МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра "ПОВТ и АС"

Классы в Java

Методические указания к лабораторной работе

по дисциплинам "Программирование", "Объектно-ориентированное программирование"

Ростов-на-Дону 20 г.

Составитель: к.ф.-м.н., доц. Габрельян Б.В.

УДК 512.3

Классы в Java: методические указания – Ростов н/Д: Издательский центр ДГТУ, 20 . – 8 с.

В методической разработке рассматриваются общий синтаксис описания классов, описание статических и неизменяемых полей и методов, отношения между классами в Java. Даны задания по выполнению лабораторной работы. Методические указания предназначены для студентов направлений 090304 "Программная инженерия", 020303 "Математическое обеспечение и администрирование информационных систем".

Ответственный редактор:

Издательский центр ДГТУ, 20

1. Синтаксис описания классов.

Описание класса в Java размещается в пакете. Если пакет не указан, то класс размещается в глобальном безымянном пакете. Классы, размещенные в пакете, могут быть доступны из других пакетов, если они объявлены как открытые (public), в противном случае они будут доступны только в пределах своего пакета. Исключением являются классы, размещенные в безымянном глобальном пакете. такие классы даже если они не объявлены открытыми доступны в любом другом пакете. Поэтому, использование безымянного глобального пакета не приветствуется.

Файл Test1.java

```
package lab4.test1;
// класс доступен и внутри пакета lab4.test1 и вне его
public class Test1 {
// в этом классе можно создавать и использовать объекты класса A
    public static void main(String[] args) {
        A aObj = new A();
        Test1 t1Obj = new Test1();
        …
    }
}
// класс доступен только внутри пакета lab4.test1
class A {
…
}
```

Файл Test2.java

```
package lab4.test2;
// класс доступен и внутри пакета lab4.test2 и вне его
public class Test2 {
/* в этом классе можно создавать и использовать объекты классов B, lab4.test1.Test1, но нельзя использовать класс lab4.test1.A
*/
    public static void main(String[] args) {
        B bObj = new B();
        lab4.test1.Test1 t1Obj = new lab4.test1.Test1();
        Test2 t2Obj = new Test2();
Error   lab4.test1.A aObj = new lab4.test1.A();
        …
    }
// класс доступен только внутри пакета lab4.test2
class B {
…
}
```

В Java класс может быть объявлен с помощью зарезервированного слова class (перечислимые типы - это специализированные версии классов и они, объявляются с использованием enum, кроме того, начиная с версии Java 14 можно объявлять также ограниченные версии классов - записи с помощью зарезервированного слова record).

Класс является областью видимости, поэтому все имена, объявленные внутри описания класса доступны только в контексте этого класса. В классе можно объявлять переменные и именованные константы (поля класса), функции (методы класса), вложенные классы и интерфейсы задавая для них имена внутри описания класса. Каждое из этих имен в классе может быть объявлено как открытое (public), то есть доступное методам других классов, закрытое (private) - доступное только в методах своего класса, защищенное (protected) - доступное классам своего пакета и классам- наследникам, и, если не указан никакой модификатор доступа (ни public, ни private, ни protected) доступное всем классам своего пакета (на уровне пакета).

```
package lab4.test2;
class A {
        public int a1;          // открытое поле
        private byte a2;        // закрытое поле
        protected short a3;         // защищенное поле
        long a4;                // поле, доступное на уровне пакета
        public final int fi = 100; // открытое неизменяемое (константное) поле
        // методы
        public void f1() {}
        private int f2(int a, int x) { return a*x; }
        protected float f3() { return Math.PI*2.; }
        double f4(double x) { return x*x; }
}
//
```

```
public class Test {
	public static void main(String[] args) {
		A obj = new A();
		obj.a1 = 10;
Error	obj.a2 = 1;		// нет доступа
		obj.a3 = 125;		// доступ есть - один пакет
		obj.a4 = 1_000_000;	// доступ есть - один пакет
Error	obj.fi = 12;		// нельзя изменить константу
		obj.f1();
Error	obj.f2(2, 10);		// нет доступа
		float f = obj.f3();	// доступ есть - один пакет
		double d = obj.f4(5);	// доступ есть - один пакет
	}
}
```

