**Содержание:**

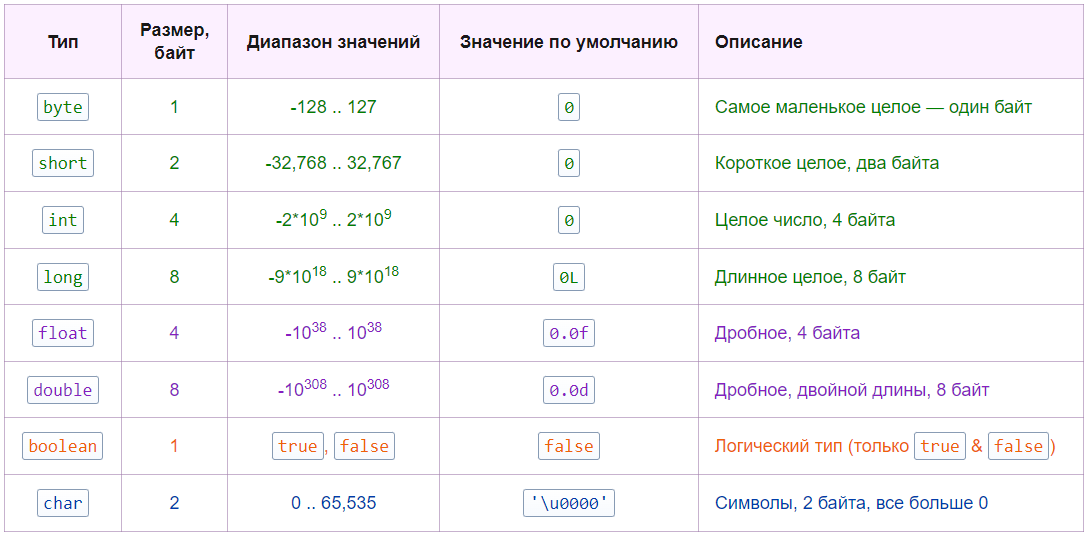
1. Типы данных
2. Литералы
3. String и Wrapper’s
4. Побитовые операции
5. Приоритеты операторов
6. Циклы for и for-each
7. Операторы перехода
8. Varargs и var
9. Switch
10. Enum
11. Comparable и Comparator
12. Iterable и Iterator
13. This и super

**Типы данных:**

Делятся на примитивные типы и ссылочные типы.

Примитивы – 4 группы и 8 типов данных, диапазон значений считается по формуле

, кроме char/boolean. char имеет только положительные значения, а у Boolean их всего 2.



Ссылочные типы данных – все остальные типы данных.

**Сравнение примитивов и ссылочных типов:**

|  |  |  |
| --- | --- | --- |
|  | **Примитивные переменные** | **Ссылочные переменные** |
| Что хранит | Хранит значение | Хранят адрес объекта в памяти |
| Создание | Создаются путем объявления и присваивания им значения. | Создаются через конструкторы классов (присваивание только создаёт вторую ссылку на существующий объект) |
| Значение по умолчанию – речь о полях. | Ноль, для boolean false. | По умолчанию их значение — null |
| Значения | Имеют строго заданный диапазон допустимых значений | Ограничены форматом ссылок, не имеют фиксированного диапазона значений. |
| Значения при передаче в методы | В аргументы методов попадают копии значения переменной (по значению) | В методы передаётся значение ссылки — операция выполняется над оригинальным объектом, на который ссылается переменная (по значению) |

\*\*Локальные переменные не имеют значений по умолчанию, необходима явная инициализация для использования.

