# Java-язык программирования vs Java-платформа

Java— строго типизированный объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Разработка ведётся сообществом, организованным через Java Community Process, язык и основные реализующие его технологии распространяются по лицензии GPL. Права на торговую марку принадлежат корпорации Oracle.

Приложения Java обычно транслируются в специальный байт-код, поэтому они могут работать на любой компьютерной архитектуре с помощью виртуальной Java-машины. Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

Платформа [Java](https://ru.bmstu.wiki/Java" \o "Java) - это набор программ, которые облегчают разработку и запуск программ, написанных на языке программирования [Java](https://ru.bmstu.wiki/Java" \o "Java) . Платформа [Java](https://ru.bmstu.wiki/Java" \o "Java) будет включать механизм выполнения (называемый виртуальной машиной), компилятор и набор библиотек ; могут быть также дополнительные серверы и альтернативные библиотеки, которые зависят от требований. [Java](https://ru.bmstu.wiki/Java" \o "Java) не относится ни к одному процессору или операционной системе, поскольку платформы [Java](https://ru.bmstu.wiki/Java" \o "Java) были реализованы для широкого спектра аппаратных и операционных систем, чтобы позволить программам [Java](https://ru.bmstu.wiki/Java" \o "Java) работать одинаково на всех из них.

# Связь java-файлов и классов

Javasoft JDKs предполагает, что исходный код класса с модификатором public хранится в файле с точно таким же именем, как и имя класса, и расширением .java.

# JVM JDK and JRE

**Java Development Kit** (сокращенно JDK) —[комплект разработчика приложений](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BB%D0%B5%D0%BA%D1%82_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA%D0%B0_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) на языке [Java](https://ru.wikipedia.org/wiki/Java" \o "Java), включающий в себя [компилятор](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) Java ([javac](https://ru.wikipedia.org/wiki/Javac" \o "Javac)), стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему Java ([JRE](https://ru.wikipedia.org/wiki/JRE)). В состав JDK не входит [интегрированная среда разработки](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты [командной строки](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%BE%D0%B9_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8).

Все современные интегрированные среды разработки приложений на Java, такие, как [JDeveloper](https://ru.wikipedia.org/wiki/JDeveloper" \o "JDeveloper), [NetBeans IDE](https://ru.wikipedia.org/wiki/NetBeans_IDE" \o "NetBeans IDE), [Sun Java Studio Creator](https://ru.wikipedia.org/w/index.php?title=Sun_Java_Studio_Creator&action=edit&redlink=1" \o "Sun Java Studio Creator (страница отсутствует)), [IntelliJ IDEA](https://ru.wikipedia.org/wiki/IntelliJ_IDEA" \o "IntelliJ IDEA), [Borland JBuilder](https://ru.wikipedia.org/wiki/JBuilder" \o "JBuilder), [Eclipse](https://ru.wikipedia.org/wiki/Eclipse_(%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8)" \o "Eclipse (среда разработки)), опираются на сервисы, предоставляемые JDK. Большинство из них для компиляции Java-программ используют компилятор из комплекта JDK. Поэтому эти среды разработки либо включают в комплект поставки одну из версий JDK, либо требуют для своей работы предварительной инсталляции JDK на машине разработчика.

Доступны полные исходные тексты JDK, включая исходные тексты самого Java-компилятора [javac](https://ru.wikipedia.org/wiki/Javac" \o "Javac).

**Java Virtual Machine** (сокращенно Java VM, JVM) — [виртуальная машина](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0) Java — основная часть исполняющей системы [Java](https://ru.wikipedia.org/wiki/Java" \o "Java), так называемой Java Runtime Environment ([JRE](https://ru.wikipedia.org/wiki/JRE)). Виртуальная машина Java исполняет [байт-код Java](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4_Java), предварительно созданный из [исходного текста](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D1%82%D0%B5%D0%BA%D1%81%D1%82) Java-программы [компилятором](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) Java ([javac](https://ru.wikipedia.org/wiki/Javac" \o "Javac)). JVM может также использоваться для выполнения программ, написанных на других языках программирования. Например, исходный код на языке [Ada](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \o "Ада (язык программирования)) может быть откомпилирован в байт-код Java, который затем может выполниться с помощью JVM.

JVM является ключевым компонентом платформы Java. Так как виртуальные машины Java доступны для многих аппаратных и программных платформ, Java может рассматриваться и как связующее программное обеспечение, и как самостоятельная платформа. Использование одного [байт-кода](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) для многих платформ позволяет описать Java как «скомпилировано однажды, запускается везде» (compile once, run anywhere).

