ЛАБОРАТОРНАЯ РАБОТА №1 ПО ПРЕДМЕТУ «ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ»
ТЕМА: «ОБЩИЕ ПРИНЦИПЫ РАЗРАБОТКИ КОНСОЛЬНЫХ ПРИЛОЖЕНИЙ JAVA. РАБОТА В ИНТЕГРИРОВАННОЙ СРЕДЕ РАЗРАБОТКИ, РАЗРАБОТКА КОНСОЛЬНЫХ ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ БАЗОВОГО СИНТАКСИСА ЯЗЫКА JAVA. КЛАССЫ И ОБЪЕКТЫ, СВОЙСТВА И МЕТОДЫ В JAVA»

Как и во всех современных языках программирования, в *Java* поддерживается несколько типов данных. Их можно применять для объявления переменных и создания массивов. В языке *Java* применяется простой, эффективный и логический подход к этим языковым средствам.

***Java* – строго типизированный язык.** Именно этим объясняется безопасность и надежность программ на *Java*. **Во-первых, каждая переменная и каждое выражение имеет конкретный тип, и каждый тип строго определен. Во-вторых, все операции присваивания, как явные, так и через параметры, передаваемые при вызове методов, проверяются на соответствие типов.**

В *Java* отсутствуют средства автоматического приведения или преобразования конфликтующих типов, как это имеет место в некоторых языках программирования. Компилятор *Java* проверяет все выражения и параметры на соответствие типов. Любые несоответствия типов считаются ошибками, которые должны быть исправлены до завершения компиляции класса.

В языке *Java* определены **восемь примитивных типов данных**: *byte, short, int, long, char, boolean, float, double*. Примитивные типы называют также простыми типами данных.

Эти типы данных можно использовать непосредственно или для создания собственных классов. Таким образом, они служат основанием для всех других типов данных, которые могут быть созданы в приложении.

Примитивные типы представляют одиночные значения, а не сложные объекты. Язык *Java* является полностью объектно-ориентированным, кроме примитивных типов данных, которые аналогичны простым типам данных, которые можно встретить в большинстве других не объектно-ориентированных языков программирования. Эта особенность *Java* объясняется стремлением

обеспечить максимальную эффективность. **Превращение примитивных типов в объекты привело бы к слишком заметному снижению производительности.**

Примитивные типы определены таким образом, чтобы **обладать явно выражаемым диапазоном допустимых значений и математически строгим поведением,** в то время как, например, языки типа *C* и *C++* **допускают** варьирование длины целочисленных значений в зависимости от требований исполняющей среды.

Но в языке *Java* дело обстоит иначе. **В связи с требованием переносимости** программ на *Java* все типы данных обладают **строго определенным диапазоном допустимых значений**. Например, значения типа *int* всегда оказываются 32-разрядными, независимо от конкретной платформы. **Это позволяет создавать программы, которые гарантированно будут выполняться на любой машинной архитектуре без специального переноса**. В некоторых средах строгое указание длины целых чисел может приводить к незначительному снижению производительности, но это требование абсолютно необходимо для переносимости программ. Рассмотрим каждый из примитивных типов данных в отдельности.

Для целых чисел в языке *Java* определены четыре типа: *byte, short, int* и *long*. **Все эти типы данных представляют целочисленные значения со знаком: как положительные, так и отрицательные. В *Java* не поддерживаются только положительные целочисленные значения без знака.** Во многих других языках программирования поддерживаются целочисленные значения как со знаком, так и без знака, но разработчики *Java* посчитали целочисленные значения без знака ненужными.

В частности, они решили, что понятие числовых значений без знака служило в основном для того, чтобы обозначить состояние старшего бита, определяющего знак целочисленного значения. В *Java* управление состоянием старшего бита осуществляется иначе: с помощью специальной операции "сдвига вправо без знака". Благодаря этому отпадает потребность в целочисленном типе данных без знака.

Длина целочисленного типа означает не занимаемый объем памяти, а поведение, определяемое им для переменных и выражений данного типа. В исполняющей среде *Java* может быть использована любая длина, при условии, что типы данных ведут себя так, как они объявлены. Как показано в таблице, длина и диапазон допустимых значений целочисленных типов данных изменяются в широких пределах.

Наименьшим по длине является целочисленный тип *byte*. Это 8-разрядный тип данных со знаком и диапазоном допустимых значений от -128 до 127.

Переменные типа *byte* особенно удобны для работы с потоками ввода-вывода данных в сети или файлах. Они удобны также при манипулировании необработанными двоичными данными, которые могут и не быть непосредственно совместимы с другими встроенными типами данных в *Java*.

Для объявления переменных типа *byte* служит ключевое слово *byte*. Например, в следующей строке кода объявляется переменная *age* типа *byte*:

```
byte age;
```

Тип *short* представляет 16-разрядные целочисленные значения со знаком в пределах от -32 768 до 32 767. Этот тип данных применяется в *Java* реже всех остальных. На экране приведен пример объявления переменных типа *short*.

```
short roll_no;
```

Наиболее употребляемым целочисленным типом является *int*. Это тип 32-разрядных целочисленных значений со знаком в пределах от - 2 147 483 648 до 2 147 483 647. Переменные типа *int* зачастую используются для управления циклами и индексирования массивов. На первый взгляд может показаться, что пользоваться типом *byte* или *short* эффективнее, чем типом *int*, в тех случаях, когда не требуется более широкий диапазон допустимых значений, предоставляемый типом *int*, но в действительности это не всегда так.

Дело в том, что при использовании значений типа *byte* и *short* в выражениях их тип **продвигается** к типу *int* при вычислении выражения. Поэтому тип *int* зачастую оказывается наиболее подходящим для обращения с целочисленными значениями.

Этот тип 64-разрядных целочисленных значений со знаком удобен в тех ситуациях, когда длины типа *int* недостаточно для хранения требуемого значения. Диапазон допустимых значений типа *long* достаточно велик, что делает его удобным для обращения с большими целыми числами. На слайде приведен пример программы, которая вычисляет количество миль, проходимых лучом света за указанное число дней.

```
//Пример №1. Вычислить расстояние, проходимое светом, используя переменные типа long
class Light {
    public static void main(String args[]) {
        int lightSpeed, days;
        long distance, seconds;
        lightSpeed = 300000;// приблизительная скорость света, километров в секунду
```

```
        days = 1000;//количество дней
        seconds = days * 24 * 60 * 60;//преобразовать в секунды
        distance = lightSpeed * seconds;//вычислить расстояние
        System.out.print("За " + days);
        System.out.print(" дней свет пройдет около ");
        System.out.println(distance + " километров. ");
    }}
```

Результаты работы программы:

```
За 1000 дней свет пройдет около 25920000000000 километров.
```

Очевидно, что такой результат не поместился бы в переменной типа *int*.

Числа с плавающей точкой, называемые также действительными числами, используются при вычислении выражений, которые требуют результата с точностью до определенного знака после десятичной точки. Например, вычисление квадратного корня или тригонометрических функций вроде синуса или косинуса приводит к результату, который требует применения числового типа с плавающей точкой.

В *Java* реализован стандартный ряд типов и операций над числами с плавающей точкой. Существуют два числовых типа с плавающей точкой: *float* и *double*, которые соответственно представляют числа одинарной и двойной точности. Их длина и диапазон допустимых значений приведены в таблице.

Тип *float* определяет числовое значение с плавающей точкой одинарной точности, для хранения которого в оперативной памяти требуется 32 бита. В некоторых процессорах обработка числовых значений с плавающей точкой одинарной точности выполняется быстрее и требует в два раза меньше памяти, чем обработка аналогичных значений двойной точности. Но если числовые значения слишком велики или слишком малы, то тип данных *float* не обеспечивает требуемую точность вычислений. Этот тип данных удобен в тех случаях, когда требуется числовое значение с дробной частью, но без особой точности. На слайде приведен пример объявления переменных типа *float*.

```
float pi=3.14f;
```

Для хранения числовых значений с плавающей точкой двойной точности типа *double* в оперативной памяти требуется 64 бита.

На самом деле в некоторых современных процессорах, оптимизированных для выполнения математических расчетов с высокой скоростью, обработка значений двойной точности выполняется быстрее, чем обработка значений одинарной точности. Все трансцендентные математические функции наподобие *sin(), cos(), sqrt()* возвращают значения

типа *double*. Рациональнее всего пользоваться типом *double*, когда требуется сохранять точность многократно повторяющихся вычислений или манипулировать большими числами.

