Лекция №2
«Classes»

Автор: доц. каф. ПИ ХНУРЭ Колесников Д.О. v0.2

2.1. Оператор instaceof

Первым операндом бинарного оператора instanceof является *объект*, вторым – *имя класса, интерфейса или тип массива*.

```
boolean flag;
String s = "Hello!";
flag = s instanceof String; // flag = true
flag = s instanceof Object; // flag = true
flag = s instanceof Comparable; // flag = true

// String наследует Object, реализует Comparable
```

```
boolean flag;
int[] x = new int[3];
flag = x instanceof Object; // flag = true
flag = null instanceof Object; // flag = false
```

В качестве первого операнда может быть использован также null.

2.2. Конструкторы

Конструкторы классов представляют из себя специальные методы, которые имеют отличия от простых методов классов, не являющихся конструкторами.

свойство	***конструкторы***	***методы***
назначение	создает экземпляр класса	группирует операторы Java
допустимые модификаторы	*не допускается* **abstract**, **final**, **static**, **synchronized** и **native**	допускается **abstract**, **final**, **static**, **synchronized** и **native**
тип возвращаемого результата	не имеет типа возвращаемого результата, не может быть даже **void**	**void** или любой другой корректный тип

свойство	*конструкторы*	*методы*
имя	такое же имя как у класса (обычно существительное с заглавной буквы)	любое имя за исключением имени класса (обычно глагол с прописной буквы)
ссылка this	ссылается на другой конструктор в этом же классе;если используется, то обращение должно быть первой строкой конструктора	ссылается на экземпляр класса-владельца данного метода.
ссылка super	вызывает конструктор родительского класса; если используется, то вызов должен быть первой строкой конструктора	используется для вызова методов и полей родительского класса, которые перекрыты в данном классе

свойство	***конструкторы***	***методы***
наследование	конструкторы *не наследуются*	при наследовании классу потомку переходят методы предка
автоматическое добавление кода конструктора	если в классе не определен конструктор, компилятор автоматически добавит в код класса конструктор без параметров	-
автоматическое добавление вызова конструктора класса-предка	если конструктор не делает вызов конструктора с помощью ключевого слова super (с аргументами или без), компилятор автоматически добавит в код класса вызов конструктора предка без параметров.	-

2.3. Уровни доступа конструкторов

Конструкторы классов могут иметь любой из четырех уровней доступа: public, protected, default и private.

Private конструктор используется тогда, когда не желательно создание экземпляра конструктора с помощью оператора new. Объект такого класса создается при помощи статического метода класса, возвращающего объект данного класса.

Подобная схема применяется, например, в singleton классе, для которого может существовать не более одного экземпляра.

```
class A {
    // статическое поле – закрытый экземпляр класса:
    private static A instance = null;

    // private конструктор:
    private A() {}

    // возвращает единственный экземпляр класса A:
    synchronized public static A getA() {
        // создается instance если он еще не создан:
        if (instance == null) instance = new A();
        return instance;
    }
}
```

Создание объекта singleton-а:

```
A a = A.getA(); // a – экземпляр класса A

A b = A.getA(); // b – экземпляр класса A,
                // a и b ссылаются на один и тот же объект
```

Замечание. Инструкция **synchronized** в заголовке метода getA используется для того, чтобы не допустить выполнение тела этого метода одновременно несколькими потоками. Если не синхронизировать метод getA, то при определенных стечениях обстоятельств возможна ситуация, когда будут созданы два различных экземпляра класса A.

2.4. Ключевое слово this

Ключевое слово this используется в двух разных случаях:

для вызова конструктора класса из другого конструктора

как переменная ссылающаяся на экземпляр класса, который использует эту ссылку.

```
class A {
    private int x;
    public A() {
        // вызов конструктора:
        this(0);
    }
    public A(int x) {
        // использование ссылки на экземпляр класса:
        this.x = x;
    }
}
```

Замечание. Вызов конструктора с помощью this(...) может осуществляться *только* из конструктора и *должен быть* всегда первой строкой в нем.

2.5. Default конструктор (конструктор с уровнем доступа «пакет»)

Default конструктор, т.е. конструктор, перед которым не стоит никакой модификатор уровня доступа, может быть вызван только из кода, который относится к тому же пакету.

Из других пакетов данный конструктор видим не будет.