\*\*Вещественные типы могут принимать значения Infinity, -Infinity, NaN (см. системы счисления).

double a = 1 / 0.0; //Деление на 0 дает Infinity;  
double infinity = a / a; //Деление Infinity ÷ Infinity дает NaN;  
double infin = 0.0 / 0.0; //Дает NaN;

\*\*С типом char можно производить математические операции.

char a = 'a';  
a++; // 'b'

a = 97 + 1;// 'b'

**\*\***Сравнение вещественных чисел – математические операции с вещественными типами всегда дают неточный результат из-за их неточного представления в двоичном виде. Например, код вида не выдаст ожидаемый результат.

double a = 0.0;  
for (int i = 1; i <=10 ; i++) {  
 a+=0.1;  
}//0.9999999999999999

Поэтому вещественные числа следует сравнивать с учетом этой погрешности.

double inaccuracy = 0.01;//Значение допустимой погрешности  
double num1 = 1.0;  
double num2 = 0.999;  
 Math.*abs*(num2 - num1) < inaccuracy //Вычисляем модуль разности двух чисел и считаем их равными, если это значение меньше значения погрешности.

Еще один способ сравнения — это использование класса BigDecimal, так как этот класс использует вместо математических операций специальные методы.

**Литералы, типы данных, приведение типов и операции с типами.**

Литерал – представление исходного кода с фиксированным значением (заданные в коде значения, не требующие вычислений).

Типы литералов: Числовые (Целочисленные, вещественные), символьные, строковые, логические.

По умолчанию целочисленные литералы интерпретируются как **int,** вещественные **double.**

Присвоение целочисленного литерала к **byte, short, char** – компилятор анализирует диапазон значений типа, к которому присваивается литерал, если литерал входит в этот диапазон, то происходит неявное приведение литерала к переменной. Литерал возможно расширить до **long** явно указав это суффиксом **L**. Математические операции с **byte**, **short**, **char** автоматически расширяют каждый тип до **int**, а затем проводятся операция.

byte b = 1; //литерал int 1 неявно приведен к byte  
long l = 3147483647L; //максимальное значение int 2147483647, литерал расширен до long  
  
byte by = 1;  
byte sum = by + b; //ошибка, требует или явного каста к byte, или переменную типа int

Вещественные литералы по умолчанию **double** – для сужения литерала необходимо явно использовать суффикс **f.** Присвоение вещественным типам целочисленного литерала приводит к его неявному приведению к вещественной переменной.

float f = 1.0; //ошибка, т.к. литерал по умолчанию double  
float ff = 1; // литерал int неявно приведен к float

**Формы записи литералов:**

double a = 1.23; //Обычная запись  
double b = 0.123E1; //Научная запись, где Е - десятичная степень числа.

Символьные литералы – для их представления используется кодировка Unicode или любая форма для представления целочисленного числа.

char letter\_O = '\u004F'; // Буква 'O' в шестнадцатеричной форме  
char letter\_a = '\141'; // Буква 'a' в восьмеричной форме  
char letter\_b = 98;//Буква 'b' в десятичной форме  
char letter\_c = 0b1100011;//Буква 'c' в двоичной форме

Строковые – можно использовать шестнадцатеричную, восьмеричную формы или конкатенацию char символов.

String a = "\u004F\141"+(char)98+(char)0b1100011;

**Класс String и Wrappers**

- строка представляет собой последовательность символов, для работы со строками определен класс String, который является final и immutable, и определяет методы для работы со строками. Внутренний массив хранится в виде byte [] начиная с Java 9.

**Immutable строки дают следующие преимущества:**

1. Потокобезопасность
2. Кэширование хэш-кода и возможность использования в качестве ключа для HashMap.
3. Возможность использовать String pool

String pool – область памяти в Heap, предназначенная для хранения уникальных строковых литералов. Если в ходе работы программы создается строка, которая уже присутствует в SP, то такая строка не создается, а ей присваивается ссылка на ранее созданную строку из SP.

String pool возможно использовать только благодаря immutable свойству String, т.к. если бы переменные хранили бы ссылки на одну и туже строку, то изменение одной из них приводило бы к нежелательному изменению другой.

Создание строк:

1. С помощью литерала – такой способ приводит к автоматическому помещению строки в String pool.
2. Создание с помощью конструктора приводит к созданию строки в Heap (не в SP), что предоставляет возможность создать разные объекты строк с идентичным содержанием. Созданную таким способом строку можно поместить в SP с помощью метода intern().

