GeekBrains

Java Core для Android

Урок 1

Введение в платформу Java

Введение в платформу Java, инструменты разработчика, написание первой программы. Переменные, типы данных, арифметические операции. Методы. Условные операторы. Оператор switch, циклы, кодовые блоки, массивы. Разбор практических примеров использования базовых элементов языка Java, работа с консолью. Введение в объектно-ориентированное программирование, классы, объекты, конструкторы, инкапсуляция, модификаторы доступа. Разбор практических примеров применения ООП при разработке приложений. Работа с классами String, StringBuilder, StringBuffer.

Особенности платформы Java

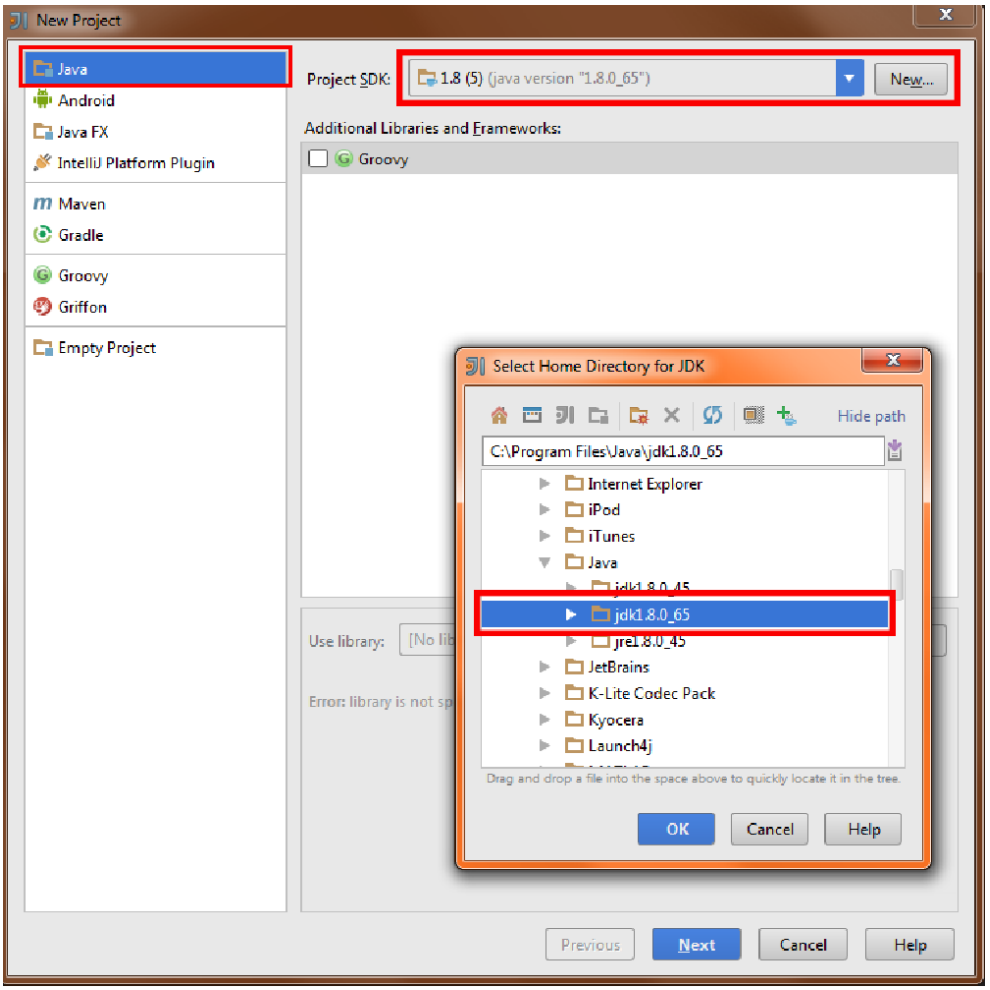
Простота	Язык Java обладает лаконичными, тесно связанными друг с другом и легко усваиваемыми языковыми средствами, был задуман как простой в изучении и эффективный в употреблении язык программирования.
Безопасность	Java предоставляет безопасные средства для создания интернет-приложений.
Переносимость	Программы на Java могут выполняться в любой среде, где есть исполняющая система Java (например, Windows, Linux, Android, MacOS и т.д.).
Объектно-ориентированный характер программирования	В Java воплощена современная философия объектно-ориентированного программирования.
Надежность	Java уменьшает вероятность появления ошибок в программах благодаря строгой типизации данных и выполнению соответствующих проверок во время выполнения. Java исключает ошибки по работе с памятью за счет автоматического управления резервированием и освобождением памяти.
Многопоточность	Язык Java обеспечивает встроенную поддержку многопоточного программирования и предоставляет множество удобных средств для решения задач синхронизации процессов. Это позволяет строить устойчиво работающие интерактивные системы.
Архитектурная независимость	Язык Java не привязан к конкретному типу вычислительной машины или архитектуре операционной системы и следует принципу «написано однажды – работает всегда».
Интерпретируемость и высокая производительность	Java предоставляет байт-код, обеспечивающий независимость от платформы. Компилируя программы в промежуточное представление, называемое байт-кодом, Java позволяет создавать межплатформенные программы, которые будут выполняться в любой системе, где реализована виртуальная машина JVM. Байт-код Java максимально оптимизируется для повышения производительности.

Инструменты разработчика

Для написания программ необходимо установить инструменты разработчика Java Development Kit (JDK), которые свободно предоставляется компанией Oracle. Загрузите JDK версии 8 можно загрузить Восьмая версия JDK не является самой последней в настоящий момент, однако ее более чем достаточно для того, чтобы начать изучение языка Java.

Бесплатную версию среды разработки IntelliJ IDEA Community Edition можно скачать с сайта разработчиков.

При создании первого проекта в IntelliJ IDEA необходимо указать путь к установленному JDK, как показано на рисунке ниже. Project SDK -> New -> JDK -> путь к папке jdk1.x.x_xxx.



Первая программа

Давайте посмотрим на то, как выглядит самая простая программа, написанная на языке Java. Создадим новый проект и добавим в него FirstApp.java.

```
/**
 * Created by GeekBrains on 15.11.2018.
 */
public class FirstApp {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

В первых строчках кода мы видим так называемые комментарии, они могут быть как однострочные (начинаются с символов //), так и многострочные (блок текста, заключенный в /* … */). Комментарии

упрощают работу с кодом, разработчик может делать пометки, которые не будут влиять на размер и скорость выполнения программы.

```
// Однострочный комментарий
/*
...
Блочный комментарий
...
*/
```

После комментариев идёт объявление класса FirstApp.

Важно! Имя класса должно совпадать с именем файла, в котором он объявлен, т.е. класс FirstApp должен находиться в файле FirstApp.java

```
public class FirstApp {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Далее идёт объявление метода main(), с которого начинается исполнение приложения.

Важно! Выполнение любой Java-программы начинается с вызова метода main(). В одном классе может быть объявлен только один метод main(). Если методы main() объявлены в нескольких классах, то вы можете выбирать с какого класса необходимо запускать ваше приложение.

Ключевое слово void сообщает компилятору, что метод main() не возвращает значений. Методы могут также возвращать конкретные значения. Что такое модификаторы доступа (public, private), ключевое слово static и String[] args, будет объяснено позже, когда зайдет разговор об основах ООП и массивах.

Важно! При написании кода на языке Java обязательно необходимо учитывать регистр символов. (Main не равнозначно main, Void и void не одно и то же, System не равнозначно system и т.д.)

Важно! Если вы работаете в IntelliJ IDEA, для упрощения/ускорения написания метода main, вы можете пользоваться сокращением psvm.

В следующей строке кода System.out.println("**Hello, World!**"), которая находится в теле метода main(), в консоль выводится текстовая строка **Hello, World!** с последующим переводом каретки на новую строку. На самом деле вывод текста на экран выполняется встроенным методом println(), который кроме текста может также выводить числа, символы, значения переменных и т.д.