2.6. Перегрузка методов

Перегрузить (overload) метод класса – это значит объявить в классе еще один метод с таким же именем, но с другим списком параметров. Не допускается наличие в классе двух методов с одним и тем же именем и списком параметров (в независимости от возвращаемого типа).

```
class A {
    public void method() {}
    public void method(int x) {}

    // ошибка компиляции
    // «method(int) is already defined in A»:
    public float method(int y) {}
}
```

Замечание. Если перегружаемый метод достался классу от класса предка, то допускается объявление метода с таким же именем и таким же списком параметров, при этом происходит *перекрытие* метода: метод потомка перекрывает метод предка, последний при этом доступен в классе потомке через конструкцию super.

Замечание. Конструкторы класса (если их несколько) являются перегруженными.

2.7. Перекрытие методов

Методы и поля класса предка могут быть перекрыты в классе потомке.

При этом перекрытые поля и методы доступны с помощью конструкции super (учитывая уровень их доступа: private элементы не доступны в потомке, а default элементы доступны только если класс потомок принадлежит тому же пакету).

```
class A {
    protected int x;
    public A() {x = 7;}
    int getX() {return x;}
}
class B extends A {
    public int x = 1; // поле x класса A перекрыто
    public B() {
        // ошибка если A и B опред. в разных пакетах:
        x = super.getX(); // x = 7
        x = getX(); // x = 1;
        x = super.x; // x = 7
    }
    // метод getX класса A перекрыт:
    public int getX() {return x;}
}
```

2.8. Преобразование типов между классами

Между классами, которые находятся в отношении наследования возможно преобразование типов двух видов: *восходящее* и *нисходящее*.

Восходящее преобразование типов это преобразование от потомка к предку.

Может выполняться *неявно* и иллюстрирует одну из основных особенностей ООП: потомок может заменить предка в любом контексте.

После восходящего преобразования типов те поля и методы потомка, которые отсутствуют в классе предке становятся не доступными.

```
class A {}
class B extends A {
    public int get() {return 7;}
}
...
B b = new B();
A a = b; // эквивалентно: A a = (A)b;

// ошибка, т.к. класс A не содержит метод getX():
int x = a.get();
```

Однако, никакого усечения объекта не происходит, при обратном, *нисходящем* преобразовании типов, которое *всегда должно выполняться явно*, поля и методы снова становятся доступными.

```
...
B b = new B();
A a = b;
b = (B)a;
int x = b.get(); // x = 7
```

Нисходящее преобразование типов может осуществляться от класса предка к классу, который находится в любом узле иерархии наследования от предка к потомку, при этом методы и поля которые были определены при дальнейшем наследовании данного класса вплоть до конечного потомка будут недоступны.

```
class A {}
class B extends A {}
class C extends B {}
...
// c – экземпляр класса потомка:
C c = new C();

// a – экземпляр класса C, в котором скрыты все поля и методы отсутствующие в A:
A a = c;

// b – экземпляр класса C, в котором скрыты все поля и методы отсутствующие в B:
B b = (B)c;
```

Замечание. При нисходящем и восходящем преобразованиях типов фактически тип объекта (т.е. класс, экземпляром, которого он является) не меняется.

```
class A {}
class B extends A {}
class C extends B {}
...
C c = new C();
A a = c; // восходящее преобразование типов
String s = a.getClass().getName(); // s = "C"
boolean flag = a instanceof C; // flag = true
// нисходящее преобразов. типов в промежуточный тип:
B b = (B)a;
s = b.getClass().getName(); // s = "C"
flag = b instanceof C; // flag = true
```

О том, что подобные манипуляции с объектами и объектными переменными справедливо называть *преобразованием типов* говорит доступность только той реализации, которая была определена на соответствующем уровне иерархии наследования (т.е. в том классе, к типу которого «преобразовали» объект), код, который был определен в классах от данного уровня иерархии до конечного потомка, становится недоступным; чтобы получить доступ к этому коду следует выполнить соответствующее *нисходящее* преобразование.

Фактически нисходящее и восходящее преобразования типов являются преобразованиями *без потери информации*.

Замечание. Нисходящее преобразование типов всегда должно выполняться явно с использованием оператора преобразования типов. При этом для некоторых преобразований возможность их осуществления не проверяется на этапе компиляции. В таком случае, если преобразование невозможно, JVM выбросит исключение времени выполнения java.lang.ClassCastException. Чтобы избежать подобных ситуаций, перед преобразованием следует сделать проверку на возможность его осуществления с помощью оператора instanceof.

```
B b = null;
if (a instanceof B) b = (B)a;
```

2.9. Влияние уровня доступа конструктора на возможность наследовать класс

Уровень доступа и наличие/отсутствие в классе *конструкторов с параметрами и без* влияет на возможность наследовать данный класс.

