# Лекция 11. Коллекции в Java

Для хранения большого количества однотипных данных могут использоваться массивы, но они не всегда являются идеальным решением. **Во-первых**, длина массива задается заранее и в случае, если количество элементов заранее неизвестно, придется либо выделять память «с запасом», либо предпринимать сложные действия по переопределению массива.   
**Во-вторых**, элементы массива имеют жестко заданное размещение в его ячейках, поэтому, например, удаление элемента из массива не является простой операцией.

В программировании давно и эффективно используются такие структуры данных как стек, очередь, список, множество и т.д., объединенные общим названием коллекция.

***Коллекция*** — это группа элементов с операциями добавления, извлечения и поиска элемента. Механизм работы операций существенно различается в зависимости от типа коллекции. Например, элементы ***стека*** упорядочены в последовательность, добавление нового элемента может происходить только в конец этой последовательности, и получить можно только элемент, находящийся в конце (то есть, добавленный последним). ***Очередь*** позволяет получить лишь первый элемент (элементы добавляются в один конец последовательности, а «забираются» с другого). Другие коллекции (например, ***список***) позволяют получить элемент из любого места последовательности, а ***множество*** вообще не упорядочивает элементы и позволяет (помимо добавления и удаления) только узнать, содержится ли в нем данный элемент.

Java предоставляет библиотеку стандартных коллекций, ***collection framework***, которые собраны в пакете ***java.util***.

***Java Collection Framework*** — иерархия интерфейсов и их реализаций, которая является частью JDK и предоставляет разработчику набор средств для обработки массивов данных (см. рис. 1).

|  |
| --- |
| image |
| Рис. 1. Иерархия интерфесов в Java Collection Framework |

Чтобы выбрать коллекцию, которая лучше всего подходит условию задачи, необходимо знать особенности каждой из них. Каждый класс в Java Collection Framework реализует некоторую коллекцию со специфичным для нее набором операций доступа к элементам. Чтобы использовать коллекцию в своей программе, нужно создать объект соответствующего класса.

Элементы коллекций могут быть любого ссылочного типа. Однако, реализация коллекций с использованием generics позволяет указывать тип ссылок, которые будут содержаться в ней. Generics-тип коллекции указывается в специальных скобочках (« <Е> », где E – описание типа элементов). Свершенно не важно, какой именно будет указанная заглавная бука в скобках, это просто идентификатор типа. Так, на самом деле, все коллекции типизируются через generics и описываются по следующим правилам:

|  |
| --- |
| 1. Запись вида Collection<E> говорит о том, что объект типа Collection содержит и работает с элементами типа E. Если эту коллекцию типизировать реальным типом, например, String, запись примет следующий вид Collection<String> и положить в такую коллекцию объекты любого другого типа будет нельзя и попытка приведет к ошибке компиляции. |
| 1. В generics работает правило наследования: так если коллекция типизирована, как Object или любой другой супертип, то в нее можно положить объекты любого типа, которые являются наследниками указанного супертипа. |

*(!) Использование Object в качестве типа для элементов коллекции может привести к трудностям: при совершении каких-то операций над элементом из коллекции (в коллекцию объекты именно для того помещаются, чтобы потом быть извлеченными и обработанными), будет невозможно воспользоваться его методами, не приведя объект к его «настоящему» типу посредством явного приведения типов.*

Коллекция, хранящая элементы типа ***Object***, сразу же «забывает» реальный тип объекта, поскольку при его добавлении было осуществлено автоматическое приведение его типа к ***Object***.

На вершине иерархии Java Collection Framework располагаются интерфейсы ***Collection*** и ***Map***. Эти интерфейсы разделяют все коллекции на две части по принципу хранения данных: простые наборы элементов наследуются от Collection и представляют собой одномерную последовательность элементов; наборы пар «ключ — значение» наследуются от Map и представляют собой одномерную последовательность пар «ключ-значение» .

# [Collection](http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)

[***Collection***](http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)определяет основные методы работы с простыми наборами элементов, которые будут общими для всех его реализаций (см. табл. 1).

|  |  |
| --- | --- |
| Таблица 1. Методы интерфейса Collection<E> | |
| **Метод** | **Значение** |
| boolean add(E item) | добавляет в коллекцию объект item типа E; при удачном добавлении возвращает true, при неудачном – false |
| void clear() | удаляет все элементы из коллекции |
| boolean contains(Object item) | возвращает true, если объект item содержится в коллекции, иначе возвращает false |
| boolean isEmpty() | возвращает true, если коллекция пуста, иначе возвращает false |
| boolean remove(Object item) | возвращает true, если объект item удачно удален из коллекции, иначе возвращается false |
| int size() | возвращает количество элементов в коллекции |
| Object[] toArray() | возвращает массив, содержащий все элементы коллекции |
| Iterator<E> iterator() | возвращает итератор коллекции |

