# Массив

Массив — это структура данных, в которой хранятся элементы одного типа. Его можно представить, как набор пронумерованных ячеек, в каждую из которых можно поместить какие-то данные (один элемент данных в одну ячейку). Доступ к конкретной ячейке осуществляется через её номер. Номер элемента в массиве также называют **индексом**. В случае с Java массив однороден, то есть во всех его ячейках будут храниться элементы одного типа. Так, массив целых чисел содержит только целые числа (например, типа int), массив строк — только строки, массив из элементов созданного нами класса Dog будет содержать только объекты Dog. То есть в Java мы не можем поместить в первую ячейку массива целое число, во вторую String, а в третью — “собаку”.

Обращение к элементу массива происходит по имени массива, за которым следует значение индекса элемента, заключённого в квадратные скобки. Например, на первый элемент массива cats можно ссылаться как на cats[0], на пятый элемент как cats[4].

В качестве индекса можно использовать числа или выражения, которые создают положительное значение типа int. Поэтому при вычислении выражения с типом long, следует преобразовать результат в int, иначе получите ошибку. С типами short и byte проблем не будет, так как они полностью укладываются в диапазон int.

ак и любой другой объект, создать массив Java, то есть зарезервировать под него место в памяти, можно с помощью оператора **new**. Делается это так:

**new** typeOfArray [length];

Где **typeOfArray** — это тип массива, а **length** — его длина (то есть, количество ячеек), выраженная в целых числах (int). Однако здесь мы только выделили память под массив, но не связали созданный массив ни с какой объявленной ранее переменной. Обычно массив сначала объявляют, а потом создают, например:

**int**[] myArray; // объявление массива

myArray = **new** int[10]; // создание, то есть, выделение памяти для массива на 10 элементов типа int

Здесь мы объявили массив целых чисел по имени myArray, а затем сообщили, что он состоит из 10 ячеек (в каждой из которых будет храниться какое-то целое число). Однако гораздо чаще массив создают сразу после объявления с помощью такого сокращённого синтаксиса:

**int**[] myArray = **new** int[10]; // объявление и выделение памяти “в одном флаконе”

Обратите внимание: После создания массива с помощью **new**, в его ячейках записаны значения по умолчанию. Для численных типов (как в нашем примере) это будет 0, для boolean — false, для ссылочных типов — null.

String[] seasons = **new** String[4]; /\* объявили и создали массив. Java выделила память под массив из 4 строк, и сейчас в каждой ячейке записано значение null (поскольку строка — ссылочный тип)\*/

seasons[0] = "Winter"; /\* в первую ячейку, то есть, в ячейку с нулевым номером мы записали строку Winter. Тут мы получаем доступ к нулевому элементу массива и записываем туда конкретное значение \*/

seasons[1] = "Spring"; // проделываем ту же процедуру с ячейкой номер 1 (второй)

seasons[2] = "Summer"; // ...номер 2

seasons[3] = "Autumn"; // и с последней, номер 3

Теперь во всех четырёх ячейках массива записаны названия сезонов. Инициализацию также можно провести по-другому, совместив с инициализацией и объявлением:

String[] seasons = **new** String[] {"Winter", "Spring", "Summer", "Autumn"};

Более того, оператор new можно опустить:

String[] seasons = {"Winter", "Spring", "Summer", "Autumn"};

## Быстрый вывод двумерного массива

Самый короткий способ вывести список элементов двумерного массива на экран — применение метода deepToString класса Arrays. Пример:

**int**[][] myArray = {{18,28,18},{28,45,90},{45,3,14}};

System.out.printLn(Arrays.deepToString(myArray));

Результат работы программы — следующий вывод: [[18, 28, 18], [28, 45, 90], [45, 3, 14]]

Объявление многомерных массивов:

**int**[][] twoDimArray; //двумерный массив

Процедура объявления и создания двумерного массива практически такая же, как и в случае одномерного:

**int**[][] twoDimArray = **new** int[3][4];

## Непрямоугольные двумерные массивы

Поскольку в Java двумерные массивы — это массивы массивов, каждый из внутренних массивов может быть разной длины. Создавая массив, мы можем указать только количество строк и не указывать количество столбцов (то есть, по сути, длину этих самых строк). Рассмотрим пример.

//объявляем и создаём массив, указывая только количество строк

**int** [][] twoDimArray = **new** int[5][];

//инициализируем массив, заполняя его массивами разной длины

twoDimArray[0] = **new** int[]{1, 2, 3, 4, 5};

twoDimArray[1] = **new** int[]{1,2,3,4};

twoDimArray[2] = **new** int[]{1,2,3};

twoDimArray[3] = **new** int[]{1,2};

twoDimArray[4] = **new** int[]{1};

//выведем получившийся непрямоугольный двумерный массив на экран

**for** (**int** i = 0; i < twoDimArray.length; i++) {

**for** (**int** j = 0; j < twoDimArray[i].length; j++) {

System.out.print(" " + twoDimArray[i][j] + " ");

}

System.out.println();

}

Вывод программы: 1 2 3 4 5 1 2 3 4 1 2 3 1 2 1 Таким образом, нулевая строка нашего массива содержит массив {1,2,3,4,5}, а четвёртая — массив {1}.