Важно! В языке Java все операторы обычно должны оканчиваться символом ; . После закрытия кодовых блоков (там, где блок закрывается скобкой }) точка с запятой не ставится, потому что эти фигурные скобки не являются операторами.

Практика! Попробуйте создать новый Java проект в IntelliJ IDEA, добавить в него класс FirstApp, прописать в этом классе метод main() и код для вывода сообщения в консоль. Как только все это будет готово - запустите проект, и проверьте что в консоли выведено, то что указано в System.out.println(); Если есть расхождения - сравните с кодом, представленным выше.
Дополнительно: Попробуйте написать в круглых скобках System.out.println() какой-нибудь другой текст, целое или дробное число, и проверьте что получится.

Теперь вы знаете как написать самую простую программу на языке Java и запустить ее в IntelliJ IDEA.

Переменные и типы данных

Переменные представляют собой зарезервированную область памяти для хранения данных. В зависимости от типа переменной операционная система выделяет память и решает, что именно должно в ней храниться.

В Java существует две группы типов данных.

- Примитивные.
- Ссылочные (объектные).

Существует 8 примитивных типов данных.

Тип	Описание	Возможные значения	Пример
byte	8-битное знаковое целое число	от -128 до 127	byte val = -120;
short	16-битное знаковое целое число	от -32768 до 32767	short val = 12442;
int	32-битное знаковое целое число	от -2147483648 до 2147483647	int val = 1000;
long	64-битное знаковое целое число	от -9223372036854775808 до 9223372036854775807	long val = 200000L;
float	32-битное знаковое число с плавающей запятой одинарной точности		float val = 12.23f;
double	64-битное знаковое число с плавающей запятой двойной точности		double val = -123.123;
char	16-битный тип данных, предназначенный для хранения символов в кодировке Unicode	от '\u0000' или 0 до '\uffff' или 65,535	char val1 = ‘*’; char val2 = ‘\u2242’;
boolean	логический тип данных	false, true	boolean val = false;

Важно! Если вы хотите указать float величину, то после числа необходимо поставить букву f.
float floatVal = 12.24f;
Если буква указана не будет, то компилятор будет считать такое дробное число типом double.
При использовании переменной типа long, после числа необходимо ставить букву L.

```
long longValue = 20000000000L;
```

Важно! Если вы только начинаете программировать, то нет необходимости запоминать количество бит для хранения переменной определенного типа, или точные границы значений. Достаточно понять какие значения, в каком типе можно хранить.

Заметка. Если вы только начали осваивать программирование, то можете на первых этапах пользоваться для работы с целыми числами переменными типа int, для дробных чисел float. А при дальнейшем обучении разберетесь с остальными типами.

Ссылочных типов данных существует большое количество, кроме того, можно создавать новые ссылочные типы, но обо всем этом будет рассказано на следующих занятиях.

Общую структуру объявления переменной можно описать так:

```
[тип_данных] [идентификатор(имя_переменной)]; // объявление переменной
[тип_данных] [идентификатор(имя_переменной)] = [начальное_значение]; // объявление переменной с инициализацией
```

```
public class FirstApp {
    public static void main(String[] args) {
        int a = 20;
        float b;
        b = 2.25f;
    }
}
```

Идентификаторы – это имена переменных, которые начинаются с буквы, $ или _, после чего может идти любая последовательность символов. Идентификаторы чувствительны к регистру.

Важно! Имена переменных должны быть написаны в camelCase - первая буква строчная, каждое следующее слово в имени с заглавной буквы и без нижних подчеркиваний или пробелов.
Правильные имена:
floatValue, name, enginePower, firstName, lastName;
Неправильные имена:
Name, Title, EnginePower, First_name, last_name

В качестве идентификаторов нельзя использовать ключевые слова Java .

Заметка. К ключевым словам языка Java относятся: abstract assert boolean break byte case catch char class const continue default do double else enum extends final finally float for goto if implements import instanceof int interface long native new package private protected public return short static strictfp super switch synchronized this throw throws transient try void volatile while

Чтобы переменная не могла менять свое значение в процессе выполнения программы, можно определить её как константу с помощью ключевого слова final, если написать его перед указанием типа данных переменной.

```
public class FirstApp {
    public static void main(String[] args) {
        final int a = 20;
    }
}
```

Динамическая инициализация переменных. Начальные значения переменных могут рассчитываться на основе значений других переменных. В примере ниже, для заполнения переменной volume используются значения переменных radius и height.

```
public class FirstApp {
    public static void main(String[] args) {
        float radius = 2.0f, height = 10.0f;
        // volume инициализируется динамически во время выполнения программы
        float volume = 3.1416f * radius * radius * height;
        System.out.println("Объем цилиндра равен " + volume);
    }
}
```

Инициализация нескольких переменных в одну строку. Если требуется несколько переменных одного типа, их можно объявить в одном операторе через запятую.

```
public class FirstApp {
    public static void main(String[] args) {
        int x, y, z;
        x = y = z = 10; // присвоить значение 10 переменным x, y и z
        float d = 2.2f, e = 7.2f;
    }
}
```

Арифметические операции

Вы можете выполнять обычные арифметические операции над числовыми переменными.

Операция	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Деление по модулю
++	Инкремент (приращение на 1)
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
*=	Умножение с присваиванием

/=	Деление с присваиванием
%=	Деление по модулю с присваиванием
--	Декремент (отрицательное приращение на 1)

Посмотрим, как это будет выглядеть в коде.

```
public class FirstApp {
    public static void main(String args[]) {
        int a = 10;
        int b = 20;
        int c = (a + b - 5) * 2;
        System.out.println("c = " + c);
    }
}
```

В примере выше мы объявили и проинициализировали две целочисленные переменные a и b, а затем объявили и проинициализировали переменную c, которая вычисляется с использованием выражения, состоящего из набора простых арифметических операций.

Ещё одна простая программа

Вот так может выглядеть ещё одна программа, написанная на языке Java.

```
public class FirstApp {
    public static void main(String args[]) {
        int a;
        int b;
        a = 128;
        System.out.println("a = " + a);
        b = a / 2;
        System.out.println("b = a / 2 = " + b);
    }
}
```

Важно!

- System.out.println(); отвечает за печать сообщений в консоль. В качестве аргумента вы можете подавать любые значения: строки, числа, объекты, логические типы данных и т.д.
- В процессе обучения вам постоянно придется пользоваться этим методом, поэтому с первого же занятия надо запомнить как он пишется.
- Если вы работаете в IntelliJ IDEA, то для быстрого набора этого метода можно использовать сокращение sout.

Выполнение начинается с первой строки метода main() и идет последовательно сверху вниз. Первые две строки в методе main() означают объявление двух целочисленных переменных с идентификаторами a и b. Затем в переменную a записывается число 128 и выводится сообщение в консоль – «a = 128». Затем значение переменной b вычисляется через значение переменной a, как b = a / 2 (т.е. b = 128 / 2 = 64), и в консоль выводится сообщение «b = a / 2 = 64».

Методы

Общая форма объявления метода выглядит следующим образом.

```
тип_возвращаемого_методом_значения имя_метода (список_параметров) {
   тело_метода;
}
```

Тип_возвращаемого_методом_значения обозначает конкретный тип данных (int, float, char, boolean, String и т.д.), возвращаемых методом. Если метод ничего не должен возвращать, указывается ключевое слово void. Для возврата значения из метода используется оператор return.

```
return значение;
```

Для указания имени метода служит идентификатор «имя». Список параметров обозначает последовательность пар «тип_данных + идентификатор», разделенных запятыми. По существу, параметры – набор данных, необходимых для работы метода. Если у метода отсутствуют параметры, то список их оказывается пустым.