Конкатенация:

1. С помощью перегруженного оператора + (Java оптимизирует эту операцию до пункта 3 в некоторых случаях)
2. С помощью методов concat или join
3. С помощью использования изменяемых строк типа StringBuilder/StringBuffer – является предпочтительным вариантом при конкатенации строк в цикле, т.к. не происходит создание новой строки, а изменяется текущая.

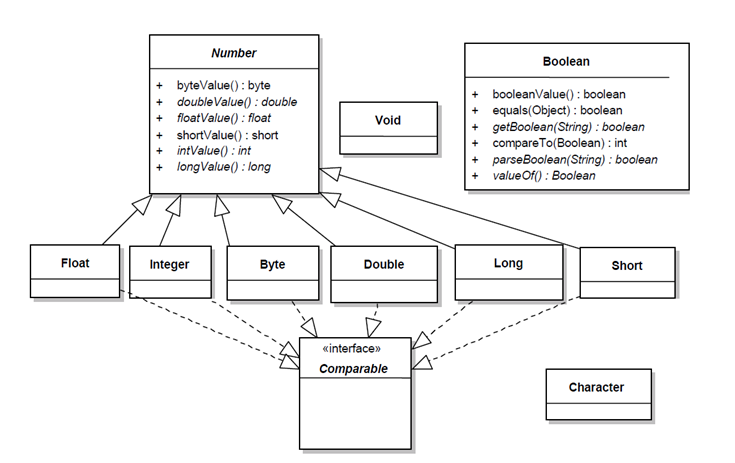
Способ 1 и 2 являются менее предпочтительными для частой конкатенации по причине создания большого кол-ва избыточных объектов (из-за иммутабельности String любая изменяющая операция генерирует новую строку).

**StringBuilder/StringBuffer –** классы для работы со строками, является mutable, то есть позволяет изменять состояние строки, без создания нового объекта. Внутренний массив динамически расширяется по мере необходимости и копирует содержимое в новый массив. То есть является более эффективным с точки зрения использования памяти по сравнению со String в ситуациях, когда нужно изменять строку.

Builder и Buffer отличаются только работой в многопоточной среде – Buffer содержит synchronized методы.

**Wrappers Классы обертки**

Классы для хранения примитивных типов (значений) в виде ссылочных. Являются immutable и final классами.



Реализуют интерфейсы serializable и comparable, кроме класса Void, числовые типы оберток наследуются от абстрактного класса Number.

Основная функция – оборачивание примитивных типов, с целью использовать их как объекты в тех контекстах, где это может быть необходимо (коллекции), возможность добавить к ним состояние, поведение и тип (в Java все объекты, кроме примитивов), возможность использовать их в ООП парадигме.

Сайд эффекты: Требуется дополнительные усилия для создания, больший вес за счет обертки, скорость доступа медленнее чем к примитивам (примитивы на стеке, если это локальная переменная, объекты в хипе), необходимость в Autoboxing и Unboxing.

Инициализация:

*С помощью метода valueOf()* – переданное значение присваивается внутреннему полю обертки.

Особенности:

Кэш – содержит внутренний класс Cache, который инициирует создание массива объектов этого класса со значениями в определенном диапазоне (нет кэширования для long, double и float). Например, для Integer при первом обращении к классу создается Integer [] со значениями от -128 до 127, то есть 256 объектов, и если создается объект в диапазоне этих значений, то вернется ссылка из кэша.

Autoboxing и Unboxing – автоматическое преобразование примитивного типа к его Wrapper и наоборот.

*С помощью конструктора* – при таком способе объект не связан с кэшем (всегда создается новый объект). С 9 Java Deprecated, с Java 14 запрещен.