Единственный способ реализации многомерных массивов в этих языках — создавать массивы массивов, а именно это и представляют собой зубчатые массивы.

Однако поскольку каждый элемент массива верхнего уровня является отдельным экземпляром массива, каждый экземпляр массива верхнего уровня может быть любого размера. Поэтому такой массив не обязательно прямоуголен — отсюда и термин **зубчатые массивы.**

Что бы узнать длину зубчатого массива необходимо суммировать длину каждого “вложенного” массива

## Главное о массивах

* Главные характеристики массива: тип помещённых в него данных, имя и длина.  
  Последнее решается при инициализации (выделении памяти под массив), первые два параметра определяются при объявлении массива.
* Размер массива (количество ячеек) нужно определять в int
* Изменить длину массива после его создания нельзя.
* Доступ к элементу массива можно получить по его индексу.
* В массивах, как и везде в Java, элементы нумеруются с нуля.
* После процедуры создания массива он наполнен значениями по умолчанию.
* Массивы в языке Java устроены не так, как в C++. Они почти совпадают с указателями на динамические массивы.

# java.lang.ArrayIndexOutOfBoundsException

Это исключение выброшено, чтобы указать, что массив был доступен с помощью незаконного индекс. Индекс является либо отрицательным, либо большим или равным размер массива.

## В чем причина этого?

Это исключение означает, что вы пытались получить доступ к индексу в массив или массив, и этот индекс не существует.

Java использует индексы на основе 0. Это означает, что все индексы начинаются с 0 как индекс первого элемента, если он содержит какие-либо элементы.

Сообщение IndexOutOfBoundsException очень явное, обычно оно принимает форму:

java.lang.IndexOutOfBoundsException: Index: 1, Size: 1

Где Index - запрошенный вами индекс, который не существует, а Size - это длина структуры, в которую вы индексировали.

### Исключение ArrayStoreException

Если попытаться записать в ячейку массива ссылку на объект неправильного типа, возникнет исключение ArrayStoreException.

Ниже мы создаем массив класса Object, предназначенный для хранения строк класса String:

Object szStr[] = new String[10];

try

{

szStr[0] = new Character('\*');

}

catch(Exception ex)

{

System.out.println(ex.toString());

}

При попытке записать в первый элемент этого массива ссылку на объект класса Character возникает исключение:

java.lang.ArrayStoreException

# Линейные структуры

# **Массив**

Когда вам нужен один объект, вы создаёте один объект. Когда нужно несколько объектов, тогда есть несколько вариантов на выбор. Я видел, как многие новички в коде пишут что-то типа такого:

*// Таблица рекордов*

int score1 = 0;

int score2 = 0;

int score3 = 0;

int score4 = 0;

int score5 = 0;

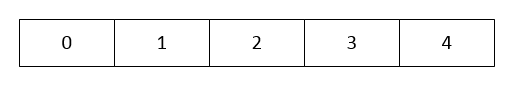
Это даёт нам значение пяти рекордов. Этот способ неплохо работает, пока вам не потребуется пятьдесят или сто объектов. Вместо создания отдельных объектов можно использовать массив.

*// Таблица рекордов*

const int NUM\_HIGH\_SCORES = 5;

int highScore[NUM\_HIGH\_SCORES] = {0};

Будет создан буфер из 5 элементов, вот такой:



Заметьте, что индекс массива начинается с нуля. Если в массиве пять элементов, то они будут иметь индексы от нуля до четырёх.

## **Недостатки простого массива**

Если вам нужно неизменное количество объектов, то массив вполне подходит. Но, допустим, вам нужно добавить в массив ещё один элемент. В простом массиве этого сделать невозможно. Допустим, вам нужно удалить элемент из массива. В простом массиве это так же невозможно. Вы привязаны к одному количеству элементов. Нам нужен массив, размер которого можно менять. Поэтому нам лучше выбрать…

# **Динамический массив**

Динамический массив — это массив, который может менять свой размер. Основные языки программирования в своих стандартных библиотеках поддерживают динамические массивы. В C++ это *vector*. В Java это *ArrayList*. В C# это *List*. Все они являются динамическими массивами. В своей сути динамический массив — это простой массив, однако имеющий ещё два дополнительных блока данных. В них хранятся действительный размер простого массива и объём данных, который может на самом деле храниться в простом массиве. Динамический массив может выглядеть примерно так:

*// Внутреннее устройство класса динамического массива*

sometype \*internalArray;

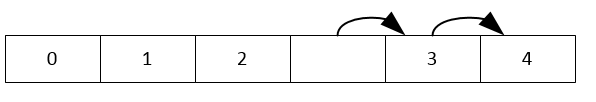
unsigned int currentLength;

unsigned int maxCapacity;

Элемент *internalArray* указывает на динамически размещаемый буфер. Действительный массив буфера хранится в *maxCapacity*. Количество использовуемых элементов задаётся *currentLength*.