Несколько примеров работы с методами.

```
public class FirstApp {
    public static void main(String[] args) {
        // для вызова метода необходимо передать ему 2 аргумента типа int,
        // результатом работы будет целое число, которое напечатается в консоль
        System.out.println(summ(5, 5));

        // для вызова метода ему не нужно передавать аргументы,
        // и он не возвращает данные (метод объявлен как void)
        printSomeText();

        // для вызова метода передаем ему в качестве аргумента строку "Java",
        // которую он выведет в консоль
        printMyText("Java");
    }

    // метод возвращает целое число, принимает на вход два целых числа
    public static int summ(int a, int b) {
        // возвращаем сумму чисел
        return a + b;
    }

    // метод ничего не возвращает, не требует входных данных
    public static void printSomeText() {
        // печатаем Hello в консоль
        System.out.println("Hello");
    }

    // метод ничего не возвращает, принимает на вход строку
    public static void printMyText(String txtToPrint) {
```

```
        // выводим строку txtToPrint в консоль
        System.out.println(txtToPrint);
    }
}
```

При объявлении всех методов внутри основного класса программы после public должно идти слово static. Если ключевое слово static будет отсутствовать, этот метод не получится вызвать из метода main(). Смысл этого ключевого слова будет пояснен на следующих занятиях в теме «Объектно-ориентированное программирование».

Условный оператор if

Условный оператор if позволяет выборочно выполнять отдельные части программы. Ниже приведена простейшая форма оператора if.

```
if (условие) {
   последовательность_операторов;
}
```

Здесь условие обозначает логическое выражение. Если (условие) истинно (true), последовательность операторов выполняется, если ложно (false) – не выполняется, например.

```
public class FirstApp {
    public static void main(String args[]) {
        if (5 < 10) {
            System.out.println("5 меньше 10");
        }
    }
}
```

В данном примере числовое значение 5 меньше 10, и поэтому условное выражение принимает логическое значение true, а следовательно, выполняется метод println().

Рассмотрим ещё один пример с противоположным условием.

```
public class FirstApp {
    public static void main(String args[]) {
        if (10 < 5) {
            System.out.println("Это сообщение никогда не будет выведено");
        }
    }
}
```

Теперь числовое значение 10 не меньше 5, а следовательно, метод println() не вызывается, и в консоль ничего не выводится.

Могут быть использованы следующие операторы сравнения.

Оператор	Значение
<	Меньше

<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно

Обратите внимание, что для проверки на равенство указывается два знака равно.

Для проверки значения типа boolean используется запись.

```
public class FirstApp {
    public static void main(String args[]) {
        boolean bool = true;
        if (bool) { // если bool == true
            // ...
        }
        if (!bool) { // если bool == false
            // ...
        }
    }
}
```

Ниже приведён пример программы, демонстрирующий применение оператора if.

```
public class FirstApp {
    public static void main(String args[]) {
        // объявляем и инициализируем три переменные
        int a = 2, b = 3, c = 0;

        if (a < b) {      // если a меньше b
            System.out.println("a меньше b");
        }
        if (a == b) {    // если a равно b
            System.out.println("a равно b. Это сообщение не будет выведено");
        }
        c = a - b; // переменная c = 2 - 3 = -1
        System.out.println("c = " + c);
        if (c >= 0) {
            System.out.println("c не отрицательно");
        }
        if (c < 0) {
            System.out.println("c отрицательно");
        }
        c = b - a; // переменная c = 3 - 2 = 1
        System.out.println("c = " + c);
        if (c >= 0) {
            System.out.println("c неотрицательно");
        }
        if (c < 0) {
            System.out.println("c отрицательно");
        }
```

```
    }
}

// Результат:
// a меньше b
// c = -1
// c отрицательно
// c = 1
// c неотрицательно
```

Ещё один вариант условного оператора if-else представлен ниже. Если условие верно, выполняется последовательность_операторов_1, если нет – последовательность_операторов_2 (из блока else).

```
if (условие) {
  последовательность операторов 1
} else {
  последовательность операторов 2
}
```

При использовании условий можно составлять более сложные конструкции с помощью логических операторов И(&&) и ИЛИ(||).

```
if (условие1 && условие2) {
   ...
}
if (условие1 || условие2) {
   ...
}
if ((условие1 && условие2) || условие3) {
   ...
}
```

В первом случае (логическое И) для выполнения кода из блока if, необходимо чтобы и условие 1, и условие 2 одновременно были верны. Во втором случае (логическое ИЛИ) достаточно, чтобы хотя бы одно из условий было верно.

Оператор switch

Оператор switch позволяет делать выбор между несколькими вариантами дальнейшего выполнения программы. Выражение последовательно сравнивается со списком значений оператора switch. При совпадении выполняется набор операторов, связанных с этим условием. Если совпадений не было, выполняется блок default (блок default является необязательной частью оператора switch). Оператор break останавливает выполнение блока case, если break убрать – выполнение кода продолжится дальше.

```
switch (выражение) {
   case значение1:
      набор_операторов1;
      break;
   case значение 2:
      набор_операторов2;
      break;
   ...
```

```
   default:
      набор_операторов;
}
```

Например, последовательность if-else-if-...

```
public static void main(String[] args) {
    int a = 3;
    if (a == 1) {
        System.out.println("a = 1");
    } else if (a == 3) {
        System.out.println("a = 3");
    } else {
        System.out.println("Ни одно из условий не сработало");
    }
}
```

может быть заменена на следующее.

```
public static void main(String[] args) {
    int a = 3;
    switch (a) {
        case 1:
            System.out.println("a = 1");
            break;
        case 3:
            System.out.println("a = 3");
            break;
        default:
            System.out.println("Ни один из case не сработал");
    }
}
```

Циклы for

Циклы позволяют многократно выполнять последовательность кода.

```
for (инициализация; условие; итерация) {
   набор_операторов;
}
```

Инициализация представлена переменной, выполняющей роль счётчика и управляющей циклом (например, int i = 0;). Условие определяет необходимость повторения цикла. Итерация задаёт шаг изменения переменной, управляющей циклом.

Важно! Инициализация, условие и итерация в круглых скобках должны быть разделены ;.
Важно! После закрытой круглой скобки точки с запятой **нет**. Если вы там ее поставите, цикл будет работать некорректно.

Выполнение цикла for продолжается до тех пор, пока проверка условия даёт истинный результат (true). Пример.

```
public static void main(String args[]) {
    for (int i = 0; i < 5; i++) {
        System.out.println("i = " + i);
    }
    System.out.println("end");
}
Результат:
i = 0
i = 1
i = 2
i = 3
i = 4
end
```

В начале каждого шага цикла проверяется условие i < 5. Если это условное выражение верно, вызывается тело цикла (в котором прописан метод System.out.println(...)), затем выполняется итерационная часть цикла. Как только условное выражение примет значение false, цикл закончит свою работу.

Пример цикла с отрицательным приращением счётчика

Ниже приведён пример цикла с отрицательным приращением цикла. Еще одной особенностью цикла является «вынос» объявления управляющей переменной до начала цикла, хотя обычно она объявляется внутри for.

```
public static void main(String args[]) {
    int x;  // объявление управляющей переменной вынесено до начала цикла
    for (x = 10; x >= 0; x -= 5) { // Шаг -5
        System.out.print(x + " ");
    }
}
Результат:
10 5 0
```

Условное выражение цикла for всегда проверяется в начале цикла. Это означает, что код в цикле может вообще не выполняться, если проверяемое условие с начала оказывается ложным. Пример.

```
public static void main(String args[]) {
    int x = 0;
    for (int count = 10; count < 5; count++) {
        x += count; // этот оператор не будет выполнен, так как 10 > 5
    }
}
```

Этот цикл вообще не будет выполняться, поскольку начальное значение переменной count больше 5. Это значит, что условное выражение count < 5 оказывается ложным с самого начала.