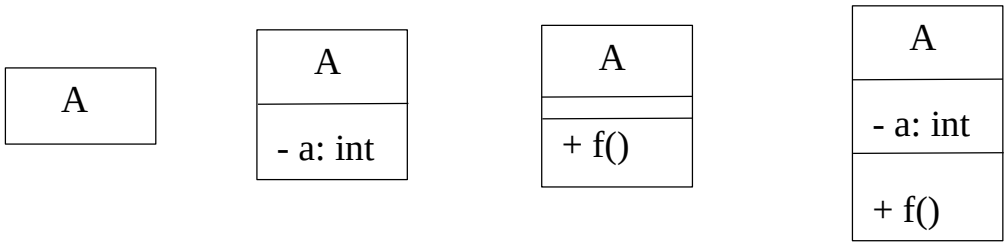
Каждый метод получает скрытый аргумент - ссылку на тот объект, для которого сейчас вызван метод. В теле метода эта ссылка доступна по имени this.

```
class A {
	private int a;
	public void f() {
		this.a = 10;
	}
}
```

Если в теле метода происходит обращение к полю или методу класса, компилятор генерирует обращение к этому имени через this

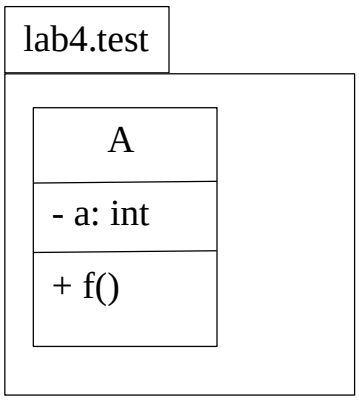
```
class A {
        private int a;
        public void f() {
                a = 10; // генерируется this.a = 10;
        }
}
```

На UML-диаграмме классов класс можно изобразить с разной степенью детализации.



В первой секции указывается имя класса, во второй - поля класса (свойства), в третьей методы класса. Если поля не указываются, а методы указываются, вторая секция должна быть пустой, но опускать ее нельзя. Значки -, +, # задают модификатор доступа - закрытый, открытый, защищенный соответственно.

Можно указать также пакет и классы, содержащиеся в пакете. Например, если класс A размещен в пакете lab4.test



1.1. Статические поля и методы

Объявление переменных в описании класса (полей класса) это лишь описание. Память под эти переменные не выделяется (то есть они не существуют) до тех пор, пока не будут созданы объекты класса. Каждый объект будет содержать свой набор всех полей класса. Поэтому вопрос о том, где в памяти расположено некоторое поле класса не имеет смысла, нужно указать объект, которому принадлежит поле. Таким образом поля "привязаны" к объектам класса и называются полями экземпляра класса. То же самое касается методов класса. Метод можно вызвать не сам по себе, но только для какого-то конкретного объекта класса. При этом у метода есть дополнительный скрытый аргумент - ссылка на тот объект для которого метод вызван. Метод "привязан" к объекту через эту ссылку (this).

Если поля или методы объявлены статическими, они не будут "привязаны" к объектам класса. Статическое поле будет существовать в единственном экземпляре независимо от того сколько объектов класса создано и создан ли вообще хотя бы один объект класса. Статическому методу не передается скрытый аргумент this, поэтому его можно вызывать, не указывая объект класса.

```
class A {
	public static int sa = 100;
	public static int getSA() {
		return sa;
	}
}
```

```
public class Test {
    public static void main(String[] args) {
        // не создаются объекты A
        System.out.println(A.getSA());
        A.sa += 20;
    }
}
```

1.2. Конструкторы и блоки инициализации

При создании объекта класса для задания начальных значений его полей (инициализации) вызывается специальный метод класса конструктор. Если конструктор не задается для класса явно, он будет создан компилятором. Это будет метод класса без явных аргументов (но с обязательным неявным аргументом this). В Java этот конструктор по умолчанию обнуляет все поля класса (в отличие, например, от конструктора по умолчанию в C++). Для числовых полей это значение ноль соответствующего типа, для логического типа - значение false, для ссылочных типов - пустая ссылка null. Инициализирующее значение для поля можно задать непосредственно при объявлении этого поля, в блоках инициализации и в теле конструктора.

```
class A {
    private int a = 10; // объявление поля
    public A(int x) { // конструктор
        a = x;
    }
```

```
        { // блок инициализации
                a = 100;
        }
}
```

Здесь вначале a получит значение 10, затем, в блоке инициализации, значение 100 и, наконец, получит значение, переданное конструктору через аргумент x. Для инициализации статических полей можно использовать статические блоки инициализации.

```
class A {
        private static int aa = 10;
        static { // статический блок инициализации
                aa *= 2;
        }
}
```

