code URJC

Programación Concurrente

Tema 5

Programación Concurrente en Java

Micael Gallego
micael.gallego@urjc.es
@micael_gallego

Universidad Rey Juan Carlos

code URJC

Programación Concurrente

Tema 5.5

Estructuras de datos concurrentes

Micael Gallego
micael.gallego@urjc.es
@micael_gallego

Universidad Rey Juan Carlos

Estructuras de datos concurrentes

- **Introducción**
- Valores atómicos concurrentes
- Estructuras de datos en Java
- Estructuras de datos sincronizadas
- Estructuras de datos concurrentes
- Colas (*Queues*)
- Streams

Introducción

- Como ocurre con cualquier otro objeto, compartir estructuras de datos entre varios hilos es muy **delicado**
- Hay que elegir algunas de las siguientes alternativas para **evitar condiciones de carrera**:
  - Poner bajo **exclusión mutua** el acceso a las estructuras compartidas
  - Usar estructuras de datos **thread-safe**, que estén preparadas para ser usadas desde varios hilos

Introducción

- En la API de Java existe un conjunto de clases e interfaces para la gestión de estructuras de datos llamado ***Java Collections Framework***
- Este *framework* permite usar varios tipos de estructuras de datos **(listas, mapas, conjuntos, colas…)**
- Existen versiones optimizadas para ser usadas por un **único hilo** y otras versiones diseñadas para ser **compartidas entre hilos**

Introducción

- Al igual que existen **estructuras de datos** preparadas para ser compartidas entre hilos, la API de Java también ofrece clases para compartir **valores atómicos** entre hilos (**thread-safe**).
- Con estas clases se pueden compartir valores de **tipo entero y objetos**
- Disponen de métodos que realizan operaciones básicas bajo **exclusión mutua**

Estructuras de datos concurrentes

- Introducción
- **Valores atómicos concurrentes**
- Estructuras de datos en Java
- Estructuras de datos sincronizadas
- Estructuras de datos concurrentes
- Colas (*Queues*)
- Streams

Valores atómicos concurrentes

- El paquete **java.util.concurrent.atomic** tiene varias clases que permiten compartir un valor atómico entre varios hilos (**Ver documentación del paquete**).
- Dispone de clases para:
  - Tipos primitivos: Integer, Long, Boolean
  - Objetos
  - Arrays de tipos primitivos y objetos
  - Otras clases más avanzadas

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

AtomicInteger

code URJC

- Se usa para compartir una variable de **tipo entero** entre varios hilos.
- Habitualmente usada para tener un **contador**
- Métodos:
  - **int get():** Devuelve el valor del entero
  - **int getAndIncrement():** Devuelve e incrementa
  - **int getAndSet(int newValue):** Devuelve y establece un nuevo valor
  - **boolean compareAndSet(int expect, int update):** Establece el nuevo valor si el anterior era el esperado.

AtomicInteger

- Ejemplo de uso

```
class SynchronizedCounter {

  private int c = 0;


  public synchronized void inc(){
    c++;
  }
  public synchronized void dec(){
    c--;
  }
  public synchronized int get(){
    return c;
  }
}
```

```
class AtomicCounter {

  private AtomicInteger c
     = new AtomicInteger(0);

  public void increment() {
     c.incrementAndGet();
  }
  public void decrement() {
     c.decrementAndGet();
  }
  public int value() {
     return c.get();
  }
}
```

Estructuras de datos concurrentes

- Introducción
- Valores atómicos concurrentes
- **Estructuras de datos en Java**
- Estructuras de datos sincronizadas
- Estructuras de datos concurrentes
- Colas (*Queues*)
- Streams

Estructuras de datos en Java

- El ***Java Collections Framework*** es un conjunto de clases e interfaces de la librería estándar para gestionar colecciones de elementos.
- Permite gestionar **listas**, **conjuntos**, **colas** y **mapas**

http://docs.oracle.com/javase/8/docs/technotes/guides/collections/

Estructuras de datos en Java

- Genéricos
- Listas, conjuntos y mapas
- Recorrer una colección
- Ordenación y Búsqueda
- *Equals* y *hashcode*
- Colecciones con tipos primitivos
- Ventajas de la colecciones

Estructuras de datos en Java

- **Genéricos**
- Listas, conjuntos y mapas
- Recorrer una colección
- Ordenación y Búsqueda
- *Equals* y *hashcode*
- Colecciones con tipos primitivos
- Ventajas de la colecciones

Genéricos

- 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**

Genéricos

- Clase pila de fracciones implementada con un array

```
class PilaFracciones {

   private Fraccion[] elementos;
   private int numElementos;

   public PilaFracciones(int tope) {
      this.elementos = new Fraccion[tope];
      this.setNumElementos(0);
   }

   public Fraccion getElemento(int indice) {
      return this.elementos[indice];
   }

   public void addElemento(Fraccion fraccion) {
      if(numElementos < elementos.length){
         this.elementos[numElementos] = fraccion;
         numElementos++;
      }
   }
   ...
}
```

Genéricos

- Uso de la clase pila de fracciones

```
public static void main(String[] args) {

   PilaFracciones pila = new PilaFracciones(5);

   Fraccion fraccion1 = new Fraccion(1,2);
   Fraccion fraccion2 = new Fraccion(1,3);
   Fraccion fraccion3 = new Fraccion(1,4);

   pila.addElemento(fraccion1);
   pila.addElemento(fraccion2);
   pila.addElemento(fraccion3);

   ...
}
```

