Конструирование программного обеспечения

Лекция 10.
Контейнеры и коллекции объектов

к.т.н. Гринкруг Е.М. (email: egrinkrug@hse.ru)

Структуры данных и коллекции

- При программировании выбор используемых структур данных влияет на качество результата:
  - на естественность реализации;
  - на эффективность использования памяти и производительность.
- Это особенно важно при обработке больших объемов данных (“Big Data”…).
- В JDK с самого начала присутствовали стандартные реализации наиболее употребительных структур данных, причем **с учетом возможности их параллельной обработки**. В современном JDK эти возможности существенно расширены и согласованы с новыми средствами обработки потоков данных.
- У вас параллельно проходит курс «Алгоритмы и структуры данных» с практикой их использования в языке С++.
  - Иногда такой курс базируется целиком на Java (см. Moodle -> Readings): Data Structures & Algorithms in Java (до и после использования generics; эти средства совершенствуются в JDK (+)).
- Полезно сопоставить возможности С++ и Java по обработке структур данных.
- Мы начнем (и позже - продолжим) обзор соответствующих средств JDK...

Контейнеры и их назначение

- Часто требуется создавать, хранить и обрабатывать наборы объектов, в том числе таких, количество и типы которых статически не известны (*что это означает?*).
- В Java имеются различные способы хранения ссылок на объекты (*укажите уже известные способы... «Самый простой» контейнер – это что?*).
- **В пакете java.util имеется библиотека утилит** (*что это?*)**, которая содержит набор *классов контейнеров* (или *классов коллекций*), предназначенных для решения всевозможных задач, требующих организации работы с различными множествами объектов.**
- До Java SE 5 все такие классы оперировали с любыми объектами, ссылаясь на объекты в терминах **java.lang.Object**; начиная с JDK5 компилятор позволяет работать с параметризованными типами, собственно добавление которых в язык Java было в значительной степени стимулировано желанием контролировать на стадии компиляции (в статике) правильность работы с коллекциями (но не только это...).

Массивы – тоже хранилища наборов объектов

- Встроенные в язык Java массивы позволяют хранить и контролировать типизированные наборы объектов:

```
MyType[ ] myObjects = new MyType[ how_many];
…
myObjects[ index] = someObject;
…
MyType sample = myObjects[ indexOfElement];
```

- Какие тут достоинства и недостатки?
  - **Элементом массива может быть только объект нужного типа (или null)**
  - **Размер объекта-массива фиксирован при его создании**
  - **Обращение к элементам массива только по контролируемым индексам**

```java
package com.grinkrug.arrays;
import java.lang.reflect.Array;

public class ArrayTest {
    static Object[] myObjects = (Object[])Array.newInstance(A.class, 5);
    public static void main(String[] args){
        myObjects[3] = new B();
    }
}

class A{}
class B{}
```

- Контроль типов элементов массива происходит и в статике и в динамике:
  - Если компилятор может определить несоответствие типов, программа не скомпилируется;
  - Если компилятор не может (см. выше): java.lang.ArrayStoreException

The Java Collections Framework

- **Коллекция (Collection)** позволяет рассматривать группу объектов как единое целое. При этом объекты могут сохраняться, извлекаться и обрабатываться как элементы коллекции. Массивы – это примеры коллекций.
- **The Java Collections Framework** – предоставляет набор стандартных классов для работы с различными коллекциями. Они содержатся в пакете **java.util** и делятся на три основные части:
  - Основные **интерфейсы**, позволяющие манипулировать коллекциями вне зависимости от их реализаций; эти родовые (generic) интерфейсы определяют общую функциональность и обмен данными между коллекциями;
  - Специфические **реализации этих интерфейсов**, предоставляющие структуры данных, которые можно непосредственно использовать в программах;
  - **Набор статических методов** в утилитных классах Collections и Arrays с различными полезными операциями над коллекциями и массивами.

Подход к разработке Collection Framework

- Изначально определен в JDK 1.2 (до того – несколько полезных классов).
- Цели разработки библиотеки коллекций:
  - она должна была быть сравнительно небольшая и легко используемая;
  - без сложностей STL в C++, но с возможностями «переиспользования» с разными типами данных, как в STL;
  - поддерживать обратную совместимость (для ранее имевшихся классов);
  - поддерживать расширяемость (в том числе – пользовательскими средствами).
- Библиотека содержит ряд своеобразных проектных решений, определяющих ее архитектуру и пути развития.
- **Framework** – или каркас (рус.) - это:
  - Программная основа (платформа), определеяющая структуру программной системы, облегчающая разработку и объединение компонентов большого программного проекта;
  - Подход к построению программ, где выделяются две основные части:
    - Постоянный каркас, определяющий общую архитектуру, и
    - Переменная часть, состоящая из сменных модулей (точек расширения).