2. Вложенные классы

Внутри класса можно объявлять другие - вложенные классы. Они представляют в программе некоторые подчиненные понятия, связанные с тем понятием, которое представляет внешний класс. Вложенные классы могут быть объявлены как открытые, закрытые, защищенные и доступные на уровне пакета. Объект вложенного класса можно создать только для некоторого объекта внешнего класса. Если внешний класс A, а вложенный B, то объект вложенного класса создается так:

```
A aObj = new A();
```

A.B bObj = **aObj.new** B(); // если объект B создается вне класса A

B bObj = **aObj.new** B(); // если объект B создается в каком-нибудь методе A

Если вложенный класс объявлен закрытым, то его можно использовать только в методах внешнего класса.

```
class Outer {
	private int o = 10;
	private class Inner1 {
		public int fun1() {
			System.out.println("o="+o);
			return 2*o;
		}
	}
	public class Inner2 {
		public int i2 = 12;
		public int fun2() {
			System.out.println("o="+o);
			// this.o = 101; this.i2 = 1;
			return 3*o;
		}
	}
}
```

```
class Test {
    public static void main(String[] args) {
        Outer out = new Outer();
Error   Outer.Inner1 in1 = out.new Outer.Inner1();  // нет доступа
        Outer.Inner2 in2 = out.new Outer.Inner2();
    }
}
```

Методы вложенного класса имеют доступ ко всем (открытым, закрытым и т.д.) полям и методам внешнего класса, но не наоборот.

2.1. Статические вложенные классы

Вложенный класс можно объявить статически вложенным. Это приведет к тому, что объекты вложенного класса не будут связаны с объектами внешнего класса, они создаются независимо от объектов внешнего класса. Такими являются вложенные классы, например, в C++.

```
public class List { // понятие список целых значений
    /* частное понятие - узел списка, используется только в List */
    private static class Node {
        int data;
        Node next;
    }
    private Node first;
```

```
    /* получить ссылку на узел со значением value */
    private Node findNode(int value) {
        Node current = first;
        while(current != null) {
            if(current.data == value) return current;
            current = current.next;
        }
        return null;
    }
    …
}
```

Если статический вложенный класс открытый, его объекты можно создавать вне вложенного класса даже если объектов внешнего класса нет, но квалифицированное имя вложенного класса должно включать имя внешнего класса. Статический вложенный класс также имеет доступ ко всем (открытым, закрытым и т.д.) полям и методам внешнего класса, но у его методов нет скрытой (неявной) ссылки this на объект внешнего класса, поэтому получить такой доступ можно лишь если метод статического вложенного класса получит ссылку явно - получит через свой аргумент или создаст объект внешнего класса.

```
class A {
    private int a;
```

```
        public A(int x) {
                a = x;
        }
        public A() {
                this(0);
        }
        //
        public static class B {
                public void fun1(A obj) {
                        System.out.println(obj.a);
                }
                public void fun2() {
                        A obj = new A(100);
                        System.out.println(obj.a);
                }
        }
}
class Test {
        public static void main(String[] args) {
                A.B b = new B();
                b.fun2();
```

```
        A a = new A();
        b.fun1(a);
    }
}
```

2.2. Внутренние классы

Класс можно объявить также в отдельном методе и даже в отдельном блоке отдельного метода. Очевидно, что в таком случае этот класс можно использовать только в блоке. Здесь уже не может быть никаких модификаторов доступа, так как блок - это часть метода внешнего класса, а для методов своего класса модификаторы доступа не имеют смысла.

```
class A {
    private int a;
    public A(int x) {
        a = x;
    }
    public void fun() {
        class B {
            public void test() {
                System.out.println("a="+a+"\tc="+c);
            }
        }
        final int c = 120;
```

```
        B obj = new B();
        obj.tets();
    }
}
```

3. Отношение агрегирования

Если полем одного класса является объект другого класса (или полями являются объекты других классов) между классами возникает отношение агрегирования, в котором первый класс является агрегатом (целым), а второй (или остальные) представляет часть (части) агрегата.