Genéricos

- ¿Si queremos implementar pila de **Intervalos**?
- ¿Creamos **otra** clase cambiando Fracción por Intervalo?
- Duplicar código es malo

```
class PilaIntervalos {

   private Intervalo[] elementos;
   private int numElementos;

   public PilaIntervalos(int tope) {
      this.elementos = new Intervalo[tope];
      this.setNumElementos(0);
   }

   public Intervalo getElemento(int indice) {
      return this.elementos[indice];
   }

   public void addElemento(Intervalo intervalo) {
      if(numElementos < elementos.length){
         this.elementos[numElementos] = intervalo;
         numElementos++;
      }
   }
   ...
}
```

Genéricos

- Uso de la clase pila de intervalos

```
public static void main(String[] args) {

    PilaIntervalos pila = new PilaIntervalos(5);

    Intervalo intervalo1 = new Intervalo(1,2);
    Intervalo intervalo2 = new Intervalo(1,3);
    Intervalo intervalo3 = new Intervalo(1,4);

    pila.addElemento(intervalo1);
    pila.addElemento(intervalo2);
    pila.addElemento(intervalo3);

    ...
}
```

Genéricos

- ¿Si queremos implementar pila de Intervalos?
- Podemos usar el **polimorfismo**
- ¿Hacemos que los elementos sean de tipo **Object**?

```
class PilaObjects {

   private Object[] elementos;
   private int numElementos;

   public PilaObjects(int tope) {
      this.elementos = new Object[tope];
      this.setNumElementos(0);
   }

   public Object getElemento(int indice) {
      return this.elementos[indice];
   }

   public void addElemento(Object object) {
      if(numElementos < elementos.length){
         this.elementos[numElementos] = object;
         numElementos++;
      }
   }
   ...
}
```

Genéricos

- Usar **Object** funciona
- Pero el compilador **no nos ayuda**
- Se puede escribir código **incorrecto**

```
//Queremos que la pila contenga Intervalos
PilaObjects pilaIntervalos = new PilaObjects(5);

pilaIntervalos.addElemento(new Intervalo(2,4));
pilaIntervalos.addElemento(new Intervalo(2,6));
…
//Nos equivocamos y el compilador no da error
//No se genera una excepción en ejecución
pilaIntervalos.addElemento(new Fraccion(1,2));

…
//Siempre que sacamos intervalos tenemos
//que hacer casting. Puede generar una excepción
Intervalo intervalo = (Intervalo) pilaIntervalos.getElemento()
```

Genéricos

- Lo ideal sería **definir el tipo** de los elementos cuando se **usa** la pila, no cuando se implementa la pila

```
public static void main(String[] args) {

    PilaIntervalo pilaIntervalo;
    PilaFraccion  pilaFraccion;

    ...
}
```

Genéricos

- Un **tipo genérico** es un tipo usado al **implementar** una clase que se **concretará** cuando se use la clase
  - Al declarar una variable, parámetro o atributo
  - Al instanciar un objeto

Genéricos

```
class PilaIntervalos {

  public PilaIntervalos(int tope) { ... }

  public Intervalo getElemento(int indice) { ... }

  public void addElemento(Intervalo intervalo) { ... }
}
```

```
class Pila<E> {

  public Pila(int tope) { ... }

  public E getElemento(int indice) { ... }

  public void addElemento(E elem) { ... }
  ...
}
```

Genéricos

Sin usar genéricos

```
//Queremos que la pila contenga Intervalos
PilaIntervalos pilaIntervalos = new PilaIntervalos(5);

Intervalo i1 = new Intervalo(2,4);
Intervalo i2 = new Intervalo(2,6);

pilaIntervalos.addElemento(i1);
pilaIntervalos.addElemento(i2);
…
Intervalo i3 = pilaIntervalos.getElemento(1);
```

Genéricos

Sin usar genéricos

```
//Queremos que la pila contenga Intervalos
PilaIntervalos pilaIntervalos = new PilaIntervalos(5);

Intervalo i1 = new Intervalo(2,4);
Intervalo i2 = new Intervalo(2,6);

pilaIntervalos.addElemento(i1);
pilaIntervalos.addElemento(i2);
…
Intervalo i3 = pilaIntervalos.getElemento(1);
```

Usando genéricos

```
//Queremos que la pila contenga Intervalos
Pila<Intervalo> pilaIntervalos = new Pila<>(5);

Intervalo i1 = new Intervalo(2,4);
Intervalo i2 = new Intervalo(2,6);

pilaIntervalos.addElemento(i1);
pilaIntervalos.addElemento(i2);
…
Intervalo i3 = pilaIntervalos.getElemento(1)
```

Genéricos

Sin usar genéricos

```
PilaIntervalos pilaIntervalos = new PilaIntervalos(5);
PilaFracciones pilaFracciones = new PilaFracciones(5);

…

Intervalo intervalo = pilaIntervalos.getElemento(1);
Fraccion fraccion = pilaFracciones.getElemento(1);
```

Usando genéricos

```
Pila<Intervalo> pilaIntervalos = new Pila<>(5);
Pila<Fraccion> pilaFracciones = new Pila<>(5);

…

Intervalo intervalo = pilaIntervalos.getElemento(1);
Fraccion fraccion = pilaFracciones.getElemento(1);
```

Genéricos

- ¿Cómo se implementaría un método que admitiera pilas, independientemente de cual sea el tipo de sus elementos?

```
public void sacarElementos(Pila<?> pila, int numElems){
    for(int i=0; i<numElementos; i++){
        pila.pop();
    }
}
```

- Al símbolo **?** Se le denomina comodín (*wildcard*)
- Es importante conocer este símbolo porque se utiliza en diversos métodos del Java Collections Framework

Estructuras de datos en Java

- Genéricos
- **Listas, conjuntos y mapas**
- Recorrer una colección
- Ordenación y Búsqueda
- *Equals* y *hashcode*
- Colecciones con tipos primitivos
- Ventajas de la colecciones