Виртуальные машины Java обычно содержат [интерпретатор](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80) байт-кода, однако, для повышения производительности во многих машинах также применяется [JIT](https://ru.wikipedia.org/wiki/JIT)-компиляция часто исполняемых фрагментов байт-кода в [машинный код](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4).

**Java Runtime Environment** (сокр. JRE; русск. среда выполнения для Java) — минимальная реализация [виртуальной машины](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0), необходимая для исполнения [Java](https://ru.wikipedia.org/wiki/Java" \o "Java)-приложений, без компилятора и других средств разработки. Состоит из виртуальной машины — [Java Virtual Machine](https://ru.wikipedia.org/wiki/Java_Virtual_Machine" \o "Java Virtual Machine) — и [библиотеки Java-классов](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0).

JRE распространяется свободно и для большинства платформ может быть загружена с сайта Oracle[[1]](https://ru.wikipedia.org/wiki/Java_Runtime_Environment" \l "cite_note-1).

Средства разработки вместе с JRE входят в [JDK](https://ru.wikipedia.org/wiki/JDK).

# Пакеты

Как правило, в Java классы объединяются в пакеты. Пакеты позволяют организовать классы логически в наборы. По умолчанию java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет гарантировать однозначность имен.

Как правило, названия пакетов соответствуют физической структуре проекта, то есть организации каталогов, в которых находятся файлы с исходным кодом. А путь к файлам внутри проекта соответствует названию пакета этих файлов. Например, если классы принадлежат пакету mypack, то эти классы помещаются в проекте в папку mypack.

Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени.

По умолчанию импортируется текущий пакет и java.lang

Правила наименования:

Имена пакета написаны во всем нижнем регистре, чтобы избежать конфликта с именами классов или интерфейсов.

Компании используют свое инвертированное имя Интернет-домена, чтобы начать их имена пакета — например, com.example.mypackage для названного пакета mypackage создаваемый программистом в example.com.

Коллизии имени, которые происходят в пределах единственной компании, должны быть обработаны условно в пределах той компании, возможно включением области или названия проекта после названия компании (например, com.example.region.mypackage).

Пакеты на языке самого Java начинаются java. или javax.

В некоторых случаях имя интернет-домена, возможно, не допустимое имя пакета. Это может произойти, если доменное имя содержит дефис или другой специальный символ, если имя пакета начинается с цифры или другого символа, который недопустим, чтобы использовать в качестве начала имени Java, или если имя пакета содержит зарезервированное ключевое слово Java, такое как "интервал". В этом случае предложенное соглашение состоит в том, чтобы добавить подчеркивание

# процедура передачи приметивных типов как параметр

Переменная в Java — это контейнер, со значением в нем. Так что значит «передать» переменную? И какова разница между примитивными и ссылочными типами данных.

Мы доберемся до этого позже. Давайте сначала начнем с простого присваивания. Что делает данный код:

int х = 3;

int у = х;

В строке 1 создается переменная x типа int и ей присваивается значение 3. В строке 2, создается переменная y типа int и ей присвается значение переменной x. В дальнейшем переменная x никак не влияет на y. Java копирует значение х (3) и помещает эту копию в у. Это передача параметра по значению. Вы не записываете одну переменную в другую. Значение копируется и присваивается новой переменной. Выражение у = х; НЕ означает "записать x в y". Оно означает "скопировать значение внутри х и записать эту копию в у". Если позже я изменю y:

у = 34;

Повлияет ли это на x? Конечно нет. x по прежнему имеет значение 3. Если позже я изменю х:

х = 90;

Как это отразится на y? Никак. Они никак не связаны после того, как было сделано присвоение (КОПИРОВАНИЕ значения). А что насчет ссылочных типов? Как они работают? Не так уж сложно, на самом деле это правило то же самое. Ссылки делают тоже самое - вы получаете копию ссылки. Так что, если я говорю:

Cat A = new Cat ();

Cat B = A;

Ссылка А копируется в ссылку B. К объекту это не относится — у вас по прежнему всего один объект. Но теперь у вас есть две различных ссылки, контролирующие один и тот же объект Cat. Теперь давайте рассмотрим передачу параметров в методы. Java передает параметры по значению. Всегда. Это означает — "скопировать значение и передать копию." Для примитивных типов это легко:

int х = 5;

doStuff (х); / / Передать копию х (значение 5) в метод doStuff

Метод doStuff выглядит следующим образом:

void DoStuff (int у) {

/ / Действия с 'y'