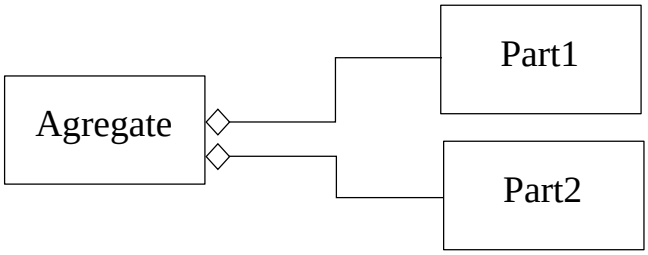
```
// класс-часть агрегата
class Part1 {
…
}
// класс-часть агрегата
class Part2 {
…
}
// класс-агрегат (целое)
class Agregate {
    private Part1 p1 = new Part1();
    private Part2 p2 = new Part2();
```

…

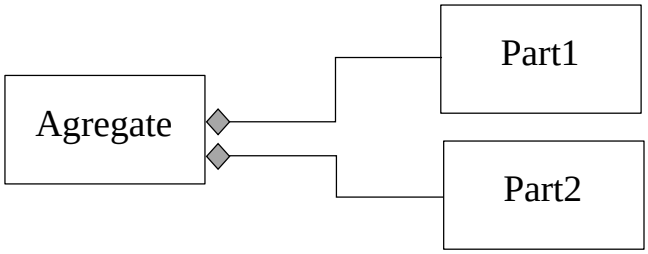
}

Класс-целое содержит в себе объекты классов-частей что обозначают как отношение "has-a".

На UML-диаграмме классов отношение агрегирования задается как стрелка, соединяющая целое с частью, направленная от части к целому, с незакрашенным ромбом на конце.



На UML-диаграмме классов можно показать особый случай отношения агрегирования - отношение композиции, возникающее если разрушение агрегата, приводит к невозможности существования частей агрегата. В этом случае используется закрашенный ромб.



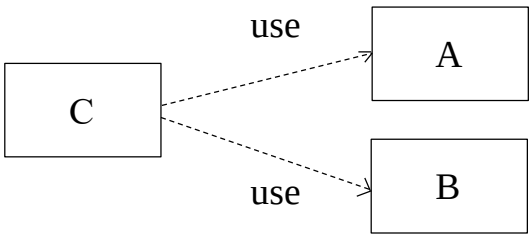
4. Отношение использования

Отношение использования возникает если методу оного класса (использующего) передается объект другого класса (используемого) или если метод создает объект другого класса или если метод обращается к статическому методу другого класса.

```
class A {
	public void fun() {}
}
class B {
	public static void f() {}
}
class C {
	public void test1(A obj) {
		obj.fun();
	}
	public void test2() {
		A obj = new A();
		obj.fun();
	}
	public void test3() {
		B.f();
	}
}
```

На UML-диаграмме классов отношение использования задается обычной стрелкой, направленной от использующего к используемому классу и соединительная линия задается пунктиром, рядом с линией помещается метка

use (или <<use>>), можно также написать по-русски "использует" (без кавычек).



5. Отношение расширения (наследования)

Отношение наследования связывает родственные классы. Один из них является родительским (базовым), другой дочерним (производным). Производный класс получает все свойства (поля) и все открытое и защищенное поведение базового класса и может добавить какие-то свои поля и методы. Несмотря на то, что объект производного класса содержит все поля, полученные от класса базового, у него нет прямого доступа к закрытым полям (и методам). Java поддерживает только единичное наследование для классов. Это означает, что у класса может быть только один прямой базовый класс. В Java все классы прямо или косвенно являются наследниками класса Object. Если при объявлении класса не указан его базовый класс, тогда базовым будет класс Object. Если указать базовый класс, то Object все равно будет хоть и не прямым, но базовым классом. Ссылку на базовый класс можно связать с объектом любого производного класса, поэтому ссылка на Object может быть связана с объектом любого класса. В Java отношение наследования задается с помощью зарезервированного слова extends. В конструкторе производного класса можно вызвать конструктор базового с помощью конструкции super(), но это обращение должно быть первой командой в конструкторе.

```
public class Base { // базовый класс
    protected int b; /* доступно классам-наследникам даже если они содержатся в другом пакете */
    public Base(int x) {
        b = x;
    }
    public int getB() {
        return b;
    }
}
class Derived extends Base { // класс-наследник
    private int d;
    public Derived(int a, int b) {
        super(a); /* вызов конструктора базового класса с одним аргументом */
        d = b;
    }
}
class Test {
    public static void main(String[] args) {
        Base b = new Base(10);
```

```
        Derived d = new Derived(1, 2);
        System.out.println(b.getB());
        b = d; /* ссылка на базовый класс связывается с объектом производного класса */
        System.out.println(b.getB());
        Object o = d; // Object базовый класс
Error   o.getB();   // в классе Object нет метода getB()
        ((Derived)o).getB(); /* приведение ссылки на Object к ссылке на Derived, теперь метод вызвать можно */
    }
}
```

Если класс объявлен как final его нельзя расширять, то есть у него не может быть наследников.

```
final class Fi {
…
}
Error   class Di extends Fi { … } // нельзя расширить final класс
```

На UML-диаграмме классов отношение наследования задается как стрелка с незакрашенным треугольником на конце, направленная от производного к базовому классу.