# List

Интерфейс List наследует интерфейс Collection и представляет собой упорядоченную коллекцию, доступ к элементам которой осуществляется по индексу и по значению. В данной коллекции допустимо хранение дубликатов (результатом поиска по значению будет первое найденное вхождение).

|  |
| --- |
| https://habrastorage.org/files/187/da1/649/187da164972c4519b6affbc4a2c6fda1.png |
| Рис. 2. Иерархия наследования интерфейса List |

# [Vector](http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html)

Реализация динамического списка объектов. Позволяет хранить любые данные, включая null в качестве элемента. В настоящее время, использование этой коллекции не рекомендуется, так как все ее методы являются синхронизированными, их быстродействие невелико. Поэтому, если не требуется достижения потокобезопасности, использования этой коллекции лучше избежать. В качестве несинхронизированной альтернативы применяется ArrayList.

|  |
| --- |
| **public** **class** Main  {  **public** **static** **void** main(String... strings)  {  Vector<String> vector=**new** Vector<String>();  vector.add("s1");  vector.add("s2");  vector.add("s3");  System.***out***.println("[0]=" + vector.get(**0**));  System.***out***.println("[1]=" + vector.get(**1**));  System.***out***.println("[2]=" + vector.get(**2**));  }  } |
|  |
| Пример 1. Использование коллекции Vector |

# [Stack](http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html)

Stack наследует Vector и реализован по принципу LIFO (last-in-first-out). Является частично синхронизированной коллекцией (кроме метода добавления push()). Однако, аналогично Vector, коллекция считается устаревшей и к использованию не рекомендуется. Вместо нее, предлагается коллекция-очередь Deque.

|  |
| --- |
| **public** **class** Main  {  **public** **static** **void** main(String... strings)  {  Stack<Integer> stack=**new** Stack<Integer>();  stack.push(**100**);  stack.push(**100**);  stack.push(**200**);    System.***out***.println("stack.size = " + stack.size());    System.***out***.println("first pop: " + stack.pop());  System.***out***.println("stack.size = " + stack.size());    System.***out***.println("second pop: " + stack.pop());  System.***out***.println("stack.size = " + stack.size());    System.***out***.println("third pop: " + stack.pop());  System.***out***.println("fourth pop: " + stack.pop());  System.***out***.println("stack.size = " + stack.size());  }  } |
|  |
| Пример 2. Использование коллекции Stack |

# [ArrayList](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

Однонаправленный список является реализацией динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элементов. Используется при частом обращении к элементам по индексу. Из-за особенностей реализации поиндексное обращение к элементам выполняется за константное время O(1).

|  |
| --- |
| **public** **class** Main  {  **public** **static** **void** main(String... strings)  {  ArrayList<Integer> aList=**new** ArrayList<Integer>();  **long** startTime=System.*currentTimeMillis*();  // add 10 000 elements to the tail  **for** (**int** i=**0**;i < **10000**;i++)  aList.add(i);  **long** finishTime=System.*currentTimeMillis*();  System.***out***.println("addition of 10000 elements to the tail took " + (finishTime - startTime) + " ms");  // add 1000 elements in the middle  startTime=System.*currentTimeMillis*();  **for** (**int** i=**0**;i < **10000**;i++)  {  **int** middleIndex=(**int**) (aList.size() / **2**);  aList.add(middleIndex, **new** Integer(i));  }  finishTime=System.*currentTimeMillis*();  System.***out***.println("addition of 10000 elements in the middle took " + (finishTime - startTime) + " ms");    // add 1000 elements to the head  startTime=System.*currentTimeMillis*();  **for** (**int** i=**0**;i < **10000**;i++)  {  aList.add(**0**, **new** Integer(i));  }  finishTime=System.*currentTimeMillis*();  System.***out***.println("addition of 10000 elements to the head took " + (finishTime - startTime) + " ms");  }  } |
| **Результат:**  addition of 10000 elements to the tail took 3 ms  addition of 10000 elements in the middle took 49 ms  addition of 10000 elements to the head took 51 ms |
| Пример 3. Использование коллекции ArrayList |

# [LinkedList](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html)

Еще один наследник List, однако, его особенностью является то, что данный список является двунаправленным, т.е. каждый элемент имеет ссылку на предыдущий и следующий элемент. Список хранит любые объекты, включая null. Благодаря тому, что список двунаправленный, добавление и удаление элементов из середины при обращении по индексу происходит за линейное время O(n), а из начала и конца за константное O(1). Так же, ввиду реализации, данную коллекцию можно использовать как стек или очередь. Для этого в ней реализованы соответсвующие методы.