Listas, conjuntos y mapas

code URJC

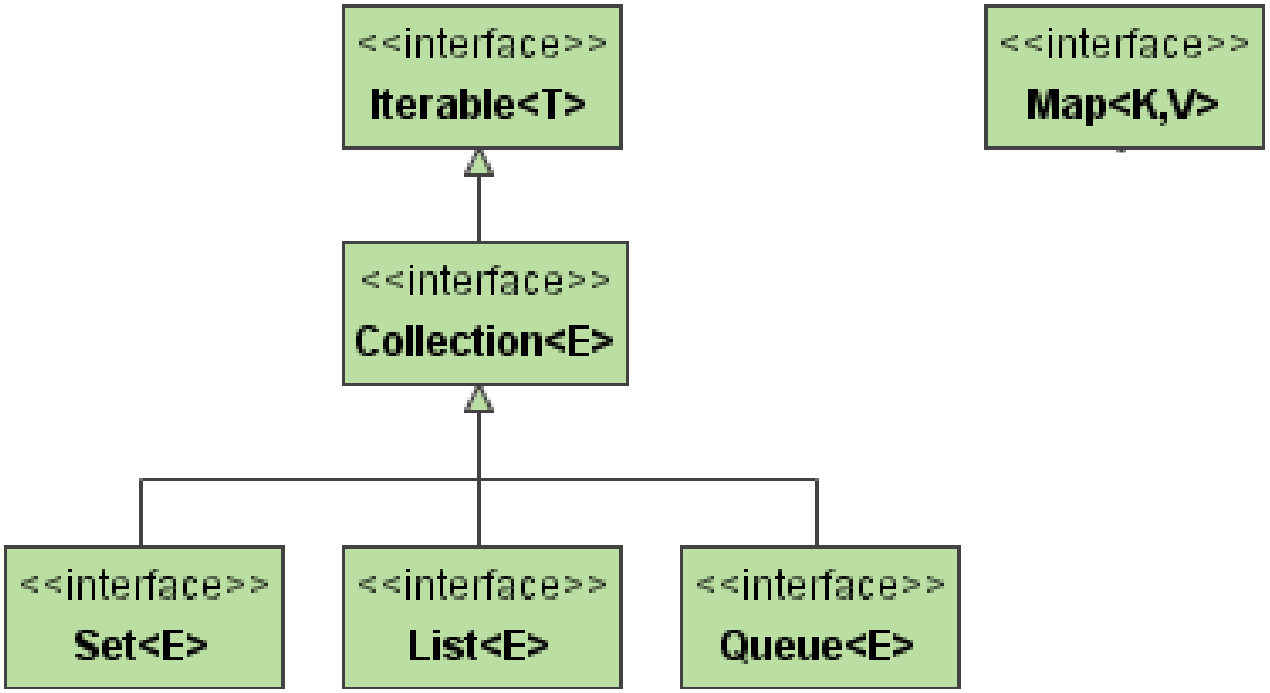
- A las estructuras de datos en Java se las denomina **colecciones** en vez de **estructuras de datos**
- Con este nombre se enfatiza que son objetos que mantienen una **colección de elementos**, independientemente de su **estructura interna**
- Esto permite **separar** la parte **pública** (interfaz)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

Listas, conjuntos y mapas

- **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.

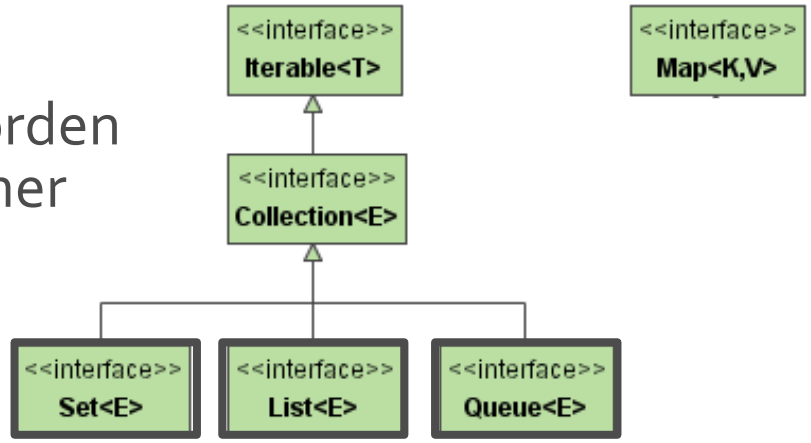
Listas, conjuntos y mapas

- **Tipos de colecciones**



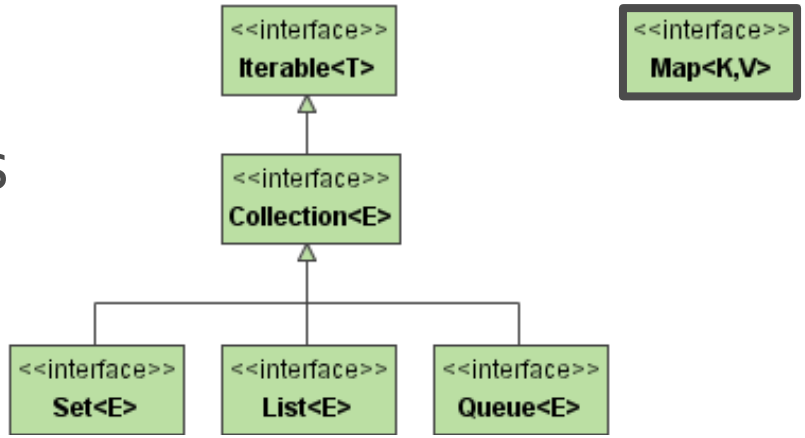
Listas, conjuntos y mapas

- **Set<E>**
  - Colección que no mantiene el orden de inserción y que no puede tener dos o más objetos iguales
- **List<E>**
  - Colección que sí mantiene el orden de inserción y que puede contener elementos duplicados
- **Queue<E>**
  - Colección para almacenar múltiples elementos antes de ser procesados. Especialmente utilizada en programas concurrentes