Цикл for с несколькими управляющими переменными

Для управления циклом можно использовать одновременно несколько переменных. В примере ниже за одну итерацию переменная i увеличивается на 1, а j уменьшается на 1.

```
public static void main(String args[]) {
    for (int i = 0, j = 10; i < j; i++, j--) {
        System.out.println("i-j: " + i + "-" + j);
    }
}
Результат:
i-j: 0-10
i-j: 1-9
i-j: 2-8
i-j: 3-7
i-j: 4-6
```

Бесконечный цикл

При использовании следующей записи цикла for можно получить бесконечный цикл. Большинство таких циклов требуют специальное условие для своего завершения.

```
for (;;) {
   // ...
}
```

Выход из работающего цикла осуществляется оператором **break** без выполнения всего кода из тела цикла, поэтому в результате нет числа 4.

```
public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
        if (i > 3) {
            break;
        }
        System.out.println("i = " + i);
    }
}
Результат:
i = 0
i = 1
i = 2
i = 3
```

Цикл foreach

Еще одной разновидностью цикла for является цикл foreach. Он используется для прохождения по всем элементам массива или коллекции, знать индекс проверяемого элемента не нужно. В приведённом ниже примере мы проходим по элементам массива sm типа String и каждому присваиваем временное имя o, то есть «в единицу времени» o указывает на один элемент массива.

```
public static void main(String[] args) {
    String[] sm = {"A", "B", "C", "D"};
    for (String o : sm) {
        System.out.print(o + " ");
    }
}
Результат:
A B C D
```

Вложенные циклы

Циклы, работающие внутри других циклов, называют вложенными. Внимательно разберите последовательность исполнения таких циклов. На всё выполнение внутреннего цикла приходится одна итерация внешнего. Пример.

```
public static void main(String args[]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            System.out.print(" " + i + j);
        }
    }
}
Результат:
 00 01 02 10 11 12 20 21 22
```

Циклы while

Цикл while работает до тех пор, пока указанное условие истинно. Как только условие становится ложным, управление программой передается строке кода, следующей непосредственно после цикла. Если заранее указано условие, которое не выполняется, программа в тело цикла даже не попадет.

```
while (условие) {
   набор_операторов;
}
```

Цикл do-while очень похож на ранее рассмотренные циклы. В отличие от for и while, где условие проверялось в самом начале (предусловие), в цикле do-while оно проверяется в самом конце (постусловие). Это означает, что цикл do-while всегда выполняется хотя бы один раз.

```
do {
   набор_операторов;
while (условие);
```

Кодовые блоки

Кодовый блок представляет собой группу операторов. Для оформления в виде блока они помещаются между открывающей и закрывающей фигурными скобками. Созданный кодовый блок становится единым логическим блоком. Такой блок можно использовать при работе с if и for. Пример.

```
public static void main(String args[]) { // <- начало кодового блока main
    int w = 1, h = 2, v = 0;
    if (w < h) {                          // <- Начало кодового блока if
        v = w * h;
        w = 0;
    }                                     // <- Конец кодового блока if
}                                         // <- Конец кодового блока main
```

В данном примере оба оператора в блоке выполняются в том случае, если значение переменной w меньше значения переменной h. Эти операторы составляют единый логический блок, и ни один из них не может быть выполнен без другого. Кодовые блоки позволяют оформлять многие алгоритмы в удобном для восприятия виде. Ниже приведён пример программы, где кодовый блок предотвращает деление на ноль.

```
public class MainClass {
   public static void main(String args[]) {
       int a = 0, b = 10, c = 0;
       if (a != 0) {
           System.out.println("a не равно нулю");
           c = b / a;
           System.out.print("b / a равно " + c) ;
       } else {
           System.out.println("a = 0. Делить на 0 нельзя");
       }
   }
}
Результат:
a = 0. Делить на 0 нельзя
```

Области видимости переменных в кодовых блоках.

```
public static void main(String args[]) { // Кодовый блок метода main()
    int x = 10;         // эта переменная доступна для всего кода в методе main
    if (x == 10) {      // Кодовый блок тела if
        int y = 20;     // Эта переменная доступна только в данном кодовом блоке
// Обе переменные x и y доступны в данном кодовом блоке
        System.out.println("x & y: " + x + " " + y);
        x = y * 2;
    }
// y = 100; // Ошибка! Переменная y недоступна за пределами тела if
    System.out.println("x = " + x);          // Переменная x по-прежнему доступна
}
```

Ещё один пример объявления переменных в цикле.

```
public static void main(String args[]) {
    for (int i = 0; i < 3; i++) {
        int y = -1;                                          // переменная y
инициализируется при каждом входе в блок
        System.out.println("y = " + y); // всегда выводится значение -1
        y++;
        System.out.println("y = " + y);
```

```
    }
}
```

Массивы

Массив представляет собой набор однотипных переменных с общим именем.

Одномерные массивы

Для объявления одномерного массива обычно применяется следующая форма.

```
тип_данных[] имя_массива = new тип_данных[размер_массива];
```

При создании массива сначала объявляется переменная, ссылающаяся на него. Затем выделяется память для массива, в Java динамически распределяется с помощью оператора new; ссылка на неё присваивается переменной. В следующей строке кода создается массив типа int, состоящий из 5 элементов, ссылка на него присваивается переменной arr.

```
int[] arr = new int[5];
```

В переменной arr сохраняется ссылка на область памяти для массива оператором new. Этой памяти должно быть достаточно для размещения в ней 5 элементов типа int. Доступ к отдельным элементам массива осуществляется с помощью индексов. Индекс обозначает положение элемента в массиве, индекс первого элемента равен нулю. Если массив arr содержит 5 элементов, их индексы находятся в пределах от 0 до 4. Индексирование массива осуществляется по номерам его элементов, заключенным в квадратные скобки. Например, для доступа к первому элементу массива arr следует указать arr[0], а для доступа к последнему элементу этого массива — arr[4]. В приведенном ниже примере программы в массиве arr сохраняются числа от 0 до 4.

```
public static void main(String args[]) {
    int[] arr = new int[5];
    for(int i = 0; i < 5; i++) {
        arr[i] = i;
        System.out.println("arr[" + i + "] = " + arr[i]);
    }
}
Результат:
arr[0] = 0
arr[1] = 1
arr[2] = 2
arr[3] = 3
arr[4] = 4
```

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
0	1	2	3	4

Заполнять созданные массивы можно последовательным набором операторов.

```
public static void main(String args[]) {
    int[] nums = new int[4];
    nums[0] = 5;
    nums[1] = 10;
    nums[2] = 15;
    nums[3] = 15;
}
```

В приведённом выше примере массив nums заполняется через четыре оператора присваивания. Существует более простой способ решения этой задачи: заполнить массив сразу при его создании.

```
тип_данных[] имя_массива = {v1, v2, v3, ..., vN} ;
```

Здесь v1-vN обозначают первоначальные значения, которые присваиваются элементам массива слева направо по порядку индексирования, при этом Java автоматически выделит достаточный объем памяти. Например.

```
public static void main(String args[]) {
    int[] nums = { 5, 10, 15, 20 };
}
```

Границы массива в Java строго соблюдаются. Если обратиться к несуществующему элементу массива, будет получена ошибка. Пример.

```
public static void main(String args[]) {
    int[] arr = new int[10];
    for(int i = 0; i < 20; i++) {
        arr[i] = i;
    }
}
```

Как только значение переменной i достигнет 10, будет сгенерировано исключение ArrayIndexOutOfBoundsException и выполнение программы прекратится.

Распечатать одномерный массив в консоль можно с помощью конструкции Arrays.toString().

```
import java.util.Arrays;

public class MainClass {
   public static void main(String args[]) {
       String[] arr = {"A", "B", "C", "D"};
       System.out.println(Arrays.toString(arr));
   }
}
Результат:
[A, B, C, D]
```

Двумерные массивы

Среди многомерных массивов наиболее простыми являются двумерные. Двумерный массив – это ряд одномерных массивов. При работе с двумерными массивами проще их представлять в виде таблицы, как будет показано ниже. Объявим двумерный целочисленный табличный массив table размером 10x20.

```
int[][] table = new int[10][20];
```

В следующем примере создадим двумерный массив размером 3x4, заполним его числами от 1 до 12 и отпечатаем в консоль в виде таблицы.

```
public static void main(String args[]) {
    int counter = 1;
    int[][] table = new int[3][4];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            table[i][j] = counter;
            System.out.print(table[i][j] + " ");
            counter++;
        }
        System.out.println();
    }
}
```

	j = 0	j = 1	j = 2	j = 3
i = 0	1	2	3	4
i = 1	5	6	7	8
i = 2	9	10	11	12

При работе с отладкой и двумерными массивами для их распечатки можно пользоваться следующим методом. На вход метода необходимо подать ссылку на любой двумерный целочисленный массив. Первый индекс массива указывает на строку, второй – на столбец.

```
public static void printArr(int[][] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[i].length; j++) {
            System.out.print(arr[i][j]);
        }
        System.out.println();
    }
}
```

Нерегулярные массивы

Выделяя память под многомерный массив, достаточно указать лишь первый (крайний слева) размер. Память под остальные размеры массива можно выделять по отдельности.

```
int[][] table = new int[3][];
table[0] = new int[1];
table[1] = new int[5];
table[2] = new int[3];
```

Поскольку многомерный массив является массивом массивов, существует возможность установить разную длину массива по каждому индексу. В некоторых случаях такие массивы могут значительно повысить эффективность работы программы и снизить потребление памяти, например, если требуется создать очень большой двумерный массив, в котором используются не все элементы.