Модифицируем пример 3, а именно заменим ArrayList на LinkedList. Как видно из результатов запуска кода, время на добавление элементов в середину списка занимает больше времени, а время на добавление в «голову» и «хвост» списка занимает приблизительно одинаковое время, тогда как в ArrayList добавление в середину и в «голову» занимает приблизительно одинаковое время, а запись в «хвост» происходит довольно быстро.

|  |
| --- |
| **public** **class** Main  {  **public** **static** **void** main(String... strings)  {  LinkedList<Integer> aList=**new** LinkedList<Integer>();  **long** startTime=System.*currentTimeMillis*();  // add 10 000 elements to the tail  **for** (**int** i=**0**;i < **10000**;i++)  aList.add(i);  **long** finishTime=System.*currentTimeMillis*();  System.***out***.println("addition of 10000 elements to the tail took " + (finishTime - startTime) + " ms");  // add 1000 elements in the middle  startTime=System.*currentTimeMillis*();  **for** (**int** i=**0**;i < **10000**;i++)  {  **int** middleIndex=(**int**) (aList.size() / **2**);  aList.add(middleIndex, **new** Integer(i));  }  finishTime=System.*currentTimeMillis*();  System.***out***.println("addition of 10000 elements in the middle took " + (finishTime - startTime) + " ms");    // add 1000 elements to the head  startTime=System.*currentTimeMillis*();  **for** (**int** i=**0**;i < **10000**;i++)  {  aList.add(**0**, **new** Integer(i));  }  finishTime=System.*currentTimeMillis*();  System.***out***.println("addition of 10000 elements in the middle took " + (finishTime - startTime) + " ms");  }  } |
| **Результат:**  addition of 10000 elements to the tail took 4 ms  addition of 10000 elements in the middle took 165 ms  addition of 10000 elements to the head took 2 ms |
| Пример 4. Использование коллекции LinkedList |

# Set

|  |
| --- |
| https://habrastorage.org/files/aca/208/428/aca20842816a48628772bd23d2bb0f24.png |
| Рис. 3. Иерархия наследования интерфейса Set |

Set (множество) представляет собой неупорядоченную коллекцию, в которой не допускается наличие дублирующихся элементов. Собственно, полностью соответствует математическому понятию «множества без повтороящихся элементов».

# [HashSet](http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html)

Реализация интерфейса Set, базирующаяся на HashMap. Внутри использует объект HashMap для хранения данных. В качестве ключа используется добавляемый элемент, а в качестве значения — объект-пустышка (new Object()). Из-за особенностей реализации порядок элементов не гарантируется при добавлении.

|  |
| --- |
| **public** **class** MainSet  {  **public** **static** **void** main(String... strings)  {  Set<Date> setOfDate=**new** HashSet<>();  Date firstDate=**new** Date();  setOfDate.add(firstDate);  System.***out***.println("setOfDate.size = " + setOfDate.size());  **for** (**int** i=**0**;i < **10000**;i++)  {  Date date=**new** Date();  setOfDate.add(date);  }  System.***out***.println("setOfDate.size = " + setOfDate.size());  setOfDate.add(firstDate);  System.***out***.println("setOfDate.size = " + setOfDate.size());    setOfDate.add(**null**);  System.***out***.println("setOfDate.size = " + setOfDate.size());  System.***out***.println("ORDER:");  Iterator<Date> iterator=setOfDate.iterator();  **while** (iterator.hasNext())  {  Date next=iterator.next();  System.***out***.println(next == **null** ? **null** : next.getTime());  }  }  }  Вывод коллекции на экран:  ORDER:  null  1456999614284  1456999614285  1456999614282  1456999614283  1456999614280  1456999614281  1456999614278  1456999614279 |
| Пример 5. Использование коллекции HashSet |

Как видно из примера, лементы коллекции находятся в произвольном порядке.

# [LinkedHashSet](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html)

Отличается от HashSet только тем, что в основе лежит LinkedHashMap вместо HashSet. Благодаря этому отличию порядок элементов при обходе коллекции является идентичным порядку добавления элементов.

|  |
| --- |
| При замене HashSet на LinkedHashSet результаты в примере 5 будут упорядоченными:  Вывод коллекции на экран:  ORDER:  1457004551376  1457004551377  1457004551378  1457004551379  1457004551380  1457004551381  1457004551382  1457004551383  1457004551384  1457004551385  null |
| Пример 5. Использование коллекции HashSet |

# [TreeSet](http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html)

Как и другие реализации интерфейса Set, основан на реализации интерфейса Map, а именно NavigableMap, что и обуславливает его поведение: предоставляет возможность управлять порядком элементов в коллекции при помощи Comparator-а (который представлет собой закон упорядочивания элементов во множестве). По умолчанию используется Comparator, реализующий алгоритм «[natural ordering](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)».