5.1. Переопределение методов базового класса

Класс-наследник может не только расширять базовый класс своими полями и методами, но и переопределять методы (не закрытые), доставшиеся ему от базового класса. Это обеспечивает полиморфное поведение системе классов. В Java методы классов-наследников, сигнатуры которых совпадают с сигнатурой методов базового класса автоматически переопределяются (с точки зрения C++ они являются виртуальными функциями).

```
class Base {
	public void f(String s) {}
}
class Derived extends Base {
	public void f(String s) {}
}
class Test {
	public static void main(String[] args) {
		Base b;
		b = new Derived();
		b.f("Hello"); // вызывается метод класса Derived
	}
}
```

Если у методов базового и производного классов совпадают только имена, а списки типов аргументов разные (сигнатуры не совпадают), то перегрузки метода не произойдет. Чтобы заставить компилятор проверять, правильно ли задана сигнатура переопределяемого метода можно использовать аннотацию @Override.

```
class Derived extends Base {
```

```
    @Override
    public void f(String s) {}
}
```

Теперь, если по ошибке попытаться переопределить метод как-то так

```
class Derived extends Base {
        @Override
Error   public void f() {}
}
```

Возникнет ошибка на этапе компиляции.

Если метод класса объявлен как final, его нельзя переопределять. Во всех классах-наследниках может использоваться только реализация базового класса.

5.2. Безымянные вложенные классы

В Java можно использовать безымянные вложенные классы. Например, есть метод табулирующий функцию, ему передаются диапазон изменения аргумента функции - два действительных числа, шаг изменения аргумента, и объект класса Func, содержащий метод fun(), вычисляющий значение функции

```
class Func {
    public double fun(double x) {
        return Math.sin(x);
    }
}
class Tabulator {
    public void tabulate(double xmin, double xmax, double step, Func f) {
        for(double x=xmin; x<=xmax; x += step)
```

```
                System.out.println(""+x+"\t"+f.fun(x));
        }
}
…
Func f = new Func();
Tabulator t = new Tabulator();
t.tabulate(-0.1, 0.1, 0.01, f);
…
```

Если построить класс-наследник Func и переопределить в нем метод fun, можно будет передать ссылку на объект этого класса в метод tabulate и протабулировать другую функцию. Можно продолжить и создать столько классов-наследников Func с переопределенным методом fun сколько функций нам надо протабулировать.

```
class Func2 extends Func {
        @Override
        public double fun(double x) {
                return Math.cos(x);
        }
}
…
t.tabulate(-0.1, 0.1, 0.01, new Func2());
…
```

Плохо то, что для каждой новой функции нужно придумывать имя для класса-наследника Func. Этого можно избежать следующим образом

```
…
t.tabulate(-0.1, 0.1, 0.01, new Func() {
```

```
        @Override
        public double fun(double x) {
                return Math.exp(x*.1);
        }
});
…
```

Данная конструкция заставляет компилятор создать новый безымянный класс как класс-наследник Func, при этом мы переопределяем метод fun.

5.3. Абстрактные классы

Абстрактный класс - это класс для которого нельзя создавать экземпляры. Он используется как общий предок для семейства родственных классов и содержит их общие состояния и поведение, но объекты этого класса не представляют какого-то понятия, имеющего конкретный смысл. В Java класс становится абстрактным по объявлению. Если при объявлении класса использовать зарезервированное слово abstract, то класс станет абстрактным.

```
abstract class A {
        protected int a;
        public void fun() {}
}
                A obj; // объекта нет, только ссылка, поэтому можно
Error           obj = new A(); // нельзя создать объект абстрактного класса
```

Абстрактный класс может (но не обязан) содержать нереализованные (абстрактные) методы. Тогда он обязательно должен быть объявлен абстрактным. Классы наследники должны давать свою реализацию всех

