Estructuras de datos concurrentes

PROGRAMACIÓN CONCURRENTE EN JAVA - TEMA 4.3

Universidad
Rey Juan Carlos

Introducción

Estructuras de datos en Java

- El ***Java Collections Framework*** fue introducido en Java 1.2 para mejorar las clases básicas de colecciones (listas, tablas hash) de las versiones anteriores
- **Colecciones (*Collections*)**
  - Agrupación de objetos con operaciones de inserción, borrado, consulta, etc…
- **Marco de trabajo (*Framework*)**
  - Término usado en programación orientada a objetos para designar a un conjunto de clases que definen un diseño y una estructura para solucionar un conjunto de problemas relacionados

Introducción

Estructuras de datos en Java

- Los **genéricos** son un mecanismo utilizado en los lenguajes de programación con **tipado estático** para especificar el **tipo** de los **elementos** de una estructura de datos
- En C++ se les denomina plantillas (*templates*)
- Aparte de las estructuras de datos, también puede utilizarse en **otros contextos**
- Los genéricos se incorporaron en Java 5

Introducción

Estructuras de datos en Java

- A las estructuras de datos en Java se las denomina **colecciones** en vez de **estructuras de datos**
- Con este nombre se enfatiza que son objetos que mantienen una **colección de elementos**, independientemente de su **estructura** interna
- Esto permite **separar** la parte **pública** (interfaz) de los detalles de **implementación** internos

Listas, conjuntos y mapas

Estructuras de datos en Java

- Las colecciones (*Collections*)
  - Acceso por **posición**: Lista (*List*)
  - Acceso **secuencial**: Conjunto (*Set*)
  - Acceso por **clave**: Mapa (*Map*)
- *Framework* de colecciones
  - Arquitectura **unificada** para representar y manejar colecciones
  - Independiente de los detalles de **implementación**