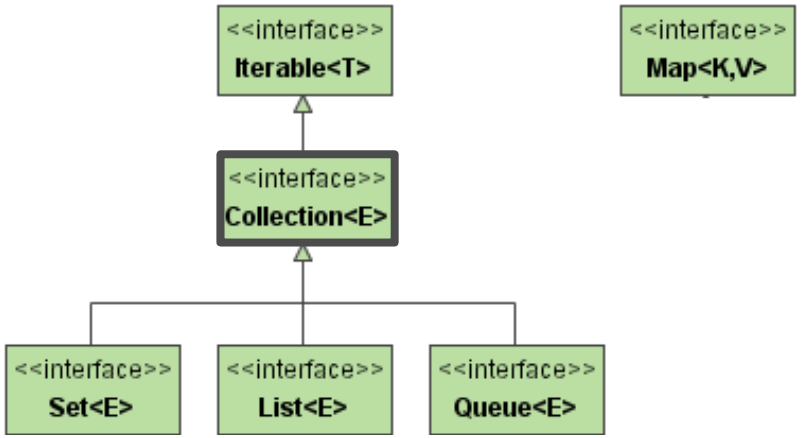
Listas, conjuntos y mapas

- **Map<K,V>**
  - Estructura que guarda los elementos (valores) asociados a una clave



Listas, conjuntos y mapas

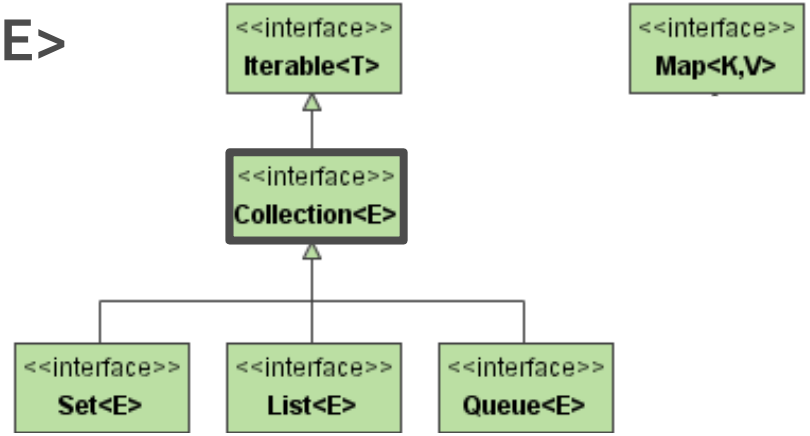
- **Collection<E>**
  - Colección genérica
  - Se pueden consultar el número de elementos
  - Se pueden añadir y eliminar elementos:
    - Hay colecciones que permiten elementos duplicados y otras no
    - Hay colecciones ordenadas o desordenadas
    - Hay colecciones que permiten el valor null, otras no



Listas, conjuntos y mapas

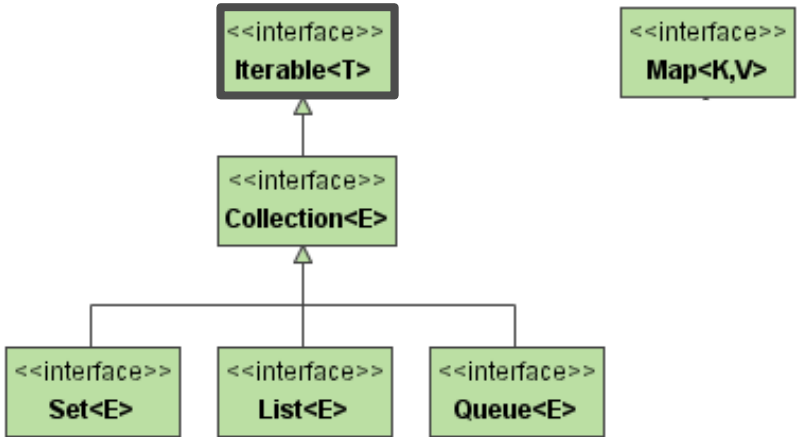
code URJC

- Algunos métodos de **Collection<E>**
  - Para agregar y eliminar elementos
    - boolean add(E e)
    - boolean remove(Object o)
  - Para realizar consultas
    - int size()
    - boolean isEmpty()
    - boolean contains(Object o)
  - Para realizar varias operaciones de forma simultánea
    - boolean containsAll(Collection<?> collection)
    - void clear()
    - boolean removeAll(Collection<?> collection)