абстрактных методов родительского класса, иначе они сами должны быть объявлены абстрактными.

```
abstract class A { // базовый класс
    public abstract void fun(int x); /* объявление абстрактного метода, у метода не моет быть реализации (тела) */
}
class B extends A { // класс-наследник
    @Override
    public void fun(int x) {} /* сигнатура метода в точности как у базового класса */
}
class Test {
    public static void main(String[] args) {
        A a; // ссылка на базовый класс
        a = new B();
        a.fun(10);
    }
}
```

Если посмотреть на описанную выше систему родственных классов с базовым классом Func, можно увидеть, что сам класс Func удобно сделать абстрактным, он будет представлять понятие "некоторая функция одного действительного аргумента, возвращающая действительное значение".

```
abstract class Func {
    public abstract double fun(double x);
}
```

Все классы, созданные по нижеприведенным заданиям, должны размещаться в пакетах (например, lab4.car, lab4.complex).

ЗАДАНИЕ 1. Создайте класс Car, представляющий понятие "автомобиль". Каждый автомобиль должен иметь, как минимум, следующие характеристики: регистрационный знак, марка, вид, цвет, мощность двигателя, количество колес. Для вновь созданного, конкретного автомобиля такие характеристики как марка, вид, цвет, мощность двигателя, количество колес должны быть заданы непременно, но регистрационного номера у него до поры до времени может и не быть (а может и быть). Все характеристики автомобиля, кроме марки, вида и количества колес можно изменять в процессе его эксплуатации. Вид автомобиля - легковой, грузовой, автобус. Создайте и используйте для задания вида автомобиля перечислимый тип. Для легковых, грузовых автомобилей и автобусов с нормальным креплением знака (тип 1) согласно ГОСТ Р 50577-2018 [6] знак имеет следующий формат: X 000 XX 00 RUS или X 000 XX 000 RUS. Здесь 0 в реальном знаке заменяется какой-то арабской цифрой, а X - одной из 12 букв кириллицы (в верхнем регистре), написание которой совпадает с написанием латинской буквы: A, B, E, K, M, H, O, P, C, T, У, X. Попытка задания неправильного знака должна пресекаться соответствующим методом класса. Для проверки используйте регулярное выражение. Создайте код для тестирования класса Car с заданием начальных характеристик, запросом новых значений для тех характеристик, которые можно изменять и выводом на экран текущих значений всех характеристик.

ЗАДАНИЕ 2. В Java отсутствует стандартный класс для представления комплексных чисел. Создайте такой класс (Complex), поддерживающий операции получения действительной, мнимой частей числа, сложения, вычитания, умножения, деления, комплексного сопряжения, проверки двух чисел на равенство, вывода значения комплексного числа в алгебраической и тригонометрической формах. Так как действительное число, это комплексное число с нулевой мнимой частью, должны поддерживаться также арифметические операции, в которых один из аргументов действительное число (тип double).
Алгебраическая форма: z = x + i*y, x - действительная часть, y - мнимая часть, i - мнимая единица

Тригонометрическая форма: $z = r*(\cos(\varphi)+i*\sin(\varphi))$, r - модуль числа $|z| = \sqrt{x^2 + y^2}$, φ - аргумент числа $\arg(z) = \operatorname{atctg}\frac{y}{x}$

z_1 и z_2 равны, если $x_1 = x_2$ и $y_1 = y_2$

комплексно сопряженное $\bar{z} = x - i*y$

Сумма z_3 чисел $z_1 = x_1 + i*y_1$ и $z_2 = x_2 + i*y_2$: $z_3 = (x_1+x_2) + i*(y_1+y_2)$

Разность $z_1 - z_2$: $z_3 = (x_1-x_2) + i*(y_1-y_2)$

Произведение $z_1 * z_2$: $z_3 = (x_1*x_2-y_1*y_2) + i*(x_1*y_2+x_2*y_1)$

Частное z_1 / z_2: $z_3 = \frac{x_1*x_2+y_1*y_2}{x_2^2+y_2^3} + i * \frac{x_2*y_1-x_1*y_2}{x_2^2+y_2^3}$

