Общие сведения

Интерфейс – описание тех методов, которые ДОЛЖЕН реализовывать класс, имплементирующий данный интерфейс.

Зачем это нужно:

- удобно при проектировании – описание поведения отделено от реализации;
- очень удобно в Test-Driven Development для эмуляции недостающего кода через т.н. «мок-объекты».

Пример

```
public interface SocialBehavior
{
 public String sayHello();
 public String hateSomebody(Person somebody);
 public String sayGoodBye();
}
```

Объявление

```
public class Person implements SocialBehavior
{
 // Здесь должна быть реализация
 // всех методов, объявленных
 // в интерфейсе
}
```

Имплементация (реализация)

2.5.1. Определение интерфейса

Интерфейсы в Java применяются для добавления к классам новых возможностей, которых нет и не может быть в базовых классах. Интерфейсы говорят о том, что класс может делать, но не говорят, как он должен это делать. Интерфейс только гарантирует (определяет контракт), какие методы должен выполнять класс, но как класс выполняет контракт интерфейс контролировать не может.

Определение интерфейса

Объявление интерфейса имеет вид:

[спецификаторы] interface имя_интерфейса
extends имя_базового_интерфейса {
/*объявление интерфейса*/
}

Поля интерфейса по умолчанию являются **final static**. Все методы по умолчанию открыты (**public**).

Example

```
public interface Square {
    double PI = 3.1415926;
      double square();
}
```

Задание: Создать классы Dog, Cat, Main и интерфейс Voice с методом doVoice(). В Dog и Cat имплементировать данный интерфейс и реализовать метод doVoice(). В классе Main создать через интерфейсную ссылку объекты Dog, Cat и вызвать метод Voice.

Методы интерфейсов по умолчанию

Java 8 позволяет вам добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово `default`. Эта фича также известна, как методы расширения. Вот наш первый пример:

```
interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

Кроме абстрактного метода `calculate` интерфейс `Formula` также определяет метод по умолчанию `sqrt`. Классы, имплементирующие этот интерфейс, должны переопределить только абстрактный метод `calculate`. Метод по умолчанию `sqrt` будет доступен без переопределения.

Использование интерфейсной ссылки:

```java
public interface Voice {
    void doVoice();
}
```

```java
public class Cat implements Voice {

    @Override
    public void doVoice() {
        System.out.println("Meow");
    }
}
```

```java
public class Dog implements Voice{

    @Override
    public void doVoice() {
        System.out.println("Gav");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.doVoice();   // Meow
        Dog dog = new Dog();
        dog.doVoice();   // Gav

        //Использование интерфейсной ссылки

        Voice cat1 = new Cat();
        cat1.doVoice();   // Meow

        Voice dog1 = new Dog();
        dog1.doVoice();   //Gav
    }
}
```

Класс Object

В Java есть специальный суперкласс **Object** и все классы являются его подклассами. Поэтому ссылочная переменная класса **Object** может ссылаться на объект любого другого класса. Так как массивы являются тоже классами, то переменная класса **Object** может ссылаться и на любой массив.

```
// применимо к любому классу
Object obj = new Cat("Barsik");
```

В таком виде объект обычно не используют. Чтобы с объектом что-то сделать, нужно выполнить приведение типов.