Основные интерфейсы

«interface»
java.lang.Iterable<E>

«interface»
Collection<E>

«interface»
List<E>

«interface»
Queue<E>

«interface»
Set<E>

«interface»
Deque<E>

«interface»
SortedSet<E>

«interface»
NavigableSet<E>

(a)

«interface»
Map<K,V>

«interface»
SortedMap<K,V>

«interface»
NavigableMap<K,V>

(b)

interface	Описанние	Конкретные классы
Collection<E>	Базовый интерфейс с операциями для работы с коллекциями	
Set<E>	Множество уникальных элементов	HashSet<E> LinkedHashSet<E>
SortedSet<E>	Множество с сортированными элементами	TreeSet<E>
NavigableSet<E>	Наследует и заменяет SortedSet<E> (предпочтителен в новом коде)	TreeSet<E>
List<E>	Обеспечивает последовательность (возможно - одинаковых) элементов	ArrayList<E> Vector<E> LinkedList<E>
Queue<E>	Коллекция элементов, обеспечивающая их очередность	PriorityQueue<E> LinkedList<E>
Deque<E>	Наследует очередь с обработкой элементов с двух концов	ArrayDeque<E> LinkedList<E>

interface	Описанние	Конкретные классы
Map<K,V>	Базовый интерфейс отображения ключей на значения	HashMap<K,V> Hashtable<K,V> LinkedHashMap<K,V>
SortedMap<K,V>	Отображения отсортированы по ключам	TreeMap<K,V>
NavigableMap<K,V>	Наследует и заменяет SortedMap	TreeMap<K,V>

- Map не наследует Collection; концептуально отображение не есть коллекция – в нем нет элементов, но есть отображения (entries – записи, вхождения); ключ может быть связан только с одним значением и должен быть уникален.
- В новом коде следует использовать NavigableMap.

Реализации

- Пакет java.util содержит реализации абстрактных типов данных, основанных на основных интерфейсах. Эти реализации не реализуют интерфейс Collection непосредственно: для реализаций используются абстрактные базовые классы.
- По соглашению, каждый класс реализации предоставляет конструктор для создания коллекции из элементов другого объекта-коллекции, переданного как параметр. Это позволяет заменять реализацию коллекции путем создания ее из тех же элементов. Такая замена работает и для реализаций Map (отображения). Но коллекции и отображения не являются взаимно заменяемыми.
- Коллекции и отображения хранят ссылки на объекты, а не сами объекты.
- Collections Framework – основана на интерфейсах (interface based): обработка производится в соответствии с типами-интерфейсами, а не реализациями; разные реализацмм являются взаимозаменяемыми.
- Все конкретные классы реализаций являются Serializable и Cloneable.

Интерфейсы и реализации Collection<E>

«interface» java.lang.Iterable<E>

«interface» Iterator<E>

«interface» Collection<E>

«interface» ListIterator<E>

«interface» List<E>

«interface» Set<E>

«interface» Deque<E>

«interface» Queue<E>

«interface» SortedSet<E>

«interface» NavigableSet<E>

Interfaces

Concrete Classes

Vector<E>

ArrayList<E>

LinkedList<E>

PriorityQueue<E>

HashSet<E>

TreeSet<E>

ArrayDeque<E>

LinkedHashSet<E>

Интерфейсы и реализации Map<K, V>

Interfaces

«interface» Map.Entry<K,V>

«interface» Map<K,V>

«interface» SortedMap<K,V>

«interface» NavigableMap<K,V>

Concrete Classes

HashMap<K,V>

TreeMap<K,V>

Hashtable<K,V>

LinkedHashMap<K,V>

Реализация	Интерфейс	Элементы	Порядок	Исп. методы элементов	Структуры данных
HashSet<E>	Set<E>	уникальны	Нет	equals() hashCode()	Hash table, Linked List
LinkedHash- Set<E>	Set<E>	уникальны	Порядок вставки	equals() hashCode()	Hash table, doubly-linked list
TreeSet<E>	SortedSet<E> NavigableSet<E>	уникальны	Сортирова н	compareTo()	Balanced tree
ArrayList<E>	List<E>	дублируемы	Порядок вставки	equals()	Resizable array
LinkedList<E>	List<E> Queue<E> Deque<E>	дублируемы	Вставка/ приоритет/ очередь	equals() compareTo()	Linked list
Vector<E>	List<E>	дублируемы	Порядок вставки	equals()	Resizable array
Priority- Queue<E>	Queue<E>	дублируемы	Приоритет	compareTo()	Priority heap (tree-like struct)
ArrayDeque<E>	Queue<E> Deque<E>	дублируемы	FIFO / LIFO	equals()	Resizable array