# Queue

|  |
| --- |
| https://habrastorage.org/files/3ee/116/dc0/3ee116dc0d6e4775a72187f551001869.png |
| Рис. 4. Иерархия наследования интерфейса Queue |

Интерфейс, описывающий коллекции, как очереди, предполагает реализацию коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out). Помимо методов, определённых в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь. Большинство реализаций данного интерфейса находится в пакете java.util.concurren.

# [PriorityQueue](http://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html)

Является единственной прямой реализацией интерфейса Queue. Особенностью данной очереди является возможность управления порядком элементов. По умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи использования необходимого Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.

# [ArrayDeque](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html)

Реализация интерфейса [Deque](http://docs.oracle.com/javase/8/docs/api/java/util/Deque.html), который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Эта коллекция представляет собой реализацию с использованием массивов, подобноArrayList, но не позволяет обращаться к элементам по индексу и хранение null. Как заявлено в документации, коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO.

# Интерфейс Map

Интерфейс [Map](http://docs.oracle.com/javase/8/docs/api/java/util/Map.html) предоставляет разработчику базовые методы для работы с данными вида «ключ — значение».Также как и Collection, он был дополнен дженериками

|  |
| --- |
| https://habrastorage.org/files/40a/eca/09a/40aeca09ac1c4cc7bdbd475a3c12fd95.png |
| Рис. 5. Иерархия наследования интерфейса Map |

# [Hashtable](http://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html)

Реализация такой структуры данных, как хэш-таблица. Она не позволяет использовать null в качестве значения или ключа. Hashtable является синхронизированной. Из-за этой особенности у неё имеются существенные проблемы с производительностью и, в большинстве случаев, рекомендуется использовать другие реализации интерфейса Map ввиду отсутствия у них синхронизации.

# [HashMap](http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)

Альтернатива Hashtable, двумя основными отличиями от Hashtable которой являются то, что HashMap не синхронизирована и позволяет использовать null как в качестве ключа и значения. Так же как и Hashtable, данная коллекция не является упорядоченной: порядок хранения элементов зависит от хэш-функции. Добавление элемента выполняется за константное время O(1), но время удаления, получения зависит от распределения хэш-функции.

|  |
| --- |
| public class MainMap  {  public static void main(String[] args)  {  Map<String, Integer> map=new LinkedHashMap<String, Integer>();  map.put("1", 1);  map.put("1", 1);  map.put("2", 2);  map.put("3", 1);  map.put("4", 1);  for (String key:map.keySet())  {  System.out.println("Key=" + key + ", value=" + map.get(key));  }  for (Map.Entry<String, Integer> entry:map.entrySet())  {  System.out.println("Key=" + entry.getKey() + ", value=" + entry.getValue());  }  }  } |
| Пример 6. Использование коллекции LinkedHashMap |

# [LinkedHashMap](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html)

Упорядоченная реализация хэш-таблицы. В отличие от HashMap, порядок итерирования равен порядку добавления элементов. Данная особенность достигается благодаря двунаправленным связям между элементами. Но это преимущество имеет также и недостаток — увеличение памяти, которое занимет коллекция.

# [TreeMap](http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html)

Реализация Map, основанная на бинарных деревьях. Как и LinkedHashMap является упорядоченной. По умолчанию, коллекция сортируется по ключам с использованием принципа «[natural ordering](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)», но это поведение может быть изменено путем замены Comparator-а коллекции, которые указывается в качестве параметра при создании объекта TreeMap.

# [WeakHashMap](http://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html)

Реализация хэш-таблицы, которая организована с использованием [weak references](http://docs.oracle.com/javase/8/docs/api/java/lang/ref/WeakReference.html) (слабых ссылок). Другими словами, Garbage Collector автоматически удалит элемент из коллекции при следующей сборке мусора.

**САМОСТОЯТЕЛЬНО:**

**Создайте эмуляцию картотеки, в которой содержится следующая информация:**

1. **Пользователи библиотеки с информацией о книгах, которые взяты ими.**
2. **Все книги библиотеки, упорядоченные по имени автора.**
3. **Информации о свободных книгах.**

**При этом:**

1. **У каждого пользователя может быть несколько книг, но только по одному экземляру конкрутного издания.**
2. **На какую-либо книгу может быть очередь из пользователей, ожидающих ее освобождения.**
3. **Экземлятор одной книги может быть несколько.**