}

Копия значения x, тоесть 5, передается в метод doStuff (). Метод doStuff () имеет свою собственную переменную, которая называется y. Переменная y — новая, другая переменная. С копией того, что было в х на момент передачи его в метод. С этого момента, у и х не влияют друг на друга. При изменении у, вы не затрагиваете х.

void doStuff (int у) {

у = 27; / / Это не влияет на 'х'

}

И наоборот — при изменении х, вы не измените y. Единственное что сделал x в этом деле это скопировал свое значение и передал его в метод doStuff(). Как "передача по значению" работает со ссылками? Слишком многие люди говорят, "Java передает примитивные типы по значению, а объекты по ссылке". Это не так как говорят. Java передает все по значению. С примитивами, вы получаете копию содержимого. Со ссылками вы тоже получаете копию содержимого. Но что такое содержимое ссылки? Пульт дистанционного управления. Средства для управления / доступа к объекту. Когда вы передаете ссылку на объект в метод, вы передаете копию ссылки. Клон пульта дистанционного управления. Объект все еще сидит в куче где был создан, ожидая кого-то, чтобы использовали пульт. Объект не волнует сколько пультов "запрограммированы" чтобы контролировать его. Это волнует только сборщика мусора и вас, программиста. Поэтому, когда вы говорите:

Cat A = new Cat ();

doStuff (А);

void DoStuff (Cat B) {

/ / Использование B

}

Существует только один объект Cat. Но теперь два пульта управления (ссылки) могут получить доступ к одному и тому же объекту Cat. Так что теперь все, что B делает объекту Cat, повлияет на Cat, на который указывает A, но это не повлияет на содержимое A! Вы можете изменить Cat, используя новую ссылку B (скопированную непосредственно с А), но вы не можете изменить А. Какого черта это значит? Вы можете изменить объект, на который ссылается А, но вы не можете взять и изменить ссылку А — переадресовать её на другой объект или null. Так что если вы измените ссылку B (не сам объект Cat на который ссылается B, а само значение ссылки) вы не измените значение А. И наоборот. Так что:

Cat A = new Cat ();

doStuff (А);

void doStuff (Cat B) {

B = new Cat (); / / Не повлияет на ссылку A

}

Это просто значит, что B указывает на другой объект. A по-прежнему счастлива. Так что повторяйте за мной: Java передает все по значению. (Хорошо, еще раз... с чувством.) Java передает все по значению. Для примитивных типов — вы передаете копию текущего значения, для ссылок на объекты — вы передаете копию ссылки (дистанционного управления). Вы никогда не передаете объект. Все объекты хранятся в куче. Всегда.

# Приведение примитивных типов

Вот краткий список того, что мы о них знаем:

1. Они не являются объектами и представляют собой значение, хранящееся в памяти
2. Примитивные типы бывают нескольких видов:
   * Целые числа — byte, short, int, long
   * Числа с плавающей точкой (дробные) — float и double
   * Логический — boolean
   * Символьный (для обозначения букв и цифр) — char
3. Каждый из них имеет свой диапазон значений:

| Примитивный тип | Размер в памяти | Диапазон значений |
| --- | --- | --- |
| byte | 8 бит | от -128 до 127 |
| short | 16 бит | до -32768 до 32767 |
| char | 16 бит | от 0 до 65536 |
| int | 32 бита | от -2147483648 до 2147483647 |
| long | 64 бита | от -9223372036854775808 до 9223372036854775807 |
| float | 32 бита | от (2 в степени -149) до ((2-2 в степени -23)\*2 в степени 127) |
| double | 64 бита | от (-2 в степени 63) до ((2 в степени 63) - 1) |
| boolean | 8 (при использовании в массивах), 32 (при использовании не в массивах) | true или false |

Но, помимо значений, типы отличаются еще и размером в памяти. int занимает больше, чем byte. А long — больше, чем short. Объем занимаемой примитивами памяти можно сравнить с матрешками: Внутри матрешки есть свободное место. Чем больше матрешка — тем больше места. Внутрь большой матрешки long мы легко можем положить меньшую по размеру int. Она легко уместится, и ничего делать дополнительно не нужно. В Java при работе с примитивами это называется автоматическим преобразованием. По-другому его называют расширением. Вот простой пример расширения:

public class Main {

public static void main(String[] args) {

int bigNumber = 10000000;

byte littleNumber = 16;

bigNumber = littleNumber;

System.out.println(bigNumber);

}

}

Здесь мы присваиваем значение byte в переменную int. Присваивание прошло успешно и безо всяких проблем: значение, хранящееся в byte, занимает меньший объем в памяти, чем “влезает” в int. “Маленькая матрешка” (значение byte) легко влезает в “большую матрешку” (переменную int). Другое дело, когда ты пытаешься сделать наоборот — положить значение большого размера в переменную, которая на такие размеры не рассчитана. С настоящими матрешками такой номер в принципе не пройдет, а в Java — пройдет, но с нюансами. Давай попробуем положить значение int в переменную short:

public static void main(String[] args) {

int bigNumber = 10000000;

short littleNumber = 1000;

littleNumber = bigNumber;//ошибка!

