Объявление классов

Классы в Java объявляются с помощью ключевого слова class. Пример самого простого объявления класса:

Java **← это слово в документе будет означать начало кода на java**.

class Goblin {

}

Здесь мы объявляем новый класс с именем Goblin.

Внутри фигурных скобок объявляются все **поля, конструкторы и методы класса**.

Перед ключевым словом class может стоять модификатор public, который делает класс доступным из всех пакетов. Если модификатора public нет, как в нашем случае, то класс доступен только в том пакете, в котором он объявлен.

**Объявление полей**

Пример объявления полей:

Java

class Goblin {

private int money;

double health;

protected int diamonds = 10;

public String name;

}

В этом примере мы объявили четыре поля:

поле money с типом int ;

поле health с типом double ;

поле diamonds с типом int;

поле name с типом String.

Каждый экземпляр класса Goblin будет иметь своё значение полей money , health , diamonds и name.

В самом начале объявления поля указывается модификатор доступа к полю ( private , protected или public ), либо не указывается, и тогда используется доступ по умолчанию package-private. Затем указывается, если нужно, ключевое слово static (будет объяснено позже), а также, если нужно, ключевое слово final (будет объяснено позже). Затем тип поля и имя. Затем поле может сразу инициализироваться начальным значением, например как поле diamonds инициализируется числом 10 в нашем примере.

Модификатор доступа, static и final могут располагаться в любом порядке, но согласно соглашению о кодировании принят именно такой порядок, который описан в статье.

Поле money объявлено с модификатором доступа private, и оно будет доступно только внутри этого класса.

Поле health объявлено без модификаторов доступа, и для него будет использоваться уровень доступа package-private (поле будет доступно только внутри своего пакета).

Поле diamonds объявлено с модификатором доступа protected, и оно будет доступно в этом пакете, этом классе и классах наследниках от этого класса (как объявлять наследников будет объяснено позже).

Поле name объявлено с модификатором доступа public, и оно будет доступно во всех классах всех пакетов.

Уровни доступа

Модификатор Класс Пакет Дочерний класс Все классы

public Есть Есть Есть Есть

protected Есть Есть Есть Нет

без модификатора Есть Есть Нет Нет

private Есть Нет Нет Нет

Имя поля следует давать в соответствии с соглашениями.

Обращаться к полю внутри класса, которому оно принадлежит можно просто по имени поля:

Java

class Goblin {

int ammo = 10;

//... другие поля класса

// метод стрельбы

public void fire() {

// уменьшаем количество пуль.

// Обратите внимание, что к полю класса

// обращаемся просто по имени

ammo --;

// ... остальной код

}

// ... другие методы

}

Из других классов обращение к полю класса происходит через точку, например:

Java

// Создаём объект GoblinObj

Goblin goblinObj = new Goblin();

// Обращение к полю ammo

goblinObj.ammo ++;

Имейте в виду, что прямое обращение к полю другого класса является плохим стилем, поскольку оно нарушает принципы ООП. Все обращения должны происходить к методам, которые уже сами меняют значения полей в соответствии с заложенной в них логике.

Также рекомендуется давать всем полям класса самый минимальный из возможных уровней доступа, что означает, что большинство полей класса должны иметь уровень доступа private. Остальные уровни доступа должны даваться отдельным переменным только в том случае, если это действительно нужно.

При объявлении полей можно в одной инструкции объявить несколько полей с одинаковым типом и одинаковыми модификаторами, но согласно соглашению о кодировании так делать не стоит:

Java

// Так делать НЕ рекомендуется

int x, y, z; // Объявляем три переменные: x, y, z

Объявление методов

Мы уже видели примеры объявления методов в статье «Введение в Java 8» и в примере доступа к полям в этой статье.

Пример объявления метода:

Java

public int fire(boolean withAim, double windDirection,

double windPower) {

// some operators;

return hitPoints;

}

Объявление метода состоит из следующих частей:

Модификатор доступа: private , без модификатора ( package-private ), protected , public .