Многомерные массивы

В Java допускаются n-мерные массивы, ниже показана форма объявления.

```
тип_данных[][]...[] имя_массива = new тип_данных[размер1][размер2]...[размерN];
```

В качестве примера ниже приведено объявление трехмерного целочисленного массива размерами 2x3x4.

```
int[][][] mdarr = new int[2][3][4];
```

Многомерный массив можно инициализировать. Инициализирующую последовательность нужно заключить в отдельные фигурные скобки.

```
тип_данных[][] имя_массива = {
   { val, val, val, ..., val },
   { val, val, val, ..., val },
   { val, val, val, ..., val }
};
```

Альтернативный синтаксис объявления массивов

Помимо рассмотренной выше общей формы для объявления массива можно также пользоваться следующей формой.

```
тип_данных имя_массива[];
```

Два следующих объявления массивов равнозначны.

```
public static void main(String[] args) {
    int arr[] = new int[3];
    int[] arr2 = new int[3];
}
```

Получение длины массива

При работе с массивами имеется возможность программно узнать его размер. Для этого можно воспользоваться записью *имя_массива.length*. Это удобно использовать, когда нужно пройти циклом for по всему массиву.

```
public static void main(String[] args) {
    int[] arr = {2, 4, 5, 1, 2, 3, 4, 5};
    System.out.println("arr.length: " + arr.length);
    for (int i = 0; i < arr.length; i++) {
        System.out.print(arr[i] + " ");
    }
}
Результат:
arr.length: 8
2 4 5 1 2 3 4 5
```

Ввод данных из консоли

Для ввода данных из консоли можно воспользоваться объектом класса Scanner (вопрос, что такое классы и объекты, будет подробно рассмотрен на 5 занятии).

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in); // создание объекта класса Scanner
     int a = sc.nextInt();                                    // чтение целого числа в
переменную a
    String b = sc.nextLine();                          // чтение введенной строки
     String c = sc.next();                                    // слово до следующего
пробела
    sc.close();    // после завершения работы со сканером его необходимо закрыть,
}
```

Пример программы, запрашивающей у пользователя ввод целого числа и выводящей в консоль число в 2 раза больше.

```
import java.util.Scanner;
public class MainClass {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Введите число: ");
        int a = sc.nextInt();
        a *= 2;
        System.out.println("Введенное вами число, умноженное на 2, равно " + a);
```

```
        sc.close();
    }
}
```

Как же сделать ввод данных в заданных пределах?

```
import java.util.Scanner;
public class MainClass {
    public static Scanner sc = new Scanner(System.in);

    public static void main(String[] args) {
          int d = getNumberFromScanner("Введите число в пределах от 5 до 10", 5,
10);
        System.out.println("d = " + d);
    }

    public static int getNumberFromScanner(String message, int min, int max) {
        int x;
        do {
            System.out.println(message);
            x = sc.nextInt();
        } while (x < min || x > max);
        return x;
    }
}
Результат:
Введите число в пределах от 5 до 10
8
d = 8
```

Метод getNumberFromScanner() будет запрашивать у пользователя целое число до тех пор, пока оно не окажется в пределах от min до max включительно. Перед каждым запросом будет выводится сообщение, которое передано в message. Повторный запрос осуществляется с помощью цикла do/while. Мы будем запрашивать у пользователя ввод числа до тех пор, пока он будет пытаться указать число меньше минимального или больше максимального.

Что такое класс

Класс определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Наиболее важная особенность класса состоит в том, что он определяет новый тип данных, которым можно воспользоваться для создания объектов этого типа, т.е. класс — это шаблон (чертеж), по которому создаются объекты (экземпляры класса). Для определения формы и сущности класса указываются данные, которые он должен содержать, а также код, воздействующий на эти данные.

Если мы хотим работать в нашем приложении с документами, то необходимо для начала объяснить что такое документ, описать его в виде класса (чертежа) Document. Рассказать какие у него должны быть свойства: название, содержание, количество страниц, информация о том, кем он подписан и т.д. В этом же классе мы описываем что можно делать с документами: печатать в консоль, подписывать, изменять содержание, название и т.д. Результатом такого описания и будет наш класс Document. Однако это по-прежнему всего лишь чертеж. Если нам нужны конкретные документы, то необходимо создавать объекты: документ №1, документ №2, документ №3. Все эти документы будут иметь одну и

ту же структуру (название, содержание, …), с ними можно выполнять одни и те же действия, НО наполнение будет разным (в первом документе содержится приказ о назначении работника на должность, во втором, о выдаче премии отделу разработки и т.д.).

Заметка. Приведенный пример является абстрактным, и нет необходимости говорит о том, что структура таких документов будет совершенно разной.

Посмотрим на упрощенную форму объявления класса:

```
модификатор_доступа class имя_класса {
    модификатор_доступа тип_переменной имя_поля; // первое поле
    модификатор_доступа тип_переменной имя_поля; // второе поле
    // ...
    модификатор_доступа тип_переменной имя_поля; // n-е поле

    модификатор_доступа имя_коструктора(список_аргументов) {
        // ...
    }

    модификатор_доступа тип_метода имя_метода(список_аргументов) {
        // тело метода
    }
    // ...
    модификатор_доступа тип_метода имя_метода(список_аргументов) {
        // тело метода
    }
}
```

Пример класса User (пользователь):

```
public class User {
    private int id;
    private String name;
    private String position;
    private int age;

    public User(int id, String name, String position, int age) {
        this.id = id;
        this.name = name;
        this.position = position;
        this.age = age;
    }

    public void info() {
        System.out.println("id: " + id + "; Имя пользователя: " + name+ "; 
Должность: " + position + "; Возраст: " + age);
    }
```

```
    public void changePosition(String position) {
        this.position = position;
        System.out.println("Пользователь " + name + " получил новую должность: "
+ position);
    }
}
```

Как правило, переменные, объявленные в классе, описывают свойства будущих объектов, а методы - их поведение. Например, в классе User (пользователь) можно объявить переменные: int id, String name, String position, int age; которые говорят о том, что у пользователя есть идентификационный номер (id), имя (name), должность (position) и возраст (age). Методы info() и changePosition(), объявленные в классе User, означают что мы можем выводить информацию о нем в консоль (info) и изменять его должность (changePosition).

Переменные, объявленные в классе, называются полями экземпляра, каждый объект класса содержит собственные копии этих переменных, и изменение значения поля у одного объекта никак не повлияет на это же поле другого объекта.

Важно! Код должен содержаться либо в теле методов, либо в блоках инициализации и не может “висеть в воздухе”, как показано в следующем примере.

```
public class User {
    // ...

    public void info() {
        System.out.println("id: " + id + "; Имя пользователя: " + name + "; 
Должность: " + position + "; Возраст: " + age);
    }

    age++; // Ошибка
    System.out.println(age); // Ошибка

    public void changePosition(String position) {
        this.position = position;
        System.out.println("Пользователь " + name + " получил новую должность: " +
position);
    }
}
```

Поля экземпляра и методы, определённые в классе, называются членами класса. В большинстве классов действия над полями осуществляются через его методы.

Первый класс

Представим, что нам необходимо работать в нашем приложении с котами. Java ничего не знает о том, что такое коты, поэтому нам необходимо создать новый класс (тип данных), и объяснить что же такое кот. Для этого начнем потихоньку прописывать класс Cat. Пусть у котов есть три свойства: name (кличка), color (цвет) и age (возраст); и они пока ничего не умеют делать.