Listas, conjuntos y mapas

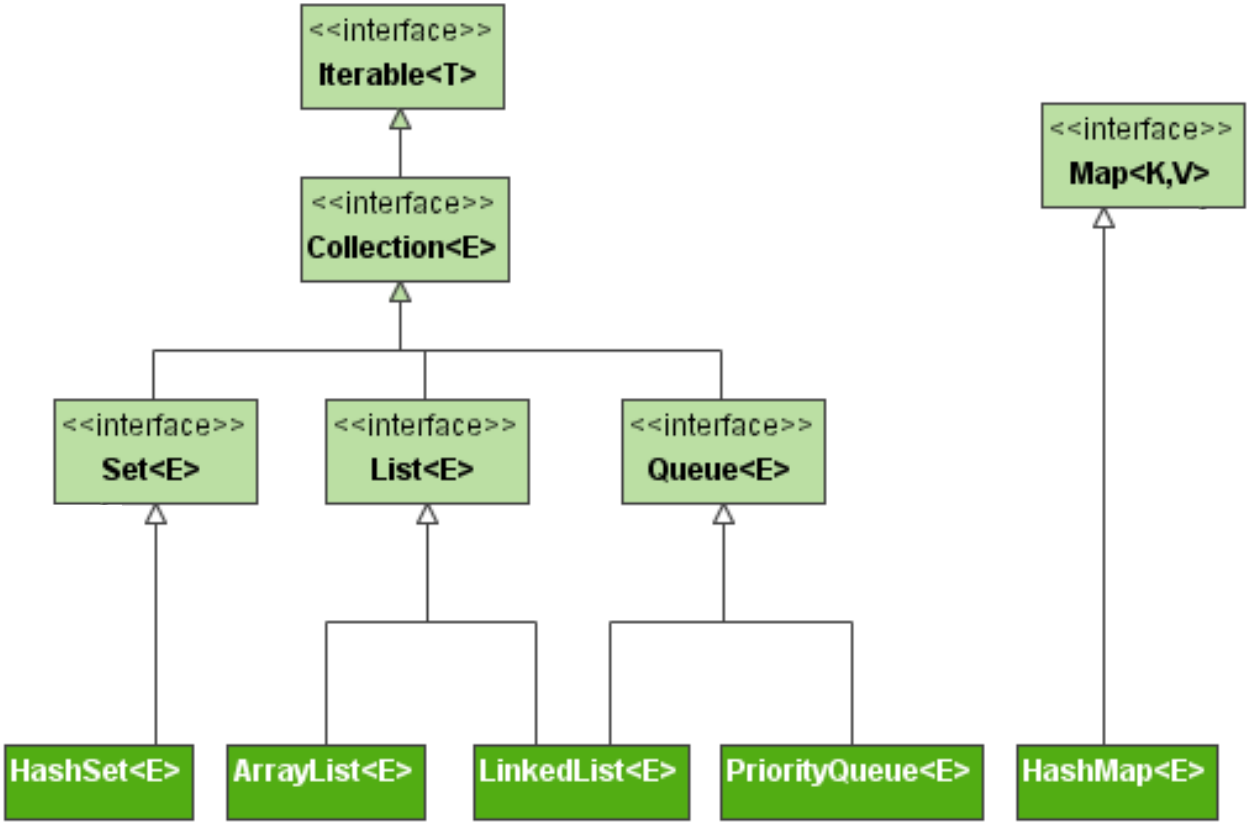
- **Iterable<T>**
  - Permite acceder a los elementos uno por uno
  - No permite consultar el número de elementos
  - No permite añadir o eliminar elementos
  - Se utiliza para **recorrer una colección**



Listas, conjuntos y mapas

- Estos interfaces tienen una **implementación por defecto** que puede ser utilizada en la mayoría de los casos
  - **List<E>:** ArrayList<E>
  - **Set<E>:** HashSet<E>
  - **Map<K,V>:** HashMap<K,V>
  - **Queue<E>:**
    - LinkedList<E>: Cola FIFO
    - PriorityQueue<E>: Ordena sus elementos por prioridad

Listas, conjuntos y mapas



Listas (List<E>)

- **Interfaces vs Implementaciones**
  - Las variables, parámetros y atributos se declaran con el tipo de las **interfaces**
  - La clase de **implementación** sólo se usa para instanciar los objetos
  - Se abstrae lo más posible de la implementación concreta (y se puede cambiar fácilmente en el futuro)

```
List<String> nombres = new ArrayList<>();
```

Listas (List<E>)

- Colección que **mantiene el orden de inserción** y que puede contener elementos **duplicados**
- Similar a un array pero que crece de forma dinámica
- Se accede a los elementos indicando su posición
- Algunos métodos:
  - void add(int index, E element)
  - boolean addAll(int index, Collection<? extends E> c)
  - E get(int index)
  - E remove(int index)

Listas (List<E>)

- Es la estructura de datos **más usada**
- Es la estructura de datos **más eficiente para la inserción** de elementos (al final)
- No obstante, **no** es muy **eficiente** para **búsquedas** (porque son secuenciales)

Listas (List<E>)

- **ArrayList<E>** es la clase por defecto que implementa **List<E>**

```
//Declaro la variable del tipo de la interfaz,
//y le asigno un objeto del tipo de la clase de
//implementación.
List<Intervalo> listaIntervalos = new ArrayList<>();
List<Fraccion> listaFracciones = new ArrayList<>();

listaIntervalos.add(new Intervalo(2,4));
listaFracciones.add(new Fraccion(2,6));
…
Intervalo intervalo = listaIntervalos.get(0);
Fraccion fraccion = listaFracciones.get(0);
```

Ejercicio 1

- Crear un ejemplo básico para probar el funcionamiento de las listas
  - Declarar una lista de **String**.
  - Añadir y eliminar elementos de la lista
  - Definir un método **addElemToList(…)** que reciba una lista de **String** y un **String** como parámetro y añada el **String** a la lista

Conjuntos (Set<E>)

- No mantiene el orden de inserción
- No es posible recuperar los elementos en el orden en que fueron insertados
- No admite elementos duplicados
- Si se añade un objeto al conjunto y ya había otro igual, no se produce ningún cambio en el conjunto

Conjuntos (Set<E>)

- Es la estructura de datos **más eficiente buscando elementos**
- Pero eso hace que la **inserción** sea más **costosa** que en las listas

Conjuntos (Set<E>)

- **HashSet<E>** es la implementación por defecto de **Set<E>** y se implementa utilizando una tabla **hash**

```
//Declaro la variable del tipo de la interfaz,
//y le asigno un objeto del tipo de la clase de
//implementación.
Set<Intervalo> intervalos = new HashSet<>();

Intervalo intervalo = new Intervalo(2,4);
intervalos.add(intervalo);

//Esta inserción no tiene efecto
intervalos.add(intervalo);
int numIntervalos = intervalos.size(); // Devuelve 1
```