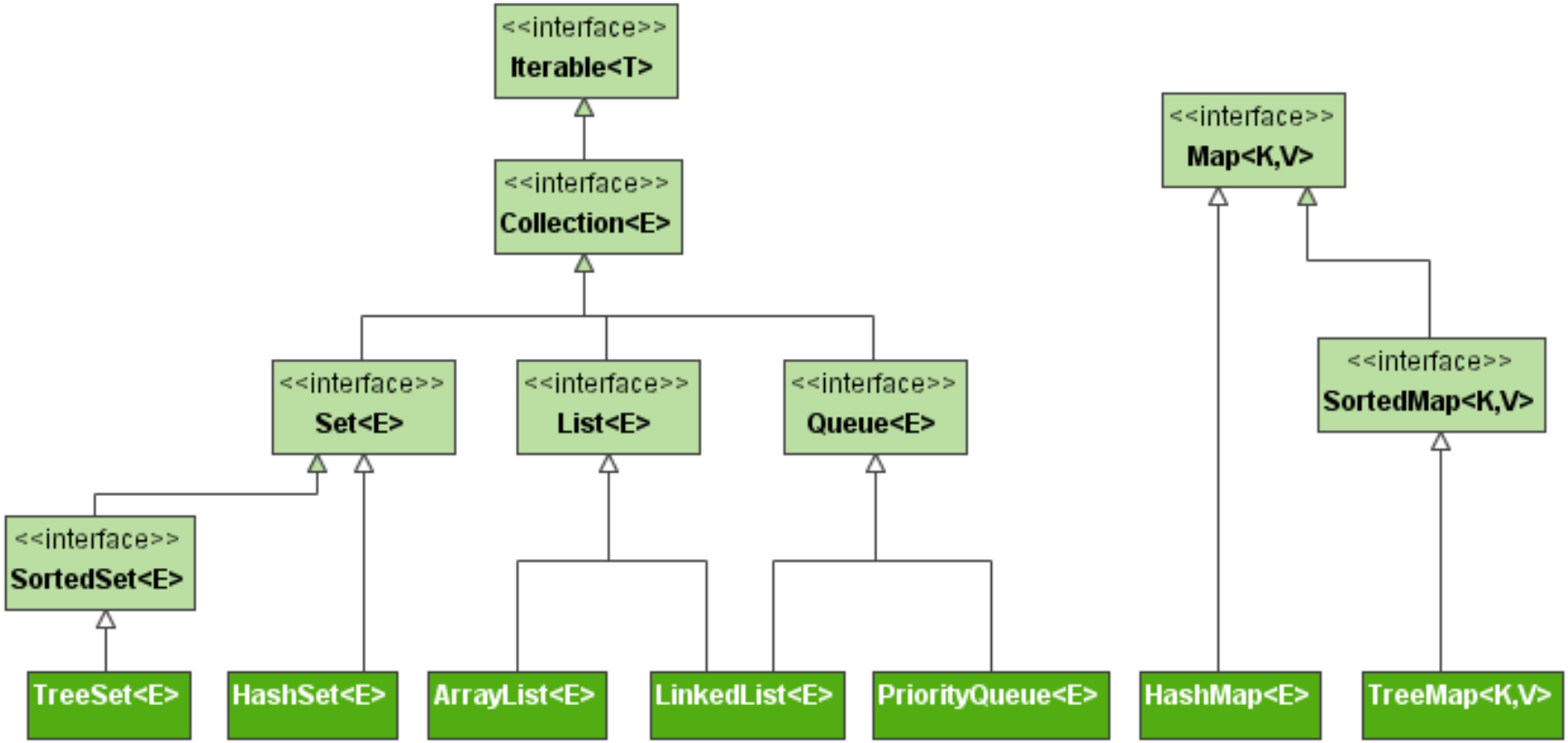
Listas, conjuntos y mapas

Estructuras de datos en Java

- Interfaces
  - Permiten **manipular** las colecciones independientemente de la implementación particular.
  - Definen la **funcionalidad**, no cómo debe implementarse esa funcionalidad
- Implementación
  - Clases que **implementan** los interfaces que definen los tipos de colecciones (listas, mapas y conjuntos)
- Algoritmos
  - Métodos que realizan **computaciones** sobre colecciones de elementos
  - Búsqueda, ordenación, etc.

Listas, conjuntos y mapas

Estructuras de datos en Java



Listas, conjuntos y mapas

Estructuras de datos en Java

- Es necesario saber la complejidad de las operaciones de cada colección
  - http://bigocheatsheet.com/

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

	get	*add*	*contains*	*next*	*remove(O)*	*Iterator.remove*
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

	get	*containsKey*	*next*	*Note*
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

	add	*contains*	*next*	*Note*
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

	offer	*peek*	*poll*	*size*
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)
DelayQueue	O(log n)	O(1)	O(log n)	O(1)
LinkedList	O(1)	O(1)	O(1)	O(1)
ArrayDeque	O(1)	O(1)	O(1)	O(1)
LinkedBlockingDequeue	O(1)	O(1)	O(1)	O(1)

Estructuras de datos sincronizadas

- Desde Java 1.2, en el *Java Collections Framework* existen implementaciones de las colecciones denominadas **colecciones sincronizadas** (*synchronized collections*)
- Estas implementaciones están diseñadas para que sean compartidas entre varios hilos
- Tienen todos sus **métodos** bajo **exclusión mutua** (**sincronizados**)
- Ningún hilo podrá ejecutar ningún método mientras otro hilo esté ejecutando otro método (o el mismo)

Estructuras de datos sincronizadas

Estructuras de datos concurrentes

- Para crear una colección sincronizada primero hay que crear un objeto de una colección
- Luego se invoca algún método estático de la clase **Collections**

```
public static <T> Collection<T>  synchronizedCollection(Collection<T> c)
public static <T> Set<T> synchronizedSet(Set<T> s)
public static <T> List<T> synchronizedList(List<T> list)
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
public static <T> SortedSet<T>  synchronizedSortedSet(SortedSet<T> s)
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)
```

- A las colecciones sincronizadas se las denomina **envolturas de sincronización** (*Synchronization Wrappers*) porque envuelven a la colección original