```
Cat cat = (Cat) obj;
```

У класса есть несколько важных методов.

- Object clone() - создаёт новый объект, не отличающий от клонируемого
- boolean equals(Object obj) - определяет, равен ли один объект другому
- void finalize() - вызывается перед удалением неиспользуемого объекта
- Class<?> getClass() - получает класс объекта во время выполнения
- int hashCode() - возвращает хеш-код, связанный с вызывающим объектом
- void notify() - возобновляет выполнение потока, который ожидает вызывающего объекта
- void notifyAll() - возобновляет выполнение всех потоков, которые ожидают вызывающего объекта
- String toString() - возвращает строку, описывающий объект
- void wait() - ожидает другого потока выполнения
- void wait(long millis) - ожидает другого потока выполнения
- void wait(long millis, int nanos) - ожидает другого потока выполнения

Методы **getClass()**, **notify()**, **notifyAll()**, **wait()** являются финальными и их нельзя переопределять.

2.3.15. Переопределение метода equals()

Метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае.

При переопределении должны выполняться соглашения:

- **рефлексивность** – объект равен самому себе;
- **симметричность** – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- **транзитивность** – если метод **equals()** возвращает значение true при сравнении объектов **x** и **y,** а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- **непротиворечивость** – при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- **ненулевая ссылка** при сравнении с литералом **null** всегда возвращает значение **false**.

2. Что такое метод `equals()`. Чем он отличается от операции ==.

Метод `equals()` обозначает отношение эквивалентности объектов. Эквивалентным называется отношение, которое является симметричным, транзитивным и рефлексивным.

- **Рефлексивность**: для любого ненулевого `x`, `x.equals(x)` вернет `true`;
- **Транзитивность**: для любого ненулевого x, y и z, если x.equals(y) и y.eqals(z) вернет true, тогда и x.equals(z) вернет true;
- **Симметричность**: для любого ненулевого `x` и `y`, `x.equals(y)` должно вернуть `true`, тогда и только тогда, когда `y.equals(x)` вернет `true`.

Также для любого ненулевого `x`, `x.equals(null)` должно вернуть false.
Отличия equals() от операции == в классе Object нет. Это видно, если взглянуть исходный код метода equals класса Object:

```
public boolean equals(Object obj) {
return (this == obj);
}
```

Однако, нужно не забывать, что, если объект ни на что не ссылается(null), то вызов метода equals этого объекта приведет к NullPointerException. Также нужно помнить, что при сравнении объектов оба они могут быть null и операция obj1 == obj2 в данном случае будет true, а вызов equals приведет к исключению NullPointerException.

Как мы видим, при помощи операции == сравниваются ссылки на объекты. Но мы можем переопределять метод equals, тем самым задавая логику сравнения двух объектов. Например, рассмотрим сравнение двух одинаковых чисел, созданных при помощи класса Integer:

```
Integer a = new Integer(6);
Integer b = new Integer(6);
System.out.println(a == b); // false т.к. это разные объекты с разными ссылками
System.out.println(a.equals(b)); // true, здесь уже задействована логика сравнения
```

Syhi-подсветка кода

Если взглянуть внутрь метода equals класса Integer, то мы увидим:

```
public boolean equals(Object obj) {
if (obj instanceof Integer) {
return value == ((Integer)obj).intValue();
}
return false;
}
```

Syhi-подсветка кода

Понятно, что тут уже нет сравнения ссылок, а сравниваются int значения.

3. **Что будет, если переопределить equals, но не переопределить `hashCode`?**

   Изначально `hashCode` — случайное число.

   Коллекции в Java перед тем как сравнить объекты с помощью `equals` всегда ищут/сравнивают их с помощью метода `hashCode()`. И если у одинаковых объектов будут разные `hashCode`, то объекты будут считаться разными — до сравнения с помощью `equals` просто не дойдет.

Метод hashCode()

Хеш-код - это целое число, генерируемое на основе конкретного объекта. Его можно рассматривать как шифр с уникальным значением.

Для вычисления хеш-кода в классе **String** применяется следующий алгоритм.

```
int hash = 0;
for(int i = 0; i < length(); i++)
        hash = 31 * hash + charAt(i);
```

У любого объекта имется хеш-код, определяемый по умолчанию, который вычисляется по адресу памяти, занимаемой объектом.

Значение хеш-кода возвращает целочисленное значение, в том числ и отрицательное.

Если в вашем классе переопределяется метод **equals()**, то следует переопределить и метод **hashCode()**.

2.3.16. Переопределение метода hashCode()

Метод **int hashCode()** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;
- все одинаковые по содержанию объекты одного типа должны иметь одинаковые хэш-коды;
- различные по содержанию объекты одного типа могут иметь различные хэш-коды.

Следует переопределять всегда, когда переопределен метод **equals()**.

Метод toString()

Очень важный метод, возвращающий значение объекта в виде символьной строки.

Очень часто при использовании метода **toString()** для получения описания объекта можно получить набор бессмысленных символов, например, [I@421199e8. На самом деле в них есть смысл, доступный специалистом. Он сразу может сказать, что мы имеем дело с одномерным массивом (одна квадратная скобка), который имеет тип **int** (символ I). Остальные символы тоже что-то означают, но вам знать это не обязательно.

Если же вам нужно научное объяснение, то метод работает по следующему алгоритму (из документации).

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Обычно принято переопределять метод, чтобы он выводил результат в читаемом виде.

2.3.17. toString()

Метод **toString()** следует переопределять таким образом, чтобы кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (то есть всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**.

Example