Mapas (Map<K,V)

- Define una estructura de datos que asocia (mapea) claves con valores
- No permite claves repetidas
- Varias claves distintas pueden estar asociadas al mismo valor (valores repetidos)
- La búsqueda de un valor asociado a una clave es muy eficiente

Mapas (Map<K,V)

- **Métodos más importantes:**
  - **V put(K key, V value):** Insertar un valor asociado a la clave
  - **V get(Object key):** Obtener un valor asociado a la clave
  - **Collection<V> values():** Devuelve la colección de valores
  - **Set<K> keySet():** Devuelve el conjunto de claves
  - **Entry<K,V> entrySet():** Devuelve el conjunto de pares clave-valor (entradas del mapa)

Mapas (Map<K,V)

- **HashMap<K,V>** es la implementación por defecto de **Map<K,V>** que implementa el conjunto de datos utilizando una tabla **hash**

```
Map<String, Coche> propietarios = new HashMap<>(5);

Coche toledo = new Coche("Seat", "Toledo", 110)
Coche punto = new Coche("Fiat", "Punto", 90);

propietarios.put("M-1233-YYY", toledo);
propietarios.put("M-1234-ZZZ", punto);

Coche c = propietarios.get("M-1234-ZZZ");
```

Mapas (Map<K,V)

```
Map<String, String> configuracion = new HashMap<>();

configuracion.put("lenguaje", "ingles");
configuracion.put("servidor", "http://...");
Configuracion.put("correo", "a.b@xyz");

…

String lenguaje = configuracion.get("lenguaje");
String servidor = configuracion.get("servidor");
```

Ejercicio 2

- Se tiene una colección de aeropuertos (objetos de la clase Aeropuerto), y se desea poder obtener un aeropuerto dado su nombre
- Declarar la estructura de datos adecuada para asociar el nombre de cada aeropuerto con el objeto aeropuerto correspondiente
- Introducir varios aeropuertos asociados a sus nombres: “El Prat”, “Barajas”, “Castellón”
- Obtener el objeto aeropuerto dado su nombre: “Barajas”

Estructuras de datos ordenadas

- Cuando los elementos son comparables entre sí, puede ser útil insertar de forma **ordenada** los elementos
- **SortedSet<E>**: Ordena los elementos
  - Implementación TreeSet<E>
- **SortedMap<E>**: Ordena las claves
  - Implementación TreeMap<E>

Otras implementaciones

- Aparte de las implementaciones por defecto, existen otras implementaciones para situaciones especiales
  - De propósito general
  - De propósito específico
  - Para soporte de concurrencia
  - Combinadas
  - Optimizadas para el acceso secuencial
  - Optimizadas para el acceso aleatorio

Estructuras de datos en Java

- Genéricos
- Listas, conjuntos y mapas
- **Recorrer una colección**
- Ordenación y Búsqueda
- *Equals* y *hashcode*
- Colecciones con tipos primitivos
- Ventajas de la colecciones

Recorrer una colección

- Acceder a cada elemento de una colección depende de su tipo:
  - Lista
    - Acceso por posición con **bucle for**
    - Acceso **secuencial**
  - Conjunto
    - Acceso **secuencial**
  - Mapa
    - Acceso secuencial a la **colección de valores**
    - Acceso secuencial al **conjunto de claves**
    - Acceso secuencial al **conjunto de entradas**

Recorrer una Lista

- Acceso por posición con **bucle for**

```
List<String> ciudades = new ArrayList<>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

for (int i=0; i < ciudades.size(); i++) {
   String ciudad = ciudades.get(i);
   System.out.println(ciudad + "\n");
}
```

- No se recomienda, sobre todo en estructuras de datos basadas en listas enlazadas (**LinkedList<E>**) puede ser ineficiente frente al acceso secuencial

Recorrer una Lista

- Acceso secuencial
  - Se puede realizar con el **for mejorado**
  - Se puede realizar con el **iterador**

Recorrer una Lista

- Acceso secuencial con **for mejorado**

```
List<String> ciudades = new ArrayList<>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

for (String ciudad: ciudades) {
    System.out.println(ciudad + "\n");
}
```

- En general es la forma preferida

Recorrer una Lista

- Acceso secuencial con **iterador**

```
List<String> ciudades = new ArrayList<>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

Iterator<String> it = ciudades.iterator();
while (it.hasNext()){
    String s = it.next();
    System.out.println(s + "\n");
}
```

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

Recorrer una Lista

- Acceso secuencial con **iterador**
  - Es la única forma de borrar elementos de una lista mientras se recorre

```
List<String> ciudades = new ArrayList<>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

Iterator<String> it = ciudades.iterator();
while (it.hasNext()){
    String ciudad = it.next();
    if(ciudad.equals(“Madrid”)){
       it.remove();
    }
}
```

Recorrer un Conjunto

- Sólo se pueden recorrer los elementos de un **conjunto** con acceso secuencial
- Con **for mejorado** o con **iteradores**

```
Set<String> ciudades = new HashSet<>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

for (String ciudad: ciudades) {
    System.out.println(ciudad + "\n");
}
```

Recorrer un Mapa

- Formas de recorrer un mapa
  - Acceso secuencial a la **colección de valores**

```
Map<String, Coche> propietarios = new HashMap<>(5);

coches.put("M-1233-YYY", new Coche("Seat", "Toledo", 110));
coches.put("M-1234-ZZZ", new Coche("Fiat", "Punto", 90));

for (Coche coche : coches.values()) {
    System.out.println("Coche: "+coche);
}
```

