для его многочисленных полноценных наследников, которые будут созданы на его основе.

- Но могут быть и более абстрактные аналогии. Больше похожие на интерфейсы, с несколькими реализованными методами. Например, профессия **переводчик**. Без уточнения, с какого, и на какой язык, получим «**абстрактного переводчика в вакууме**». Или телохранитель. Про него может быть известно, что он владеет боевыми искусствами и может защитить клиента. Но какими именно единоборствами, и каким способом защитить клиента – это уже «особенности реализации» каждого конкретного телохранителя.

Код на Java	Описание

abstract class BodyGuard
{
abstract void applyMartialArts(Attacker attacker);
 	
void shoot(Attacker attacker)
{
gun.shoot(attacker);
}
 	
void saveClientLife(Attacker attacker)
{
if (attacker.hasGun())
shoot(attacker);
else
applyMartialArts(attacker);
}
}
	В классе «телохранитель» определено, как поступать в случае нападения: стрелять или применить боевые искусства.
	
	Но не определено, какие именно боевые искусства, хотя точно известно, что этот навык есть.
	Мы можем создать несколько разных телохранителей (унаследовав этот класс). Все они будут уметь защищать клиента и стрелять в нападающего.

Множественное наследование интерфейсов

- Представьте, что вы пишете компьютерную игру. И ее герои — ваши объекты — должны демонстрировать очень сложное поведение: ходить по карте, собирать предметы, выполнять квесты, общаться с другими героями, кого-то убивать, кого-то спасать. Допустим, вы смогли разделить все объекты на 20 категорий. Это значит, что если вам повезет, вы можете обойтись всего 20-ю классами, для их описания. А теперь вопрос на засыпку: сколько всего уникальных видов взаимодействия у этих объектов. Объект каждого типа может иметь уникальные взаимодействия с 20-ю видами других объектов (себе подобных тоже считаем). Т.е. всего нужно запрограммировать 20 на 20 — 400 взаимодействий! А если уникальных видов объектов будет не 20, а 100, количество взаимодействий может достигнуть 10,000!

- Очень часто можно упростить взаимодействие объектов, если взаимодействовать будут не объекты, а их роли и/или способности. А способности, как мы уже знаем, легко добавляются в класс, когда он реализует некоторый интерфейс.

- Когда пишется большая программа, обычно с этого сразу и начинают:
  1) Определяют все существующие способности/роли.
  2) Затем описывают взаимодействие между этими ролями.
  3) А потом просто наделяют все классы их ролями.

- Зная всего эти три роли (интерфейса) можно написать программу и описать корректное взаимодействие этих ролей. Например, объект будет гнаться (посредством интерфейса Moveable) за тем, «кого ты можешь съесть» и убегать от того, «кто может съесть тебя». И все это без знаний о конкретных объектах. Если в программу добавить еще объектов (классов), но оставить эти роли, она будет прекрасно работать – управлять поведением своих объектов.

	Код на Java	Описание
1	**interface** Moveable {}	— роль/способность передвигаться.
2	**interface** Eatable {}	— роль/способность быть съеденным.
3	**interface** Eat {}	— роль/способность съесть кого-нибудь.
4	**class** Tom **extends** Cat **implements** Moveable, Eatable, Eat {}	Tom – это кот, у которого есть три роли: 1) может передвигаться 2) может кого-то съесть 3) может быть съеденным кем-то (собакой)
5	**class** Jerry **extends** Mouse **implements** Moveable, Eatable {}	Jerry – это мышь, у которого есть две роли: 1) может передвигаться 2) может быть съеденным кем-то
6	**class** Killer **extends** Dog **implements** Moveable, Eat {}	Killer – это собака, у которого есть две роли: 1) может передвигаться 2) может кого-то съесть

Абстрактный класс vs. интерфейс

Абстрактный класс	Интерфейс
Наследование	
Абстрактный класс может унаследоваться только от одного класса и любого количества интерфейсов.	Интерфейс не может наследоваться от классов, но может от любого количества интерфейсов.
Абстрактные методы	
Абстрактный класс может содержать абстрактные методы. Но может и не содержать их вообще.	Все не статические и не default методы интерфейса – абстрактные – не содержат реализации. Интерфейс может не содержать никаких методов вообще.
Методы с реализацией	
Абстрактный класс может содержать методы с реализацией.	Интерфейс может содержать методы по умолчанию (default methods).
Данные	
Никаких ограничений.	Интерфейс содержит только public final static данные.
Создание объекта	
Нельзя создать объект абстрактного класса.	Нельзя создать объект интерфейса.

Стандартные интерфейсы: InputStream, OutputStream

- Объявлены они как абстрактные классы, но если начать разбираться, то можно увидеть, что **по своей сути — это интерфейсы.** Почти все их методы абстрактные, кроме нескольких незначительных методов. Очень похожи на нашего «телохранителя», которого мы рассматривали.

- Это очень интересные интерфейсы. Пока что я специально буду называть их интерфейсы, чтобы было понятно, зачем они нужны. А потом мы поговорим, почему же их все-таки сделали абстрактными классами.