Реализация	Интерфейс	Элементы	Порядок	Исп. методы элементов	Структуры данных
HashMap<K,V>	**Map<K,V>**	**Уникальные ключи**	**нет**	**equals() hashCode()**	**Hash table и array**
LinkedHash-Map<K,V>	**Map<K,V>**	**Уникальные ключи**	**Key insertion order/ Access order of entries**	**equals() hashCode()**	**Hash table и doubly-linked list**
Hashtable< K,V>	**Map<K,V>**	**Уникальные ключи**	**нет**	**equals() hashCode()**	**Hash table**
TreeMap<K,V>	**SortedMap<K,V> NavigableMap<K,V>**	**Уникальные ключи**	**Sorted in key order**	**equals() compareTo()**	**Balanced tree**

- Все перечисленные выше реализации, *кроме Vector и Hashtable*, не являются thread-safe (*что это?*);
- Имеются возможности получить thread-safe реализации.

Основные концепции библиотеки

- Концепция ООП – инкапсуляция данных *(что это?)*, однако способы структуризации наборов данных для разных применений тоже важны.
- Различают интерфейс структуры данных и его реализации, которые могут быть разными
  - **При использовании структуры данных в программе после создания структуры данных не обязательно знать, какая реализация применяется; поэтому указывать конкретный класс реализации следует только при конструировании объекта, реализующего набор данных, а все ссылки определять как интерфейсные.**
- Интерфейс не дает ответа на вопрос об эффективности реализации
  - **Имеются разные реализации одного интерфейса, отличающиеся своими достоинствами и недостатками; их следует знать для правильного выбора реализации при решении конкретных задач.**
- Впрочем, такой подход годится не всегда, так как иногда классы реализации интерфейса содержат дополнительную функциональность, не предусмотренную в интерфейсе
  - **Если требуется использовать такие дополнительные методы, производить восходящее преобразование к более общему типу интерфейса не получится.**

Коллекции и отображения (Collections and Maps)

- **Коллекция (Collection)**: группа отдельных элементов, сформированная по некоторым правилам:
  - **List** (список) хранит элементы в соответствии с тем, как они заносились в список;
  - В **Set** (множество) нельзя хранить повторяющиеся элементы;
  - **Queue** (очередь) выдает элементы в порядке, определяемом спецификой очереди (*какие могут быть варианты?*)
- **Отображение (Map,** иногда плохо переводят -“карта”**)**: группа пар объектов key-value (ключ - значение), позволяющая искать значение по ключу.
  - Массив тоже позволяет искать объект значение – с помощью числа, – ассоциируя числа с объектами.
  - Map ищет один объект с помощью другого объекта. Иногда это называют **ассоциативным массивом** или **словарем (dictionary)**, так как объект-значение ищется по объекту-ключу подобно поиску значения в словаре *(а словари бывают разные...)*

Замечание о параметризации типов

- Generic types (родовые и параметризованные типы) появились в Java 5 для того, чтобы
  - можно было писать более универсальный код,
  - на этапе компиляции (в статике!) автоматически выполнялись некоторые преобразования типов и
  - выявлялись ошибки, связанные с типизацией.