Ключевое слово static , если нужно (будет описано позже).

Ключевое слово final , если нужно (будет описано позже).

Тип возвращаемого значения (в данном примере int ) или void , если метод не возвращает значение.

Имя метода ( в этом примере fire ).

Список параметров (в нашем примере три параметра: withAim , windDirection , windPower ).

Список исключений (будет описано в последующих статьях).

Тело метода в фигурных скобках.

Модификатор доступа, static и final могут располагаться в любом порядке, но согласно соглашению о кодировании принят именно такой порядок, который описан в статье.

Имя метода может содержать символы подчёркивания, знак доллара, цифры и многие другие символы Юникода (даже русские буквы), но не может начинаться с цифры, так же как и имя переменной. Однако согласно соглашению о кодировании на именование методов распространяются почти такие же правила, что и на именование переменных с тем отличием, что оно должно быть глаголом:

fire

buildObject

connect

compareTo

Если в объявлении метода не используется ключевое слово void, то метод должен явно вернуть значение с помощью оператора return. Пример:

Java

public int returnSix() {

return 6;

}

В операторе return можно указать выражение, тогда оно будет вызвано и из результатом вызова метода будет результат этого выражения:

Java

public double sum(double x1, double x2) {

return x1 + x2;

}

Тип возвращаемого значения должен совпадать с типом возвращаемого значения, указанного в объявлении, иначе будет ошибка компиляции. Если в объявлении указано ключевое слово void , то использование return не обязательно, но можно его указать, если хотим досрочно завершить выполнение метода:

Java

public void myVoidMethod1() {

...

if (someValue > 3) {

return;

}

...

}

В качестве типа возвращаемого значения может использоваться имя интерфейса или класса.

**Если в качестве возвращаемого значения указан интерфейс, то метод должен вернуть экземпляр любого класса, реализующего этот интерфейс, или null.**

Если в качестве возвращаемого значения указан класс, то метод должен вернуть экземпляр этого класса, экземпляр класса потомка или null.

**Сигнатура метода** — это имя метода вместе со списком параметров.

Java различает методы между собой по их сигнатуре. В одном классе не может быть двух и более методов с одинаковой сигнатурой, но поскольку список параметров входит в сигнатуру методов, то можно перегружать методы.

**Перегрузка методов** — создание нескольких методов с одинаковым именем, но разным списком параметров.

Перегрузка методов позволяет создавать, например, для каждого типа свою реализацию метода с тем же именем, но другим типом параметра, либо создать отдельный метод, принимающий другое количество параметров.

Пример перегрузки методов:

Java

class Goblin {

public void hit(Axe axe) {

// ... operators

}

public void hit(Flail flail) {

// ... operators

}

public void hit(Scimitar scimitar) {

// ... operators

}

public void hit(Torch torch) {

// ... operators

}

public void hit(Sword sword) {

// ... operators

}

public void hit(Sword sword, int comboCount) {

// ... operators

}

}

Все методы в примере выше — это разные методы. Компилятор различает их по списку параметров. Возвращаемое значение не входит в сигнатуру метода, а значит вы не можете создать несколько методов с одинаковым именем и параметрами, но разным возвращаемым значением.

Примеры вызова методов hit :

Java

Scimitar scimitar = new Scimitar();

goblin.hit(scimitar); // будет вызван public void hit(Scimitar scimitar)

Sword sword = new Sword();

goblin.hit(sword); // будет вызван public void hit(Sword sword)

Не стоит злоупотреблять перегрузкой методов. Используйте её только там, где это действительно нужно, иначе это может усложнить понимание вашего кода другими разработчиками.

**Конструкторы**

Конструкторы вызываются для создания объектов. Они похожи на классы, но они не имеют возвращаемого значения (даже void ), и они имеют то же самое имя, что и сам класс.