```
class Pen {
	private int price;
	private String producerName;

	public Pen(int price, String producerName) {
		this.price = price;
		this.producerName = producerName;
	}

	public boolean equals(Object obj) {
		if (this == obj) {
			return true;
		}
		if (null == obj) {
			return false;
		}
		if (getClass() != obj.getClass()) {
			return false;
		}

		Pen pen = (Pen) obj;
		if (price != pen.price) {
			return false;
		}
		if (null == producerName) {
			return (producerName == pen.producerName);
		} else {
			if (!producerName.equals(pen.producerName)) {
				return false;
			}
		}

		return true;
	}

	public int hashCode() {
		return (int) (31 * price + ((null == producerName) ? 0 : producerName
				.hashCode()));
	}

	public String toString() {
		return getClass().getName() + "@" + "price: " + price
				+ ", producerName: " + producerName;
	}

}
```

5. **Как правильно клонировать объект?**

Два типа клонирования.

Для клонирования объекта по умолчанию нужно:

a. Добавить интерфейс `Cloneable` своему классу
b. Переопределить метод `clone` и вызвать в нем базовую реализацию:

```
1  class Point implements Cloneable
2  {
3   int x;
4   int y;
5
6   public Object clone()
7   {
8    return super.clone();
9   }
10 }
```

Или можно написать реализацию метода `clone` самому:

```
class Point
{
 int x;
 int y;

 public Object clone()
 {
  Point point = new Point();
  point.x = this.x;
  point.y = this.y;
  return point;
 }
}
```

Задание 1: создать класс Person с 3-мя полями разного типа. Переопределить для этого класса equals, hachCode, toString.

Задание 2: добавить поле более сложного типа (Cat e.g.). Соответственно изменить вышеперечисленные методы

2.8. Классы внутри классов

В Java можно объявлять классы внутри других классов и даже внутри методов.

Они делятся на внутренние нестатические, вложенные статические и анонимные классы.

Такая возможность используется, если класс более нигде не используется, кроме как в том, в который он вложен.

2.8.1. Внутренние (inner) классы

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса.

Example

```
import java.util.Date;
public class Outer1 {
        private String str;
        Date date;

        Outer1() {
                str = "string in outer";
                date = new Date();
        }
        class Inner {
                public void method() {
                        System.out.println(str);
                        System.out.println(date.getTime());
                }
        }
}
```

Доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего класса.

Example

```
import java.util.Date;
public class Outer2 {
	private Inner inner;
	private String str;
	private Date date;

	Outer2() {
		str = "string in outer";
		date = new Date();
		inner = new Inner();
	}

	class Inner {
		public void method() {
			System.out.println(str);
			System.out.println(date.getTime());
		}
	}

	public void callMethodInInner() {
		inner.method();
	}
}
```

Создать у класса Фрукт(Fruit) внутренний класс Косточки(Pip) с полем amount и методом getAmount(). Вывести количество косточек на экран, используя внешний класс

Внутренние классы не могут содержать **static**-полей, кроме **final static**.

Example

```
import java.util.Date;
public class Outer3 {
	private Inner inner;
	private String str;
	private Date date;

	class Inner {
		private int i;
		public static int static_pole; // ERROR
		public final static int pubfsi_pole = 22;
		private final static int prfsi_polr = 33;

		public void method() {
			System.out.println(str);
			System.out.println(date.getTime());
		}
	}

	public void callMethodInInner() {
		inner.method();
	}
}
```

Доступ к таким полям можно получить извне класса, используя конструкцию:

имя_внешнего_класса.имя_внутреннего класса.имя_статической_переменной

```
Outer outer = new Outer();
System.out.println(Outer.Inner.pubfsi_pole);
```

Доступ к переменной типа **final static** возможен во внешнем классе через имя внутреннего класса.

Example