ЗАДАНИЕ 3. Реализуйте методы для вычисления элементарных функций комплексного переменного $z = x + i*y$: $e^z = e^x(\cos(y)+i*\sin(y))$, $\sin(z) = \frac{e^{i\cdot z}-e^{-i\cdot z}}{2\cdot i}$, $\cos(z) = \frac{e^{i\cdot z}+e^{-i\cdot z}}{2}$, $\tan(z) = \frac{\sin(z)}{\cos(z)} = \frac{e^{i\cdot z}-e^{-i\cdot z}}{(e^{i\cdot z}+e^{-i\cdot z})\cdot i}$, $\operatorname{arctan}(z) = \frac{\cos(z)}{\sin(z)} = \frac{(e^{i\cdot z}+e^{-i\cdot z})\cdot i}{e^{i\cdot z}-e^{-i\cdot z}}$, $\operatorname{sh}(z) = \frac{e^z-e^{-z}}{2}$, $\operatorname{ch}(z) = \frac{e^z+e^{-z}}{2}$, $\operatorname{th}(z) = \frac{\operatorname{sh}(z)}{\operatorname{ch}(z)} = \frac{e^z-e^{-z}}{e^z+e^{-z}}$, $\operatorname{cth}(z) = \frac{\operatorname{ch}(z)}{\operatorname{sh}(z)} = \frac{e^z+e^{-z}}{e^z-e^{-z}}$. Формула Эйлера: $e^{i*x} = \cos(x)+i*\sin(x)$, x - действительное число. Организуйте методы так, чтобы их можно было вызывать, не создавая объекты класса, которому принадлежат эти методы. Прежде чем писать код, ответьте себе на вопрос, должны ли эти методы быть методами класса Complex, созданного в предыдущем задании?

ЗАДАНИЕ 4. В Задании 1 двигатель автомобиля представлен только одной своей характеристикой - мощностью. А самом деле каждый двигатель имеет серийный (заводской) номер, и, помимо мощности, рабочий объем, расход топлива, вид топлива, число цилиндров и т.д. Поэтому для такого важного понятия удобно ввести отдельный класс. Создайте класс Engine, включив в него несколько важных характеристик двигателя и методы доступа к этим характеристикам. Определите и реализуйте нужный конструктор (или конструкторы) класса Engine. Замените в классе Car поле "мощность" на поле "двигатель" (engine). Измените код тестирования для проверки работоспособности новой версии класса Car и класса Engine.

ЗАДАНИЕ 5. В Заданиях 1, 4 вид автомобиля задается как предопределенное значение. Появление новых видов будет приводить к необходимости модифицировать класс Car. Кроме того, для новых видов автомобилей правила формирования регистрационного знака могут быть другими. Более гибкое решение - каждый вид автомобиля представлять собственным классом. С другой стороны, все автомобили

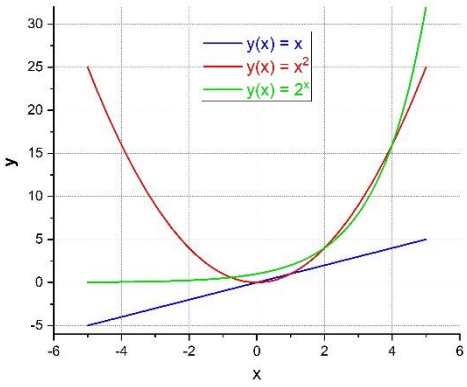
будут обладать одинаковым набором некоторых базовых характеристик. Не разумно при возникновении нового вида автомобиля всякий раз повторять в нем объявления этих базовых характеристик и определять методы доступа к ним. Нужно использовать семейство родственных классов, в котором один (Car) будет содержать все общие характеристики, а другие - классы для конкретных видов автомобилей дополнительные характеристики и/или конкретные значения для базовых характеристик. Измените нужным образом класс Car, создайте классы для следующих видов автомобилей: легковой, грузовой, автобус, специальный (например, пожарная машина, или автомобиль для дипломатических миссий). Для специальных автомобилей придумайте свою схему формирования регистрационного знака (или возьмите из ГОСТ Р 50577-2018 [6]).

ЗАДАНИЕ 6. В Задании 5 класс Car содержит общую функциональность семейства автомобилей разных видов, но не представляет какие-то конкретные автомобили. Не логично разрешать пользователям создавать в программе экземпляры класса Car. Кроме того, некоторые методы доступа к базовым характеристикам автомобиля, заданные в классе Car не должны переопределяться в классах-наследниках. Если это будет сделано случайно или преднамеренно, изменится базовое поведение некоторых автомобилей, а этого не должно происходить по логике организации семейства классов автомобилей. Нужно запретить наследникам переопределять такие методы базового класса. Наконец, нужно задаться вопросом, насколько расширяемой должна быть наша система родственных классов. Например, есть ли какие-то особые формы автобусов, для которых нужно построить класс-наследник имеющегося класса? А для других классов, представляющих автомобили? Сделайте так, чтобы, по крайней мере, класс представляющий автобусы нельзя было расширять.