Все основные интерфейсы коллекций являются *родовыми (generic):*

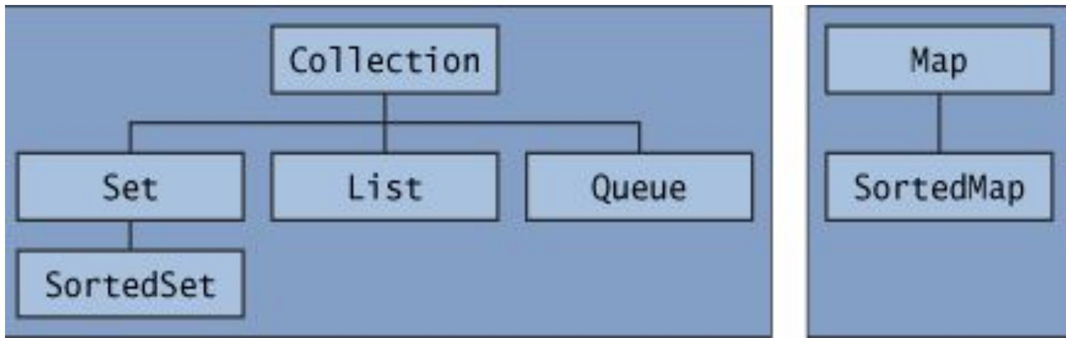
public interface Collection<E>…

Синтаксис <E> показывавет, что интерфейс – generic.
Когда декларируется экземпляр Collection, указывается тип объектов, содержащихся в коллекции.
Спецификации типа позволяет компилятору («рано», до исполнения) проверить правильность типа объектов, помещаемых в коллекцию.

Достоинства Java Collections Framework

- Уменьшение усилий при программировании:
  - Освобождает от реализации деталей для концентрации на основном.
- Повышается скорость и качество программы:
  - Дает качественные алгоритмы и их реализации.
- Облегчается функциональная совместимость различных API:
  - Интерфейсы коллекций позволяют сопрягать разнородные, независимо разработанные API.
- Уменьшаются усилия при изучении и использовании новых API:
  - Многие API используют интерфейсы коллекций для спецификации входных и выходных данных
- Упрощается разработка новых API:
  - Нет необходимости «изобретать велосипед».
- Стимулируется переиспользование ПО:
  - Соответствующие коллекциям структуры даннных естественно переиспользуются (т.е. - используются повторно, многократно...).

Основные интерфейсы (с JDK1.6 + NavigableSet/Map)



Иерархия интерфейсов состоит из двух **отдельных** деревьев:
- Collection
- Map (**не есть Collection**)

- Для сокращения объема библиотеки разные варианты (immutable-варианты, варианты с фиксированной длиной, append-only, и т.п.) для коллекций не предоставляются (они м.б. добавлены wrapper'ами, в новых JDK, и т.п.).
- Вместо этого, модифицирующие операции в каждом интерфейсе помечаются как optional (необязательные) или – в новых JDK(8+) – как default:
  - **Конкретная реализация может не поддерживать все операции;**
  - **При вызове операции, которая не поддерживается, коллекция выкидывает UnsupportedOperationException**
  - **Реализации обязаны документировать, какие опциональные операции есть;**
  - **Реализации из Java API (из java.util), обычно подерживают все операции.**

- **Collection** – группа элементов – самый общий интерфейс коллекций:
  - Некоторые типы коллекций могут иметь дублирующиеся элементы;
  - Некоторые могут быть упорядочены.
- **Set** – коллекция, которая не может иметь повторяющихся элементов:
  - Моделирует математическое понятие множества.
- **List** – упорядоченная коллекция (иногда – последовательность, sequence):
  - Может иметь повторяющиеся элементы;
  - Пользователь обычно полностью контролирует, куда вставляется элемент, и может обращаться к элементам по интексу;
- **Queue** – коллекция, упорядочивающая элементы перед их обработкой:
  - Всякая реализация обязана специфицировать это упорядочение.
- **Map** – объект, отображающий ключи в значения:
  - Не может содержать повторяющихся ключей,
  - Каждый ключ отображается не более чем на одно значение.

Интерфейс Collection

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Все реализации коллекций имеют конструктор с параметром Collection.

Это позволяет легко делать из одной коллекции другую.

Поэтому, такой конструктор называют *conversion constructor*.

Все коллекции имеют generic-методы, частично реализованные в их абстрактном суперклассе **AbstractCollection**

Ряд default – методов добавлен для обработки потоков данных (с JDK8)…

Проход по коллекциям

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

```
for (Object o : collection)
    System.out.println(o);
```

- Используйте итератор, а не for-each, когда нужно:
  - Убрать текущий элемент;
  - Проходить сразу по нескольким коллекциям.
- Метод *remove* можно вызвать только раз для одного вызова *next*, и должно быть Exception при нарушении этого правила.
- В JDK8 сделаны усовершенствования...