```
public class Cat {
    String name;
    String color;
    int age;
}
```

Важно! Имя класса должно совпадать с именем файла, в котором он объявлен, т.е. класс Cat должен находиться в файле Cat.java

Итак, мы рассказали Java что такое коты, теперь если мы хотим создать в нашем приложении кота, следует воспользоваться следующим оператором.

```
Cat cat1 = new Cat();
```

Подробный разбор того, что происходит в этой строке, будет проведен в следующем пункте. Пока же нам достаточно знать, что мы создали объект типа Cat (экземпляр класса Cat), и для того чтобы с ним работать, положили его в переменную, которой присвоили имя cat1. На самом деле, в переменной не лежит весь объект, а только ссылка где его искать в памяти, но об этом позже.

Объект cat1 создан по “чертежу” Cat, и значит у него есть поля name, color, age, с которыми можно работать (получать или изменять их значения). Для доступа к полям объекта служит операция-точка, которая связывает имя объекта с именем поля. Например, чтобы присвоить полю color объекта cat1 значение “White”, нужно выполнить следующий оператор:

```
cat1.color = "Белый";
```

Операция-точка служит для доступа к полям и методам объекта по его имени. Рассмотрим пример консольного приложения, работающего с объектами класса Cat.

```
public class CatDemoApp {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
        Cat cat2 = new Cat();
        cat1.name = "Барсик";
        cat1.color = "Белый";
        cat1.age = 4;
        cat2.name = "Мурзик";
        cat2.color = "Черный";
        cat2.age = 6;
        System.out.println("Кот1 имя: " + cat1.name + " цвет: " + cat1.color + "
возраст: " + cat1.age);
        System.out.println("Кот2 имя: " + cat2.name + " цвет: " + cat2.color + "
возраст: " + cat2.age);
    }
}
```

Результат работы программы:

```
Кот1 имя: Барсик цвет: Белый возраст: 4
Кот2 имя: Мурзик цвет: Черный возраст: 6
```

Вначале мы создали два объекта типа Cat: cat1 и cat2, соответственно они имеют одинаковый набор полей (name, color, age), однако каждому из них мы в эти поля записали разные значения. Как видно

из результатом печати в консоле, изменение значения полей одного объекта, никак не влияет на значения полей другого объекта. Данные объектов cat1 и cat2 изолированы друг от друга.

Создание объектов

Как создавать новые типы данных (классы) мы разобрались, мельком посмотрели и как создаются объекты наших классов. Давайте теперь поподробнее разберем как создавать объекты, и что при этом происходит.

Создание объекта проходит в два этапа. Сначала создается переменная, имеющая интересующий нас тип (в данном случае Cat), в нее мы сможем записать ссылку на будущий объект (поэтому при работе с классами и объектами мы говорим о ссылочных типах данных). Затем необходимо выделить память под наш объект, создать и положить объект в выделенную часть памяти, и сохранить ссылку на этот объект в памяти в нашу переменную.

Заметка. Область памяти, в которой создаются и хранятся объекты, называется кучей (heap).

Для непосредственного создания объекта применяется оператор new, который динамически резервирует память под объект и возвращает ссылку на него, в общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором new.

```
public static void main(String[] args) {
    Cat cat1;
    cat1 = new Cat();
}
```

В первой строке кода переменная cat1 объявляется как ссылка на объект типа Cat и пока ещё не ссылается на конкретный объект (первоначально значение переменной cat1 равно null). В следующей строке выделяется память для объекта типа Cat, и в переменную cat1 сохраняется ссылка на него. После выполнения второй строки кода переменную cat1 можно использовать так, как если бы она была объектом типа Cat. Обычно новый объект создается в одну строку (*Cat cat1 = new Cat()*).

Подробное рассмотрение оператора new

Оператор new динамически выделяет память для нового объекта, общая форма применения этого оператора имеет следующий вид:

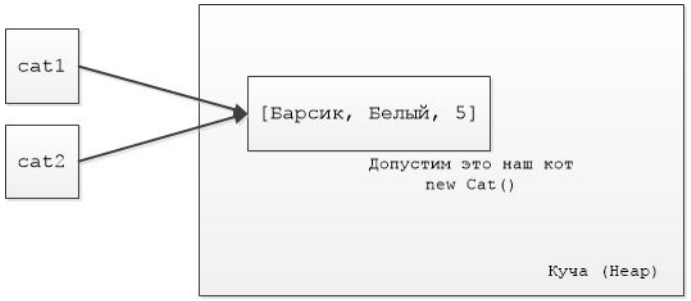
```
Имя_класса имя_переменной = new Имя_класса();
```

Имя_класса() в правой части выполняет вызов конструктора данного класса, который позволяет подготовить наш объект к работе.

Рассмотрим еще один пример, в котором создаются новые объекты.

```
public static void main(String[] args) {
    Cat cat1 = new Cat();
    Cat cat2 = cat1;
}
```

На первый взгляд может показаться, что переменной cat2 присваивается ссылка на копию объекта cat1, т.е. переменные cat1 и cat2 будут ссылаться на разные объекты в памяти. **Но это не так**. На самом деле cat1 и cat2 будут ссылаться на один и тот же объект. Присваивание переменной cat1 значения переменной cat2 не привело к выделению области памяти или копированию объекта, лишь к тому, что переменная cat2 ссылается на тот же объект, что и переменная cat1.



Таким образом, любые изменения, внесённые в объекте по ссылке cat2, окажут влияние на объект, на который ссылается переменная cat1, поскольку это один и тот же объект в памяти.

Конструкторы

Давайте еще раз взглянем на один из предыдущих примеров.

```
public class CatDemoApp {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
        cat1.name = "Барсик";
        cat1.color = "Белый";
        cat1.age = 4;
        System.out.println("Кот1 имя: " + cat1.name + " цвет: " + cat1.color + "
возраст: " + cat1.age);
    }
}
```

Чтобы создать объект мы тратим одну строку кода (*Cat cat1 = new Cat()*). Поля этого объекта заполнятся автоматически значениями по-умолчанию (целочисленные - 0, логические - false, ссылочные - null и т.д.). Нам бы хотелось дать коту какое-то имя, указать его возраст и цвет, поэтому мы прописываем еще три строки кода. В таком подходе есть несколько недостатков: во-первых, мы напрямую обращаемся к полям объекта (чего не стоит делать, в соответствии с принципами инкапсуляции, о которых речь пойдет чуть позже), а во-вторых, если полей у класса будет намного больше, то для создания всего лишь одного объекта будет уходить 10-20+ строк кода, что очень неудобно. Было бы неплохо иметь возможность сразу при создании объекта указывать значения его полей.

Для инициализации объектов при создании в Java предназначены **конструкторы**. Имя конструктора обязательно должно совпадать с именем класса, а синтаксис аналогичен синтаксису метода. Если создаться конструктор класса Cat как показано ниже, он автоматически будет вызываться при создании объекта.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat() {
        System.out.println("Это конструктор класса Cat") ;
        name = "Барсик";
        color = "Белый";
        age = 2;
```

```
    }
}

public class MainClass {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
    }
}
```

Теперь, при создании объектов класса Cat, все коты будут иметь одинаковые имена, цвет и возраст (а именно, белый двухлетний Барсик).

Заметка. Еще раз обращаем внимание, что в строке `Cat cat1 = new Cat();` подчеркнутая часть кода и есть вызов конструктора класса Cat.

Важно! У классов **всегда** есть конструктор. Даже если вы не пропишите свою реализацию конструктора, то Java автоматически создаст пустой конструктор по-умолчанию. Для класса Cat, он будет выглядеть так:

```
public Cat() {
}
```

Параметризированные конструкторы

При использовании конструктора из предыдущего примера, все созданные коты будут одинаковыми, пока мы вручную не поменяем значения их полей. Чтобы можно было указывать начальные значения полей наших объектов необходимо создать параметризированный конструктор.