## **Добавление к динамическому массиву**

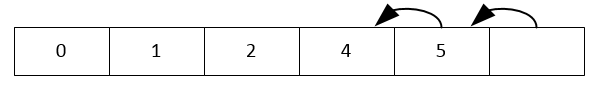
При добавлении объекта к динамическому массиву происходит несколько действий. Класс массива проверяет, достаточно ли в нём места. Если *currentLength < maxCapacity*, то в массиве есть место для добавления. Если места недостаточно, то размещается больший внутренний массив, и всё копируется в новый внутренний массив. Значение maxCapacity увеличивается до нового расширенного значения. Если места достаточно, то добавляется новый элемент. Каждый элемент после точки вставки должен быть скопирован на соседнее место во внутреннем массиве, и после завершения копирования пустота заполняется новым объектом, а значение *currentLength* увеличивается на единицу.

  
  
Поскольку необходимо перемещать каждый объект после точки вставки, то наилучшим случаем будет добавление элемента к концу. При этом нужно перемещать ноль элементов (однако внутренний массив всё равно требует расширения). Динамический массив лучше всего работает при добавлении элемента в конец, а не в середину.

При добавлении объекта к динамическому массиву каждый объект может переместиться в памяти

## **Удаление из динамического массива**

Удаление объектов требует меньше работы, чем добавление. Во-первых, уничтожается сам объект. Во-вторых, каждый объект после этой точки сдвигается на один элемент. Наконец, currentLength уменьшается на единицу.



Как и при добавлении к концу массива, удаление из конца массива является наилучшим случаем, потому что при этом нужно перемещать ноль объектов. Также стоит заметить, что нам не нужно изменять размер внутреннего массива, чтобы сделать его меньше. Выделенное место может оставаться таким же, на случай, если мы позже будем добавлять объекты.

Удаление объекта из динамического массива приводит к смещению в памяти всего после удалённого элемента.

## **Недостатки динамических массивов**

Допустим, массив очень велик, а вам нужно часто добавлять и удалять объекты. При этом объекты могут часто копироваться в другие места, а многие указатели становиться недействительными. Если вам нужно вносить частые изменения в середине динамического массива, то для этого есть более подходящий тип линейной структуры данных…

# **Связные списки**

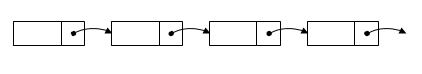
Массив — это непрерывный блок памяти, и каждый элемент его расположен после другого. Связанный список — это цепочка объектов. Связанные списки тоже присутствуют в стандартных библиотеках основных языков программирования. В C++ они называются *list*. В Java и C# это *LinkedList*. Связанный список состоит из серии узлов. Каждый узел выглядит примерно так:

*// Узел связанного списка*

sometype data;

Node\* next;

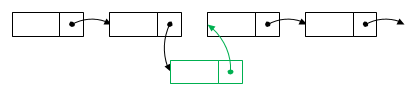
Он создаёт структуру такого типа:



Каждый узел соединяется со следующим.

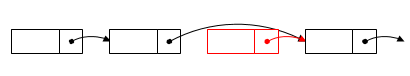
## **Добавление к связанному списку**

Добавление объекта к связанному списку начинается с создания нового узла. Данные копируются внутрь узла. Затем находится точка вставки. Указатель нового узла на следующий объект изменяется так, чтобы указывать на следующий за ним узел. Наконец, узел перед новым узлом изменяет свой указатель, чтобы указывать на новый узел.



## **Удаление из связанного списка**

При удалении объекта из связанного списка находится узел перед удаляемым узлом. Он изменяется таким образом, чтобы указывать на следующий после удалённого объекта узел. После этого удалённый объект можно безопасно стереть.



## **Преимущества связанного списка**

Самое большое преимущество связанного списка заключается в добавлении и удалении объектов из списка. Внесение изменений в середину списка выполняется очень быстро. Помните, что динамический массив теоретически мог вызывать смещение каждого элемента, а связанный список сохраняет каждый другой объект на своём месте.

## **Недостатки связанного списка**

Вспомните, что динамический массив — это непрерывный блок памяти.  
  
Если вам нужно получить пятисотый элемент массива, то достаточно просто посмотреть на 500 «мест» вперёд. В связанном списке память соединена в цепочку. Если вам нужно найти пятисотый элемент, то придётся начинать с начала цепочки и следовать по её указателю к следующему элементу, потом к следующему, и так далее, повторяя пятьсот раз.  
  