- Есть такая интересная вещь в Java как «**поток**». **Поток** — это очень простая сущность. И его простота есть залог очень мощного механизма обмена данными. Потоки бывают двух видов: **поток для чтения и поток для записи.**

- В поток для записи **можно записывать данные**. Для этого у него есть метод **write**(). Из потока для чтения **можно данные читать**. Для этого у него есть метод **read**().

- **InputStream** – это интерфейс потока чтения, описывающий такую способность: «из меня можно читать байты».

- А **OutputStream**– это, соответственно, интерфейс потока записи, описывающий способность: «в меня можно записывать байты».

- В Java есть очень много классов, которые умеют работать с интерфейсами **InputStream** и **OutputStream**. Например, вы хотите прочитать файл с диска и вывести его содержимое на экран. Нет ничего проще.

- Для того, чтобы прочитать данные из файла на диске, есть специальный класс **FileInputStream**, который реализует интерфейс **InputStream**. Хотите записать прочитанные данные в другой файл? Для этого есть класс **FileOutputStream**, который реализует интерфейс **OutputStream**. Вот как выглядит код копирования [данных одного] файла в другой.

```
public static void main(String[] args) throws IOException
{
 InputStream inStream = new FileInputStream("c:/source.txt");
 OutputStream outStream = new FileOutputStream("c:/result.txt");

 while (inStream.available() > 0)
 {
  int data = inStream.read(); //читаем один байт из потока для чтения
  outStream.write(data); //записываем прочитанный байт в другой поток.
 }

 inStream.close(); //закрываем потоки
 outStream.close();
}
```

- Представим, что мы написали класс, и добавили ему способности **InputStream** и **OutputStream**.

- Если мы корректно реализовали поддержку этих интерфейсов, то объекты нашего класса теперь можно сохранить в файл на диске. Просто вычитав их содержимое через метод **read**. Или загрузить из файла, создав объект и записав в него содержимое файла через метод **write**.

Код на Java	Описание

class MyClass
{
private ArrayList<Integer> list;
}
	Для простоты представим, что наш класс содержит в себе один объект – ArrayList типа Integer.

- Теперь добавим в него методы read и write

- Это, конечно, не реализация интерфейсов InputStream и OutputStream, но очень похоже.

Код на Java	Описание

class MyClass
{
private ArrayList<Integer> list;
public void write(**int** data)
{
list.add(data);
}
public int read()
{
int first = list.get(0);
list.remove(0);
return first;
}
 	
public int available()
{
return list.size();
}
}
	Теперь у нас в классе реализован метод **read**, который позволяет последовательно вычитать все содержимое нашего списка **list**.
	
	И метод **write**, который позволяет записывать в наш list значения.

- Сохранение содержимого в файл:

Код на Java	Описание

public static void main(String[] args)
{
MyClass myObject = **new** MyClass();
OutputStream outStream = **new** FileOutputStream ("c:/my-object-data.txt");
 	
while (myObject.available() > 0)
{
int data = myObject.read(); //читаем один int из потока для чтения
outStream.write(data); //записываем прочитанный int в другой поток.
}
 	
outStream.close();
}
	Запись объекта MyClass в файл

- Сохранение содержимого в файл:

Код на Java	Описание

public static void main(String[] args)
{
InputStream inStream = **new** FileInputStream("c:/my-object-data.txt");
MyClass myObject = **new** MyClass();
 	
while (inStream.available() > 0)
{
int data = inStream.read(); //читаем один int из потока для чтения
myObject.write(data); //записываем прочитанный int в другой поток.
}
 	
inStream.close(); //закрываем потоки
}
	Чтение объекта MyClass из файла

Задачи

1. Реализуйте интерфейс Drink в классах **Fanta** и **Cola**.

```
public class Solution {
   public static void main(String[] args) throws Exception {
      print(new Fanta());
      print(new Cola());
   }

   private static void print(Drink drink) {
      System.out.println(drink.getClass().getSimpleName());
   }

   public interface Drink {
      boolean isOrange();
   }

   public static class Fanta {
   }

   public static class Cola {
   }
}
```

2. Создайте класс **Screen** и реализуйте в нем интерфейсы Selectable и Updatable. Не забудьте реализовать методы!

```
public class Solution {
   public static void main(String[] args) throws Exception {
   }

   interface Selectable {
      void onSelect();
   }

   interface Updatable {
      void refresh();
   }

   //напишите тут ваш класс
}
```

3. Исправьте код так, чтобы программа выполнялась и выводила фразу: "**Я переводчик с английского**". Метод main менять нельзя!

```
public class Solution {
   public static void main(String[] args) throws Exception {
      Translator translator = new Translator();
      System.out.println(translator.translate());
   }

   public abstract static class Translator {
      public String translate() {
         return "Я переводчик с английского";
      }
   }
}
```

4. Реализуйте в классе **Fox** интерфейс Animal. Поменяйте код так, чтобы в классе **Fox** был только один метод - getName. Учтите, что создавать дополнительные классы и удалять методы нельзя!

```
public class Solution {
   public static void main(String[] args) throws Exception {
   }

   public interface Animal {
      Color getColor();
   }

   public static class Fox {
      public String getName() {
         return "Fox";
      }
   }
}
```