Важно! В приведенном ниже примере, в аргументах конструктора используется нижнее подчеркивание _, это сделано для упрощения понимания логики заполнения полей объекта. И в будущем будет заменено на более корректное использование ключевого слова this.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String _name, String _color, int _age) {
        name = _name;
        color = _color;
        age = _age;
    }
}
```

При такой форме конструктора, когда мы будем кота, необходимо будет обязательно указать его имя, цвет и возраст. Набор полей, которые будут заполнены через конструктор, вы определяете сами, то есть вы не обязаны все поля, которые есть в классе записывать в аргументы конструктора.

```
public static void main(String[] args) {
    Cat cat1 = new Cat("Барсик", "Коричневый", 4);
    Cat cat2 = new Cat("Мурзик", "Белый", 5);
}
```

Наборы значение (Барсик, Коричневый, 4) и (Мурзик, Белый, 5) будут переданы в качестве аргументов конструктора (_name, _color, _age), а конструктор уже перезапишет полученные значения в поля объект (name, color, age). То есть начальные значения полей каждого из объектов будет определяться тем, что мы передадим ему в конструкторе. Как видите, теперь нам нет необходимости обращаться напрямую к полям объектов, и мы в одну строку можем проинициализировать наш новый объект.

Ключевое слово this

Иногда требуется, чтобы метод ссылался на вызвавший его объект. Для этой цели в Java определено ключевое слово this. Им можно пользоваться в теле любого метода для ссылки на текущий объект, т.е. объект у которого был вызван этот метод.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }
}
```

Эта версия конструктора действует точно так же, как и предыдущая. Ключевое слово this применяется в данном случае для того, чтобы отличить аргумент конструктора от поля объекта.

Перегрузка конструкторов

Наряду с перегрузкой обычных методов возможна перегрузка и конструкторов. Мы можем как не объявлять ни одного конструктора, так и объявить их несколько. Также как и при перегрузке методов, имеет значение набор аргументов, не может быть нескольких конструкторов с одним и тем же набором аргументов.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    public Cat(String name) {
        this.name = name;
```

```
        this.color = "Неизвестно";
        this.age = 1;
    }
}
```

Важно! Как только вы создали в классе свою реализацию конструктора, конструктор по-умолчанию автоматически создаваться не будет. И если вам понадобится такая форма конструктора (в которой нет аргументов, и которая ничего не делает), необходимо будет конструктор по-умолчанию вручную.

```
  public Cat() {
  }
```

В этом случае допустимы будут следующие варианты создания объектов:

```
public static void main(String[] args) {
    // Cat cat1 = new Cat(); этот конструктор больше не работает
    Cat cat2 = new Cat("Барсик");
    Cat cat3 = new Cat("Мурзик", "Белый", 5);
}
```

Соответствующий перегружаемый конструктор вызывается в зависимости от аргументов, указываемых при выполнении оператора new.

Важно! Не лишним будет напомнить что у классов **всегда** есть конструктор, даже если вы не пропишите свою реализацию конструктора.

Инкапсуляция

Инкапсуляция связывает данные с манипулирующим ими кодом и позволяет управлять доступом к членам класса из отдельных частей программы, предоставляя доступ только с помощью определенного ряда методов, что позволяет предотвратить злоупотребление этими данными.

То есть класс должен представлять собой «черный ящик», которым можно пользоваться, но его внутренний механизм защищен от повреждений.

Способ доступа к члену класса определяется модификатором доступа, присутствующим в его объявлении. Некоторые аспекты управления доступом связаны, главным образом, с наследованием и пакетами, и будут рассмотрены позднее. В Java определяются следующие модификаторы доступа: public, private и protected, а также уровень доступа, предоставляемый по умолчанию.

Любой **public** член класса доступен из любой части программы. Компонент, объявленный как **private**, доступен только внутри класса, в котором объявлен. Если в объявлении члена класса отсутствует явно указанный модификатор доступа (**default**), то он доступен для подклассов и других классов из данного пакета. Если же требуется, чтобы элемент был доступен за пределами его текущего пакета, но только классам, непосредственно производным от данного класса, то такой элемент должен быть объявлен как **protected**.

Модификатор доступа предшествует остальной спецификации типа члена.

```
public int num;
protected char symb;
boolean active;

private void calculate(float x1, float x2) {
    // ...
}
```

Как правило, доступ к данным объекта должен осуществляться только через методы, определённые в классе этого объекта. Поле экземпляра вполне может быть открытым, но на то должны иметься веские основания. Для доступа к данным обычно используются методы: геттеры и сеттеры.

Геттер позволяет узнать содержимое поля, его тип совпадает с типом поля для которого он создан, а имя, как правило, начинается со слова get, к которому добавляется имя поля.

Сеттер используется для изменения значения поля, имеет тип void и именуется по аналогии с геттером, только get заменяется на set. Сеттер позволяет добавлять ограничения на изменение полей — в примере ниже с помощью сеттера не получится указать коту отрицательный или нулевой возраст.

```
public class Cat {
    private String name;
    private int age;

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        } else {
            System.out.println("Введен некорректный возраст");
        }
    }
```

```
    public int getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Заметка. Если для поля создан только геттер, то вне класса это поле будет доступно только для чтения. Если ни создан ни геттер, ни сеттер, то работа с полем снаружи класса может осуществляться только косвенно, через другие методы этого класса. (Пусть в классе Cat есть поле вес, оно private и для него нет геттеров и сеттеров. Тогда мы не можем через сеттер изменить вес кота напрямую, но если мы его покормим, то кот может сам набрать вес. Мы не можем запросить вес через геттер и получить конкретное значение, но у кота может быть метод info(), который выведет в консоль нам величину, записанную в поле weight.

Особенности всех уровней доступа в языке Java сведены в таблицу:

	private	Модификатор отсутствует	protected	public
Один и тот же класс	+	+	+	+
Подкласс, производный от класса из того же самого пакета	-	+	+	+
Класс из того же самого пакета, не являющийся подклассом	-	+	+	+
Подкласс, производный от класса другого пакета	-	-	+	+
Класс из другого пакета, не являющийся подклассом, производный от класса данного пакета	-	-	-	+

Работа с символьными строками

В Java символьная строка представляет собой последовательность символов.В отличие от других языков программирования, где символьные строки представлены последовательностью символов, в Java они являются объектами класса String.
В результате создания объекта типа String получается неизменяемая символьная строка, т.е. невозможно изменить символы имеющейся строки. При любом изменении строки создается новый объект типа String, содержащий все изменения. Если требуются изменяемые строки, могут быть использованы классы StringBuilder и StringBuffer.

Создание символьных строк

Экземпляр класса String можно создать множеством способов.

```
public class Main {
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = new String("Home");
        String s3 = new String(new char[]{'A', 'B', 'C'});
        String s4 = new String(s3);
        String s5 = new String(new byte[]{65, 66, 67});
        String s6 = new String(new byte[]{0, 65, 0, 66}, StandardCharsets.UTF_16);
         System.out.printf("s1 = %s, s2 = %s, s3 = %s, s4 = %s, s5 = %s, s6 = %s",
s1, s2, s3, s4, s5, s6);
    }
}
Результат:
s1 = Java, s2 = Home, s3 = ABC, s4 = ABC, s5 = ABC, s6 = AB
```

В примере выше представлена лишь часть возможных вариантов. Самым простым способом является создание строки по аналогии с s1. Обычно строки требуется создавать с начальными значениями. Для этого предоставлены разнообразные конструкторы. При использовании варианта s2 в памяти создается новый экземпляр строки с начальным значением “Home”. Если в программе есть массив символьных строк, то из них можно собрать строку с помощью конструктора, которому в качестве аргумента передается ссылка на массив char[]. Если нужно создать объект типа String, содержащий ту же последовательность символов, что и другой объект типа String, можно использовать вариант s4. Варианты s5 и s6 позволяют «построить» строку из байтового массива без указания кодировки или с ней. Если кодировка не указана, будет использована ASCII.