конструктор без параметров	***конструкторы с параметрами***	***возможность наследовать данный класс***
отсутствует	присутствуют	+ если конструктор потомка вызывает один из конструкторов предка с параметрами через конструкцию super
	отсутствуют	+

конструктор без параметров	*конструкторы с параметрами*	*возможность наследовать данный класс*
public	не имеет значения	+
protected	не имеет значения	+
default	присутствуют	+ если конструктор потомка вызывает один из конструкторов предка с параметрами через конструкцию super
	отсутствуют	+ если потомок из того же пакета

конструктор без параметров	*конструкторы с параметрами*	*возможность наследовать данный класс*
private	присутствуют	+ если конструктор потомка вызывает один из конструкторов предка с параметрами через конструкцию super
	отсутствуют	-

2.10. Инициализация нестатических полей

В Java существует три механизма инициализации *нестатических* полей:

1) присваивание полю значения при его объявлении;
2) в конструкторе;
3) в *блоках инициализации*.

Блок инициализации выполняется каждый раз при создании нового экземпляра класса. Присваивания значений полю выполняется сверху вниз по мере их появления, причем присваивания в объявлении и в блоках инициализации (которых может быть несколько) имеют равный приоритет.

```
class A {
    { // блок инициализации
        x = 1; // значение поля x равно 1
    }
    private int x = 3; // x становится равным 3

    // еще один блок инициализации:
    {x = 5;} // значение поля x становится равным 5
}
```

В блоках инициализации не допускается использование в правых частях присваиваний неинициализированных полей.

```
class A {
    // ошибка компиляции (illegal forward reference):
    {x = y + 1;}
    private int x = 3;
    private int y = 2;
}
```

Замечание. Присваивание полю значения в блоке инициализации, который предшествует объявлению поля ошибки не вызывает, т.к. само объявление с инициализацией в одной строке на самом деле выполняется в два прохода.

Первый проход – определяется, что данное поле объявлено в классе, второй проход – сверху вниз, учитывая присваивания в блоках инициализации и в объявлении поля, осуществляется инициализация поля.

```
private int x = 3;
```
эквивалентно
```
private int x;
{x = 3;}
```

Замечание. В блоках инициализации можно инициализировать статические поля, при этом их значения будут обновляться при каждом создании нового экземпляра класса.

Также в них можно вызывать методы данного класса и других классов, в том числе статические, но в них нельзя передавать неинициализированные еще поля.

2.11. Инициализация статических полей

Для инициализации *статических* полей используется присваивание полю значения при его объявлении и присваивания в *статических блоках инициализации*. Статический блок инициализации выполняется один раз при первом обращении к классу (или при загрузке класса). Инициализация статических полей выполняется сверху вниз, учитывая присваивания в объявлении полей и в статических блоках инициализации.

```
class A {
    static {x = 1;} // x = 1
    public static int x = 2; // x = 2
    static {x = 3;} // x = 3
}
```

2.12. Порядок вызова блоков инициализации и конструкторов при создании объекта

Статические и нестатические блоки инициализации а также конструкторы классов при наличии иерархии наследования вызываются в строго определенном порядке.

Пусть A → B → ... → Z иерархия наследования, в которой класс A является суперклассом для всех остальных классов B, ..., Z. При создании экземпляра класса Z происходит следующее:

1) вызываются статические блоки инициализации всех классов от предка к потомку (от A до Z), при этом если ранее уже был создан экземпляр некоторого класса из иерархии (или этот класс был загружен), то статические блоки инициализации будут вызываться начиная от потомка этого класса и заканчивая классом Z;

2) для всех классов от предка к потомку вызывается вначале блок инициализации, затем тело конструктора.

```
class A {
    {...} // 1 – блок инициализации класса A
    static {...} // 2 – статич. блок инициализ. класса A
    A() {...} // 3 – конструктор класса A
}
class B extends A {
    {...} // 4 – блок инициализации класса B
    static {...} // 5 – статич. блок инициализ. класса B
    B() {...} // 6 – конструктор класса B
}
...
new A(); // будут выполнены строки: 213
new B(); // будут выполнены строки: 51346, т.к. блок инициализации 2 был уже выполнен ранее
```

2.13. Значение полей по умолчанию

Значения неинициализированных полей класса выставляются следующим образом по умолчанию:

тип: byte, short, int, long, floag, double
значение: 0

тип: char
значение: '\u0000'

тип: boolean
значение: false

тип: ссылочная (объектная) переменная
значение: null

Замечание. Вплоть до первой инициализации поля его значение равно значению по умолчанию начиная от строки с его объявлением.

```
class A {
    int x; // начиная с этой строки x = 0
    ...
    {x = 7;} // начиная с этой строки x = 7
}
```

Выше объявления поля можно производить его инициализацию в блоках инициализации, но нельзя использовать значение, которое оно получает в этих блоках.

```
class A {
    {x = 7;} // x = 7, здесь ошибки нет

    {y = x;} // ошибка: выше объявления поля
             //его значение использовать нельзя
    int x, y;
}
```

Замечание. Элементы массивов являющихся в java объектами, можно рассматривать как поля таких объектов, поэтому при выделении памяти для массива с помощью оператора new (т.е. при создании объекта-массива) элементам массива присваиваются соответствующее значение по умолчанию (в зависимости от типа массива), которые были приведены выше в таблице.

```
String[] s = new String[3];
boolean[] f = new boolean[5];

String s = s[1]; // s = null
boolean f = f[2]; // f = false
```

2.14. Абстрактные классы

Абстрактный метод – это метод не имеющий реализации. Такие методы должны быть объявлены с модификатором *abstract*.

Если класс содержит хотя бы один абстрактный метод, он также должен быть объявлен с модификатором *abstract*.

```
public abstract class test {
    abstract void someMethod();
}
```

Замечание. Модификатор *abstract* должен стоять перед ключевым словом *class*, но может стоять после модификатора уровня доступа класса или метода.

Замечание. Для классов не допускается сочетание *abstract* с модификатором *final*. Для методов не допускается сочетание *abstract* с модификаторами *final*, *private* и *static*.

2.15. Вызов конструкторов в классе

В коде класса конструктор класса может быть вызван только из конструктора и только с помощью ключевого слова this, за которым в скобках должны следовать параметры соответствующего конструктора, причем такой вызов должен быть первой строкой в конструкторе.

Замечание. Из конструктора может быть вызван только один конструктор из оставшихся. Не допускается вызов сразу нескольких конструкторов, например двух.

```
class A {
    A() {
        this(3);

        this(3.); // ошибка: call to this must
            //be first statement in constructor
    }
    A(int x) {}
    A(float x) {}
}
```

Замечание. В коде класса можно создавать экземпляры этого класса с помощью оператора *new*, при этом всегда будет происходить выполнение соответствующего конструктора.

```
class A {
    A(int x) {...}
    public void someMethod() {
        A a = new A(3);
    }
}
```

2.16. Конструктор без параметров в контексте наследования

Если в классе определен хотя бы один конструктор, имеющий один или более параметров, то компилятор в код класса конструктор без параметров *не добавляет*.

Поэтому потомок такого класса *обязан* включать в себя конструктор (с параметрами или без), причем один из его конструкторов должен вызывать конструктор с параметрами предка через конструкцию *super*. Это связано с тем, что конструктор потомка всегда вызывает конструктор предка, если же такой вызов не прописан явно, компилятор добавит в код конструктора потомка вызов *конструктора без параметров* предка.

```
class A {
    A(int x) {}
}

// код класса C написан без ошибок:
class C extends A {
    C() {super(0);}
}

// ошибка компиляции
// в классе A отсутствует конструктор A()
class B extends A {
    B() {}
}
```

2.17. Уровни доступа класса

Класс может иметь два уровня доступа:

открытый - public (имя класса предваряется модификатором **public**)

по умолчанию – default (модификатор уровня доступа не ставится). Каждый отдельный java-файл может содержать несколько классов, но только один из них может быть public (все классы в файле могут быть default).

Замечание. Если класс имеет уровень доступа по умолчанию, то вне пакета в котором он определен, данный класс виден *не будет*.

2.18. Обращение к статическим полям и методам

В статических методах нельзя вызывать нестатические методы и использовать нестатические переменные, т.к., статический метод является принадлежностью класса (который один), а нестатические методы и поля принадлежат объектам этого класса (которых может быть много).