На слайде приведен пример программы, где переменные типа *double* используются для вычисления площади круга.

```
//Пример №2. Вычислить площадь круга
class Area {
    public static void main(String args[]) {
        double pi, square;
        float r;
        r = 10.8f;//радиус окружности
        pi = 3.1416;//приблизительное значение числа Пи
        square = pi * r * r;//вычислить площадь круга
        System.out.println("Площадь круга равна " + square);
    }
}
```

Результаты работы программы:

```
Площадь круга равна 366.4362369429933
```

Для хранения символов в *Java* используется тип данных *char*. Но тем, у кого имеется опыт программирования на *C/C++*, следует иметь в виду, что тип *char* в *Java* не равнозначен типу *char* в языках *C/C++*. **Если в C/C++ тип *char* является целочисленным и имеет длину 8 бит, то в *Java* для представления символов типа *char* используется двухбайтовая кодировка Юникод (*Unicode*), определяющая полный набор международных символов на всех известных языка мира.**

Разумеется, пользоваться кодировкой в Юникоде для локализации программ на таких языках, как английский, немецкий, испанский или французский, не совсем эффективно, поскольку для представления символов из алфавита этих языков достаточно и 8 бит. Но это та цена, которую приходится платить за переносимость программ в глобальном масштабе. Подробнее о кодировке Юникод можно посмотреть по адресу http://www.unicode.org.

Использование переменных типа *char* демонстрируется в следующем примере программы:

```
//Пример №3. Применение типа char. Символьные переменные ведут себя как целочисленные значения
class CharDemo {
    public static void main(String args[]) {
        char ch1, ch2;
        ch1 = 88;//код символа X
        ch2 = 'Y';
```

```
        System.out.print("ch1 и ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

Результаты работы программы:

```
ch1 и ch2: X Y
```

Обратите внимание на то, что переменной *ch1* присвоено значение 88, обозначающее код *ASCII* (и Юникод) латинской буквы *X*. Так как набор символов в коде *ASCII* занимает первые 127 значений из набора символов в Юникоде. Поэтому все прежние приемы, применяемые для обращения с символами в других языках программирования, вполне пригодны и для *Java*.

Несмотря на то, что тип *char* предназначен для хранения символов в Юникоде, им можно пользоваться и как целочисленным типом для выполнения арифметических операций. Например, он допускает сложение символов или приращение значения символьной переменной.

В формальной спецификации *Java* тип *char* упоминается как целочисленный, а это означает, что он относится к той же общей категории, что и типы *int*, *short*, *long* и *byte*. Но поскольку основное назначение типа *char* – представлять символы в Юникоде, то он относится к собственной отдельной категории.

В языке *Java* имеется примитивный тип *boolean*, предназначенный для хранения логических значений. Переменные этого типа могу принимать только одно из двух возможных значений: *true* (истинное) или *false* (ложное). Значения логического типа возвращаются из всех операций сравнения вроде *a* > *b*.

Тип *boolean* обязателен для употребления и в условных выражениях, которые управляют такими операторами, как *if, while* и *for*. В следующем примере программы демонстрируется применение логического типа *boolean*:

```
//Пример №4. Демонстрация применения значений типа boolean
class BoolTest {
    public static void main(String args[]) {
        boolean b = false;
        System.out.println("b равно " + b);
        b = true;
        System.out.println("b равно " + b);
        //значение типа boolean может управлять оператором if
        if (b) System.out.println("Этот код выполняется.");
        b = false;
        if (b) System.out.println("Этот код не выполняется.");
        //результат сравнения - значение типа boolean
        System.out.println("10 > 9 равно " + (10 > 9));
    }}
```

Результаты работы программы:

```
b равно false
b равно true
Этот код выполняется.
10 > 9 равно true
```

В приведенном примере программы особое внимание обращают на себя три момента. Во-первых, при выводе значения типа *boolean* с помощью метода *println()* на экране, как видите, появляется слово "*true*" или "*false*". Во-вторых, одного лишь значения переменной типа *boolean* оказывается достаточно для управления условным оператором *if*. Записывать условный оператор *if* так, как показано ниже, совсем не обязательно.

```
if (b == true)...
```

И в-третьих, результатом выполнения операции сравнения является логическое значение типа *boolean*. Именно поэтому выражение 10>9 приводит к выводу слова "*true*". **Более того, выражение 10 > 9 должно быть заключено в дополнительный ряд круглых скобок, поскольку операция + имеет более высокую степень предшествования, чем операция >.**

Переменная служит основной единицей хранения данных в программе на *Java*. Переменная определяется в виде сочетания идентификатора, типа и необязательного начального значения. Кроме того, у всех переменных имеется своя область действия, которая определяет их доступность для других объектов и продолжительность существования переменной.

В *Java* все переменные должны быть объявлены до их использования.

Основная форма объявления переменных выглядит следующим образом:

```
тип идентификатор [=значение];
```

где параметр **тип** обозначает один из примитивных типов данных в *Java*, имя класса или интерфейса. Любой переменной можно присвоить начальное значение (инициализировать ее) через знак равенства. **Следует, иметь в виду, что инициализирующее выражение должно возвращать значение того же самого (или совместимого) типа, что и у переменной.** Для объявления нескольких переменных указанного типа можно воспользоваться списком, разделяемым запятыми.

В языке *Java* допускается наличие любого объявленного типа в каком угодно правильно оформленном идентификаторе.

В *Java* допускается объявление переменных в любом блоке кода. **Блок кода заключается в фигурные скобки {}, задавая тем самым область действия.** Код любого метода в *Java* должен начинаться открывающей фигурной скобки { и завершаться закрывающей фигурной скобкой }. Таким образом, при открытии каждого нового блока кода создается новая область действия. Область действия определяет, какие именно объекты доступны для других частей программы. Область действия также определяет продолжительность существования этих объектов.

Во многих других языках программирования различаются две основные категории области видимости (*scope*): глобальная и локальная. Но эти традиционные области действия не вполне вписываются в строгую объектно-ориентированную модель *Java*.

Несмотря на возможность задать глобальную область действия, в настоящее время такой подход является скорее исключением, чем правилом. **Две основные области действия в *Java* определяются классом и методом.**

Область действия, определяемая методом, начинается с его открывающей фигурной скобки. Но если у метода имеются параметры, то они также включаются в область действия метода.

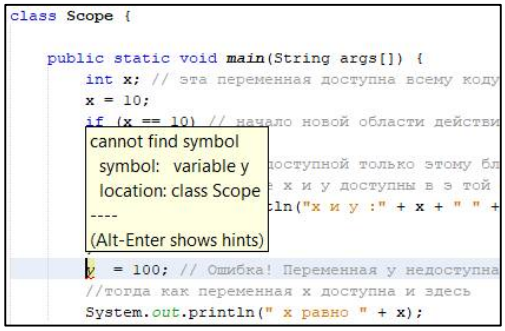
Переменные, объявленные в области действия, не доступны из кода за пределами этой области. Таким образом, объявление переменной в области действия обеспечивает ее локальность и защиту от несанкционированного доступа или внешних изменений.

Области действия могут быть вложенными. Так, **вместе с каждым блоком кода, по существу, создается новая, вложенная область действия.** В таком случае внешняя область действия включает в себя внутреннюю область. **Это означает, что объекты, объявленные во внешней области действия, будут доступны для кода из внутренней области действия, но не наоборот. Объекты, объявленные во внутренней области действия, будут недоступны за ее пределами.** Для того, чтобы разобраться, каким образом функционируют вложенные области действия, рассмотрим пример:

```
//Пример №5. Продемонстрировать область действия блока кода
class Scope {
    public static void main(String args[]) {
        int x; //переменная доступна всему коду из метода main
        x = 10;
        if (x == 10){// начало новой области действия,
            int y = 20;
            //переменные x и y доступны в этой области действия
            System.out.println("x и y :"+x+" "+y);
            x = y * 2;
        }
        //y = 100;//Ошибка! Переменная недоступна в этой области
действия. Переменная x доступна здесь
```

```
        System.out.println(" x равно "+x);
    }}
```

Ошибка:



Результаты работы программы:

```
x и y :10 20
 x равно 40
```

Переменная x объявлена в начале области действия метода *main()* и доступна всему последующему коду из этого метода. Объявление переменной y делается в блоке кода условного оператора if. А поскольку этот блок кода задает область действия, то переменная y доступна только коду из этого блока. Именно поэтому закомментирована строка кода y =100, находящаяся за пределами этого блока. Если удалить символы комментария, то это приведет к ошибке во время компиляции, поскольку переменная y недоступна за пределами своего блока кода. А к переменной x можно обращаться из блока условного оператора if, поскольку коду в этом блоке (т.е. во вложенной области действия) доступны переменные, объявленные в объемлющей области действия.

Переменные можно объявлять в любом месте блока кода, но они действительны только после объявления. Так, если переменная объявлена в начале метода, то она доступна всему коду в теле этого метода. И наоборот, если переменная объявлена в конце блока кода, она, по существу, бесполезна, так как вообще недоступна для использования. Например, следующий фрагмент кода ошибочен, поскольку переменной *count* нельзя пользоваться до ее объявления:

```
// Этот фрагмент кода написан неверно!
// Переменную count нельзя использовать до ее объявления!
count = 100;
int count;
```

Следует иметь в виду еще одну важную особенность: переменные создаются при входе в их область действия и уничтожаются при выходе из

нее. **Это означает, что переменная утратит свое значение сразу же после выхода из ее области действия.**

Следовательно, **переменные, объявленные в теле метода, не будут хранить свои значения в промежутках между последовательными обращениями к этому методу.** Кроме того, переменная, объявленная в блоке кода, утратит свое значение после выхода из него. Таким образом, срок действия переменной ограничивается ее областью действия.

Если объявление переменной включает в себя ее инициализацию, то инициализация переменной будет повторяться при каждом вхождении в блок кода, где она объявлена.