Пример конструктора для нашего класса Goblin :

Java

public Goblin(int initialMoney, double initialHealth) {

money = initialMoney;

health = initialHealth;

}

Теперь чтобы создать экземпляр класса Goblin нужно вызвать конструктор с ключевым словом new :

Java

Goblin myGoblin = new Goblin(8, 100.0);

Эта конструкция создаст экземпляр класса Goblin с помощью нашего конструктора и присвоит ссылку на этот класс переменной myGoblin.

С помощью перегрузки конструкторов можно создать несколько конструкторов. Главное чтобы они имели различное количество или тип параметров:

Java

class Goblin {

private int money;

double health;

protected int diamonds = 10;

public String name;

// Конструктор без параметров.

public Goblin() {

}

// Конструктор с двумя параметрами

public Goblin(int initialMoney, double initialHealth) {

money = initialMoney;

health = initialHealth;

}

// Конструктор с одним параметром.

public Goblin(String goblinName) {

name = goblinName;

}

// Приватный конструктор. Его можно будет вызвать

// только внутри этого метода.

private Goblin(int initialDiamonds) {

diamonds = initialDiamonds;

}

//... ещё конструкторы и методы

}

Теперь мы можем создавать экземпляры класса Goblin , используя любой из этих конструкторов, но приватный конструктор можно вызывать только внутри самого класса Goblin (например, в одном из его методов или полей инициализации).

Java

Goblin goblin0 = new Goblin();

Goblin goblin1 = new Goblin("Vasya");

Goblin goblin2 = new Goblin(3, 45.0);

Если мы не объявим ни одного конструктора в описании класса, то компилятор добавит один конструктор по умолчанию без параметров и с модификатором доступа public. Если же мы объявим хотя бы один конструктор, даже приватный, то конструктор без параметров добавляться не будет, но мы можем объявить его сами, если нужно.

Хитрость: Операция new возвращает ссылку на объект. Можно сразу же вызвать какой-нибудь метод этого объекта или обратиться к свойству, не присваивая эту ссылку переменной:

Java

new Goblin(myParam1).someMethod1(myParam2);

Если метод тоже возвращает ссылку на объект, то можно сразу вызвать метод этого объекта:

Java

new Goblin(myParam1).someMethod1(myParam2).someMethod2(); //... и т. д.

Ключевые слова static , final и abstract будут описаны позднее, но если вы перечитываете учебник второй раз, то запомните:

Конструктор НЕ может быть static , final или abstract.

**Передача параметров в метод или конструктор**

Метод может иметь любое число параметров, но каждый параметр должен иметь уникальное имя в пределах описания этого метода. Нельзя объявить метод, у которого два параметра имеют одинаковое имя, даже если они имеют разный тип. Имя параметра не может совпадать с именем локальной переменной, объявленной в методе. Однако имя параметра может совпадать с именем поля класса, в этом случае параметр затеняет (shadow) поле, поскольку при прямом обращении к этому имени мы будем обращаться к параметру, а не к полю класса.

Параметры, описанные в объявлении метода, называются формальными параметрами. Значения, передаваемые в формальные параметры при вызове метода или конструктора, называются аргументами или фактическими параметрами.

Вы можете использовать любой примитивный тип для параметров, объект или массив.

Примитивные типы передаются по значению — изменения внутри метода или конструктора не отражаются на значении переменной, которую передали:

Java

public void tryChangeParameterValue(int val1) {

// Мы можем менять значение val1,

// но val1 содержит копию переданного значения.

// Все изменения val1 видны только внутри этого метода

System.out.println("Inside method 2: " + val1); // 100

val1++;

System.out.println("Inside method 2: " + val1); // 101

}

public void test1() {

int parameter1 = 100;

System.out.println("parameter1 = " + parameter1); // 100

tryChangeParameterValue(parameter1);

System.out.println("parameter1 = " + parameter1); // 100

// изменения внутри метода происходили

// с копией переменой, а не с нашим

// parameter1.