Произвольный доступ к связанному списку выполняется очень медленно.  
  
Ещё один серьёзный недостаток связанного списка не особо очевиден. Каждому узлу необходимо небольшое дополнительное место. Сколько ему нужно места? Можно подумать, что для него нужен только размер указателя, но это не совсем так. При динамическом создании объекта всегда существует небольшой запас. Некоторые языки программирования, например, C++, работают со страницами памяти. Обычно страница занимает 4 килобайта. При использовании операторы добавления и удаления, размещается целая страница памяти, даже если вам нужно использовать только один байт.  
  
В Java и C# всё устроено немного иначе, в них есть специальные правила для небольших объектов. Для этих языков не требуется вся 4-килобайтная страница памяти, но всё равно у них есть небольшой запас. Если вы используете стандартные библиотеки, то о втором недостатке волноваться не нужно. Они написаны таким образом, чтобы минимизировать занимаемое впустую место.

# **Заключение**

Эти три типа (массив, динамический массив и связанный список) создают основу почти для всех более сложных контейнеров данных.   
Эти структуры являются в программировании фундаментальными. Не важно, какой язык вы будете изучать, для работы с данными вы будете их использовать.

**Линейные структуры данных с конечными точками**

# **Стек**

Представьте, что у вас есть куча листов бумаги.  
  
Мы кладём один лист в стопку. Теперь мы можем получить доступ только к верхнему листу.  
  
Мы кладём ещё один лист в стопку. Предыдущий лист теперь скрыт и доступ к нему невозможен, мы можем использовать верхний лист. Когда мы закончим с верхним листом, мы можем убрать его со стопки, открыв доступ к лежащему под ним.  
  
В этом заключается идея стека. Стек — это структура LIFO. Это расшифровывается как Last In First Out («последним вошёл, первым вышел»). При добавлении и удалении из стека последний добавленный элемент будет первым удаляемым.  
  
Для стека нужно всего три операции: Push, Pop и Top.  
  
Push добавляет объект в стек. Pop удаляет объект из стека. Top даёт самый последний объект в стеке. Эти контейнеры в большинстве языков являются частью стандартных библиотек. В C++ они называются *stack*. В Java и C# это *Stack*. (Да, единственная разница в названии с заглавной буквы.) Внутри стек часто реализуется как динамический массив. Как вы помните из этой структуры данных, самыми быстрыми операциями на динамических массивах являются добавление и удаление элементов из конца. Поскольку стек всегда добавляет и удаляет с конца, обычно push и pop объектов в стеке выполняется невероятно быстро.

# **Очередь**

Представьте, что вы стоите в очереди за чем-то.  
  
Первого человека в очереди обслуживают, после чего он уходит. Потом обслуживается и уходит второй в очереди. Другие люди подходят к очереди и встают в её конец. Вот в этом заключается идея структуры данных «очередь».

Очередь — это структура FIFO (First In First Out, «первым зашёл, первым вышел»).  
  
При добавлении и удалении из очереди первый добавляемый элемент будет первым извлекаемым. Очереди нужно только несколько операций: Push\_Back, Pop\_Front, Front и Back. Push\_Back добавляет элемент к концу очереди. Pop\_Front удаляет элемент из начала очереди. Front и Back позволяют получить доступ к двум концам очереди.  
  
Программистам часто нужно добавлять или удалять элементы из обоих концов очереди. Такая структура называется двухсторонней очередью (double ended queue, deque). В этом случае добавляется ещё пара операций: Push\_Front и Pop\_Back. Эти контейнеры тоже включены в большинство основных языков. В C++ это *queue* и *deque*. Java определяет интерфейсы для очереди и двухсторонней очереди, а затем реализует их через *LinkedList*. В C# есть класс *Queue*, но нет класса Deque.  
  
Внутри очередь и двухсторонняя очередь могут быть устроены довольно сложно. Поскольку объекты могут поступать и извлекаться с любого конца, внутренний контейнер должен уметь наращивать и укорачивать очередь с начала и с конца. Во многих реализациях используются множественные страницы памяти. Когда любой из концов разрастается за пределы текущей страницы, добавляется дополнительная страница. Если страница больше не нужна, то она удаляется. В Java используется следующий способ: для связанного списка требуется немного дополнительной памяти, а не страниц памяти, но для этого языка такая реализация вполне работает.

# **Очередь с приоритетом**

Это очень распространённая вариация очереди. Очередь с приоритетом очень похожа на обычную очередь.  
  
Программа добавляет элементы с конца и извлекает элементы из начала. Разница в том, что можно задавать приоритеты определённым элементам очереди. Все самые важные элементы обрабатываются в порядке FIFO. Потом в порядке FIFO обрабатываются элементы с более низким приоритетом. И так повторяется, пока не будут обработаны в порядке FIFO элементы с самым низким приоритетом.  
  