System.out.println(bigNumber);

}

Ошибка! Компилятор понимает, что ты пытаешься сделать что-то нестандартное, и засунуть большую матрешку (int) внутрь маленькой (short). Ошибка компиляции в данном случае — предупреждение от компилятора: “*Эй, ты точно уверен, что хочешь это сделать?*” Если ты уверен, говоришь об этом компилятору: “*Все ок, я знаю, что делаю!*” Этот процесс называется явным преобразованием типов, или **сужением**. Чтобы сделать сужение, тебе необходимо явно указать тип, к которому ты хочешь привести свое значение. Иными словами, ответить компилятору на его вопрос: “*Ну и в какую из этих маленьких матрешек ты хочешь засунуть эту большую матрешку?*” В нашем случае это будет выглядеть так:

public static void main(String[] args) {

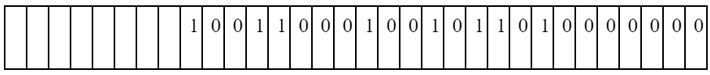
int bigNumber = 10000000;

short littleNumber = 1000;

littleNumber = (short) bigNumber;

System.out.println(littleNumber);

}

Мы явно указали, что хотим уместить значение int в переменную short и берем ответственность на себя. Компилятор, видя явное указание на более узкий тип, проводит преобразование. Каков же будет результат? **Вывод в консоль:** *-27008* Немного неожиданно. Почему именно такой? На самом деле все просто. У нас было изначальное значение — 10000000 Оно хранилось в переменной int, которая занимала 32 бита, и в двоичной форме оно выглядело так: Мы записываем это значение в переменную short, но она может хранить только 16 бит! Соответственно, только первые 16 бит нашего числа и будут туда перемещены, остальные — отбросятся. В итоге в переменную short попадет значение Расширение и сужение примитивных типов - 4, которое в десятичной форме как раз равно -27008 Именно поэтому компилятор “просил подтверждения” в форме явного приведения к конкретному типу. Во-первых, оно показывает, что ты берешь ответственность за результат на себя, а во-вторых, указывает компилятору сколько места выделить при приведении типов. Ведь если бы мы в последнем примере приводили int к типу byte, а не к short, в нашем распоряжении было бы только 8 бит, а не 16, и результат был бы уже другим. Для дробных типов (float и double) сужение происходит по-своему. Если попытаться привести такое число к целочисленному типу, у него будет отброшена дробная часть.

public static void main(String[] args) {

double d = 2.7;

long x = (int) d;

System.out.println(x);

}

**Вывод в консоль:** *2*

## Тип данных char

Ты уже знаешь, что тип char используется для отображения отдельных символов.

public static void main(String[] args) {

char c = '!';

char z = 'z';

char i = '8';

}

Но у него есть ряд особенностей, которые важно понимать. Давай еще раз посмотрим в таблицу с диапазонами значений:

| Примитивный тип | Размер в памяти | Диапазон значений | | |
| --- | --- | --- | --- | --- |
| byte | 8 бит | | от -128 до 127 |
| short | 16 бит | | от -32768 до 32767 |
| char | 16 бит | | от 0 до 65536 |
| int | 32 бита | | от -2147483648 до 2147483647 |
| long | 64 бита | | от -9223372036854775808 до 9223372036854775807 |
| float | 32 бита | | от (2 в степени -149) до ((2-2 в степени -23)\*2 в степени 127) |
| double | 64 бита | | от (-2 в степени 63) до ((2 в степени 63)-1) |
| boolean | 88 (при использовании в массивах), 32 (при использовании не в массивах) | | true или false |

