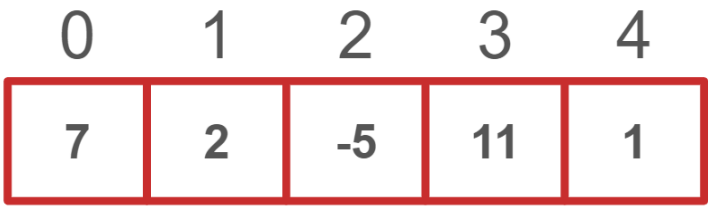
Массивы. Цикл for

Массивы

Массивы являются фундаментальной концепцией в программировании и информатике. Они представляют собой структуру данных, состоящую из набора элементов одного и того же типа. Эти элементы упорядочены и доступны по их **индексу**, который является числовым представлением их позиции в массиве. Индексы обычно начинаются с нуля, что означает, что первый элемент массива находится под индексом 0, второй элемент - под индексом 1 и так далее.

Особенностью массива является его статический характер: размер массива задается при его создании и не может быть изменен после этого. Это означает, что для добавления или удаления элементов в массиве необходимо создать новый массив и, если нужно, скопировать элементы из старого массива в новый.

Массивы находят широкое применение: они используются для хранения и последовательного доступа к разнообразным данным, например, спискам имен или числовым значениям; играют ключевую роль в математических и алгоритмических задачах, представляя векторы и матрицы; служат основой для создания более сложных структур данных, включая списки, стеки, очереди и деревья; обеспечивают эффективность обработки больших объемов данных за счет последовательного хранения элементов в памяти.



Объявление массива:

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

Запись значений по индексам:

```
a[0] = 7;
a[1] = 2;
a[2] = -5;
a[3] = 11;
a[4] = 1;
```

Получение значения по индексу с последующим выводом:

```
System.out.println(a[2]);
```

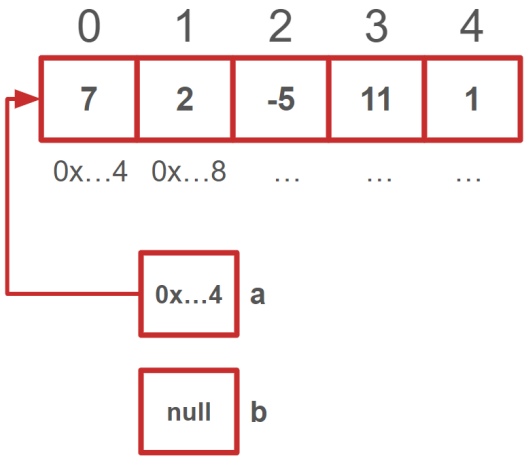
Явная инициализация массива - это процесс, при котором элементы массива указываются непосредственно в момент его объявления. Этот метод инициализации чрезвычайно удобен и эффективен в ситуациях, когда вы точно знаете, какие значения должны содержаться в массиве до запуска программы. Например, если вам нужно создать массив с фиксированным набором данных, таких как дни недели, месяцы года или предустановленные настройки, явная инициализация позволяет сразу же определить эти значения, упрощая код и повышая его читаемость. Однако стоит помнить, что данный метод имеет ограничения по гибкости, так как размер массива фиксируется и значения определяются на этапе компиляции.

```
int[] numbers = {7, 2, -5, 11, 1};

String words = {"Hello", "Bye", "Java"};
```

Массив в Java, подобно объектам классов (например, классу String в языках программирования), **является ссылочным типом данных**. Это означает, что переменная массива фактически хранит адрес (или ссылку) на первый элемент массива в памяти, а не сам массив. Таким образом, при работе с массивом, вы оперируете ссылкой на него, что позволяет эффективно управлять большими наборами данных, не затрачивая ресурсы на копирование значений элементов массива.

Подобно другим ссылочным типам данных, переменная массива может находиться в состоянии, когда она не ссылается ни на какой адрес в памяти, то есть имеет значение **null**. В этом случае, она не указывает на какой-либо массив и попытка доступа к элементам массива через такую переменную приведет к ошибке выполнения программы, такой как **NullPointerException** в Java. Это особенно важно при написании надежного и безопасного кода, так как необходимо учитывать и обрабатывать ситуации, когда переменная массива может быть `null`.

```
int[] a = {7, 2, -5, 11, 1};
int[] b = null;
```



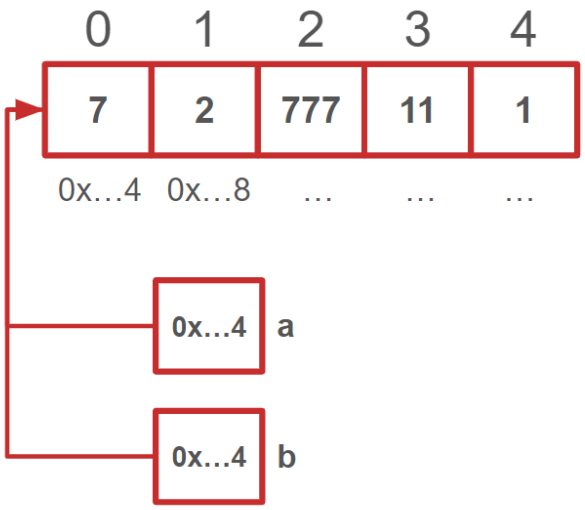
Рассмотрим код:

```
int[] a = {7, 2, -5, 11, 1};

int[] b = null;

b = a; // копирование ссылки

b[2] = 777;

System.out.println(a[2]); // 777
```

Особенность данного кода заключается в том, что он демонстрирует ключевой аспект работы с массивами. В этом коде происходит не копирование элементов массива `a` в массив `b`, а копирование ссылки на массив. Это означает, что после выполнения операции `b = a;` обе переменные `a` и `b` ссылаются на один и тот же массив в памяти.