Конкатенация строк

С помощью операции + можно соединять две символьные строки, порождая в итоге объект типа String. Операции сцепления символьных строк можно объединять в цепочку. Ниже приведено несколько примеров.

```
public class Main {
    public static void main(String[] args) {
        String a1 = "Hello ";
        String a2 = "World";
        String a3 = a1 + a2;
        System.out.println(a3);
        String b1 = "Число 10: ";
        int b2 = 10;
        String b3 = b1 + b2;
        System.out.println(b3);
        String c1 = "Home";
        String c2 = "[" + c1 + "] = " + 1;
        System.out.println(c2);
    }
}
Результат:
Hello World
Число 10: 10
[Home] = 1
```

Как видно из примера выше, строки можно сцеплять с другими типами данных, например, целыми числами(int). Так происходит потому, что значение типа int автоматически преобразуется в своё

строковое представление в объекте типа String. После этого символьные строки сцепляются, как и прежде.
Сцепляя разные типы данных с символьными строками, следует быть внимательным, иначе можно получить неожиданные результаты.

```
public class Main {
    public static void main(String[] args) {
        String str = "Десять: " + 5 + 5;
        System.out.println(str);
    }
}
Результат:
Десять: 55
```

В данном примере выводится следующий результат сцепления: «Десять: 55» вместо: «Десять: 10». Благодаря предшествованию операций сначала выполняется сцепление символьной строки «Десять:» со строковым представлением первого числа 5. Затем полученный результат сцепляется со строковым представлением второго числа 5. Чтобы выполнить сначала целочисленное сложение, следует заключить эту операцию в круглые скобки, как показано ниже.

```
public class Main {
    public static void main(String[] args) {
        String str = "Десять: " + (5 + 5);
        System.out.println(str);
    }
}
Результат:
Десять: 10
```

Методы для работы с символьными строками

Ниже представлено несколько часто используемых методов для работы со строками. Это не полный список.

int length()	Получение длины строки
char charAt(int pos)	Извлечение из строки символа, находящегося на позиции pos, индексация символов начинается с нуля
char[] toCharArray()	Преобразование строки в массив символов
boolean equals(Object obj)	Посимвольное сравнение строк
boolean equalsIgnoreCase(Object obj)	Сравнение строк без учета регистра
Strinq concat(String obj)	Объединение двух строк в одну. Этот метод создает новый строковый объект, содержащий вызываемую строку, в конце которой добавляется содержимое параметра строка. Метод выполняет то же действие, что и операция +.
String toLowerCase(), String toUpperCase()	Преобразование всех символов строки из верхнего регистра в нижний, из нижнего регистра в верхний

Классы StringBuilder и StringBuffer

В отличие от класса String класс StringBuilder представляет собой **изменяемые** последовательности символов. В классе StringBuilder определены следующие четыре конструктора: StringBuilder(), StringBuilder (int size), StringBuilder (String obj), StringBuilder (CharSequence obj).

Первый конструктор по умолчанию резервирует место для 16 символов. Второй конструктор принимает целочисленный аргумент, задающий размер буфера. Третий конструктор принимает аргумент типа String, задающий начальное содержимое объекта типа StringBuilder и резервирующий место для 16 символов. Выделение места для 16 дополнительных символов позволяет сэкономить время, затрачиваемое на перераспределение памяти при незначительных изменениях начальной строки. Четвертый конструктор создает объект из последовательности символов.

Методы для работы с StringBuilder и StringBuffer

int length()	Получение длины строки.
int capacity()	Получение объема выделенной памяти.
void ensureCapacity(int minimumCapacity)	Предварительное выделение места для определенного количества символов после создания объекта типа StringBuffer, чтобы установить емкость буфера. Это удобно, если заранее известно, что к объекту типа StringBuffer предполагается присоединить большое количество мелких символьных строк.
void setLength(int length)	Установка длины строки. Если указанная длина больше текущей, то в конце строки добавляются пустые символы, если меньше - символы, оказавшиеся за пределами вновь заданной длины строки, удаляются.
void append(...)	Присоединение любого типа данных в конце вызывающего объекта.

Класс StringBuffer отличается от класса StringBuilder только тем, что он является потокобезопасным. Более подробно этот вопрос будет изучен в будущем при рассмотрении темы «Многопоточность».

Примеры взаимодействия объектов

Рассмотрим несколько примеров взаимодействия классов и объектов между собой. Допустим, в программе есть класс Кот. У кота есть кличка и аппетит (сколько единиц еды он потребляет за приём пищи). Эти поля мы заполняем с помощью конструктора. Также есть метод eat(), который заставляет кота покушать, но пока что он пустой, так как неизвестно, откуда кот должен брать еду.

```
public class Cat {
    private String name;
    private int appetite;
    public Cat(String name, int appetite) {
        this.name = name;
        this.appetite = appetite;
    }
    public void eat() { }
}
```

Чтобы можно было хранить еду, создадим класс Тарелка с полем food (еда измеряется в целых числах и не важно, что это за единицы измерения). При создании тарелки мы можем указать начальное значение food. В процессе работы с помощью метода info() можно вывести в консоль информацию о тарелке.

```
public class Plate {
    private int food;
    public Plate(int food) {
        this.food = food;
    }
    public void info() {
```

```
        System.out.println("plate: " + food);
    }
}
```

Если в методе main() создать объекты этих классов, то можно узнать информацию о тарелке и вызвать пустой метод eat() у кота. Эти объекты пока никак не могут взаимодействовать между собой.

```
public class MainClass {
    public static void main(String[] args) {
        Cat cat = new Cat("Barsik", 5);
        Plate plate = new Plate(100);
        plate.info();
        cat.eat();
    }
}
```

Можно добавить класса геттеры и сеттеры и получить код вроде.

```
public class MainClass {
    public static void main(String[] args) {
        Cat cat = new Cat("Barsik", 5);
        Plate plate = new Plate(100);
        plate.info();
        cat.eat();
        plate.setFood(plate.getFood() - cat.getAppetite());
    }
}
```

В этом случае получится, что из тарелки мы вычитаем количество еды, потребляемое котом, но при этом метод eat() все также бесполезен. Желательно сделать так, чтобы мы могли указывать коту, из какой тарелки он должен покушать, и при этом сам кот должен уменьшить количество еды в тарелке. Чтобы удобно было уменьшать количество еды в тарелке без использования геттеров и сеттеров, добавим в класс Тарелка метод decreaseFood(int n), который уменьшает food на указанную величину n. Чтобы кот мог взаимодействовать с тарелкой, в метод eat передадим в качестве параметра ссылку на объект класса Тарелка. Ниже приведён код, учитывающий описанные выше моменты. (*Код целиком перенесен на следующую страницу, чтобы его удобно было просматривать.*).

```
public class Plate {
    private int food;
    public Plate(int food) {
        this.food = food;
    }
    public void decreaseFood(int n) {
        food -= n;
    }
    public void info() {
        System.out.println("plate: " + food);
    }
}
public class Cat {
    private String name;
    private int appetite;
    public Cat(String name, int appetite) {
        this.name = name;
```

```
        this.appetite = appetite;
    }
    public void eat(Plate p) {
        p.decreaseFood(appetite);
    }
}
public class MainClass {
    public static void main(String[] args) {
        Cat cat = new Cat("Barsik", 5);
        Plate plate = new Plate(100);
        plate.info();
        cat.eat(plate);
        plate.info();
    }
}
Результат:
plate: 100
plate: 95
```

Как видите, теперь мы можем взять любой объект класса Кот и «попросить» его покушать из любого объекта класса Тарелка с помощью строки вида cat.eat(plate). То есть можно заставить каждого кота кушать из своей миски или взять много котов и заставить есть из общей. В текущей реализации есть несколько вопросов, которые необходимо решить, вы можете сделать это самостоятельно.

Дополнительные материалы

1. К. Сьерра, Б. Бейтс Изучаем Java // Пер. с англ. – М.: Эксмо, 2012. – 720 с.
2. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. – М.: Вильямс, 2014. - 864 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 1376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.