ЗАДАНИЕ 7. Создайте класс "Автобаза". На базе может размещаться некоторое фиксированное количество автомобилей разных видов (классы производные от Car). Для хранения объектов, представляющих автомобили нужно использовать массив фиксированного размера. Максимально возможный размер для конкретной автобазы (конкретного объекта класса) задается при создании объекта. Каждый автомобиль может находиться в одном из трех допустимых состояний: на базе, в рейсе, в ремонте. Нужно обеспечить добавление нового автомобиля (если еще есть место для размещения автомобиля). Удаление

(списание) автомобиля. Отправку исправного автомобиля в рейс. Отправку неисправного автомобиля в ремонт. Возврат автомобиля из рейса или из ремонта. Отображение на экране списка находящихся на базе исправных автомобилей, списка автомобилей, находящихся в рейсе, списка неисправных автомобилей (отдельные методы).

ЗАДАНИЕ 8. Средство построения графиков функций может генерировать рисунки подобные следующему. Отвлекаясь пока от



графического изображения спроектируйте систему классов для решения такой задачи в объектно-ориентированном стиле (в виде UML-диаграммы классов). Каждое важное понятие задачи нужно представить собственным классом. Например, на рисунке есть собственно весь график, на нем отдельные кривые, оси координат, координатная сетка, пояснения. Это все важные понятия в решаемой задаче. Каждый класс отвечает за свой функционал. В частности, каждый класс отвечает за прорисовку своих объектов. Определите, какие параметры должны хранить соответствующие классы, какие у них должны быть методы и изобразите на UML-диаграмме классов сами классы, их поля и методы, отношения между классами.

ЗАДАНИЕ 9. Создайте классы для построенной в Задании 7 UML-диаграммы классов. Вместо реального рисования нужно обеспечить вывод на экран текстовой информации со значениями параметров объектов соответствующих классов.

Библиотека графического пользовательского интерфейса AWT предлагает использовать в качестве главного окна программы объект класса Frame. Frame представляет окно с полосой заголовка на которой, помимо заголовка, размещаются также элементы управления для сворачивания окна, разворачивания его на весь экран и закрытия окна, рамкой, позволяющей нужным образом изменять размеры окна и рабочей областью, которая является контейнером, то есть может содержать другие элементы пользовательского интерфейса. Для рисования в AWT есть класс Canvas ("полотно"). Этот класс представляет подчиненное окно без заголовка и рамки. Можно разместить объект Canvas в рабочей области Frame так, чтобы "полотно" занимало всю рабочую область. Чтобы Canvas всегда занимал всю рабочую область Frame

нужно при изменении размера Frame соответствующим образом изменять и размер Canvas. AWT предлагает стандартные менеджеры размещения (компоновки) которые берут на себя задачу изменения размеров элементов управления, размещенных в окне Frame. При этом возможны разные варианты изменений размеров, только по горизонтали, только по вертикали и более сложные. Например, менеджер компоновки BorderLayout позволяет так разместить элемент управления в окне, чтобы "подстраивались" одновременно и ширина, и высота Canvas.

Литература

1. Дж.Гослинг, Б.Джой, Г.Стил, Г.Брача, А.Бакли "Язык программирования Java SE 8. Подробное описание, 5-е изд.". – М.: Вильямс. – 2016, 672с.
2. Н.А.Прохоренок "Основы Java, 2-е изд.". – СПб.:BHV-Петербург. – 2019, 768с.
3. Б.Эккель "Философия Java. Библиотека программиста". – СПб.: Питер. – 2001, 880с.
4. П.Ноутон, Г.Шилдт "Java 2". – СПб.:BHV-Петербург. – 2001, 1072с.
5. Д.Бишоп "Эффективная работа: Java 2". – СПб.:Питер; К.:BHV. – 2002, 592с.

Редактор А.А. Литвинова

ЛР № 04779 от 18.05.01.	В набор	В печать
Объем 0,5 усл.п.л., уч.-изд.л.	Офсет.	Формат 60x84/16.
Бумага тип №3.	Заказ №	Тираж 120. Цена

Издательский центр ДГТУ

Адрес университета и полиграфического предприятия:

344010, г. Ростов-на-Дону, пл. Гагарина, 1.