```
public class Outer5 {
        Inner inner;
        Outer5() {
                inner = new Inner();
        }
        class Inner {
                public final static int pubfsi_pole = 22;
                private final static int prfsi_polr = 33;
        }
        public void callMethodInInner() {
                System.out.println(Inner.prfsi_polr);
                System.out.println(Inner.pubfsi_pole);
        }
}
```

Предыдущее задание модифицировать таким образом, чтобы увидеть количество косточек, не используя методы.

Внутренние классы могут быть производными от других классов.
Внутренние классы могут быть базовыми (в пределах внешнего класса).
Внутренние классы могут реализовывать интерфейсы.

Если необходимо создать объект внутреннего класса где-нибудь, кроме метода внешнего класса, то нужно определить тип объекта как:

имя_внешнего_класса.имя_внутреннего_класса

Объект в этом случае создается по правилу:

ссылка_на_внешний_объект.new конструктор_внутренего_класса([параметры]);

```
Outer.Inner1 obj1 = new Outer().new Inner1();
Outer.Inner2 obj2 = new Outer().new Inner2();
obj1.print();
obj2.print();
```

Внутренний класс может быть объявлен внутри метода или логического блока внешнего класса; видимость класса регулируется видимостью того блока, в котором он объявлен; однако класс сохраняет доступ ко всем полям и методам внешнего класса, а также константам, объявленным в текущем блоке кода.

Example

```
public class Outer6 {
        public void method() {
                final int x = 3;
                class Inner1 {
                        void print() {
                                System.out.println("Inner1");
                                System.out.println("x=" + x);
                        }
                }
                Inner1 obj = new Inner1();
                obj.print();
        }


        public static void main(String[] args) {
                Outer6 out = new Outer6();
                out.method();
        }
}
```

Локальные внутренние классы не объявляются с помощью модификаторов доступа.

Example

```
public class Outer7 {
        public void method() {
                public class Inner1 {
                } // ОШИБКА
        }
}
```

Ссылка на внешний класс имеет вид:

имя_внешнего_класса.this

Example

```
public class Outer8 {
        int count = 0;

        class InnerClass {
                int count = 10000;

                public void display() {
                        System.out.println("Outer: " + Outer8.this.count);
                        System.out.println("Inner: " + count);
                }
        }
}
```

2.8.2. Статические (nested) классы

Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса.

Example

```
public class Outer9 {
        private int x = 3;

        static class Inner1 {
                public void method() {
                        Outer9 out = new Outer9();
                        System.out.println("out.x=" + out.x);
                }
        }
}
```

Вложенный класс имеет доступ к статическим полям и методам внешнего класса.

Example

```
public class Outer10 {
        private int x = 3;
        private static int y = 4;
        public static void main(String[] args) {
                Inner1 in = new Inner1();
                in.method();
        }
        public void meth() {
                Inner1 in = new Inner1();
                in.method();
        }

   static class Inner1 {
          public void method() {
                  System.out.println("y=" + y);
                  // System.out.println("x="+x); // ERROR
                  Outer10 out = new Outer10();
                  System.out.println("out.x=" + out.x);
          }
   }
}
```

Переделать предыдущий код так, чтобы внутренний класс стал вложенным статическим.

Статический метод вложенного класса вызывается при указании полного относительного пути к нему.

Example

```
public class Outer11 {
        public static void main(String[] args) {
                Inner1.method();
        }

        public void meth() {
                Inner1.method();
        }

        static class Inner1 {
                public static void method() {
                        System.out.println("inner static method");
                }
        }
}
```

Example

```
public class Outer12 {
        public static void main(String[] args) {
                Outer11.Inner1.method();
        }
}
```

Продемонстрировать вызов статического метода вложенного класса.

Подкласс вложенного класса не наследует возможность доступа к членам внешнего класса, которым наделен его суперкласс.

Example

```
public class Outer13 {
	private static int x = 10;

	public static void main(String[] args) {
		Inner1.method();
	}

	public void meth() {
		Inner1.method();
	}

	static class Inner1 {
		public static void method() {
			System.out.println("inner1 outer.x=" + x);
		}
	}
}
```

Example

```
public class Outer14 extends Outer13.Inner1{
	public static void main(String[] args) {
	}

	public void outer2Method() {
		// System.out.println("x="+x); // ERROR
	}
}
```

Сегодня мы изучили:

- Класс Object
- Объявление и реализация интерфейсов
- Использование ссылок на интерфейсы
- Переменные в составе интерфейсов
- Наследование интерфейсов
- Внутренние классы