Для типа char указан числовой диапазон — от 0 до 65536. Но что это значит? Ведь char — это не только цифры, но и буквы, знаки препинания… Дело в том, что значения char хранятся в Java в формате Юникода. Мы уже сталкивались с Юникодом в одной из прошлых лекций. Ты, наверное, помнишь, что **Unicode** — это стандарт кодирования символов, включающий в себя знаки почти всех письменных языков мира. Иными словами, это список специальных кодов, в котором найдется код почти для любого символа из любого языка. Общая таблица Юникодов очень большая, и, конечно, ее не нужно учить наизусть. Вот, например, ее кусочек: Главное — понимать принцип хранения значений char, и помнить, что **зная код конкретного символа** всегда можно получить его в программе. Давай попробуем это сделать с каким-нибудь случайным числом:

public static void main(String[] args) {

int x = 32816;

char c = (char) x ;

System.out.println(c);

}

**Вывод в консоль: 耰** Именно в таком формате в Java хранятся символы char. Каждому символу соответствует число — числовой код размером 16 бит, или два байта. Юникоду 32816 соответствует иероглиф 耰. Обрати внимание вот на какой момент. В этом примере мы использовали переменную int. Она занимает в памяти **32 бита**, в то время как char — **16**. Здесь мы выбрали int, потому что нужное нам число 32816 находится за пределами диапазона short. Хотя размер char, как и short, равен 16 битам, но в диапазоне char нет отрицательных чисел, поэтому “положительный” диапазон char в два раза больше (65536 вместо 32767 у short). Мы можем использовать int, пока наш код укладывается в диапазон до 65536. Но если создать число int >65536, оно будет занимать больше 16 битов. И при сужении типов:

char c = (char) x;

лишние биты будут отброшены, и результат будет весьма неожиданным.

## Особенности сложения char и целых чисел

Давай рассмотрим вот такой необычный пример:

public class Main {

public static void main(String[] args) {

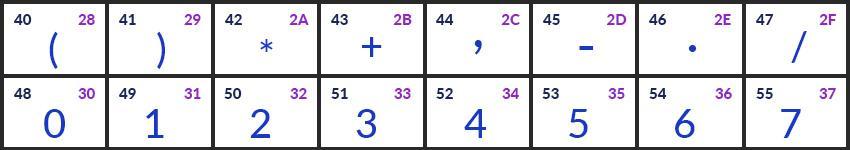
char c = '1';

int i = 1;

System.out.println(i+c);

}

}

**Вывод в консоль:** *50* O\_О Где логика? 1+1, откуда взялось 50?! Ты уже знаешь, что значения char хранятся в памяти как числа в диапазоне от 0 до 65536, обозначающие Юникод нашего символа. Так вот. Когда мы производим сложение char и какого-то целочисленного типа, char преобразуется к числу, которое соответствует ему в Юникоде. Когда в нашем коде мы складывали 1 и ‘1’ — символ ‘1’ преобразовался к своему коду, который равен 49 (можешь проверить в таблице выше). Поэтому результат и стал равен 50. Давай еще раз возьмем для примера нашего старого друга — **耰**, и попробуем сложить его с каким-нибудь числом.

public static void main(String[] args) {

char c = '耰';

int x = 200;

System.out.println(c + x);

}

**Вывод в консоль:** *33016* Мы уже выяснили, что **耰** соответствует коду 32816. А при сложении этого числа и 200 мы получаем как раз наш результат — 33016 :) Механизм работы, как видишь, достаточно простой.

# Литералы

**Литералы** — это явно заданные значения в коде программы — константы определенного типа, которые находятся в коде в момент запуска.

class Test {

public static void main(String[] args) {

System.out.println("Hello world!");

}

}

В этом классе “Hello world!” — литерал. Литералы бывают разных типов, которые определяются их назначением и способом написания.

## **Типы литералов и их применение**

Все литералы — это примитивные значения (строки, числа, символы, булевы значения). Нельзя создать литерал-объект. Единственный литерал, связанный с объектом — это null. В соответствии с примитивами, литералы также делятся на разделы:

1. Числовые:
   1. Целочисленные;
   2. С плавающей точкой;
2. Строковые;
3. Символьные;
4. Логические.

## **Числовые литералы**

### Целочисленные литералы

Данный вид литералов — самый простой. Числа записываются в их стандартном виде без указывающих символов и прочего. Любое целое число — по умолчанию целочисленный литерал. То есть можно явно задать значение переменной или количество итераций в цикле. В Java поддерживается 4 системы счисления:

* Двоичная
* Восьмеричная
* Десятичная
* Шестнадцатеричная

В JDK 7 появилась возможность записи двоичных значений. Делается это с помощью префикса 0b или 0B. Далее идет запись с использованием 0 и 1. Числа в восьмеричной системе записываются с помощью ведущего 0. Допустимые цифры — от 0 до 7. Запись 09 или 08 вызовет ошибку компиляции. С десятичной системой счисления нет никаких проблем: числа указываются в привычном нам виде. Единственное ограничение — число не может начинаться с 0, так как компилятор примет его за восьмеричное. Числа в шестнадцатеричной системе записываются с помощью префиксов 0x и 0X. Допустимые цифры — от 0 до 15, где числа 10-15 обозначаются символами A-F соответственно.