Estructuras de datos sincronizadas

- Ejemplo de creación de **colecciones sincronizadas**

```
List<String> sharedList =
   Collections.synchronizedList(new ArrayList<String>());

Map<String,String> sharedMap =
   Collections.synchronizedMap(new HashMap<String,String>());

Set<String> sharedSet =
   Collections.synchronizedSet(new HashSet<String>());
```

Estructuras de datos sincronizadas

- Ejemplo de acceso compartido a una lista de *Strings*

```
public class SynchronizedCollectionsSample {

  static List<String> sharedList;

  protected static void process(int num) {
    for (int i = 0; i < 5; i++) {
      try {
         Thread.sleep((long) (Math.random() * 500));
      } catch (InterruptedException e) {}
      sharedList.add("H"+num+"_I"+i);
    }
  }

  public static void main(String[] args) {
    sharedList =
       Collections.synchronizedList(new ArrayList<String>());

    //Create threads and wait for it to finish

    System.out.println("List: "+sharedList);
  }
}
```

Instrucciones atómicas

Estructuras de datos sincronizadas

- ¿Es el siguiente código seguro si **list** es una colección sincronizada compartida por varios hilos?

```
public String deleteLast(List<String> list) {
    int lastIndex = list.size() - 1;
    return list.remove(lastIndex);
}
```

- La colección está preparada para llamadas concurrentes porque está **sincronizada**
- Pero es posible que se produzca una **condición de carrera** y la colección compartida sea **modificada** entre la primera consulta del tamaño y devolver el elemento

Instrucciones atómicas

- Cualquier **instrucción atómica de grano grueso**, en la que se realice una **acción compuesta** requiere que el cliente de la colección **sincronice** su **acceso** a la misma
  - Recorrer la colección
  - Borrar/Devolver el último elemento
  - Operaciones condicionales como "insertar si ausente" (put-if-absent)
  - …
- El cliente tiene que poner las acciones compuestas bajo **exclusión mutua** con el mismo **cerrojo (*lock*)** que se sincroniza **internamente** la colección

Instrucciones atómicas

ESTRUCTURAS DE DATOS SINCRONIZADAS

- Borrar el último elemento de una **lista sincronizada**

```
public String deleteLast(List<String> list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        return list.remove(lastIndex);
    }
}
```

- Recorrer una **lista sincronizada**

```
synchronized(sharedList) {
    for (String elem : sharedList){
        System.out.print(elem+",");
    }
    System.out.println();
}
```

Instrucciones atómicas

- Insertar un elemento sólo si no existe en un **mapa sincronizado**

```
synchronized (map) {
    if (!map.containsKey(key)) {
        map.put(key, value);
    }
}
```

Instrucciones atómicas

ESTRUCTURAS DE DATOS SINCRONIZADAS

- Al recorrer un mapa sincronizado hay que usar como **cerrojo** (*lock*) al **mapa**, no a las colecciones auxiliares

```
synchronized(sharedMap) {
    for (Entry<String,Integer> elem : sharedMap.entrySet()){
        System.out.print(
          elem.getKey()+">"+elem.getValue()+",");
    }
    System.out.println();
}
```

- Si se recorre una colección sin exclusión mutua y otro hilo la modifica antes de finalizar el recorrido, se puede producir una **ConcurrentModificationException**

Problemas con las colecciones sincronizadas

- Todas las acciones que se realizan sobre la colección deben **sincronizarse** (están bajo **exclusión mutua**)
- Esto puede **limitar** innecesariamente la **concurrencia** en operaciones de lectura simultáneas
- Se limita el aprovechamiento de los recursos, se **limita la escalabilidad**
- Esto es especialmente problemático al poner la colección **bajo exclusión** mutua para **recorrerla** y realizar operaciones en los elementos

Estructuras de datos concurrentes

- Para solucionar los problemas de las colecciones sincronizadas, en Java 5 se crearon las **colecciones concurrentes**
- Estas clases están diseñadas para el acceso concurrente desde varios hilos de ejecución.
- Ofrecen mucho **mejor rendimiento** que las **colecciones sincronizadas** y deberían usarse siempre que se necesite una **colección compartida** entre hilos