Recorrer un Mapa

- Formas de recorrer un mapa
  - Acceso secuencial al **conjunto de claves**

```
Map<String, Coche> propietarios = new HashMap<>(5);

coches.put("M-1233-YYY", new Coche("Seat", "Toledo", 110));
coches.put("M-1234-ZZZ", new Coche("Fiat", "Punto", 90));

for (String matricula : coches.keySet()) {
    System.out.println("Matricular: "+matricula);
}
```

Recorrer un Mapa

- Formas de recorrer un mapa
  - Acceso secuencial al **conjunto de entradas**

```
Map<String, Coche> propietarios = new HashMap<>(5);

coches.put("M-1233-YYY", new Coche("Seat", "Toledo", 110));
coches.put("M-1234-ZZZ", new Coche("Fiat", "Punto", 90));

for (Entry<String,Coche> e : coches.entrySet()) {
    System.out.println("Mat:"+e.getKey()+" "+e.getValue());
}
```

Estructuras de datos en Java

code URJC

- Genéricos
- Listas, conjuntos y mapas
- Recorrer una colección
- **Ordenación y Búsqueda**
- *Equals* y *hashcode*
- Colecciones con tipos primitivos
- Ventajas de la colecciones

Ordenación y Búsqueda

- Sólo se pueden ordenar las **listas**
- Ordenación de elementos con métodos estáticos en la clase **Collections**

```
List<String> nombres = new ArrayList<>();
nombres.add("Pepe");
nombres.add("Juan");
nombres.add("Antonio");

Collections.sort(nombres);
System.out.println(nombres);
```

```
[Antonio, Juan, Pepe]
```

Ordenación y Búsqueda

- Se puede especificar el **orden de comparación**
  - Se usa una lambda que devuelve un valor positivo si o1 es mayor que o2. Negativo en caso contrario

```
List<String> nombres = new ArrayList<>();
nombres.add("Juanin");
nombres.add("Pepe");
nombres.add("Antonio");

Collections.sort(nombres,(o1,o2)-> o1.length()-o2.length());
```

```
[Pepe, Juanin, Antonio]
```

Ordenación y Búsqueda

- Se puede especificar el **orden de comparación**
  - Se puede usar Comparator.comparing(…)

```
List<String> nombres = new ArrayList<>();
nombres.add("Juanin");
nombres.add("Pepe");
nombres.add("Antonio");

Collections.sort(nombres,
     Comparator.comparing(String::length));
```

```
[Pepe, Juanin, Antonio]
```

Ordenación y Búsqueda

- **¿Qué es una búsqueda?**
  - En un conjunto es saber si está el elemento
  - En un mapa es obtener el valor asociado a la clave (si está)
  - En una lista es saber su posición (si está)

Ordenación y Búsqueda

- **Búsqueda en conjuntos**
  - Método: boolean contains(Object o)
- **Búsqueda en mapas**
  - Método : V get(Object key)

Ordenación y Búsqueda

- **Búsquedas en listas**
  - Si la lista está desordenada, usamos el método **int indexOf(E e)**
  - Si la lista está ordenada, se puede usar una **búsqueda binaria** (método en la clase Collections)
    - Si el elemento está en la lista, devuelve su **posición**
    - Si el elemento no está en la lista, devuelve el lugar en el que **debería estar**

Ordenación y Búsqueda

```
List<String> nombres = new ArrayList<>();
nombres.add("Pepe");
nombres.add("Juan");
nombres.add("Antonio");

Collections.sort(nombres);

//int pos = nombres.indexOf("Mario");
int pos = Collections.binarySearch(nombres, "Mario");
if (pos < 0){
    //El nombre no está en la lista
    int insertPos = -pos-1;
    System.out.println("No está. Debería estar en: "+insertPos);
} else {
    System.out.println("Está en la posición: "+pos);
}
```

Ordenación y Búsqueda

- Hay que **elegir muy bien** la estructura de datos que se utiliza en un programa
  - Listas (basadas en arrays)
    - Eficiente la inserción al final O(1)
    - Eficiente el acceso por posición O(1)
    - Ineficiente la búsqueda O(n)
  - Conjuntos (basados en hash)
    - Eficiente la inserción O(1) (aunque menos que la lista)
    - No se puede hacer acceso por posición
    - Eficiente la búsqueda O(1)
  - Mapas (basados en hash)
    - Igual que los conjuntos

Ejercicio 3

- Implementar una aplicación que permita gestionar en memoria un conjunto de viajes de una aerolínea
- Cada viaje se representa con la ciudad origen, destino y la duración del viaje (clase Viaje)
- Se dan de alta los viajes en un gestor (clase GestorViajes)
- Al gestor de viajes se le pueden pedir:
  - Devolver todos los viajes que tienen una determinada ciudad origen
  - Devolver todos los viajes que tienen una determinada ciudad destino
  - Devolver todos los viajes
  - Devolver todas las ciudades en las que hay viajes
- Hay que conseguir el **menor tiempo de ejecución** de las consultas, aunque sean necesarias varias estructuras de datos

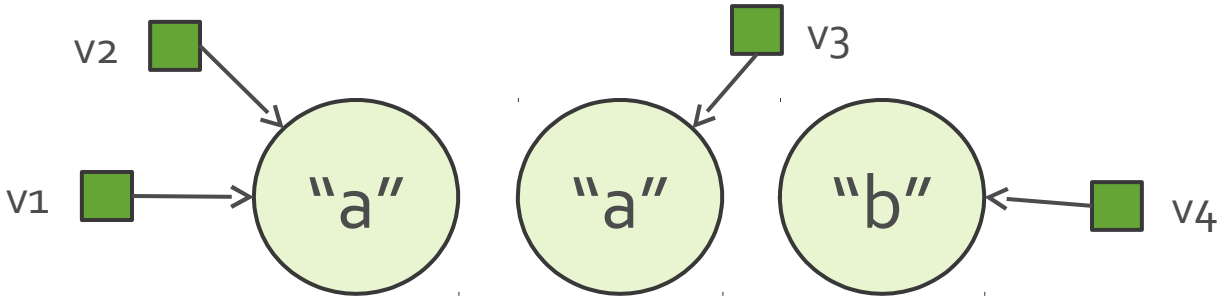
Estructuras de datos en Java

- Genéricos
- Listas, conjuntos y mapas
- Recorrer una colección
- Ordenación y Búsqueda
- ***Equals* y *hashcode***
- Colecciones con tipos primitivos
- Ventajas de la colecciones