Неожиданный момент в этом коде происходит при изменении элемента массива через одну из переменных, в данном случае `b[2] = 777;`. Поскольку `a` и `b` указывают на один и тот же массив, изменение элемента через переменную `b` также отражается на массиве, к которому ссылается переменная `a`. Таким образом, когда мы выводим `a[2]`, мы видим значение `777`, хотя кажется, что изменения были внесены только в массив `b`.

Это важное поведение следует учитывать при работе с ссылочными типами данных, так как оно может привести к неочевидным и нежелательным побочным эффектам, если программист не осознает, что разные переменные могут ссылаться на один и тот же объект в памяти. Это поведение также подчеркивает важность явного копирования массивов, когда требуется создать полностью независимую копию.



Цикл for

Цикл **for** можно рассматривать как синтаксический сахар над циклом while. Он предлагает более простой, читаемый и компактный способ для итерации, особенно при работе с массивами и коллекциями.

Структура:

```
for (инициализация; условие окончания; инкремент/декремент) {
      // блок кода
}
```

Инициализация: Здесь устанавливается начальное состояние, чаще всего это счетчик цикла.
Условие окончания: Цикл продолжается до тех пор, пока это условие истинно.
Инкремент/Декремент: Изменение счетчика после каждой итерации цикла.
Блок кода: Код, который выполняется на каждой итерации цикла.

Цикл `for` обладает рядом преимуществ. Одним из ключевых является улучшенная читаемость: он позволяет объединить инициализацию, условие и инкремент/декремент в одной строке, значительно упрощая чтение и понимание кода. Это также снижает вероятность ошибок, поскольку уменьшает шансы забыть об обновлении счетчика цикла, что является частой проблемой в циклах `while`. Кроме того, переменная счетчика цикла, как правило, ограничена областью действия самого цикла `for`, что способствует более чистому и организованному коду, уменьшая вероятность случайного взаимодействия с другими частями программы.

Пример (вывод последовательности чисел от 0 до 5):

```
for (int i = 0; i < 5; i++) {
      System.out.println(i);
}
```

Пример (вывод массива):

```
for (int i = 0; i < a.length; i++) {
      System.out.println(a[i]);
}
```

Пример (считывание массива):

```
for (int i = 0; i < a.length; i++) {
      a[i] = scanner.nextInt();
}
```

break

Оператор **break** Он используется для немедленного прерывания цикла, независимо от того, было ли нарушено условие его окончания. Это особенно полезно в ситуациях, когда во время выполнения цикла возникает сценарий, требующий немедленного выхода из него, например, когда найден нужный элемент в массиве или достигнуто специфическое условие. Применение `break` позволяет избежать лишних итераций, что не только экономит ресурсы и время выполнения программы, но и способствует написанию более чистого и эффективного кода. Вместо того чтобы дожидаться естественного завершения цикла, программист может точно указать условие, при котором цикл должен быть немедленно прерван. Это делает код более читаемым и легким для понимания, так как явные условия прерывания цикла указывают на важные моменты в логике программы.

Однако использование `break` требует внимательности и аккуратности, поскольку неосторожное его применение может привести к трудно обнаруживаемым ошибкам в логике программы. Особенно это актуально в сложных циклах с множественными уровнями вложенности, где неправильное использование `break` может привести к нежелательным результатам. Тем не менее, когда `break` используется правильно, он становится мощным инструментом для повышения эффективности и читаемости кода.

Пример (нахождение первого отрицательного числа):

```
for (int i = 0; i < a.length; i++) {
      if (a[i] < 0) {
            System.out.println(a[i]);
            break; // прерывание цикла
        }
```

```
}

// выполнение программы будет продолжено здесь
```

continue

Оператор `continue` осуществляет переход к следующей итерации цикла, при этом пропускается выполнение оставшейся части кода текущей итерации. Этот оператор наиболее полезен в ситуациях, когда некоторые условия внутри цикла не требуют выполнения всего блока кода, и программе нужно быстро перейти к следующему шагу.

В отличие от оператора `break`, который полностью прерывает цикл, `continue` оказывает влияние только на текущую итерацию. При выполнении `continue` цикл не прекращается полностью, а просто пропускает остальную часть текущей итерации и продолжает со следующей. Это делает `continue` идеальным инструментом для случаев, когда нужно игнорировать определенные значения или условия, но при этом сохранять цикл активным.

Примером использования `continue` может служить цикл, который перебирает элементы массива и выполняет действия только для тех элементов, которые удовлетворяют определенному условию. Если условие не выполняется, `continue` активируется, и цикл немедленно переходит к следующему элементу, минуя оставшуюся часть кода в текущей итерации.

Использование `continue` способствует написанию более чистого и организованного кода, так как позволяет избежать вложенных условий и делает структуру цикла более ясной. Тем не менее, как и в случае с `break`, важно осторожно использовать `continue`, чтобы избежать создания запутанных или трудно читаемых циклов, особенно при наличии множественных условий и вложенностей. Правильно применяемый, `continue` улучшает читаемость кода и способствует более эффективной реализации циклических алгоритмов.

Пример (получение суммы всех положительных четных элементов массива):

```
for (int i = 0; i < a.length; i++) {
        // для нечетного - переходим к следующему
        if (a[i] % 2 == 1) continue;

        //для отрицательного - переходим к следующему
        if (a[i] < 0) continue;

        sum = sum + a[i];
}
```

```
// выполнение программы будет продолжено здесь после прохода всех итераций
```