```
//Пример №6. Демонстрация срока действия переменной
class LifeTime {
  public static void main(String args[]) {
    int x;
    for (x = 0; x < 3; x++) {
      int y =-1;//переменная y инициализируется при каждом входе
в цикл for
      System.out.println("y равно: "+y);// здесь всегда
выводится значение -1
      y = 100;
      System.out.println("y теперь равно: "+y);
        }}}
```

Результаты работы программы:

```
y равно: -1
y теперь равно: 100
y равно: -1
y теперь равно: 100
y равно: -1
y теперь равно: 100
```

Переменная y постоянно инициализируется значением -1 при каждом вхождении в цикл for. И хотя переменной y впоследствии присваивается значение 100, тем не менее оно теряется.

Несмотря на то, что блоки могут быть вложенными, во внутреннем блоке кода нельзя объявлять переменные с таким же именем, что и во внешней области действия.

Например, следующая программа ошибочна:

```
int bar = 1;
while (true) {
     int bar = 2;
   }}
```

Ошибка:

В языке *Java* поддерживается большой набор операций. Большинство из них может быть отнесено к одной из следующих группа:

1. арифметические;
2. поразрядные;
3. логические;
4. операции отношения.

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

Арифметические операции применяются в математических выражениях таким же образом, как и в алгебре. Все арифметические операции, доступные в *Java*, перечислены в таблице.

Таблица – Арифметические операции в *Java*

Обозначение	**Описание**
+	Сложение (*addition*) унарный плюс
-	Вычитание (*subtraction)* или унарный минус
*	Умножение (*multiplication*)
/	Деление (*division*)
%	Деление по модулю (*division by modulus*)
++	Инкремент (*increment*, приращение на 1)
+=	Сложение с присваиванием (*addition with assignment*)
-=	Вычитание с присваиванием (*subtraction with assignment*)
*=	Умножение с присваиванием (*multiplication with assignment*)
/=	Деление с присваиванием (*division with assignment*)
%=	Деление по модулю с присваиванием (*modulo division with assignment*)
--	Декремент (decrement, уменьшение на 1)

Операнды арифметических операций должны иметь **числовой** тип. Арифметические операции нельзя выполнять над логическими типами данных, но допускается над типом *char*, поскольку в *Java* этот тип по существу является разновидностью типа *int*.

Логические операции выполняются только с операндами типа *boolean*. Все логические операции с двумя операндами соединяют два логически значения, образуя результирующее логическое значение. Все доступные в *Java* логические операции перечислены в таблице.

Таблица – Логические операции в *Java*

Операция	**Описание**
&	Логическая операция И

|	Логическая операция ИЛИ
^	Логическая операция исключающее ИЛИ
||	Укороченная логическая операция ИЛИ
&&	Укороченная логическая операция И
!	Логическая унарная операция НЕ
&=	Логическая операция И с присваиванием
|=	Логическая операция ИЛИ с присваиванием
^=	Логическая операция исключающее ИЛИ с присваиванием
==	Равенство
!=	Неравенство
? :	Тернарная условная операция типа если…, то …, иначе

Таблица – Результаты выполнения логических операций

A	B	A|B	A&B	A^B	!A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

ПРЕДШЕСТВОВАНИЕ ОПЕРАЦИЙ (ORDER OF OPERATIONS)

Предшествование (от высшего к низшему) операций в *Java* приведено в таблице на экране. Операции, находящиеся в одном ряду таблицы, имеют одинаковое предшествование. Операции с двумя операндами имеют порядок вычисления слева направо (за исключением операции присваивания, которая выполняется справа налево). Формально скобки [] и (), а также точка (.) считаются разделителями, но они могут служить и в качестве операций. Именно в этом качестве они имеют наивысшее предшествование. Обратите также внимание на операцию "стрелка" (->), которая была внедрена в версии *JDK* 8 и применяется в лямбда-выражениях.

Таблица – Предшествование операций в *Java*

Наивысшее предшествование
[], (), . круглые скобки, квадратные скобки, точка
++ (постфиксная операция), -- (постфиксная операция)
++ (префиксная операция), -- (префиксная операция), ~, !, + (унарная операция), (приведение типов)
*, /, %
+, -
>>, >>>, <<
>, >=, <, <=, instanceof
==, !=
&
^
|
&&

? :
-> (стрелка, JDK 8, лямбда-выражения)
=
Наинизшее предшествование

Круглые скобки повышают предшествование заключенных в них операций. Нередко это требуется для получения необходимого результата.

В языках программирования управляющие операторы применяются для реализации переходов и ветвлений в потоке исполнения команд программы, исходя из ее текущего состояния. Управляющие операторы в программе на *Java* можно разделить на следующие категории: операторы выбора, операторы цикла и операторы перехода.

Операторы выбора (*if, switch*) позволяют выбирать разные ветви выполнения команд в соответствии с результатом вычисления заданного выражения или состоянием переменной.

Операторы цикла (*for, while, do-while)* позволяют повторять выполнение одного или нескольких операторов (т.е. они образуют циклы).

Операторы перехода (*break, continue, return*) обеспечивают возможность нелинейного выполнения программы. Рассмотрим кратко управляющие операторы, доступные в *Java*.

ОПЕРАТОРЫ ВЫБОРА

В языке *Java* поддерживаются два оператора выбора: *if* и *switch*. Эти операторы позволяют управлять порядком выполнения команд программы в соответствии с условиями, которые известны только во время выполнения.

Условный оператор *if* является оператором условного ветвления. Его можно использовать с целью направить выполнение программы по двум разным ветвям.

```
if (условие) оператор1;
else оператор2;
```

где каждый **оператор** обозначает одиночный или составной оператор, заключенный в фигурные скобки (т.е. блок кода); **условие** – любое выражение, возвращающее логическое значение типа *boolean*. Оператор *else* указывать необязательно.

Условный оператор *if* действует следующим образом: если **условие** истинно, то выполняет **оператор1**, а иначе выполняется **оператор2**, если таковой имеется. Но ни в коем случае не будут выполняться обе ветки (*if* и *else*).

В программах очень часто встречаются **вложенные условные операторы. Пользуясь вложенными условными операторами *if*, не**

следует забывать, что оператор *else* всегда связан с ближайшим условным оператором *if*, находящимся в том же самом блоке кода.

Условные операторы *if* выполняются последовательно, сверху вниз. Как только одно из условий, управляющих оператором *if*, оказывается равным *true*, выполняется оператор, связанный с данным условным оператором *if*, а остальная часть конструкции *if-else-if* пропускается. Если ни одно из условий не выполняется (т.е. не равно *true*), то выполняется заключительный оператор *else*.

ОПЕРАТОР SWITCH

В языке *Java* оператор *switch* является оператором ветвления. Он предоставляет простой способ направить поток исполнения команд по разным ветвям кода в зависимости от значения управляющего выражения. Зачастую оператор *switch* оказывается эффективнее длинных последовательностей операторов в конструкции *if-else-if*.

Во всех версиях *Java* до *JDK* 7 указанное выражение должно иметь тип *byte, short, int, char* или перечисляемый тип. Начиная с *JDK* 7, выражение может также иметь тип *String*. Каждое значение, определенное в операторах ветвей *case*, должно быть однозначным константным выражением (например, литеральным значением). Каждое значение должно быть совместимо по типу с выражением, используемом в конструкции *switch*.

В операторе *switch* обнаруживается совпадение выражения с константой только в одной из ветвей *case*. Константы ни в одной из двух ветвей *case* того же самого оператора *switch* не могут иметь одинаковые значения. Внутренний оператор *switch* и содержащий его внешний оператор *switch* могут иметь одинаковые константы в ветвях *case*.

Как правило, оператор *switch* действует эффективнее ряда вложенных условных операторов *if* за счет того, что компилятор *Java* компилируя оператор *switch*, будет проверять константу в каждой ветви *case* и создавать "таблицу переходов", чтобы использовать ее для выбора ветви программы в зависимости от получаемого значения выражения. Поэтому в тех случаях, **когда требуется делать выбор среди большой группы значений, оператор *switch* будет выполняться значительно быстрее последовательности операторов *if-else*.** Ведь компилятору известно, что константы всех ветвей *case* имеют один и тот же тип, и их достаточно проверить на равенство значению выражения *switch*. В то же время компилятор не располагает подобными сведениями о длинном перечне выражений условного оператора *if*.

ОПЕРАТОРЫ ЦИКЛА

Для управления циклами, в *Java* предоставляются операторы *for*, *while* и *do-while*.

Оператор цикла *while* и *do-while* являются основополагающим для организации циклов в *Java*. Они повторяют оператор или блок операторов до

тех пор, пока значение его управляющего выражения истинно. **В цикле *while* условие проверяется при входе в цикл, а в цикле *do-while* после первого выполнения тела цикла.**

Когда цикл начинается, выполняется инициализация переменных цикла. В общем случае устанавливается значение переменной управления циклом, которая действует в качестве счетчика. **Важно понимать, что первая часть цикла *for*, содержащая инициализирующее выражение, выполняется только один раз.** Затем вычисляется заданное условие, которое должно быть логическим выражением. Как правило, в этом выражении значение управляющей переменной сравнивается с целевым значением.

Если результат этого сравнения истинный, то выполняется тело цикла. А если результат сравнения ложный, то цикл завершается. И наконец, выполняется третья часть цикла *for* – итерация. Обычно эта часть цикла содержит выражение, в котором увеличивается или уменьшается значение переменной управления циклом. Затем цикл повторяется, и на каждом его шаге сначала вычисляется условное выражение, затем выполняется тело цикла, а после этого вычисляется итерационное выражение. Этот процесс повторяется до тех пор, пока результат вычисления итерационного выражения не станет ложным.

В ряде случаев требуется указать несколько операторов в инициализирующей и итерационной частях оператора цикла *for*. Для решения этой задачи в *Java* предоставляется специальная возможность. Для того чтобы две переменные или больше могли управлять циклом *for*, в *Java* допускается указывать несколько операторов как в инициализирующей, так и в итерационной части оператора цикла *for*, разделяя их запятыми.

РАЗНОВИДНОСТИ ЦИКЛА FOR

У оператора цикла *for* имеется несколько разновидностей, расширяющих возможности его применения. Гибкость этого цикла объясняется тем, что три его части (инициализацию, проверку условий и итерацию) совсем не обязательно использовать только по прямому назначению. По существу, каждую часть оператора цикла *for* можно применять в любых требующихся целях.

В одной из наиболее часто встречающихся разновидностей цикла *for* предполагается употребление условного выражения. В частности, в этом выражении совсем не обязательно сравнивать переменную управления циклом с некоторым целевым значением. **По существу, условием, управляющим циклом *for*, может быть любое логическое выражение.**