**Побитовые операции:**

& (Побитовое И) - 1 + 1 = 1; 0 + 1 = 0; Выдает 1, только если биты в обоих операндах равны 1;

| (Побитовое ИЛИ) – 1 + 1 = 1; 1 + 0 = 1; Выдает 1, если один из операндов равен 1;

^ (XOR Исключающее ИЛИ) – 1 + 0 = 1; 1 + 1 = 0; Выдает 1, если один из операндов равен 1, если оба, то 0;

~ (Побитовое НЕ) – инвертирует биты числа;

<< - побитовый сдвиг влево, заполняет пустоты 0, остальное сдвигает влево;

Shor и byte при побитовых сдвигах автоматически расширяются до int, что не дает им терять свои байты вплоть до смещения на 31 пункт и они будут интерпретироваться как int – например:

Число byte 0b0100\_0000 (128) при смещении на 2 дает 0b1000\_0000 (-128), если рассматривать байты, но так как оно автоматом расширяется до int, то оно интерпретируется как 128, потому что старший бит у int 31й бит, если byte 0b0100\_0000 (128) << 2 , если привести к byte, то все биты потеряются и будет 0, но автоматически будет результат 0b1\_0000\_0000(256);

Если использовать оператор =<< - он автоматически сужает полученный int до byte и результат будет более ожидаем!! Сдвиг влево умножает значение на 2^n, где это кол-во битов для сдвига;

При показателе смещения более 31 используется остаток от деления на это число, поэтому <<32 означает <<0, а <<33 означает <<1;

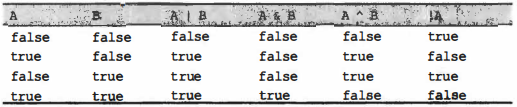
>> - побитовый сдвиг вправо, равносилен целочисленному делению на 2^n, данный оператор сохраняет старший знаковый бит (для сохранения отрицательных значений)

<< и >> используются как ускоренное целочисленное деление и умножение;

>>> - беззнаковый сдвиг вправо тоже самое что и >>, но заполняет крайний левый бит 0 => делает из отрицательных чисел положительные, в случае с byte 0b1111\_1111\_1111\_1111\_1111\_1111\_1000\_0000 (-128) – при >>> он заменит старший бит 0 и получится 0b0111\_1111\_1111\_1111\_1111\_1111\_1100\_0000 (2147483584);

Операции отношения типа ==, <, >= etc являются бинарными операторами результат которых будет Boolean;

Булевские логические операции - & | ^ выводят новое булевское значение из 2х других значений работают так же как и с битами; & отдает приоритет false, | true, ^ true, но при двух true дает false;



Оператор ! инвертирует Boolean значение;

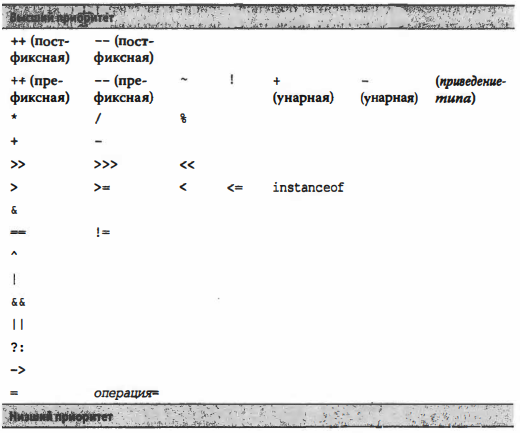
Короткозамкнутые логические операции: && и || требую слева и справа от себя операции возвращающие Boolean, например a>b || a=0 т.е. результатом вычисления будет

true/false|| true/false

&& требует, чтобы выражение слева и справа давало true, тогда общее выражение возвращает true, и будет выполняться тело выражения. т.е. если первое не дает true, то нет смысла вычислять второе!!

|| - требует выполнения хотя бы одного выражения как true, если левый дает true, то нет смысла вычислять правый, так как все-равно выражение в общем вернет true;

**Приоритет операторов:**



Если операции расположенные в одной строке имеют одинаковый приоритет, то они выполняются слева направо, для увеличения приоритета используются ( ).