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

Итератор в JDK 8 (+)

- Интерфейс Iterator E> имеет (в JDK 8(+)) четыре метода (два из них - default):

```
boolean hasNext();
E next();
default void remove() {throw new UnsupportedOperationException("remove");}
default void forEachRemaining(Consumer<? super E> action) {…}
```

  - **Повторно вызывая next(), можно последовательно перебирать элементы, а при выходе за конец коллекции – получить NoSuchElementException, если не использовать hasNext().**
- Если с – это коллекция (т.е. Iterable<E>), можно использовать цикл "for each", например - для `Collection<String> c` :

```
for (String element : c) { /* do something with element */ }
```

(компилятор транслирует это в цикл с итератором (можно посмотреть – как...))
- С JDK8, все это упрощает метод forEachRemaning(…) с лямбда-выражением.
- **Последовательность обработки элементов зависит от типа коллекции.**
- **Имеется зависимость между вызовами next() и remove(): нельзя вызвать remove() без предварительного вызова next() – будет IllegalStateException.**
  - Метод next() выдает очередной элемент и «устанавливается на следующий»...

Интерфейс Set

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

- Те же методы, что в Collection;
- Нет одинаковых элементов;
- Два множества равны, если они содержат одинаковые элементы.
- Платформа Java имеет три реализации:
  - **HashSet** (быстрая, но не гарантирует одинаковой последовательности итерации)
  - **TreeSet** (заметно медленнее, упорядочивает элементы по их значениям)
  - **LinkedHashSet** (упорядочение в порядке вставления элементов)

Интерфейс List

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

- Упорядоченная коллекция (последовательность)
- Может содержать повторяющиеся элементы
- Есть две реализации списков:
  - ArrayList
  - LinkedList

  и переделанная реализация
  - Vector

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

Интерфейс Queue

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Queue Interface Structure

	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

- Коллекция элементов, которая накапливается для их последующей обработки;
- Обычно не допускается null-элемент (LinkedList – это исключение);
- Имеются расширения для параллельной работы;

Интерфейс Map

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

- Моделирует математическое понятие *функции*;
- Не содержит дублирующихся ключей;
- Каждый ключ отображается не более чем на одно значение;
- Реализации:
  - **HashMap;**
  - **TreeMap;**
  - **LinkedHashMap;**
  - **Hashtable** (переделанная старая реализация)

Упорядочение объектов

Classes Implementing Comparable

Class	Natural Ordering
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent lexicographic on path name
String	Lexicographic
Date	Chronological
CollationKey	Locale-specific lexicographic

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- Сортировка списка list: Collections.sort(list);
- Если объекты не могут сравниваться - ClassCastException
- Результат compareTo() число:
  - (< 0) : <this> меньше o;
  - (= 0) : <this> равен o;
  - (> 0) : <this> больше o.

Что делать, если объекты надо сортировать не в «естественном» порядке», или они не реализуют Comparable-интерфейс?

Collections.sort (list, myComparator);

где myComparator – объект, инкапсулирующий сравнение/упорядочение.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Пример (как отсортируются объекты таким компаратором?):

```
static final myComparator = new Comparator() {    // что за конструкция?
        public int compare (Object o1, Object o2) {
                return o1.hashCode() – o2.hashCode();                    }
}
Collection c = …// какой-то набор объектов...
List myList = new ArrayList ( c );
Collections.sort ( myList, myComparator );
```

Интерфейсы SortedSet и SortedMap

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

- Содержат отсортированные по возрастанию (при создании) элементы
- В SortedMap – сортировка происходит по ключам
- Позволяют «выкусывать» свои подмножества

Реализации и их особенности

- См. исходные тексты соответствующих классов – это важно!
- Классы (утилитные) Arrays и Collections содержат полезные методы, которые произаодят и/или потребляют коллекции...
- Можно получать unmodifiable views на коллекции...
- Можно получатть synchronized views на коллекции...

- Это – важная тема, которая требует изучения и практики...
- Мы будем часто пользоваться коллекциями и их дальнейшими усовершенствованиями (связанными с параллелизмом обработки данных)...

Приложение: Java Collections Framework Interview Questions

- A good understanding of Collections framework is required to understand and leverage many powerful features of Java technology.
- Java Collections framework is fundamental utility tools provided by Java that are heavily used in java programming on all types of programs including web based and desktop applications.
- Your knowledge of Java will be considered incomplete without a good working experience of collections API

What is Java Collections API?