При добавлении нового элемента с более высоким приоритетом, чем остальная часть очереди, он сразу же перемещается в начало очереди. В C++ эта структура называется *priority\_queue*. В Java это *PriorityQueue*. В стандартной библиотеке C# очереди с приоритетом нет. Очереди с приоритетом полезны не только для того, чтобы встать первым на очереди к принтеру организации. Их можно использовать для удобной реализации алгоритмов, например, процедуры поиска A\*. Наболее вероятным результатам можно отдать более высокий приоритет, менее вероятным — более низкий. Можно создать собственную систему для сортировки и упорядочивания поиска A\*, но намного проще использовать встроенную очередь с приоритетом.

# **Заключение**

Стеки, очереди, двухсторонние очереди и очереди с приоритетом можно реализовать на основе других структур данных. Это не фундаментальные структуры данных, но их часто используют. Они очень эффективны, когда нужно работать только с конечными элементами данных, а серединные элементы не важны.

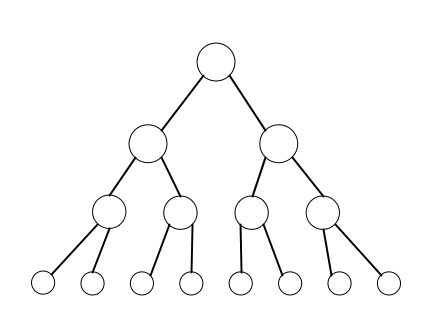
## **Деревья и кучи.**

# **Структуры данных «деревья»**

Деревья данных очень полезны во многих случаях. В разработке видеоигр структуры деревьев используются для подразделения пространства, позволяющего разработчику быстро находить находящиеся рядом объекты без необходимости проверки каждого объекта в игровом мире. Даже несмотря на то, что структуры деревьев являются фундаментальными в информатике, на практике в большинстве стандартных библиотек нет непосредственной реализации контейнеров на основе деревьев.

# **Простое дерево**

Дерево — это… дерево. У настоящего дерева есть корень, ветви, а на концах ветвей есть листья.  
Структура данных дерева начинается с корневого узла. Каждый узел может разветвляться на дочерние узлы. Если у узла нет дочерних элементов, то он называется узлом листа. Когда деревьев несколько, это называется лесом. Вот пример дерева. В отличие от настоящих деревьев они растут сверху вниз: корневой узел обычно рисуется сверху, а листья — внизу.



Одним из первых возникает вопрос: сколько каждый узел может иметь дочерних элементов?  
  
Многие деревья имеют не больше двух дочерних узлов. Они называются двоичными деревьями. На примере выше показано двоичное дерево. Обычно дочерние элементы называются левым и правым дочерними узлами. Ещё одним распространённым в играх типом деревьев является дерево с четырьмя дочерними узлами. В дереве квадрантов (quadtree), которое можно использовать для покрытия сетки, дочерние узлы обычно называются по закрываемому ими направлению: NorthWest (северо-запад) или NW, NorthEast (северо-восток) или NE, SouthWest (юго-запад) или SW и SouthEast (юго-восток) или SE.  
  
Деревья используются во многих алгоритмах (я уже упоминал о двоичных деревьях). Существуют сбалансированные и несбалансированные деревья. Бывают красно-чёрные деревья, АВЛ-деревья, и многие другие.  
  
Хотя теория деревьев и удобна, она страдает от серьёзных недостатков: места для хранения и скорости доступа. Каким способом лучше всего хранить дерево? Наиболее простым способом является построение связанного списка, он же оказывается самым худшим. Предположим, что нам нужно построить сбалансированное двоичное дерево. Мы начинаем со следующей структуры данных:

*// Узел дерева*

Node\* left;

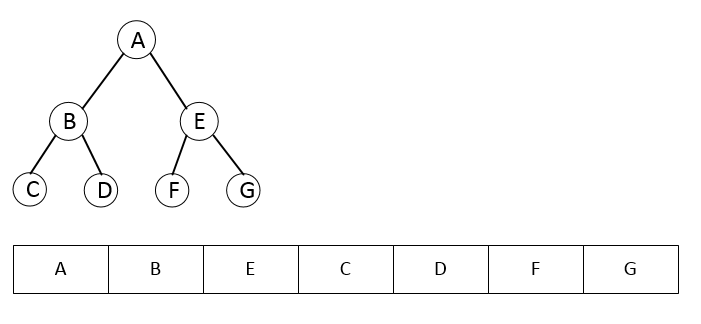
Node\* right;

sometype data;

Достаточно просто. Теперь представим, что в нём нужно хранить 1024 элемента. Тогда для 1024 узлов придётся хранить 2048 указателей.  
  
Это нормально, указатели малы и можно обойтись небольшим пространством.  
  