public static void main(String[] args) {

int a = 0b1101010110;

int b = 012314;

int c = 456;

int d = 0x141D12;

System.out.println("Число a в двоичной системе: " + a);

System.out.println("Число b в восьмеричной системе: " + b);

System.out.println("Число c в десятичной системе: " + c);

System.out.println("Число d в шестнадцатеричной системе: " + d);

}

Вывод: *Число a в двоичной системе: 854 Число b в восьмеричной системе: 5324 Число c в десятичной системе: 456 Число d в шестнадцатеричной системе: 1318162* Несмотря на то, что числа задаются в разных системах счисления, в программе они обрабатываются как десятичные числа. Выход за пределы значений приведет к ошибке компиляции:

int b = 012914; // Ошибка

int d = 0x141Z12; // Ошибка

При запуске на этапе компиляции получим следующий результат:

Error:(13, 25) java: integer number too large: 012914

Error:(14,30) java: ';' expected

А что с типизацией? Любой целочисленный литерал по умолчанию имеет тип int. Если его значение выходит за пределы присваиваемой переменной, происходит ошибка компиляции. При использовании типа long необходимо в конце ставить символ L, обозначающий этот тип:

long x = 0x1101010110; // Ошибка

long b = 1342352352351351353L; // Все в порядке

Попытка компиляции приводит к такой ошибке:

Error(11, 26) java: integer number too large: 1101010110

### **Литералы с плавающей точкой**

Числа с плавающей точкой, или дробные числа, могут записываться двумя способами. Первый — в качестве классической десятичной дроби: 3.14159 или 2.718281282459045. Второй — в научном виде, то есть обычная десятичная дробь плюс суффикс в виде символа е или Е и степени 10, на которую нужно умножить указанную дробь. Например, 4.05E-13, это значит 4.05 \* 10-13.

double a = 2.718281828459045;

double d = 4.05E-13;

System.out.println("Тип double в классическом виде: " + a);

System.out.println("Тип double в научном виде: " + d);

Вывод: *Тип double в классическом виде: 2.718281828459045 Тип double в научном виде: 4.05E-13* В отличие от целых чисел и систем счисления, научная форма записи сохраняется в переменной и обрабатывается также, как классическая форма. Что с типизацией? Любое число с плавающей точкой создает тип double. Если необходимо использовать тип float, следует добавить в конце символ f или F. В этом случае double будет сведен к типу float. Автоматически этого не происходит:

float a = 2.718281828459045; // Ошибка

float d = 4.05E-13F; // Все в порядке

При запуске на этапе компиляции видим такую ошибку:

Error:(11, 27) java: incompatible types: possible lossy conversion from double to float

## **Строковые литералы**

Строковые литералы — это набор символов, заключенных в двойные кавычки. Данный тип используется так же часто, как и числовые литералы. В строке также могут находится служебные символы, которые необходимо экранировать (так называемые escape-последовательности). Пример:

String simpleString = "Это обычная строка. Такая же стандартная и непримечательная, как и все мы.";

String uniqueString = "А это необычная \"строка\". Хотя бы потому, что часть \"слов\" в ней в \"кавычках\".";

System.out.println(simpleString);

System.out.println(uniqueString);

Вывод: *Это обычная строка. Такая же стандартная и непримечательная, как и все мы. А это необычная "строка". Хотя бы потому, что часть "слов" в ней в "кавычках".*

## **Символьные литералы**

Символьные литералы в Java представлены кодовой таблицей Unicode, то есть каждый символ — 16-битовое значение. Для обозначения символа в коде его обособляют одинарными кавычками. По опыту использования бывают два вида символов:

1. Те, что можно ввести с клавиатуры — обычные символы;
2. Символы, которые просто так с клавиатуры не ввести (символы различных языков, фигуры и так далее).