**Отличие постфиксной записи от префиксной:**

**y = ++x –** префиксная запись –сначала инкремент, затем присвоение переменной**.**

**y = x++ -** постфиксная запись – сначала присвоение, затем инкремент.

**Особенности цикла for:**

Цикл for может быть записан в виде похожем на while. Отличие от обычного объявления for только областью видимости переменной I – в стандартном for она будет видна только в теле цикла.

int i = 0;  
for (; i < 10;) {  
 i++;  
}

Также возможно объявление сразу нескольких переменных в цикле или нескольких условий выхода.

for (int i = 0, b = 11, c = 22; i < 22 || b < 11; i++, b--)

Вечный цикл с помощью for:

for (; ; ) {  
}

*For и while*

for (int i = 0; i <= 20; i++) {  
 if ((i % 7) == 0) continue;  
 System.*out*.println(i);  
}

напечатает все числа, кроме 7 и 14

int i = 1;  
while (i <= 20) {  
 if ((i % 7) == 0) {  
 continue;  
 }  
 System.*out*.println(i);  
 i++;  
}

напечатает числа до 6, дальше повиснет в вечном цикле, из-за того, что I станет равно 7 и не сможет добраться до инкремента. Видимо, в цикле for инкрементация происходит в самом конце, даже после break или continue.

**for-each –** синтаксический сахар над записью вида

Iterator<Integer> iter = list.iterator();  
while (iter.hasNext()) {  
 System.*out*.println(iter.next());  
}

Появился в Java 5, можно использовать с массивами и наследниками Iterable.

**Switch – case:**

Изначально в case возможно было использовать только примитивные типы byte, short, char, int и их обертки.

Механика: case должен иметь уникальное значение, условием совпадения является равенство, а не логическое условие, как в if. Первое совпадение с условием инициирует выполнение его кода и всех case, которые описаны ниже него до break, блок default выполняется в случае отсутствия совпадений с case.

Java 5: Добавлена возможность использовать Enum

Java 7: Добавлена возможность использовать String

Java 12: Возможность возвращать значения с помощью break (сейчас недоступно). Возможность присвоить значение switch case переменной.  
Новый синтаксис – стрелочные метки -> , убирают необходимость явно использовать break.

Java 13: yield для возврата значения из case.

Java 14: Стабилизирован синтаксис, убрана возможность возвращать значения с помощью break;

Java 17: Можно использовать null в case.

**Операторы перехода** – break; continue; return;

**Break** – полностью завершает блок кода (if, for, switch)

Метки (label) в Java - это идентификаторы, определяют место, к которому можно вернуться в программе из любой другой точки.

Одним из наиболее распространенных способов использования меток является их использование с оператором break. Если оператор break используется с меткой, он прерывает выполнение оператора, помеченного этой меткой, и переходит к следующему оператору за меткой

outerLoop:  
for (int i = 0; i < 5; i++) {  
 innerLoop:  
 for (int j = 0; j < 5; j++) {  
 if (i == 2 && j == 2) {  
 break outerLoop;  
 }  
 System.*out*.println("i = " + i + ", j = " + j);  
 }  
}

Здесь break содержит метку outerLoop, что означает, что он прервет все вложенные блоки включая эту метку, т.е. полностью прервет оба цикла в данном случае, если использовался бы break без метки, то он прервал бы только текущий цикл и снова пошел бы по внешнему;

**Continue** – завершает текущую итерацию блока, но полностью не завершает цикл, может также использоваться с метками;

**Return** – завершает выполнение текущего метода и передает управление вызвавшему его методу передавая результат метода (может использоваться в методах void, тогда метод просто прервется).

**Var - Local Variable Type Inference** (Java 10)

Используется для объявления переменной с неявным типом данных (просто для сокращения кода), язык по-прежнему остается строго типизированным.

Может быть использован только в местах, где можно явно вывести тип переменной исходя из ее значения. Не может быть использован в аргументах методов, возвращаемых значениях или, например, инициализирован null без явного приведения к какому-либо классу.

**Varargs** – Аргументы переменной длины Variable Arguments (Java 5).

Обозначается как . . . используется только как аргумент для методов.

Под капотом содержит массив, обязательно должен находится последним аргументом у метода, не обязательно передавать массив значений в метод, можно просто передать пачку параметров;

**Enum –** специальный тип класс, цель которого ограничить создание собственных экземпляров определенным списком (Java 5).