Вы можете помнить, что при каждом размещении объекта он занимает небольшую часть дополнительных ресурсов. Точное количество дополнительных ресурсов зависит от библиотеки используемого вами языка. Многие популярные компиляторы и инструменты могут использовать различные варианты — от всего лишь нескольких байтов для хранения данных до нескольких килобайтов, позволяющих упростить отладку. Я работал с системами, в которых размещение занимает не меньше 4 КБ памяти. В этом случае 1024 элементов потребуют около 4 МБ памяти. Обычно ситуация не настолько плоха, но дополнительные затраты на хранение множества мелких объектов нарастают очень быстро.  
  
Вторая проблема — скорость. Процессорам «нравится», когда объекты находятся в памяти рядом друг с другом. У современных процессоров есть участок очень быстрой памяти — кэш — который очень хорошо справляется с большинством данных. Когда программе требуется один фрагмент данных, кэш загружает этот элемент, а также элементы рядом с ним. Когда данные не загружены в очень быструю память (это называется «промахом кэша»), программа приостанавливает свою работу, и ждёт загрузки данных. В самом очевидном формате, когда каждый элемент дерева хранится в собственном участке памяти, ни один из них не находится рядом с другим. Каждый раз при обходе дерева программа приостанавливается.  
  
Если создание дерева напрямую связано с такими проблемами, то стоит выбрать структуру данных, работающую как дерево, но не обладающую его недостатками. И эта структура называется…

# **Куча**

Существует два вида куч.  
  
Первая — это куча в памяти. Это большой блок памяти, в котором хранятся объекты. Но я буду говорить о другой куче.  
  
Структура данных «куча» — это, в сущности, то же самое, что и дерево. У неё есть корневой узел, у каждого узла есть дочерние узлы, и так далее. Куча добавляет ограничения, её сортировка всегда должна выполняться в определённом порядке. Необходима функция сортировки — обычно оператор «меньше чем».  
  
При добавлении или удалении объектов из кучи структура сортирует себя, чтобы стать «полным» деревом, в котором каждый уровень дерева заполнен, за исключением, возможно, только последнего ряда, где всё должно быть смещено в одну сторону. Это позволяет очень эффективно обеспечить пространство для хранения и поиск по куче.  
  
Кучи можно хранить в простом или динамическом массиве, то есть на её размещение тратится мало места. В C++ есть такие функции, как push\_heap() и pop\_heap(), позволяющие реализовать кучи в собственном контейнере разработчика. В стандартных библиотеках Java и C# нет похожего функционала. Вот дерево и куча с одинаковой информацией:



# **Почему их нет в стандартных библиотеках**

Это простые, фундаментальные и очень полезные структуры данных. Многие считают, что они должны присутствовать в стандартных библиотеках. За несколько секунд в поисковике вы можете найти тысячи реализаций деревьев.  
  
Оказывается, что хотя деревья очень полезны и фундаментальны, существуют более хорошие контейнеры. Есть более сложные структуры данных, обладающие преимуществами дерева (стабильность и форма) и преимуществами кучи (пространство и скорость). Более совершенные структуры данных обычно являются сочетанием таблиц данных с таблицами поиска. Две таблицы в сочетании обеспечивают быстрый доступ, быстрое изменение, и хорошо проявляются себя и в плотных, и в неплотных ситуациях. Они не требуют переноса элементов при добавлении и удалении элементов, не потребляют излишней памяти и не фрагментируют память при расширенном использовании.

# **Заключение**

Важно знать о структурах данных «дерево», потому что в работе вам часто придётся их использовать. Также важно знать, что эти структуры данных при прямой реализации имеют недостатки. Вы можете реализовывать собственные структуры деревьев, просто знайте, что существуют более компактные типы. Зачем же я рассказал о них, если они на самом деле не используются в стандартных библиотеках? Они применяются в качестве внутренних структур в нашей следующей теме:

# Нелинейные структуры данных.

Эти структуры данных отличаются от массивов и списков. Массивы — это последовательные контейнеры. Элементы в них расположены по порядку. При добавлении нескольких элементов в определённом порядке, они остаются в этом порядке.  
  
Нелинейные структуры данных необязательно остаются в том порядке, в котором их добавляют. При добавлении или удалении элементов может измениться порядок других элементов. Внутри они состоят из деревьев и куч, рассмотренных в предыдущей части.  
  
Существует много вариаций таких структур данных. Самыми базовыми являются словарь данных, а также упорядоченное и неупорядоченное множества.

# **Словарь данных**

Обычный словарь состоит из набора слов (ключа) и определения (значения). Поскольку ключи находятся в алфавитном порядке, любой элемент можно найти очень быстро.  
  
Если словари не были бы отсортированы, то поиск слов в них был невероятно сложным.  
  
Существует два основных способа сортировки элементов в словаре: сравнение или хэш. Традиционное упорядочивание сравнением обычно более интуитивно. Оно похоже на порядок бумажном словаре, где всё отсортировано по алфавиту или по числам.  
  