Обычные символы можно указать в явном виде: ‘,’ или ‘@’. Если символ служебный (например, перенос строки или табуляция), такой символ необходимо экранировать обратным слешем. Символы, которые нельзя просто так ввести с консоли, можно задать в их 16-битовом виде. Для этого необходимо указать код символа с префиксом \u, например ‘\u00F7'. Также символы можно указывать в восьмеричном стиле (трехзначный номер), добавив в начале просто обратный слеш, например ‘\122’. На мой взгляд, гораздо легче использовать \u. Пример использования:

System.out.println("Амперсанд - " + '&');

System.out.println("Символ деления - " + '\u00F7');

Вывод: *Амперсанд - & Символ деления - ÷*

## **Логические литералы**

Самый простой литерал — это логический. Имеется всего 2 значения: false и true, которые указываются явно без различных символов. Такие литералы могут быть присвоены переменным типа boolean или указаны в месте, где ожидается тип boolean (например, в блоке if, хотя такая практика считается, мягко говоря, моветоном).

boolean flag = false;

if(true) {

// Действия будут выполняться всегда.

}

## Джедайская техника с литералами

Благодаря символам в Java можно делать много интересных штук, в числе которых — управление эмоджи. Например, выведем улыбающееся лицо:

int smile = 0x1F600; // Здесь шестнадцатеричный код эмоджи

StringBuilder sb = new StringBuilder();

sb.append(Character.toChars(smile)); // Собираем в StringBuilder

System.out.println("Улыбающееся лицо: " + sb.toString()); // Выводим

Вывод: *Улыбающееся лицо:* 😀 Несмотря на то, что отображение эмоджи может быть жутковатым (в зависимости от реализации), это решение неплохо справляется с задачей. Однако в стандартной таблице кодировок сложно искать нужные эмоджи, на официальном сайте раздел Эмотикон скуден. Гораздо проще воспользоваться дополнительными библиотеками.

Примитивы и обертки

У примитивов в программировании, и в Java в частности, есть множество преимуществ: они занимают мало памяти, за счет чего повышается эффективность работы программы, и четко разделены по диапазонам значений. Однако в процессе изучения Java мы уже не раз, словно мантру, повторяли — “**в Java все является объектом**”. А ведь примитивы — прямое опровержение этих слов. Объектами они не являются. Получается, принцип “все является объектом” является ложным? На самом деле нет. В Java у каждого примитивного типа есть свой брат-близнец — **класс-обертка** (Wrapper). Что такое обертка? **Обертка — это специальный класс, который хранит внутри себя значение примитива.** Но поскольку это именно класс, он может создавать свои экземпляры. Они будут хранить внутри нужные значения примитивов, при этом будут являться настоящими объектами. Названия классов-оберток очень похожи на названия соответствующих примитивов, или полностью с ними совпадают. Поэтому запомнить их будет очень легко.

| Wrapper Classes for Primitive Data Types | |
| --- | --- |
| Primitive Data Types | Wrapper Classes |
| int | Integer |
| short | Short |
| long | Long |
| byte | Byte |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

Объекты классов оберток создаются так же, как и любые другие:

public static void main(String[] args) {

Integer i = new Integer(682);

Double d = new Double(2.33);

Boolean b = new Boolean(false);

}

Классы-обертки позволяют нивелировать недостатки, которые есть у примитивных типов. Самый очевидный из них — **примитивы не имеют методов**. Например, у них нет метода toString(), поэтому ты не сможешь, например, преобразовать число int в строку. А вот с классом-оберткой Integer — запросто.

public static void main(String[] args) {

Integer i = new Integer(432);

String s = i.toString();

}

Возникнут сложности и с обратным преобразованием. Допустим, у нас есть строка, про которую мы точно знаем, что она содержит число. Тем не менее, в случае с примитивным типом int мы никак не сможем это число из строки достать и превратить, собственно, в число. Но благодаря классам-оберткам такая возможность у нас появилась.

public static void main(String[] args) {

String s = "1166628";

Integer i = Integer.parseInt(s);

System.out.println(i);

}

**Вывод:** *1166628* Мы успешно получили число из строки и присвоили его в переменную-ссылку Integer i. Кстати, по поводу ссылок. Ты уже знаешь, что параметры передаются в методы по-разному: примитивы — по значению, а объекты — по ссылке. Ты можешь использовать это знание при создании своих методов: если твой метод работает, например, с дробными числами, но тебе нужна логика именно передачи по ссылке, ты можешь передать в метод параметры Double/Float вместо double/float. Кроме того, помимо методов в классах-обертках есть очень удобные для использования статические поля. Например, представь, что перед тобой сейчас стоит задача: *вывести в консоль максимально возможное число* int*, а после — минимально возможное.* Задачка вроде элементарная, а все равно — без гугла вряд ли справишься. А классы-обертки легко позволяют решать такие “бытовые задачи”:

public class Main {

public static void main(String[] args) {

System.out.println(Integer.MAX\_VALUE);