```
boolean done = false;
for(int i=1; !done; i++){
```

```
        //…
        if(interrupted()) done = true;
}
```

В этом примере выполнение цикла *for* продолжается до тех пор, пока в переменной *done* не установится логическое значение *true*. В этой разновидности цикла *for* не выполняется проверка значения в переменной *i* управления циклом.

Так же в цикле *for* инициализирующее или итерационное выражения или оба вместе могут отсутствовать

```
int i=0;
boolean done = false;
//Отдельные части оператора цикла for могут отсутствовать
for (; !done;) {
        System.out.println(" i равно "+i);
        if(i == 10) done = true;
        i++;
}
```

В данном примере инициализирующее и итерационное выражения вынесены за пределы цикла *for*. В итоге соответствующие части оператора цикла *for* оказываются пустыми. Так, если начальное условие определяется сложным выражением где-то в другом месте программы или значение переменной управления циклом изменяется случайным образом в зависимости от действий, выполняемых в теле цикла, то эти части оператора цикла *for* имеет смысл оставить пустыми.

Ещё одна разновидность цикла *for* оставляет все три части оператора пустыми, создавая бесконечный цикл, т.е. такой цикл, который никогда не завершается:

```
for (;;)
```

Начиная с версии *JDK* 5, в *Java* можно использовать вторую форму цикла *for*, реализующую цикл в стиле *for each*. В современной теории языков программирования все большее применение находит понятие циклов в стиле *for each*, которые постепенно становятся стандартными средствами во многих языках программирования. **Цикл в стиле *for each* предназначен для строго последовательного выполнения повторяющихся действий над коллекцией объектов.** В отличие от некоторых языков, подобных C#, где для реализации циклов в стиле *for each* используется ключевое слово *foreach*, в *Java* возможность организации такого цикла реализована путем усовершенствования цикла *for*.

Преимущество такого подхода состоит в том, что для его реализации не требуется дополнительное ключевое слово, а уже существующий код не нарушается.

ОПЕРАТОРЫ ПЕРЕХОДА

В языке *Java* определены три оператора перехода: *break, continue, return*. Они передают управление другой части программы.

Кроме операторов перехода в *Java* поддерживается еще один способ изменения порядка выполнения инструкций программы, который состоит в обработке исключений. Обработка исключений предоставляет структурированный механизм, с помощью которого можно обнаруживать и обрабатывать ошибки во время выполнения программы. Для поддержки этого механизма служат ключевые слова *throw, throws, try, catch, finally*. По существу, **механизм обработки исключений позволяет выполнять нелокальные ветви программы.** Тема обработки исключений будет рассматривается отдельно.

В языке *Java* оператор *break* находит три применения. Во-первых, он завершает последовательность операторов в операторе *switch*. Во-вторых, его можно использовать для выхода из цикла. И в-третьих, этот оператор можно применять в качестве "цивилизованной" формы оператора безусловного перехода *goto*. Рассмотрим два последних применения данного оператора.

Необходимо помнить, что оператор *break* не должен использоваться в качестве обычного средства выхода из цикла. Для этого служит условное выражение в цикле. Этот оператор следует использовать для выхода из цикла при определенных условиях.

В языке *Java* оператор *goto* отсутствует, поскольку он позволяет выполнять ветвление программ произвольным и неструктурированным образом, создавая так называемый спагетти-код. Как правило, код, который управляется оператором *goto*, труден для понимания и сопровождения. Кроме того, этот оператор исключает возможность оптимизировать код для определенного компилятора. Но в некоторых случаях оператор *goto* оказывается удобным и вполне допустимым средством для управления потоком исполнения команд. Например, оператор *goto* может оказаться полезным при выходе из ряда глубоко вложенных циклов.

Для подобных случаев в *Java* определена расширенная форма оператора *break*. Используя эту форму, можно, например, организовать выход из одного или нескольких блоков кода. Они совсем не обязательно должны быть частью цикла или оператора *switch*, но могут быть любыми блоками кода. Более того, можно точно указать оператор, с которого будет продолжено выполнение программы, поскольку данная форма оператора *break* наделена метками. Оператор *break* с меткой предоставляет все преимущества оператора *goto*, не порождая присущие ему недостатки.

Чаще всего метка – это имя, обозначающее блок кода. Им может быть, как самостоятельный блок кода, так и целевой блок другого оператора. При выполнении этой формы оператора *break* управление передается блоку кода, помеченному меткой. Такой блок кода должен содержать оператор *break*, но

он не обязательно должен быть непосредственно объемлющим его блоком. В частности, это означает, что оператор *break* с меткой можно применять для выхода из ряда вложенных блоков. Но **его нельзя использовать для передачи управления внешнем блоку кода, который не содержит данный оператор *break*.**

Чтобы пометить блок, достаточно поместить в его начале метку. **Метка – это любой допустимый в *Java* идентификатор с двоеточием, указанный для объемлющего блока кода.** Как только блок помечен, его метку можно использовать в качестве адресата для оператора *break*. В итоге выполнение программы будет продолжено с конца помеченного блока. Тут важно уяснить что оператор *break* прерывает исполнение блока кода (составного оператора). Если метка не указана, то прерывает исполнение блока, к которому принадлежит, если метка указана, то прерывает исполнение указанного блока.

Иногда требуется, чтобы повторение цикла осуществлялось с пропуском какой-то части цикла. Это означает, что на данном конкретном шаге может возникнуть потребность продолжить выполнение цикла без выполнения остального кода в его теле. По существу, это означает переход в теле цикла к его окончанию.

Для выполнения этого действия служит оператор *continue*. В циклах *while* и *do-while* оператор *continue* вызывает передачу управления непосредственно условному выражению, управляющему циклом. В цикле *for* управление передается вначале итерационной части цикла *for*, а затем условному выражению. Во всех трех видах циклов любой промежуточный код пропускается (не выполняется).

Как и оператор *break*, оператор *continue* может содержать метку объемлющего цикла, который нужно продолжить.

Фрагмент, показанный на экране, демонстрирует пример работы оператора *continue*, который завершает текущую итерацию и начинает следующую итерацию внешнего цикла по метке *outer*. То есть прерывается текущая итерация как внутреннего цикла, так соответственно и внешнего. И затем начинается новая итерация внешнего цикла. Данный код выводит псевдо-графический треугольник на консоль.

ОПЕРАТОР RETURN

Оператор *return* служит для выполнения явного выхода из метода, т.е. передает управление объекту, который вызвал данный метод. Оператор *return* можно использовать в любом месте метода для возврата управления тому объекту, который вызвал данный метод. Следовательно, **оператор *return* немедленно прекращает выполнение метода, в теле которого он находится,** что и демонстрирует пример, показанный на экране. В данном примере выполнение оператора *return* приводит к возврату в точку вызова метода *getArea*().

Во втором примере на слайде условие *if* необходимо, так без него компилятор *Java* сигнализировал бы об ошибке типа "*unreachable code*" (недостижимый код), поскольку он выяснил бы, что последний вызов метода *println()* вообще не будет выполняться.

ПРИМЕНЕНИЕ ОПЕРАТОРА INSTANCEOF

Иногда тип (класс) объекта полезно выяснить во время выполнения программы для того, чтобы на основании имеющихся в этом типе полей и методов строить дальнейшее выполнение кода. Например, в одном потоке исполнения объекты разных типов могут формироваться, а в другом потоке исполнения – использоваться. В таком случае удобно выяснить тип каждого объекта, получаемого в обрабатывающем потоке исполнения. Тип объекта во время выполнения не менее важно выяснить и в том случае, когда требуется приведение типов.

В *Java* неправильное приведение типов влечет за собой появление ошибки во время выполнения. Большинство ошибок приведения типов может быть выявлено на стадии компиляции. Но приведение типов в пределах иерархии классов может стать причиной ошибок, которые обнаруживаются только во время выполнения. Например, суперкласс *Parent* может порождать подкласс *Child*.

Следовательно, приведение объекта класса *Child* к типу *Parent* вполне допустимо, но приведение объекта класса *Parent* к типу *Child* – неверно. А поскольку объект типа *Parent* может ссылаться на объекты любых дочерних классов, то возникает необходимость во время выполнения узнать, на какой именно тип делается ссылка перед тем, как выполнить приведение.

Для разрешения этого вопроса в *Java* предоставляется оператор времени выполнения *instanceof*, который имеет следующую форму:

```
public class Employee extends Object
public class Manager extends Employee
public class Engineer extends Employee
---------------------------------------
public void doSomething(Employee e){
   if(e instanceof Manager){
       //Process a Manager
   }
   else if(e instanceof Engineer){
       //Process an Engineer
   }
   else{
       //Process other type of Employee
   }
}
```

где *e* обозначает ссылку на экземпляр класса, а типом этой ссылки является – конкретный тип этого класса, например, *Manager* или *Engineer*. Если ссылка на объект относится к указанному типу или может быть приведена к нему, то вычисление оператора *instanceof* дает в итоге логическое значение *true*, а иначе – логическое значение *false*.

Таким образом, **оператор *instanceof* – это оператор, с помощью которого программа может получить сведения о типе объекта во время выполнения приложения.**

Большинство программ не нуждается в операторе *instanceof*, поскольку типы объектов обычно известны заранее. Но этот оператор может пригодиться при разработке обобщенных конструкций, оперирующих объектами из сложной иерархии классов.

КОММЕНТАРИИ В JAVA

Комментарии в языке *Java*, как и в большинстве других языков программирования, игнорируются при выполнении программы и не влияют на результирующий бинарный код. Таким образом, в программу можно добавлять столько комментариев, сколько потребуется, не опасаясь увеличить ее объем.

В языке Java есть три способа выделения комментариев в тексте. Чаще всего используются две косые черты //, при этом комментарий начинается сразу за символами//и продолжается до конца текущей строки.

Как правило, они используются для пояснения именно этой строки, например:

```
int y=1970;// год рождения
```

Если нужны длинные многострочные комментарии, то можно каждую строку начинать символами //. Хотя более удобно ограничивать блоки многострочных комментариев разделителями /* и */.

Блочные комментарии могут занимать произвольное количество строк, например:

```
/*Этот цикл не может начинаться с нуля
из-за особенностей алгоритма*/
for (int i=1; i<10; i++) {
...
}
```

Часто блочные комментарии оформляют следующим образом (каждая строка начинается с символа *):

```
/*
* Описание алгоритма работы
```

```
* следующего цикла while
*/
while (x > 0) {
...
}
```

Блочный комментарий не обязательно должен располагаться на нескольких строках, он может даже находится в середине оператора:

```
float s =2*Math.PI/*getRadius()*/;//Закомментировано для отладки
```

Комментарии не могут находиться внутри символьных и строковых литералов, идентификаторов. Следующий пример содержит случаи неправильного применения комментариев:

```
// В этом примере текст /*...*/ станет просто частью строки S
String s = "text/*just text*/";
/*
* Следующая строка станет причиной ошибки при компиляции,
* так как комментарий разбил имя метода getRadius()
*/
circle.get/*comment*/Radius();
```

А такой код допустим:

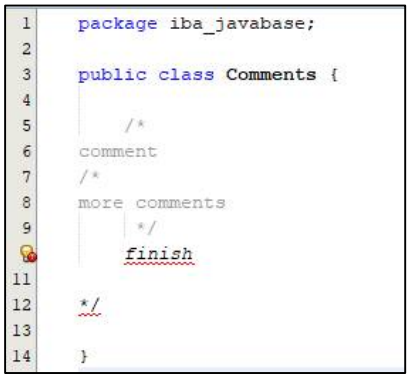
```
// Комментарий может разделять вызовы функций:
cirle./*comment*/getRadius();
// Комментарий может заменять пробелы:
int/*comment*/x=1;
```

В последней строке между названием типа данных *int* и названием переменной *x* обязательно должен быть пробел или, как в данном примере, комментарий.

Комментарии не могут быть вложенными. Символы /*, */,//не имеют никакого особенного значения внутри уже открытых комментариев, как строчных, так и блочных. Таким образом, в следующем примере

```
/* начало комментария /*///** завершение тут: */
```

описан ровно один блочный комментарий. А в следующем примере



```
1   package iba_javabase;
2
3   public class Comments {
4
5       /*
6   comment
7   /*
8   more comments
9       */
10      finish
11
12  */
13
14  }
```

компилятор выдаст ошибку. Блочный комментарий начался в строке 5 с комбинации символов /*. Вторая открывающая комбинация /* в строке 7 будет проигнорирована, так как находится уже внутри комментария. Символы */ в строке 9 завершат комментарий, а строки 10 и 12 породят ошибки.

Любые комментарии полностью удаляются из программы во время компиляции, поэтому их можно использовать неограниченно, не опасаясь, что это повлияет на бинарный код. Основное предназначение комментариев – сделать программу простой для понимания, в том числе и для других разработчиков, которым придется в ней разбираться по какой-либо причине, и для самого себя через некоторое время. Также комментарии зачастую используются для временного исключения частей кода.

Кроме этого, существует особый вид блочного комментария – комментарий разработчика (документирующий). Он применяется для автоматического создания документации кода. В стандартную поставку *JDK*, начиная с версии 1.0, входит специальная утилита *javadoc*. На вход ей подается исходный код классов, а на выходе получается удобная документация в *HTML* формате, которая описывает все классы, все их поля и методы. При этом активно используются гиперссылки, что существенно упрощает изучение программы по ее такому описанию (например, читая описание метода, можно одним нажатием мыши перейти на описание типов, используемых в качестве аргументов или возвращаемого значения). Однако понятно, что одного названия метода и перечисления его аргументов не достаточно для понимания его работы. Необходимы дополнительные пояснения от разработчика.

Комментарий разработчика записывается так же, как и блочный. Единственное различие в начальной комбинации символов – для документации комментарий необходимо начинать с /**. Например:

```
/**
* Вычисление модуля целого числа.
* Этот метод возвращает абсолютное значение аргумента x.
* @version 1.01 2020-10-01
* @author Gary Cornell
```

```
* @see java.lang.Math#PI
*/
int getAbs(int x) {
     if (x>=0) return x;
     else return -x;
}
```

Первое предложение должно содержать краткое резюме всего комментария. В дальнейшем оно будет использовано как пояснение этому методу в списке всех методов класса.

Символ * в начале каждой строки и предшествующие ему пробелы и знаки табуляции игнорируются. Их можно вообще не использовать, но они удобны, когда важно форматирование.

Так же *javadoc* поддерживает специальные теги. Они начинаются с символа @. Подробное описание этих тегов можно найти в документации. Например, можно использовать тег @see, чтобы сослаться на другой класс, поле или метод, или даже на другой интернет-сайт.