При сортировке элементов таким образом может потребоваться функция сравнения. Обычно эта функция по умолчанию является оператором «меньше чем», например a < b.  
  
Второй способ сортировки элементов — использование хэша. Хэш — это просто способ преобразования блока данных в одно число. Например, строка «blue» может иметь хэш 0xa66b370d, строка «red» — хэш 0x3a72d292. Когда словарь данных использует хэш, он обычно считается неотсортированным. В действительности он всё равно отсортирован по хэшу, а не по удобному человеку критерию. Словарь данных работает тем же образом. Есть небольшая разница в скорости между использованием словарей с традиционной сортировкой и сортировкой по хэшу. Различия так малы, что их можно не учитывать.  
  
В C++ есть семейство контейнеров *map*/*mutimap* или *unordered\_map*/*unordered\_multimap*. В Java семейство называется *HashMap*, *TreeMap* или *LinkedHashMap*. Каждое из них имеет собственные особенности реализации, например сортировка по хэшу или сравнением, допущение дубликатов, но в целом концепция одинакова. Заметьте, что в каждой из стандартных библиотек имеется их упорядоченная версия (в которой задаётся сравнение) и неупорядоченная версия (где используется хэш-функция). После добавления элементов в словарь данных вы сможете изменять значения, но не ключ.  
  
Вернёмся к аналогии с бумажным словарём: можно менять определение слова, не перемещая слово в книге; если изменить написание слова, то придётся удалять первое написание и повторно вставлять слово с новым написанием. Подробности работы вы можете узнать в учебниках. Достаточно знать, что словари очень быстры при поиске данных, и могут быть очень медленными при добавлении или удалении значений.

# **Упорядоченное и неупорядоченное множество**

Упорядоченное множество — это почти то же самое, что и словарь. Вместо ключа и значения в нём есть только ключ. Вместо традиционного словаря со словами и определениями там только слова. Множества полезны, когда вам нужно хранить только слова без дополнительных данных. В C++ семейство структур называется *set*/*multiset* или *unordered\_set*/*unordered\_multiset*. В Java это *HashSet*, *TreeSet* или *LinkedHashSet*.  
  
Как и в случае со словарями, существуют упорядоченные версии (где задаётся сравнение) и неупорядоченные версии (где используется хэш-функция). После добавления ключа его тоже нельзя изменять. Вместо этого нужно удалить старый объект и вставить новый. Часто они реализуются точно так же, как словарь данных, просто хранят только значение. Поскольку они реализуются так же, то они имеют те же характеристики. В множествах очень быстро ищутся и находятся значения, но они медленно работают при добавлении и удалении элементов.

# **Заключение**

Классы контейнеров словарей данных, упорядоченных и неупорядоченных множеств очень полезны для быстрого поиска данных. Часто они реализуются как деревья или хэш-таблицы, которые очень эффективны в этом отношении. Используйте их тогда, когда требуется один раз создать данные и часто ссылаться на них. Они не так эффективны при добавлении и удалении элементов. Внесение изменений в контейнер может вызвать смещение или изменение порядка внутри него. Если вам необходимо следовать этому паттерну использования, то лучше выбрать упорядоченный связанный список.

# **Класс**

***Класс*** – логическое описание чего-либо, шаблон, с помощью которого можно создавать реальные экземпляры этого самого чего-либо. Другими словами, это просто описание того, какими должны быть созданные сущности: какими свойствами и методами должны обладать.

***Свойства*** – характеристики сущности, ***методы*** – действия, которые она может выполнять. Хорошим примером класса из реальной жизни, дающим понимание, что же такое класс, можно считать чертежи: чертежи используются для описания конструкций (катапульта, отвертка), но чертеж – это не конструкция. Инженеры используют чертежи, чтобы создавать конструкции, так и в программировании классы используются для того, чтобы создавать объекты, обладающие описанными свойствами и методами.

## Виды классов в Java

В Java есть 4 вида классов внутри другого класса:

1. ***Вложенные внутренние классы****– нестатические классы внутри внешнего класса.*
2. ***Вложенные статические классы****– статические классы внутри внешнего класса.*
3. ***Локальные классы Java****– классы внутри методов.*
4. ***Анонимные Java классы****– классы, которые создаются на ходу.*

***Модификаторы:***

### Модификатор static

**Модификатор static** — применяется для создания методов и переменных класса.

#### Переменные static

Ключевое слово static используется для создания переменных, которые будут существовать независимо от каких-либо экземпляров, созданных для класса. Только одна копия переменной static в Java существует вне зависимости от количества экземпляров класса.

Статические переменные также известны как переменные класса. В Java локальные переменные не могут быть объявлены статическими (static).