2.19. Доступ к переменной this

Статические методы не имеют доступа к переменной this.

```
class A {
    int x;
    public static void method() {
        // ошибка компиляции:
        this.x = 3;
    }
}
```

2.20. Перегрузка статических методов при наследовании

Статические методы могут быть перегружены нестатическими методами при наследовании.

```
class A {
    public static void method() {}
}

class B extends A {
    public void method(int x) {}
}
```

2.21. Перекрытие статических методов при наследовании

Статические методы не могут быть перекрыты нестатическими методами при наследовании.

```
class A {
    public static void method() {}
}

class B extends A {
     // ошибка компиляции:
    public void method() {}
}
```

2.22. Абстрактные статические методы

Статические методы не могут быть абстрактными.

```
abstract class test {
     // ошибка компиляции:
    abstract public static void method();
}
```

2.33. Доступ к перекрытым методам родительского класса

С помощью ключевого слова *super* можно получить доступ к перекрытым методам родительского класса (непосредственного предка)

Замечание. С помощью *super* нельзя получить доступ к перекрытым методам класса, который находится более чем на одну ступень выше по иерархии наследования.

2.34. Наследование абстрактного класса

Класс, который наследует абстрактный класс должен или реализовать все его абстрактные методы или, в противном случае, должен быть объявлен абстрактным.

```
abstract class A {
    abstract public void m();
}

// ошибка компиляции:
class B extends A{}
```

2.35. Объектные переменные абстрактных классов

Нельзя создать экземпляр абстрактного класса, однако можно объявить объектную переменную этого класса и заставить ссылаться ее на объект - экземпляр класса, который наследует и реализует данный абстрактный класс.

```
abstract class A {
    abstract public void method();
    public void print() {
        System.out.println("A");
    }
}

class B extends A {
    public void method(){}

    public static void main(String[] argv) {
        A a = new B();
        a.print(); // будет напечатано A
    }
}
```

Замечание. Абстрактные классы могут быть интерпретированы как интерфейсы, у которых часть методов уже реализована.

Переменная, которая имеет тип абстрактного класса, может рассматриваться как интерфейсная переменная.

2.36. Уровни доступа и перекрытие методов

При наследовании уровень доступа к перекрываемому методу не может быть сужен.

```
class A {
    // уровень доступа protected:
    protected void method() {}
}

class B extends A {
     // ошибка компиляции, уровень доступа default
    // менее "доступен", чем protected:
    void method() {}
}
```

2.37. Уровни доступа элементов классов

Элементы класса (методы и поля) могут иметь 4 уровня доступа, которые располагаются в порядке понижения открытости следующим образом:

public
protected
default
private.

Замечание. В Java отсутствие модификатора доступа перед элементом класса указывает на то, что этот элемент имеет уровень доступа default, а не private как в языке C++.

уровень доступа	*обозна-чение*	*модифи-катор*	*элемент класса доступен*
общедоступный	public	public	1) внутри класса; 2) из любого внешнего кода.
защищенный	protected	protected	1) внутри класса; 2) из любого внешнего кода пакета; 3) в любом потомке этого класса
по умолчанию	default	отсутствует	1) внутри класса; 2) из любого внешнего кода пакета
закрытый	private	private	только внутри класса

2.38. Модификатор final

Модификатором *final* может быть помечен класс, метод, поле, локальная переменная.

помечаемая с помощью модификатора final сущность	*свойство, которое приобретает сущность*
локальная переменная	значение можно присвоить максимум один раз
поле	1) поле *должно быть инициализировано* при объявлении 2) *значение поля изменить нельзя* (final-поля это константы)
метод	*метод нельзя перекрыть* в потомке (но можно перегрузить)
класс	класс *нельзя наследовать*

Практические задания

1. Создать класс "Окружность".

Класс должен иметь следующие поля:
1) x, y - координаты центра окружности;
2) radius - радиус окружности.

Класс должен иметь следующие методы:
1) передвинуть окружность на dx и dy;
2) проверить попадание заданной точки внутрь данной окружности;
3) проверить попадание другой окружности внутрь данной;
4) вывести на экран параметры окружности.

2. Создать класс "Вектор" для хранения ссылок на объекты.

Класс должен иметь следующие поля:
1) массив ссылок, который может расти;
2) количество ссылок в массиве.

Класс должен иметь следующие методы:
1) очистить весь массив;
2) добавить ссылку в массив;
3) Получить j-й элемент;
4) Удалить j-й элемент;
5) вывести значения массива на экран.

3. Создать класс "Матрица".

Класс должен иметь следующие поля:
1) двумерный массив вещественных чисел;
2) количество строк и столбцов в матрице.

Класс должен иметь следующие методы:
1) сложение с другой матрицей;
2) умножение на число;
3) умножение на другую матрицу;
4) транспонирование;
5) вывод на печать.