}

Объекты и массивы передаются по ссылке — изменения внутри метода или конструктора меняют объект, который нам передали. Однако если внутри метода присвоить значению параметра null или ссылку на другой объект/массив, то такое изменение коснётся только параметра метода, а исходный объект или массив останутся неизменными.

Пример:

Файл «Main.java»:

Java

class Goblin {

public int money;

}

class Main {

public void tryChangeParameterValue(Goblin goblin, int[] arr1) {

// Эти изменения будут видны снаружи метода.

goblin.money++;

arr1[0] = 200;

// Эти изменения затрагивают только наш параметр ссылочного типа

// Объекты снаружи метода не будут изменены.

goblin = null;

arr1 = null;

goblin = new Goblin();

goblin.money = -400;

arr1 = new int[100];

arr1[2] = 3;

}

public void test1() {

Goblin goblin = new Goblin();

goblin.money = 45;

int[] arr1 = {3, 4, 7};

tryChangeParameterValue(goblin, arr1);

System.out.println(goblin.money); // 46

System.out.println(arr1[0]); // 200;

System.out.println(arr1[2]); // 7

}

public static void main(String[] args) {

Main main = new Main();

main.test1();

}

}

Выведет в консоль:

46

200

7

Иногда бывает полезно иметь метод с произвольным числом параметров. В таком случае можно передавать массив в качестве параметра. Однако в Java есть возможность создать метод, принимающий произвольное число параметров. Для этого после типа последнего параметра ставится троеточие, а внутри метода этот параметр обрабатывается как обычный массив. Пример:

Java

class Main {

public void sum(int par1, double par2, String par3, int... lastParameter) {

System.out.println("lastParameter[0] = " + lastParameter[0]); // 3

System.out.println("lastParameter[1] = " + lastParameter[1]); // 5

//...

}

public static void main(String[] args) {

Main main = new Main();

main.sum(100, 3.4, "par3", 3, 5, 6, 8, 9);

}

}

Троеточие является токеном само по себе, и технически корректно ставить пробел между типом и троеточием, но согласно принятому соглашению о кодировании в Java так делать не рекомендуется.

При вызове метода или конструктора можно передавать выражение, и тогда в метод или конструктор поступит вычисленное значение этого выражения. Параметры методов и конструкторов вычисляются слева направо: сначала вычисляется первый параметр, затем второй и т. д. Вызов метода происходит только после того, как все параметры будут вычислены.

Сборка мусора

В некоторых языках программирования нужно вручную освобождать память, выделенную под объекты. В Java такой необходимости нет. Виртуальная машина Java сама освобождает память от объектов, которые больше не используются. Объект считается больше не использующимся, если на него больше нет ссылок. Ссылки на объект обычно исчезают после того, как объект выходит из своей области видимости. Вы можете самостоятельно убрать ссылку на объект, присвоив переменной значение null.

Сборщик мусора автоматически освобождает память от объектов, которые больше не используются, когда сочтёт нужным.

Ключевое слово this

Если переменная, объявленная в методе, или параметр метода имеет то же самое имя, что и свойство класса, то эта переменная затеняет (shadow) свойство класса. Обращаясь по имени переменной в этом случае мы будем обращаться к переменной метода или параметру метода, а не к свойству класса. Чтобы обратиться к затенённому свойству класса нужно использовать ключевое слово this , которое означает этот класс.

Orc.javaJava

class Orc {

double health = 100.0;

public void someMethod1() {

double health = 200.0;

System.out.println(health); // 200.0

System.out.println(this.health); // 100.0

}

public void setHealth(double health) {

// присваиваем свойству класса

// переданное значение

this.health = health;

}

}

Ключевое слово this может также использоваться для вызова из конструктора класса другого конструктора этого класса. Вызов другого конструктора должен быть обязательно первым оператором/инструкцией в конструкторе:

Goblin.javaJava

class Goblin {

private double health;

private int ammo;

private int gold;

public Goblin() {

this(100.0, 0, 0);

// ... остальная инициализация.