- Java Collections framework API is a unified architecture for representing and manipulating collections. The API contains Interfaces, Implementations & Algorithm to help java programmer in programming. In nutshell, this API does 6 things at high level
  - Reduces programming efforts. - Increases program speed and quality.
  - Allows interoperability among unrelated APIs.
  - Reduces effort to learn and to use new APIs.
  - Reduces effort to design new APIs.
  - Encourages & Fosters software reuse.
- There are six collection java interfaces. The most basic interface is Collection. Three interfaces extend Collection: Set, List, and SortedSet. The other two collection interfaces, Map and SortedMap, do not extend Collection, as they represent mappings rather than true collections.

What is an Iterator?

- Some of the collection classes provide traversal of their contents via a java.util.Iterator interface.
- This interface allows you to walk through a collection of objects, operating on each object in turn.
- Remember when using Iterators that they contain a snapshot of the collection at the time the Iterator was obtained; generally it is not advisable to modify the collection itself while traversing an Iterator.

What is the difference between java.util.Iterator and java.util.ListIterator?

- Iterator : Enables you to traverse through a collection in the forward direction only, for obtaining or removing elements
- ListIterator : extends Iterator, and allows bidirectional traversal of list and also allows the modification of elements.

What is HashMap and Map?

- Map is Interface which is part of Java collections framework. This is to store Key Value pair, and HashMap is class that implements that using hashing technique.

Difference between HashMap and HashTable?

- Both Hashtable & HashMap provide key-value access to data. The Hashtable is one of the original collection classes in Java (also called as legacy classes). HashMap is part of the new Collections Framework, added with Java 2, v1.2. There are several differences as listed below:
  - The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls. (HashMap allows null values as key and value whereas Hashtable doesn’t allow nulls).
  - HashMap does not guarantee that the order of the map will remain constant over time. But one of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you can easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.
  - HashMap is non synchronized whereas Hashtable is synchronized.
  - Iterator in the HashMap is fail-fast while the enumerator for the Hashtable isn't. So this could be a design consideration.

What does synchronized means in Hashtable context?

- Synchronized means only one thread can modify a hash table at one point of time. Any thread before performing an update on a hashtable will have to acquire a lock on the object while others will wait for lock to be released.

What is fail-fast property?

At high level - Fail-fast is a property of a system with respect to its response to failures. A fail-fast system is designed to immediately report any failure or condition that is likely to lead to failure. Fail-fast systems are usually designed to stop normal operation rather than attempt to continue a possibly-flawed process. When a problem occurs, a fail-fast system fails immediately and visibly. Failing fast is a non-intuitive technique: "failing immediately and visibly" sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix. In Java, Fail-fast term can be related to context of iterators. If an iterator has been created on a collection object and some other thread tries to modify the collection object "structurally", a concurrent modification exception will be thrown. It is possible for other threads though to invoke "set" method since it doesn't modify the collection "structurally". However, if prior to calling "set", the collection has been modified structurally, "IllegalArgumentException" will be thrown.

Why doesn’t Collecion extend Cloneable and Serializable?

- Many Collection implementations (including all of the ones provided by the JDK) will have a public clone method, but it would be mistake to require it of all Collections.
- For example, what does it mean to clone a Collection that's backed by a terabyte SQL database? Similar arguments hold for serializable.
- If the client doesn't know the actual type of a Collection, it's much more flexible and less error prone to have the client decide what type of Collection is desired, create an empty Collection of this type, and use the addAll method to copy the elements of the original collection into the new one.

How we can make HashMap synchronized?

- HashMap can be synchronized by *Map m = Collections.synchronizedMap(hashMap);*

Where will you use Hashtable and where HashMap?

- There are multiple aspects to this decision:
  - The basic difference between a Hashtable and an HashMap is that, Hashtable is synchronized while HashMap is not. Thus whenever there is a possibility of multiple threads accessing the same instance, one should use Hashtable. While if not multiple threads are going to access the same instance then use HashMap. Non synchronized data structure will give better performance than the synchronized one.
  - If there is a possibility in future that there can be a scenario when you may require to retain the order of objects in the Collection with key-value pair then HashMap can be a good choice. As one of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you can easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable. Also if you have multiple thread accessing you HashMap then Collections.synchronizedMap() method can be leveraged.
  - Overall HashMap gives you more flexibility in terms of possible future changes.

Difference between Vector and ArrayList?