```
/**
* Краткое описание.
*
* Развернутый комментарий.
*
* @see java.lang.String
* @see java.lang.Math#PI
* @see <a href="http://www.oracle.com">Official Java site</a>
*/
```

Первая ссылка указывает на класс *String* (*java.lang* – название библиотеки, в которой находится этот класс), вторая – на поле *PI* класса *Math* (символ # разделяет название класса и его полей или методов), третья ссылается на официальный сайт *Java*.

Комментарии разработчика могут быть записаны перед объявлением классов, интерфейсов, полей, методов и конструкторов. Если записать комментарий /** ... */ в другой части кода, то ошибки не будет, но он не попадет в документацию, генерируемую *javadoc*.

Все классы стандартных библиотек *Java* поставляются в виде исходного текста. Стандартная документация по этим классам сгенерирована утилитой *javadoc*. Для любой программы можно также легко подготовить подобное описание, необходимы лишь грамотные и аккуратные комментарии в исходном коде.

ОСНОВНЫЕ СВЕДЕНИЯ О КЛАССАХ

Классы составляют сущность изучение *Java*, фундамент, на котором основывается вся платформа Java, поскольку класс определяет природу объекта. Следовательно, классы служат основанием для объектно-

ориентированного программирования на *Java*. В классе определяются данные и код, который выполняет действия над этими данными. Код находится внутри методов. Имея представление о классах, объектах и методах, вы сможете писать сложные программы на любом объектно-ориентированном языке.

Код любой программы, написанной на *Java*, находится в пределах класса. Именно поэтому мы начали использовать классы уже с первых примеров программ. Разумеется, мы ограничивались лишь самыми простыми классами и не пользовались большинством их возможностей. Как станет ясно в дальнейшем, классы намного более эффективное языковое средство, чем можно было бы предположить.

Класс представляет собой шаблон, по которому определяется вид объекта. **В классе указываются данные и код, который будет оперировать этими данными.** *Java* использует спецификацию («начинку») класса для конструирования *объектов*. Объекты – это *экземпляры* классов. Таким образом, класс фактически представляет собой описание, в соответствии с которым должны создаваться объекты.

Важно, чтобы вы понимали: класс – это логическая абстракция. **Физическое представление класса в оперативной памяти появится лишь после того, как будет создан объект этого класса.**

Следует так же иметь в виду, что методы и переменные, составляющие класс, принято называть *элементами* класса. Для данных-*элементов* класса существует также другое название: *переменные экземпляра*.

ОБЩАЯ ФОРМА ОПРЕДЕЛЕНИЯ КЛАССА

Определяя класс, вы объявляете его **конкретный вид и поведение**. Для этого указываются содержащиеся в нем переменные экземпляра и оперирующие ими методы. Если самые простые классы могу содержать только код или только данные, то большинство реальных классов содержат и то, и другое.

Класс создается с помощью ключевого слова *class*. Ниже приведена упрощенная общая форма определения класса.

```
class ClassName {
//переменные экземпляра (переменные класса)
    type variableName1;
    type variableName2;
    //......
    type variableNameN;
    //методы класса
    тип methodName1(parameters){
        //тело метода
    }
    тип methodName2(parameters){
        //тело метода
    }
    тип methodNameN(parameters){
```

```
        //тело метода
    }}
```

Несмотря на отсутствие соответствующего правила в синтаксисе *Java*, правильно сконструированный класс должен определять одну и только одну логическую сущность. Например, класс, в котором хранятся имена абонентов и номера их телефонов, обычно не будет содержать сведения о фондовом рынке, среднем уровне осадков, периодичности солнечных пятен или другую не относящуюся к данному объекту информацию. Таким образом, **в хорошо спроектированном классе должна быть сгруппирована логически связанная информация.** Если же в один и тот же класс помещается логически несвязанная информация, то структурированность кода быстро нарушается.

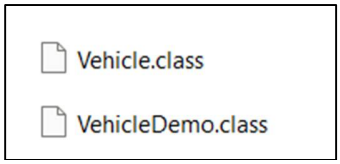
Объявление класса создает новый тип данных. В данном случае этот тип называется *Vehicle*. Мы будем использовать это имя для объявления объектов типа *Vehicle*. Вспомните, что **объявление класса – это всего лишь описание типа данных, а реальный объект при этом не создается.** Следовательно, приведенный выше код не приводит к появлению объектов типа *Vehicle*.

```
/*Example №1. Использование класса Vehicle. Присвойте файлу с
исходным кодом имя VehicleDemo.java*/
class Vehicle {
    int passengers; //количество пассажиров
    int fuelcap; //емкость топливного бака
    int mpg;//потребление топлива в милях на галлон
}
//В этом классе объявляется объект типа Vehicle
class VehicleDemo {
    public static void main(String args[]) {
        Vehicle minivan = new Vehicle();
        int range;
//Присвоить значения полям в объекте minivan
        minivan.passengers = 7;
        minivan.fuelcap = 16;//использование точечной нотации
для доступа к переменным экземпляра
        minivan.mpg = 21;
//Рассчитать дальность поездки при полном баке
        range = minivan.fuelcap * minivan.mpg;
        System.out.println("Мини-фургон может перевезти " +
minivan.passengers +" пассажиров на расстояние "+ range + "
миль");
    }}
```

Результаты работы программы:

```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:C:\Program
Мини-фургон может перевезти 7 пассажиров на расстояние 336 миль
```

Файлу, содержащему приведенный выше код, следует присвоить имя *VehicleDemo.java*, поскольку метод *main()* находится не в классе *Vehicle*, а в классе *VehicleDemo*. В результате компиляции программы будут созданы два файла с расширением *.class*: один – для класса *Vehicle*, а другой – для класса *VehicleDemo*.



Компилятор *Java* автоматически помещает каждый класс в отдельный файл с расширением *.class*. Совсем не обязательно, чтобы классы *Vehicle* и *VehicleDemo* находились в одном и том же исходном файле. Их можно расположить в двух файлах: *Vehicle.java* и *VehicleDemo.java*.

МЕТОДЫ КЛАССА

Поля класса (переменные экземпляра) и методы являются двумя основными составляющими классов.

До сих пор класс *Vehicle* содержал только данные. **Хотя классы, содержащие только данные, вполне допустимы, у большинства классов должны быть также методы. *Методы*** представляют собой подпрограммы, которые манипулируют данными, определенными в классе, а во многих случаях они представляют доступ к этим данным. Как правило, другие части программы взаимодействуют с классом посредством его методов.

Метод состоит из одной или нескольких инструкций. В правильно написанной программе на *Java* каждый метод выполняет только одну функцию или задачу, суть которой отображается в названии метода.

У каждого метода имеется свое имя, которое используется для его вызова. В целом в качестве имени метода можно использовать любой действительный идентификатор. Следует, иметь в виду, что **идентификатор *main()* зарезервирован для метода, с которого начинается выполнение программы.** Кроме того, в качестве имен методов нельзя использовать ключевые слова языка *Java*.

Часто используется соглашение, что после имени метода стоит пара круглых скобок. Так, если методу присвоено имя *getVal*, то он упоминается в тексте как *getVal()*. Такая форма записи позволяет отличать имена методов от имен переменных. Общий синтаксис объявления метода.

```
возвращаемый_тип имя(список_параметров){
    //тело метода
}
```

Здесь **возвращаемый_тип** обозначает тип данных, возвращаемых методом. **Им может быть любой допустимый тип, в том числе и тип класса, который вы создаете.** Если метод не возвращает значение, то для него указывается тип *void*. Далее следует **имя**, которое обозначает конкретное имя, присваиваемое методу. В качестве имени метода может быть использован любой допустимый идентификатор, не приводящий к конфликтам в текущей области действия. И наконец, **список_параметров** – это последовательность разделенных запятыми параметров, для каждого из которых указывается тип и имя. Параметры представляют собой переменные, которые получают значения, передаваемые им в виде **аргументов** при вызове метода. Если у метода отсутствуют параметры, список параметров оказывается пустым.

Для того чтобы добавить метод в класс *Vehicle*, его следует объявить в пределах этого класса. Например, приведенный ниже вариант класса *Vehicle* содержит метод *range()*, определяющий и отображающий дальность поездки транспортного средства.

```
//Example №3. Добавление метода range() в класс Vehicle
class Vehicle {
    int passengers;//количество пассажиров
    int fuelcap;//емкость топливного бака
    int mpg; //потребление топлива в милях на галлон
    //рассчитать и отобразить дальность поездки
    void range() {
        System.out.println("Дальность поездки транспортного средства " + fuelcap * mpg + " миль.");
        //Обратите внимание на указание переменных fuelcap и mpg без использования точечной нотации
    }
}
class AddMeth {
  public static void main(String args[]) {
    Vehicle minivan = new Vehicle();
    Vehicle sporstcar = new Vehicle();
    //Присвоить значения полям в объекте minivan
  int range1, range2;
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;
    //Присвоить значения полям в объекте sporstcar
    sporstcar.passengers = 2;
    sporstcar.fuelcap = 14;
    sporstcar.mpg = 12;
    System.out.print("Мини-фургон может перевезти " + minivan.passengers + " пассажиров. ");
    minivan.range();//отобразить дальность поездки мини-фургона
```

```
   System.out.print("Спортивный автомобиль может перевезти " +
sporstcar.passengers + " пассажров. ");
   sporstcar.range();//отобразить дальность поездки спортивного
автомобиля
    }}
```

Результат работы программы:

```
Мини-фургон может перевезти 7 пассажиров. Дальность поездки транспортного средства 336 миль.
Спортивный автомобиль может перевезти 2 пассажров. Дальность поездки транспортного средства 168 миль.
```

Рассмотрим основные элементы данной программы. Начнем с метода *range()*. Первая строка кода этого метода выглядит так:

```
void range() {
```

В этой строке объявляется метод с именем *range*, для которого не предусмотрены параметры. В качестве типа, возвращаемого этим методом, указано ключевое слово *void*. Таким образом, метод *range()* не возвращает вызывающей части программы никаких данных. И завершается строка открывающей фигурной скобкой, обозначающей начало тела метода. Тело метода *range()* состоит из единственной строки кода:

```
System.out.println("Дальность поездки транспортного средства " +
fuelcap * mpg + " миль.");
```

В этой строке на экран выводится дальность поездки транспортного средства как результат перемножения переменных *fuelcap* и *mpg*. А поскольку у каждого объекта типа *Vehicle* имеются свои копии переменных *fuelcap* и *mpg*, то при вызове метода *range()* используются данные текущего объекта.

Действие метода *range()* завершается по достижении закрывающей фигурной скобки его тела. При этом управление возвращается вызывающей части программы.

Рассмотрим подробнее следующую строку кода в методе main*()*:

```
minivan.range();
```

В этой строке кода вызывается метод *range()* для объекта *minivan*. Чтобы вызвать метод для конкретного объекта, следует указать имя этого объекта перед именем метода, используя точечную нотацию. При вызове метода ему передается управление потоком выполнения программы. Когда метод завершит свое действие, управление будет возвращено вызывающей части программы, и ее выполнение продолжится со строки кода, следующей за вызовом метода.

В данном случае в результате вызова *minivan.range()* отображается дальность поездки транспортного средства, определяемого объектом *minivan*. Точно так же при вызове *sporstcar.range()* на экран выводится дальность поездки транспортного средства, определяемого объектом *sporstcar*. При каждом вызове метода *range()* выводится дальность поездки для указанного объекта.

Необходимо отметить следующую особенность метода *range()*: обращение к переменным экземпляра *fuelcap* и *mpg* осуществляется в нем без использования точечной нотации. Если в методе используется переменная экземпляра, определенная в его классе, обращаться к ней можно напрямую, не указывая объект. Следует признать, что такой подход вполне логичен. Ведь метод всегда вызывается относительно некоторого объекта своего класса, а, следовательно, при вызове метода объект известен и нет никакой необходимости определять его еще раз. Это означает, что переменные *fuelcap* и *mpg*, встречающиеся в теле метода *range()*, неявно обозначают их копии, находящиеся в том объекте, для которого вызывается метод *range()*.

Параметризированный метод позволяет реализовать в классе *Vehicle* новую возможность: расчет объема топлива, необходимого для преодоления заданного расстояния. Назовем этот новый метод *fuelNeeded()*. Он получает в качестве параметра расстояние в милях, которое должно проехать транспортное средство, а возвращает необходимое для этого количество галлонов топлива. Метод *fuelNeeded()* определяется следующим образом.

```
double fuelNeeded(int miles) {
     return (double) miles / mpg;
}
```

Обратите внимание на то, что этот метод возвращает значение типа *double*. Это важно, поскольку объем потребляемого топлива не всегда можно выразить целым числом. Ниже приведен исходный код программы для расчета дальности поездки транспортных средств с классом *Vehicle*, содержащим метод *fuelNeeded*().

```
//Example №7. Добавление параметризированного метода, в котором производится расчет объема топлива, необходимого транспортному средству для преодоления заданного расстояния. ИЗМЕНИТЬ ПРОГРАММУ ТАК, ЧТОБЫ УБРАТЬ ПРОМЕЖУТОЧНУЮ ПЕРЕМЕННУЮ GALLONS, А РЕЗУЛЬТАТ ЕЁ ВЫПОЛНЕНИЯ ОСТАЛСЯ ПРЕЖНИМ
class Vehicle {
    int passengers;//количество пассажиров
    int fuelcap;//емкость топливного бака
    int mpg;//потребление топлива в милях на галлон
    //Определить дальность поездки транспортного средства
     int range() {
```

```
        return mpg * fuelcap;
    }
   //Рассчитать объем топлива, для преодоления заданного расстояния
    double fuelNeeded(int miles) {
        return (double) miles / mpg;
    }
}
class CompFuel {
    public static void main(String args[]) {
        Vehicle minivan = new Vehicle();
        Vehicle sporstcar = new Vehicle();
        double gallons;
        int dist = 252;
        //Присвоить значения полям в объекте minivan
        minivan.passengers = 7;
        minivan.fuelcap = 16;
        minivan.mpg = 21;
        //Присвоить значения полям в объекте sporstcar
        sporstcar.passengers = 2;
        sporstcar.fuelcap = 14;
        sporstcar.mpg = 12;
        gallons = minivan.fuelNeeded(dist);
        System.out.println("Для преодоления " + dist + " миль мини-фургону требуется "+ gallons + " галлонов топлива");
        gallons = sporstcar.fuelNeeded(dist);
        System.out.println("Для преодоления " + dist + " миль спортивному автомобилю требуется "+ gallons + " галлонов топлива");
    }}
```

Результаты работы программы:

```
Для преодоления 252 миль мини-фургону требуется 12.0 галлонов топлива
Для преодоления 252 миль спортивному автомобилю требуется 21.0 галлонов топлива
```

КОНСТРУКТОРЫ КЛАССА

Конструктор инициализирует объект при его создании. Имя конструктора совпадает с именем класса, а с точки зрения синтаксиса он подобен методу. **Но у конструкторов нет возвращаемого типа, указываемого явно**. Как правило, конструкторы используются для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта.

Конструкторы имеются у всех классов, независимо от того, определите вы их или нет, поскольку ***Java* автоматически предоставляет конструктор, используемый по умолчанию и инициализирующий все переменные экземпляра их значениями по умолчанию.**

Для большинства типов данных значением по умолчанию является нулевое значение, для типа *boolean* – логическое значение *false*, а для ссылочных типов – пустое значение *null*.

Но как только вы определите свой собственный конструктор, конструктор по умолчанию предоставляться не будет.

Теперь мы можем усовершенствовать класс *Vehicle*, добавив в него конструктор, в котором будут автоматически инициализироваться поля *passengers*, *fuelcap* и *mpg* при построении объекта. Обратите особое внимание на то, каким образом создаются объекты типа *Vehicle*.

```
//Example №10. Использование конструктора
class Vehicle {
    int passengers;//количество пассажиров
    int fuelcap;//емкость топливного бака
    int mpg;//потребление топлива в милях на галлон
    //Определить дальность поездки транспортного средства
    Vehicle(int p, int f, int m) {//Конструкор класса
VehConsDemo
        passengers = p;
        fuelcap = f;
        mpg = m;
    }
//Определить дальность поездки транспортного средства
    int range() {
        return mpg * fuelcap;
    }
//Рассчитать объем топлива, для преодоления заданного расстояния
    double fuelNeeded(int miles) {
        return (double) miles / mpg;
    }}
class VehConsDemo {
    public static void main(String args[]) {
        Vehicle minivan = new Vehicle(7, 16, 21);
        Vehicle sporstcar = new Vehicle(2, 14, 12);
        double gallons;
        int dist = 252;
        gallons = minivan.fuelNeeded(dist);
        System.out.println("Для преодоления " + dist + " миль
мини-фургону требуется " + gallons + " галлонов топлива");
        gallons = sporstcar.fuelNeeded(dist);
        System.out.println("Для преодоления " + dist + " миль
спортивному автомобилю требуется " + gallons + " галлонов
топлива");
    }}
```

При создании объекты *minivan* и *sporstcar* инициализируются конструктором *Vehicle()*. Каждый такой объект инициализируется

параметрами, указанными в конструкторе его класса. Например, в строке кода

```
Vehicle minivan = new Vehicle(7, 16, 21);
```

значения 7, 16 и 21 передаются конструктору *Vehicle()* при создании нового объекта *minivan* с помощью оператора *new*.

В итоге копии переменных *passengers*, *fuelcap* и *mpg* в объекте *minivan* будут содержать значения 7, 16 и 21 соответственно. Рассмотренная здесь версия программы дает такой же результат, как и ее предыдущая версия.

КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Покажите два способа объявления одномерного массива, состоящего из 12 элементов типа *double*.

2. Покажите все варианты, которыми можно воспользоваться для инициализации элементов одномерного массива целочисленными значениями от 1 до 5.

3. Напишите программу, в которой массив используется для нахождения среднего арифметического десяти значений типа *double*. Используйте любые десять чисел.

4. Напишите программу, которая сортирует массив символьных строк в лексикографическом порядке по убыванию и по возрастанию. Продемонстрируйте ее работу.

5. В чем состоит разница между методами *indexOf()* и *lastIndexOf()* класса *String*?

6. Перепишите приведенную ниже последовательность операторов, воспользовавшись тернарным оператором.