Estructuras de datos concurrentes

- Cada colección tiene su versión concurrente

Interface	Class
Map	ConcurrentHashMap
List	CopyOnWriteArrayList
Set	CopyOnWriteArraySet
SortedMap	ConcurrentSkipListMap
SortedSet	ConcurrentSkipListSet

ConcurrentHashMap

- La clase está diseñada para permitir acceso de lectura concurrentes
- También permite cierta concurrencia en la escritura
- En las colecciones concurrentes no es necesario (**y no se puede**) usar la exclusión mutua para recorrer la colección
- La clase está diseñada para no generar nunca una **ConcurrentModificationException**

ConcurrentHashMap

- El interfaz **ConcurrentMap** proporciona instrucciones atómicas (acciones compuestas) del alto nivel
  - **V putIfAbsent(K key, V value):** Si la clave especificada no está asociada con ningún valor, asóciala con el valor indicado
  - **boolean remove(Object key, Object value):** Borra la entrada para la clave sólo si está actualmente asociada al valor indicado
  - **V replace(K key, V value):** Reemplaza la entrada para la clave sólo si está actualmente asociada a algún valor
  - **boolean replace(K key, V oldValue, V newValue):** Reemplaza la entrada para una clave sólo si la está asociada al valor indicado

ConcurrentHashMap

- Comparación entre un ConcurrentHashMap y un mapa sincronizado
  - Tiempo empleado por varios hilos realizando el mismo número de operaciones cada uno sobre una estructura de datos compartida

Threads	ConcurrentHashMap	Mapa sincronizado
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

CopyOnWriteArrayList

- **CopyOnWriteArrayList** es un sustituto de una lista sincronizada que ofrece mejor concurrencia en el caso habitual, existen muchas lecturas y pocas modificaciones de la colección
- Las colecciones **copia-al-escribir** (*copy-on-write*) permiten un acceso concurrente seguro porque son objetos que no cambian. Cuando se modifican, se crea una copia para las sucesivas lecturas
- Como es **costoso realizar la copia** cuando se modifica, esta estructura está diseñada para casos en los que las **lecturas** son bastante más **habituales que** las **escrituras**

Ejercicio 1

- Se desea implementar de forma concurrente un programa que busca ficheros con el mismo nombre dentro de una carpeta
- La búsqueda se realiza recursivamente en unas carpetas dentro de otras
- Se proporciona la versión secuencial del programa
- Por simplicidad, en la carpeta raíz no hay ficheros y se crearán tantos hilos como carpetas

Ejercicio 1

ESTRUCTURAS DE DATOS CONCURRENTES

```
public class FindDuplicates {
   private static Map<String,String> duplicates
      = new HashMap<String,String>();

   public static void findDuplicates(File root) {
      if (root.isDirectory()) {
         for (File file : root.listFiles()) {
            if (file.isDirectory()) {
               findDuplicates(file);
            } else {
               String path = duplicates.get(file.getName());
               if(path == null){
                  duplicates.put(file.getName(), file.getAbsolutePath());
               } else {
                  System.out.println("Found duplicate file: "+file.getName());
                  System.out.println("    "+path);
                  System.out.println("    "+file.getAbsolutePath());
               }
            }
         }
      }
   }

   public static void main(String[] args) {
      findDuplicates(new File("X:\\Dir"));
   }
}
```

Colas (*Queues*)

- Las colas (*queues*) son colecciones que albergan elementos antes de ser **procesados**
- Las colas son clases que implementan el interfaz **java.util.Queue<E>**
- El orden de procesado de elementos puede ser FIFO, LIFO, por prioridades… y depende de la implementación

Colas (*Queues*)

- El interfaz de colas (**Queue**) no define métodos **bloqueantes**
- Estos métodos se usan para implementar esquemas de **productores consumidores** en programación **concurrente**
- Estos métodos están definidos en el interfaz **BlockingQueue**, que hereda de **Queue**

Colas bloqueantes (*BlockingQueue*)