Структура:

Final класс, наследник абстрактного Enum (не может наследовать классы, и нельзя наследоваться от него, только имплеменить интерфейсы).  
Конструктор – только private – необходим для параметризации объявленных констант.  
Поля – перечисления объявляются как неявно public static final объекты Enum, также могут быть определены поля этих констант и статические поля, которые определяют общие данные для всех перечислений. Поля могут иметь любые модификаторы доступа, кроме полей-перечислений (всегда public).  
Методы – кастомные (не от abstract Enum) методы также создаются без ограничений. Особенностью является возможность объявления собственных abstract методов – такой метод должен иметь собственную реализацию для каждой константы.

enum Enum {  
 *ENUM* {  
 @Override  
 void foo() {  
   
 }  
 };  
 abstract void foo();  
}

Abstract Enum – наследник Serializable и Comparable, каждый enum получает от него non-static поля name и ordinal и соответствующие методы их получения. Также компилятором генерируется static массив всех констант и static методы values() для его получения и valueOf() для получения элемента по имени.

**Comparable и Comparator –** Интерфейсы, имплементация которых определяет порядок следования объектов во время сортировки.

**Comparable** определяет естественный порядок сортировки, то есть класс самостоятельно определяет стандартный способ сравнения своих объектов. Определяет метод obj.compareTo(Object other) в котором реализуется стандартный способ сравнения.

**Comparator** интерфейс, реализации которого определяют способы сравнения объектов никак не связанных с этим классом. То есть создается сущность, ответственность которой определять способы сравнения объектов определенного типа. Определяет метод compare(Object one, Object two).

**Сравнение:**

|  |  |
| --- | --- |
| Comparable | Comparator |
| Определяет способ сравнения в классе, объекты которого будут сравниваться | Определяет способ сравнения вне класса, обычно путем внедрения своего объекта, который и будет управлять сравнением. |
| Определяет одну стандартную стратегию сравнения | Определяет различные стратегии для сравнения объекта. |
| Номинально не объявлен как функциональный интерфейс, но по спеке является таковым. | Функциональный интерфейс |

Вывод: Comparable следует использовать, когда необходимо задать некоторый стандартный порядок сортировки, Comparator – если он не определен и нет возможности вносить изменения в класс либо необходимы несколько сценариев сравнения, либо нужно предоставить сценарий отличный от стандартного.

Интерфейсы используются в реализованных сортировках типа Collections.sort или Arrays.sort, также используются в структурах данных типа TreeSet/TreeMap.

***Iterator –*** описан в документе по паттернам, особенностью реализации в коллекциях является свойство итератора отслеживать изменения в коллекции во время итерации с помощью переменной modcount, если во время итерации коллекция изменялась извне (не с помощью итератора), то выбрасывается ConcurrentModificationException – имеется в виду именно стурктура списка, а не изменение его значений. То есть коллекция не может быть изменена при обходе циклом for-each, если во время обхода требуется изменить коллекцию, то нужно изменять ее с помощью методов итератора.

***This и super***

**Super** обеспечивает доступ к функциональности суперкласса в контексте подкласса. **This** представляет текущий экземпляр класса в методе (на котором он был вызван), **super** текущий экземпляр суперкласса.

this( ) и super( ) – для вызова конструктора, этого же класса с нужными параметрами или вызова конструктора суперкласса с нужными параметрами – должны стоять первой строчкой. У дочернего класса конструктор суперкласса должен вызываться всегда.

public class Parent {  
 int a;  
 public Parent(int a) {  
 this.a = a;  
 }  
}

public class Child extends Parent {  
 public Child(int a) {  
 super(a);//Вызов конструктора суперкласса с параметром int (единственного конструктора)  
 //Если бы был конструктор без параметров, то он подставился бы автоматически  
 }  
}

Это явление называется делегирование конструктора, так же может быть использовано с this( ), для сокращения кода.

public class Parent {  
 int a;  
 int b;  
 public Parent(int a) {  
 this.a = a;  
 }  
 public Parent(int a, int b) {  
 this(a);//Обращаемся к конструктору Parent(int a)  
 this.b = b;  
 }  
}

**This.** и **super**. используются для обращения к полям и методам текущего класса или суперкласса, в основном, если имеет место затирание переменных или перекрытие статических методов при overriding. Переменным this и super в методах нельзя присваивать новые значения, они объявлены как final.