System.out.println(Integer.MIN\_VALUE);

}

}

Такие поля позволяют не отвлекаться от выполнения более серьезных задач. Не говоря уж о том, что в процессе печати числа *2147483647* (это как раз MAX\_VALUE) не мудрено и опечататься:) Кроме того, в одной из прошлых лекций мы уже обращали внимание на то, что **объекты классов-оберток являются неизменяемыми (Immutable)**.

public static void main(String[] args) {

Integer a = new Integer(0);

Integer b = new Integer(0);

b = a;

a = 1;

System.out.println(b);

}

**Вывод:** *0* Объект, на который изначально указывала ссылка а, не изменил свое состояние, иначе значение b тоже изменилось бы. Как и в случае со String, вместо изменения состояния объекта-обертки в памяти создается абсолютно новый объект. Почему же создатели Java, в конечном итоге, приняли решение оставить в языке примитивные типы? Раз уж все должно являться объектом, и у нас уже есть классы-обертки, которыми можно выразить все, что выражают примитивы, почему вообще не оставить в языке только их, а примитивы удалить? Ответ прост — производительность. Примитивные типы потому и называют примитивными, потому что они лишены многих “тяжеловесных” особенностей объектов. Да, у объекта есть много удобных методов, но ведь они не всегда тебе нужны. Иногда тебе нужно просто число 33, или 2,62, или значение true/false. В ситуациях, когда все преимущества объектов не имеют значения и не нужны для работы программы, примитивы справятся с задачей гораздо лучше.

Автоупаковка/автораспаковка

Одной из особенностей примитивов и их классов-оберток в Java является автоупаковка/автораспаковка (Autoboxing/Autounboxing) Давай разберемся с этим понятием. Как мы с тобой уже узнали ранее, Java — объектно-ориентированный язык. Это значит, что все программы, написанные на Java, состоят из объектов. Примитивы не являются объектами. Но при этом переменной класса-обертки можно присваивать значение примитивного типа. Этот процесс называется **автоупаковкой** (**autoboxing**). Точно так же переменной примитивного типа можно присваивать объект класса-обертки. **Этот процесс называется автораспаковкой (autounboxing)**. Например:

public class Main {

public static void main(String[] args) {

int x = 7;

Integer y = 111;

x = y; // автораспаковка

y = x \* 123; // автоупаковка

}

}

В строке 5 мы присваиваем примитиву x значение y, который является объектом класса-обертки Integer. Как видишь, никаких дополнительных действий для этого не нужно: **компилятор знает что** int **и** Integer**, по сути, одно и то же**. Это и есть автораспаковка. Так же происходит и автоупаковка в строке 6: объекту y легко присваивается значение примитивов (х\*123). Это пример автоупаковки. Именно поэтому добавляется слово "авто": **для присваивания ссылок-примитивов объектам их классов-оберток (и наоборот) не требуется ничего делать, все происходит автоматически**. Удобно, да? :) Еще одно очень большое удобство автоупаковки/автораспаковски проявляется в работе методов. Дело в том, что **параметры методов тоже подлежат автоупаковке и автораспаковке**. И, например, если какой-то из них методов принимает на вход два объекта Integer — мы легко можем передать туда обычные примитивы int!

public class Main {

public static void main(String[] args) {

printNumber(7);//обычный int, даже без переменной

}

public static void printNumber(Integer i) {

System.out.println("Вы ввели число " + i);

}

}

**Вывод:** *Вы ввели число 7* Точно так же работает и наоборот:

public class Main {

public static void main(String[] args) {

printNumber(new Integer(632));

}

public static void printNumber(int i) {

System.out.println("Вы ввели число " + i);

}

}

Важный момент, о котором нужно помнить: **автоупаковка и распаковка не работают для массивов**!

public class Main {

public static void main(String[] args) {

int[] i = {1,2,3,4,5};

printArray(i);//ошибка, не компилируется!

}

public static void printArray(Integer[] arr) {

System.out.println(Arrays.toString(arr));

}

}

Попытка передать массив примитивов в метод, который принимает на вход массив объектов, вызовет ошибку компиляции. Напоследок, еще раз кратко сравним примитивы и обертки **Примитивы:**

имеют преимущество в производительности

**Обертки:**

Позволяют не нарушать принцип “все является объектом”, благодаря чему числа, символы и булевы значения true/false не выпадают из этой концепции

Расширяют возможности работы с этими значениями, предоставляя удобные методы и поля

Необходимы, когда какой-то метод может работать исключительно с объектами