#### Методы static

Ключевое слово static используется для создания методов, которые будут существовать независимо от каких-либо экземпляров, созданных для класса.

В Java статические методы или методы static не используют какие-либо переменные экземпляра любого объекта класса, они определены. Методы static принимают все данные из параметров и что-то из этих параметров вычисляется без ссылки на переменные.

Переменные и методы класса могут быть доступны с использованием имени класса, за которым следует точка и имя переменной или метода.

**Модификатор final** — используется для завершения реализации классов, методов и переменных.

#### Переменные final

Переменная final может быть инициализирована только один раз. Ссылочная переменная, объявленная как final, никогда не может быть назначен для обозначения другого объекта.

Однако данные внутри объекта могут быть изменены. Таким образом, состояние объекта может быть изменено, но не ссылки.

С переменными в Java модификатор final часто используется со static, чтобы сделать константой переменную класса.

Метод final не может быть переопределен любым подклассом. Как упоминалось ранее, в Java модификатор final предотвращает метод от изменений в подклассе.

Главным намерение сделать метод final будет то, что содержание метода не должно быть изменено стороне.

#### Класс final

Основная цель в Java использования класса объявленного в качестве final заключается в предотвращении класс от быть подклассом. Если класс помечается как final, то ни один класс не может наследовать любую функцию из класса final.

#### Метод abstract

Метод abstract является методом, объявленным с любой реализацией. Тело метода (реализация) обеспечивается подклассом. Методы abstract никогда не могут быть final или strict.

Любой класс, который расширяет абстрактный класс должен реализовать все абстрактные методы суперкласса, если подкласс не является абстрактным классом.

Если класс в Java содержит один или несколько абстрактных методов, то класс должен быть объявлен как abstract. Абстрактный класс не обязан содержать абстрактные методы.

Абстрактный метод заканчивается точкой с запятой. Пример: public abstract sample();

**Модификатор synchronized** — используются в Java для потоков.

Ключевое слово synchronized используется для указания того, что метод может быть доступен только одним потоком одновременно. В Java модификатор synchronized может быть применен с любым из четырех модификаторов уровня доступа.

### Модификатор transient

Переменная экземпляра отмеченная как transient указывает виртуальной машине Java (JVM), чтобы пропустить определённую переменную при сериализации объекта, содержащего её.

Этот модификатор включён в оператор, что создает переменную, предшествующего класса или типа данных переменной.

### Модификатор volatile

**Модификатор volatile** — используются в Java для потоков.

В Java модификатор volatile используется, чтобы позволить знать JVM, что поток доступа к переменной всегда должен объединять свою собственную копию переменной с главной копией в памяти.

Доступ к переменной volatile синхронизирует все кэшированные скопированные переменные в оперативной памяти. Volatile может быть применен только к переменным экземпляра, которые имеют тип объект или private. Ссылка на объект volatile может быть null.

#### Пример

public class MyRunnable implements Runnable{

private volatile boolean active;

public void run(){

active = true;

while (active){ // линия 1

// здесь какой-нибудь код

}

}

public void stop(){

active = false; // линия 2

}

}

Как правило, run() вызывается в одном потоке (впервые начинаете использовать Runnable в Java), а stop() вызывается из другого потока. Если в линии 1 используется кэшированное значение active, то цикл не может остановиться, пока Вы не установите active false в линии 2.

# Конструктор

Конструктор - это специальный метод, который вызывается при создании нового объекта. Не всегда удобно инициализировать все переменные класса при создании его экземпляра. Иногда проще, чтобы какие-то значения были бы созданы по умолчанию при создании объекта. По сути конструктор нужен для автоматической инициализации переменных.

Конструктор инициализирует объект непосредственно во время создания. Имя конструктора совпадает с именем класса, включая регистр, а по синтаксису конструктор похож на метод без возвращаемого значения.

В отличие от метода, конструктор никогда ничего не возвращает.

Конструктор определяет действия, выполняемые при создании объекта класса, и является важной частью класса. Как правило, программисты стараются явно указать конструктор. Если явного конструктора нет, то Java автоматически создаст его для использования по умолчанию.

Имея дело с перегруженными конструкторами, удобно один конструктор вызывать из другого через ключевое слово **this**. При выполнении конструктора **this()** сначала выполняется перегруженный конструктор, который соответствует списку параметров. Затем выполняются операторы, находящиеся внутри исходного конструктора, если таковые существуют. Вызов конструктора **this()** должен быть первым оператором в конструкторе.

Иногда класс создаётся только для хранения каких-то статических полей и статических методов. Таким классам принято давать имена **Utils**, но это не обязательно. Такому классу не нужен конструктор, но если автор класса его не создал, то система сама создаст конструктор по умолчанию. Такой конструктор не имеет смысла, а также может послужить источником ошибок. Чтобы предохраниться от подобной проблемы вы сами явно должны создать пустрой конструктор и сделать его закрытым.