- Las colas bloqueantes (**java.util.concurrent.BlockingQueue**) ofrecen operaciones que se quedan bloqueadas
- Si la cola está **llena**, los métodos de **inserción** se **bloquean** hasta que haya **espacio** (o salte un *timeout*)
- Si la cola está **vacía**, los métodos de **extracción** se **bloquean** hasta que exista un **elemento** (o salte un *timeout*)

Colas bloqueantes (*BlockingQueue*)

- **Inserción bloqueante**
  - **put(e):** Bloquea hasta que se pueda realizar la operación
  - **offer(e, time, unit):** Bloquea y devuelve **false** si no se realiza la operación en el tiempo indicado
- **Extracción bloqueante**
  - **take():** Bloque hasta que se pueda realizar la operación
  - **poll(time, unit):** Bloquea y devuelve **null** si no se realiza la operación en el tiempo indicado

Colas bloqueantes (*BlockingQueue*)

- Las **colas bloqueantes** (*BlockingQueue*) son clases que permiten compartirse por varios hilos (*thread-safe*)
- Pero algunas operaciones compuestas como **addAll**, **containsAll**,… puede que **no** se implementen de forma **atómica**

Colas dobles (*Deque*)

- Las colas dobles (**java.util.Deque**) son colecciones lineales que permiten inserción y eliminación en cualquier extremo
- **Deque** proviene de cola doblemente finalizada (*double ended queue*)
- Este estructura se puede comportar como una **cola FIFO** o como una **pila LIFO** dependiendo de los métodos usados

Implementaciones de Queue

- **Queue**
  - **PriorityQueue**
    - Ordenación basada en prioridades
    - No concurrente (no *thread-safe*)
  - **ConcurrentLinkedQueue**
    - Ordenación FIFO
    - Concurrente (*thread-safe*)
  - **LinkedList**
    - Ordenación FIFO
    - No concurrente (no *thread-safe*)

Implementaciones de Deque

- **Deque**
  - **ArrayDeque**
    - Implementación FIFO basada en array
    - No concurrente (no *thread-safe*)
  - **LinkedList**
    - Implementación FIFO basada en lista enlazada
    - No concurrente (no *thread-safe*)
  - **ConcurrentLinkedDeque** (Java 7)
    - Implementación FIFO basada en lista enlazada
    - Concurrente (*thread-safe*)

Implementaciones de BlockingQueue

- **BlockingQueue**
  - **ArrayBlockingQueue**
    - Cola FIFO basada en arrays
  - **LinkedBlockingQueue**
    - Cola FIFO basada en listas enlazadas
  - **PriorityBlockingQueue**
    - Cola por prioridades

Implementaciones de BlockingQueue

- **BlockingQueue**
  - **SynchronousQueue**
    - Cola sin capacidad
    - Las inserciones deben esperar a las extracciones
    - Las extracciones deben esperar a las inserciones
  - **DelayQueue**
    - Cola FIFO que mantiene los elementos en la cola durante un tiempo especificado (*delay*)

Implementaciones

- **BlockingDeque**
  - **LinkedBlockingDeque**
    - Cola doble basada en listas enlazas
    - Concurrente (*thread-safe*)
- **TransferQueue** (Java 7)
  - Cola con métodos para esperar a clientes o consumidores al insertar un elemento
  - **LinkedTransferQueue**
    - Cola de transferencia basada en listas enlazadas
    - Concurrente (*thread-safe*)

Ejercicio 2

- Implementa el problema del productor-consumidor utilizando colas bloqueantes.
- La cola debe tener un límite de elementos que sea parametrizable.
- El productor solo puede producir si hay hueco en la cola
  - Debe producir un String con el texto "Productor IDX -> Producto N"
    - IDX es el identificador del hilo productor y N es el número de producto.
  - Debe imprimir por pantalla el mensaje "PRODUCIENDO PRODUCTO"
    - PRODUCTO es el mensaje producido.
- El consumidor solo puede consumir si hay productos en la cola
  - Cuando consuma debe imprimir un mensaje con el producto consumido "CONSUMIDOR IDX: PRODUCTO"
    - IDX es el identificador del consumidor y PRODUCTO es el producto consumido