}

public Goblin(double health) {

this(health, 0, 0);

// ... остальная инициализация.

}

public Goblin(double health, int ammo) {

this(health, ammo, 0);

// ... остальная инициализация.

}

public Goblin(double health, int ammo, int gold) {

this.health = health;

this.ammo = ammo;

this.gold = gold;

}

}

В этом примере класс Goblin имеет несколько конструкторов с разным числом параметров (перегруженные конструкторы). Конструкторы с меньшим числом параметров вызывают конструктор с самым большим количеством параметров. Компилятор Java различает эти конструкторы по параметрам (типу, количеству и порядку).

Ключевое слово static

При создании объектов каждый объект получает свой отдельный набор переменных экземпляров. Если же нужно сделать какую-то переменную общей для всех экземпляров, то используется ключевое слово static .

Пример:

Goblin.javaJava

class Goblin {

static int idCounter = 0;

int id;

Goblin() {

idCounter++;

id = idCounter;

}

}

В таком классе Goblin переменная idCounter одна, общая для всех экземпляров. Для всех экземпляров этого класса значение этой переменной будет всегда одно и то же, благодаря чему каждый экземпляр класса будет получать в поле id уникальное значение, большее значения поля id предыдущего экземпляра. Переменная idCounter называется статическим свойством/полем или переменной класса и относится к классу, а не к его экземплярам.

Обратиться к статическому свойству можно либо через имя класса, либо через имя экземпляра, однако рекомендуется всегда обращаться к статическим свойствам через имя класса, чтобы подчеркнуть, что оно относится именно к классу:

Java

System.out.println("idCounter=" + Goblin.idCounter); // предпочтительно

Goblin goblin = new Goblin();

System.out.println("idCounter=" + Goblin.idCounter); // предпочтительно

System.out.println("idCounter=" + goblin.idCounter);

Статическое свойство может тоже иметь модификаторы private , protected или public.

Модификатор static можно применить к методу, тогда он будет статическим и его можно будет вызывать через имя класса:

Java

static int getIdCounter() {

return idCounter;

}

Пример вызова:

Java

int x = Goblin.getIdCounter();

Статические методы можно вызывать и через имя экземпляра, но рекомендуется всегда вызывать их через имя класса, так как они относятся именно классу.

К статическим методам и свойствам можно обратиться даже тогда, когда ещё нет ни одного экземпляра класса.

Запомните:

Методы экземпляров могут обращаться к переменным экземпляров (нестатическим свойствам/полям) и методам экземпляров напрямую.

Методы экземпляров могут обращаться к переменным класса (статическим полям) и методам класса (статическим методам) напрямую.

Методы классов могут обращаться к методам класса (статическим методам) и переменным класса (статическим свойствам/полям) напрямую.

Методы классов не могут напрямую обращаться к переменным экземпляров (нестатическим свойствам/полям) и методам экземпляров, и они не могут использовать ключевое слово this , так как для них нет экземпляра класса. Они должны использовать ссылку на какой-нибудь экземпляр.

Ключевое слово final

Это больше относится к наследованию, которое будет рассмотрено в более поздних статьях. Здесь я опишу лишь в общих чертах.

Ключевое слово final может быть применено к локальной переменной, переменной экземпляра или параметру. Оно означает, что значение переменной не будет меняться после инициализации. Если попытаться изменить значение такой переменной, то будет ошибка компиляции.

Goblin.javaJava

class Goblin {

final String name;

public Goblin (String name) {

this.name = name; // Инициализируем переменную final.

}

public void someMethod1(final String secondName) {

final String thirdName = "Third";

// Переменные с final менять после инициализации нельзя!

this.name = secondName; //Нельзя! Ошибка компиляции.

secondName = "3"; // Нельзя! Ошибка компиляции.

thirdName = "4"; // Нельзя! Ошибка компиляции.