```
if(x < 0) y= 10;
        else y= 20;
```

8. Является ли ошибкой превышение верхней границы массива?

9. Является ли ошибкой использование отрицательных значений индекса для доступа к элементам массива?

10. Можно ли управлять оператором *switch* с помощью объектов типа *String*? Если да, то напишите пример такого использования.

Задания для выполнения на лабораторном занятии:

1. Приветствовать пользователя при вводе его имени через командную строку.

2. Отобразить в окне консоли аргументы командной строки в обратном порядке.

3. Вывести заданное количество случайных чисел с переходом и без перехода на новую строку.

4. Ввести пароль из командной строки и сравнить его со строкой-образцом. Отобразить результаты сравнения.

5. Ввести целые числа как аргументы командной строки, подсчитать их сумму и произведение. Вывести результат на консоль.

6. Ввести с консоли *n* целых чисел. На консоль вывести: четные и нечетные числа, наибольшее и наименьшее число, числа, которые делятся на 3 или на 9, элементы, расположенные методом пузырька по убыванию модулей.

ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.

2. Ответить на контрольные вопросы лабораторной работы.

3. Разработать алгоритм программы по индивидуальному заданию.

4. Написать, отладить и проверить корректность работы созданной программы.

5. Написать электронный отчет по выполненной лабораторной работе.

Отчет должен быть оформлен по стандарту БГУИР (Стандарт предприятия СТП 01-2017 "Дипломные проекты (работы). Общие требования"**) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания, разбитый на необходимые типы файлов
6. скриншоты выполнения индивидуального задания
7. диаграмму созданных классов в нотации UML
8. выводы по лабораторной работе

ВО ВСЕХ ЗАДАНИЯХ ПОЛЬЗОВАТЕЛЬ ДОЛЖЕН САМ РЕШАТЬ ВЫЙТИ ИЗ ПРОГРАММЫ ИЛИ ПРОДОЛЖИТЬ ВВОД ДАННЫХ. ВСЕ РЕШАЕМЫЕ ЗАДАЧИ ДОЛЖНЫ БЫТЬ РЕАЛИЗОВАНЫ, ИСПОЛЬЗУЯ КЛАССЫ И ОБЪЕКТЫ.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Создать массив переменных типа *String*, инициализировать массив названиями месяцев от января до декабря (в русском и английском вариантах). Создать второй массив, содержащий 12 случайных десятичных значений в интервале от 0.0 до 100.0. Уточнить у пользователя какой он язык предпочитает и вывести название каждого месяца вместе с соответствующим десятичным значением. Вычислить и вывести на экране среднее из полученных случайных 12 значений.

2. Написать программу, которая создает переменную *String*, содержащую параграф текста на выбор. Извлечь слова из текста и отсортировать их в алфавитном порядке. Вывести отсортированный список слов. Найти слово максимальной длины.

3. Создать массив из десяти переменных типа *String*, каждая из которых содержит произвольную строку – месяц/день/год в следующем формате 01/10/2020. Проанализировать каждый элемент в массиве и вывести представление даты в формате 10 января 2020 года.

4. Написать программу для создания случайной последовательности из *n* прописных букв, которая не включает гласные буквы. Число ***n*** вводит пользователь с клавиатуры.

5. Написать программу для создания случайной последовательности из *n* строчных букв, которая не включает гласные буквы. Число ***n*** вводит пользователь с клавиатуры.

6. Создать объект типа *String* и проинициализировать его текстовой строкой. Определить количество гласных, пробелов, цифр, специальных знаков и общее количество букв.

7. Создать программу, которая проверяет корректность введенного идентификатора переменной на языке *Java*. Пользователь вводит название идентификатора, а программа сообщает допустимый ли это идентификатор или нет. Момент завершения тестирования идентификаторов определяет пользователь.

8. Создать массив объектов типа *String* и проинициализировать его следующими текстовыми строками: «To be or not to be that is the question», «I am a student of economical department», «My name is John», «Hello world». Воспользоваться методом *indexOf()* класса *String* для определения в массиве подстроки «be», «not to», «am», «department», «hello». Вывести порядковый номер каждой найденной подстроки.

9. Создать массив объектов типа *String* и проинициализировать его следующими строками: «To;be:or*not;to:be*that;is:the*question», «I;am:a*student;of:economical*department», «My;name:is*John», «Hello;world». Использовать метод *indexOf()* класса *String* совместно с методом *substring()* для извлечения из исходного массива строк последовательности подстрок, которые разделены символами «;», «:»,«*».

10. Написать программу, с помощью которой создается последовательность точек и набор линий, соединяющих каждую пару последовательных точек. Затем вычисляется общая длина линии, соединяющей все точки. Найти среди рассчитанных линий отрезок минимальной и максимальной длины. Найти сумму длин всех линий.

11. Написать программу для создания прямоугольного массива, содержащего таблицу умножения от 1×1 до 12×12. Вывести таблицу как набор столбцов с числовыми значениями, выровненными как показано на

рисунке: первая строка вывода – это заголовки столбцов без заголовка для первого столбца, затем числа от 1 до 12 для остальных столбцов. Первый элемент в каждой из последующих строк является заголовком строки, изменяющимся от 1 до 12.

12. Диаметр Солнца равен приблизительно 864 000 милям (1 391 980 километров), а диаметр Земли – 7 926 милям (12 756 километров). Вычислить с помощью методов класса *Math*:

– объем Земли в кубических милях и кубических километрах;

– объем Солнца в кубических милях и кубических километрах;

– отношение объема Солнца к объему Земли.

Вывести рассчитанные значения на экран. Считать, что Земля и Солнце являются шарами. Объем шара задается формулой $4\pi r^3/3$, где r – радиус шара.

13. Написать программу, которая по трем введенным с клавиатуры точкам определит вид треугольника: прямоугольный, равнобедренный, равносторонний или разносторонний. Момент завершения ввода точек определяет пользователь.

14. Написать программу, которая в матрице произвольного порядка определит индекс строки с минимальным элементом и индекс столбца с максимальным элементом этой матрицы. Матрицу и её размер пользователь вводит с клавиатуры.

15. Написать программу, которая в матрице произвольного порядка определит отношение среднего значения элементов, расположенных на главной диагонали, к среднему значению элементов, расположенных на побочной диагонали этой матрицы. Матрицу и её размер пользователь вводит с клавиатуры.

16. Написать программу для переворачивания (обращения, reversing) строки, изменив расположение символов в строке задом наперёд.

17. Напишите программу, которая находит второе по величине число в массиве.

18. Написать программу, которая выводит на консоль все символы белорусского и польского алфавитов в верхнем и нижнем регистре.

19. Запрограммировать вычисление следующего выражения: (a + b — f / a) + f * a * a — (a + b). Числа a, b, f вводятся с клавиатуры. Организовать пользовательский интерфейс, таким образом, чтобы было понятно, в каком порядке должны вводиться числа.

20. Напишите программу, которая позволяет пользователю ввести в консоли латинскую букву нижнего регистра, переводит её в верхний регистр и результат выводит в консоль.

21. Составить программу, которая в бесконечном цикле просит ввести два числа. Если первое число больше второго, то программа печатает фразу "первое число больше второго". Если первое число меньше второго, то

программа печатает фразу "первое число меньше второго". А если числа равны, программа напечатает сообщение "числа равны".

22. Составить программу-тест по языку программирования. На экране по очереди появляются вопросы, с вариантами ответов. В конце работы программа выдает количество заработанных баллов по результатам ответов.

23. С помощью оператора цикла for, разработать программу, которая будет выводить таблицу умножения для числа, введенного пользователем с клавиатуры.

24. Создать класс Book, который содержит данные о годе издания и наименовании книги. Создать массив объектов этого класса. Предложить пользователю внести данные и записать их в объекты. Вывести сохраненные в объектах данные на экран.

25. Создать массив размера n (полученного от пользователя), заполнить массив случайными числами. Вывести на экран все нечётные числа массива и их количество.

26. Создать класс, описывающий работника, со свойствами: фамилия, стаж, часовая заработная плата, количество отработанных часов. Реализовать ввод данных работника с клавиатуры. Рассчитать с помощью методов класса заработную плату за отработанное время, и премию, размер которой определяется в зависимости от стажа (при стаже до 1 года 0%, до 3 лет 5%, до 5 лет 8%, свыше 5 лет 15%). С помощью метода печати, реализовать вывод информации о работнике на экран.

27. Результаты соревнований по прыжкам в длину представлены в виде матрицы 5x3 (5 спортсменов по 3 попытки у каждого). Найти, какой спортсмен и в какой попытке показал наилучший результат.

28. Создать классы, которые будут хранить информации о знаке зодиака человека (surname — фамилия; zodiacSign— знак зодиака; day — день рождения. Программа, должна уметь выполнять следующие действия: ввод с клавиатуры данных в массив объектов; вывод на экран информации о людях, родившихся в месяц, значения которого введено с клавиатуры, если таких нет, то выдать соответствующее сообщение.

29. Создать класс *Matrix*, в котором реализовать методы для работы с матрицами: перемножение матриц, сложение матриц. Создать массив объектов этого класса. Предложить пользователю внести данные и записать их в объекты. Вывести сохраненные в объектах данные на экран.

30. Создать класс *Square*. Поле класса хранит длину стороны квадрата. Методы-элементы класса возвращают площадь, периметр, устанавливают поля и возвращают значения полей класса, выводят данные о полях класса на экран. Создать массив объектов этого класса. Предложить пользователю внести данные и записать их в объекты. Вывести сохраненные в объектах данные на экран.