- Vector & ArrayList both classes are implemented using dynamically resizable arrays, providing fast random access and fast traversal. ArrayList and Vector class both implement the List interface. Both the classes are member of Java collection framework, therefore from an API perspective, these two classes are very similar. However, there are still some major differences between the two. Below are some key differences
  - Vector is a legacy class which has been retrofitted to implement the List interface since Java 2 platform v1.2
  - Vector is synchronized whereas ArrayList is not. Even though Vector class is synchronized, still when you want programs to run in multithreading environment using ArrayList with Collections.synchronizedList() is recommended over Vector.
  - ArrayList has no default size while vector has a default size of 10.
  - The Enumerations returned by Vector's elements method are not fail-fast. Whereas ArraayList does not have any method returning Enumerations.

What is the difference between Enumeration and Iterator?

- Enumeration and Iterator are the interface available in java.util package. The functionality of Enumeration interface is duplicated by the Iterator interface. New implementations should consider using Iterator in preference to Enumeration. Iterators differ from enumerations in following ways:
  - Enumeration contains 2 methods namely hasMoreElements() & nextElement() whereas Iterator contains three methods namely hasNext(), next(), remove().
  - Iterator adds an optional remove operation, and has shorter method names. Using remove() we can delete the objects but Enumeration interface does not support this feature.
  - Enumeration interface is used by legacy classes. Vector.elements() & Hashtable.elements() method returns Enumeration. Iterator is returned by all Java Collections Framework classes. java.util.Collection.iterator() method returns an instance of Iterator.

Why should you always use ArrayList over Vector?

- You should use ArrayList over Vector because you should default to non-synchronized access. Vector synchronizes each individual method. That's almost never what you want to do. Generally you want to synchronize a whole sequence of operations. Synchronizing individual operations is both less safe (if you iterate over a Vector, for instance, you still need to take out a lock to avoid anyone else changing the collection at the same time) but also slower (why take out a lock repeatedly when once will be enough)? Of course, it also has the overhead of locking even when you don't need to. It's a very flawed approach to have synchronized access as default.

- You can always decorate a collection using Collections.synchronizedList - the fact that Vector combines both the "resized array" collection implementation with the "synchronize every operation" bit is another example of poor design; the decoration approach gives cleaner separation of concerns. Vector also has a few legacy methods around enumeration and element retrieval which are different than the List interface, and developers (especially those who learned Java before 1.2) can tend to use them if they are in the code. Although Enumerations are faster, they don't check if the collection was modified during iteration, which can cause issues, and given that Vector might be chosen for its syncronization - with the attendant access from multiple threads, this makes it a particularly pernicious problem. Usage of these methods also couples a lot of code to Vector, such that it won't be easy to replace it with a different List implementation. Despite all above reasons we may never have officially deprecated Vector class (backward compatibility !!!).

What is the importance of hashCode() and equals() methods?

- The java.lang.Object has methods public boolean equals(Object obj) and public int hashCode(). These two methods are used heavily when objects are stored in collections. There is a contract between these two methods which should be kept in mind while overriding any of these methods. The Java API documentation describes it in detail.
- The hashCode() method returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable or java.util.HashMap. The general contract of hashCode is: Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.

- This integer need not remain consistent from one execution of an application to another execution of the same application. If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results.
- However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables. As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects.
- The equals(Object obj) method indicates whether some other object is "equal to" this one. The equals method implements an equivalence relation on non-null object references: It is reflexive: for any non-null reference value x, x.equals(x) should return true.

- It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified. For any non-null reference value x, x.equals(null) should return false.
- The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

- Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes. **A practical Example of hashcode() & equals():** This can be applied to classes that need to be stored in Set collections. Sets use equals() to enforce non-duplicates, and HashSet uses hashCode() as a first-cut test for equality. Technically hashCode() isn't necessary then since equals() will always be used in the end, but providing a meaningful hashCode() will improve performance for very large sets or objects that take a long time to compare using equals().

Q&A

Эффективная работа с коллекциями требует упражнений и практического опыта

Есть масса разных вопросов, которые важно понимать для правильной работы с коллекциями...

См. 24 Java Collections Interview Questions (материалы к семинару)

Замечание по литературе:
Читать – Хорстман т.1 глава 9 (10 издание).
Есть русский перевод, но – как всегда – его качество оставляет желать лучшего...
Выложить его в Moodle?

См. главу Collections в Sun Tutorial (он предоставлен на сайте Oracle)

См. учебник on-line:
http://tutorials.jenkov.com/java-collections/index.html