}

}

Переменные final не инициализируются значением по умолчанию. Им обязательно должно быть присвоено какое-нибудь значение, иначе возникнет ошибка компиляции.

Ключевое слово final может применяться к методу, тогда этот метод нельзя переопределять в классах-потомках для методов экземпляров и нельзя скрывать (hide) в классах потомках для случая статических методов (наследование и переопределение метода будет описано в статье про наследование):

Goblin.javaJava

class Goblin {

// Этот метод нельзя переопределять в потомках.

public final void myFinalMethod1() {

}

// Этот метод нельзя скрывать в потомках.

public static final void myFinalMethod2() {

}

}

Можно применить final ко всему классу, что означает, что у класса не может быть потомков, то есть будет нельзя наследоваться от этого класса.

Goblin.javaJava

final class Goblin {

}

Модификатор static , применённый совместно с final, к свойствам класса используется для объявления констант. Такое свойство не может быть изменено после инициализации, и оно обязательно должно быть проинициализировано.

Компилятор подставляет реальное значение констант во все места программы, где они используются. Если значение константы пришлось в последствии поменять, то нужно перекомпилировать все классы, которые её используют, так как иначе там останется старое значение.

Пример объявления константы:

Java

static final double PI = 3.141592653589793;

Согласно соглашению об именовании имена констант записываются прописными буквами, а между словами ставится символ подчёркивания.

Инициализация полей

Полям можно присвоить начальное значение сразу при объявлении:

Goblin.javaJava

class Goblin {

int x = 300;

int y = x \* 3; // Будет вычислено значение и просвоено 900.

public static void main(String[] args) {

Goblin goblin = new Goblin();

System.out.println(goblin.x);

System.out.println(goblin.y);

}

}

Инициализация происходит сверху вниз в порядке объявления в исходном коде. Сначала x присвоится 300, а затем y присвоится 300 \* 3.

Если выражение инициализации не помещается в одну строку, или требуется обработка ошибок, использование циклов и прочее, то можно использовать блоки инициализации:

Goblin.javaJava

class Goblin {

static int idCounter;

int money;

static {

// Инициализация статических полей

idCounter = 3;

}

{

// Инициализации переменных экземпляров.

money = 300;

}

}

Блоки статической инициализации выполняются один раз при инициализации класса. Может быть несколько блоков инициализации, и в таком случае они будут выполняться в порядке появления в исходном файле сверху вниз.

Блоки инициализации экземпляров выполняются при создании экземпляров объекта. Может быть несколько блоков инициализации экземпляров, в таком случае они выполняются в порядке появления в исходном файле.

Не стоит слишком сильно перемешивать блоки инициализации, конструкторы и инициализацию при объявлении, иначе код может получиться запутанным и сложным для понимания. Однако при наличии всех этих видов инициализации в одном классе инициализация экземпляра происходит так:

Вычисляются аргументы конструктора. Если конструктор начинается с вызова другого конструктора этого класса, то вычисляются аргументы его и т. д.

Если конструктор не начинается с вызова другого конструктора, то он начинается с явного или неявного вызова конструктора базового класса (будет описано в статье про наследование). Выполняется этот конструктор базового класса.

Выполняются все выражения инициализации экземпляров и блоки инициализации экземпляров в том порядке, в котором они объявлены в исходном файле, словно они идут одним блоком.

Выполняется остаток тела конструктора.

Пример:

Goblin.javaJava

class Goblin {

int x1;

{

x1 = 100;

}

int x2 = x1 + 1;

Goblin() {

x1 = 200;

}

int x3 = x1 + 2;

{

x1 = -100;

}

public static void main(String[] args) {

Goblin g = new Goblin();

System.out.println("x1=" + g.x1);

System.out.println("x2=" + g.x2);

System.out.println("x3=" + g.x3);

}

}

Выведет в консоль:

x1=200

x2=101

x3=102