Equals y *hashcode*

- **¿Objetos iguales o el mismo objeto?**
  - Dos variables apuntan al **mismo** objeto si al comparar con **==** se obtiene **verdadero** (***true***)
  - Dos variables apuntan a objetos **iguales** si al comparar con el método **equals** se obtiene **verdadero** (***true***)

Equals y hashcode



Expresión	Resultado
v1 == v2	true
v1.equals(v2)	true
v1 == v3	false
v1.equals(v3)	true
v1 == v4	false
v1.equals(v4)	false

Equals y hashcode

- **Estructuras de datos basadas en hash**
  - La estructura de datos **HashSet** utiliza los métodos **equals** y **hashcode** de los objetos que se insertan en ella
  - La estructura de datos **HashMap** utiliza los métodos **equals** y **hashcode** de las claves que se asocian a valores
  - Los métodos **equals** y **hashcode** se usan para saber si dos objetos son **iguales** o **diferentes**

Equals y hashcode

- **Estructuras de datos basadas en hash**
  - Las clases **String**, **Date**, **Double**… implementan correctamente los métodos **equals** y **hashcode** para que funcionen correctamente con **HashSet** y como claves de un **HashMap**
  - Las clases implementadas por el desarrollador **no** tienen esta funcionalidad por **defecto**

Equals y hashcode

- **Estructuras de datos basadas en hash**
  - Si en una clase propia no implementamos los métodos **equals** y **hashcode**
    - Nunca existirán dos objetos **iguales** de esa clase (aunque tengan los **mismos valores** de los atributos)
    - Si se intenta insertar **el mismo objeto** en un conjunto por segunda vez, no tendrá efecto.
    - Si se intenta insertar **un objeto con los mismos valores de atributos** que otro ya incluido, **se insertará** el nuevo valor (porque a ojos del HashSet, no son iguales)

Equals y hashcode

- **Estructuras de datos basadas en hash**
  - Si en una clase propia si implementamos los métodos **equals** y **hashcode**
    - Decidimos cuándo dos objetos de esa clase se consideran **iguales** (habitualmente si algunos de sus atributos son iguales)
    - Si se intenta insertar **el mismo objeto** en un conjunto por segunda vez, no tendrá efecto.
    - Si se intenta insertar **un objeto igual** que otro ya incluido, no tendrá efecto**.**

Equals y hashcode

- **¿Cómo se implementan los métodos equals y hashcode?**
  - Se deciden qué atributos tienen que ser iguales para que los objetos se consideren iguales (pueden ser todos).
  - El entorno de desarrollo genera automáticamente el código en base a esos atributos

* Item 9 en "Effective Java Programming Language Guide, second edition", (Addison-Wesley,2008) de Joshua Bloch

Equals y hashcode

```
public class Fraccion {

  private float numerador;
  private float denominador;

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + Float.floatToIntBits(denominador);
    result = prime * result + Float.floatToIntBits(numerador);
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Fraccion other = (Fraccion) obj;
    if (Float.floatToIntBits(denominador) != Float
        .floatToIntBits(other.denominador))
      return false;
    if (Float.floatToIntBits(numerador) != Float
        .floatToIntBits(other.numerador))
      return false;
    return true;
  }
}
```

Estructuras de datos en Java

- Genéricos
- Listas, conjuntos y mapas
- Recorrer una colección
- Ordenación y Búsqueda
- *Equals* y *hashcode*
- **Colecciones con tipos primitivos**
- Ventajas de la colecciones

Colecciones con tipos primitivos

- Existen clases que se comportan como **los tipos primitivos** (clases de envoltura, *wrapper*)
  - Integer, Double, Float, Boolean, Character…
- El *autoboxing* y *autounboxing* es la capacidad de conversión automática entre un valor de un tipo primitivo y un objeto de la clase correspondiente

```
int numero = 3;
Integer numObj = numero;
int otroNum = numObj;
```

Colecciones con tipos primitivos

- Esto permite usar las colecciones con tipos primitivos
- Hay que ser consciente de que se tienen que realizar conversiones y eso es **costoso**

```
List<Integer> enteros = new ArrayList<>();
enteros.add(3);
enteros.add(5);
enteros.add(10);

int num = enteros.get(0);
```

Colecciones con tipos primitivos

- Existen implementaciones de terceros con estructuras de datos especialmente diseñadas para tipos primitivos
- Deben usarse cuando se utilizan mucho en un programa y las conversiones sean muy numerosas
  - http://trove4j.sourceforge.net/
  - http://fastutil.dsi.unimi.it/
  - http://commons.apache.org/primitives/

Estructuras de datos en Java

- Genéricos
- Listas, conjuntos y mapas
- Recorrer una colección
- Ordenación y Búsqueda
- *Equals* y *hashcode*
- Colecciones con tipos primitivos
- **Ventajas de la colecciones**

Ventajas de las colecciones

- Reducción del esfuerzo del programador
- Incremento de la velocidad y calidad
- Interoperabilidad entre APIs no relacionadas
- Menor esfuerzo de aprendizaje y uso de otras APIs
- Fomenta la reutilización del software

Ventajas de las colecciones

- Aunque las estructuras de datos de la API son muy completas, existen librerías de terceros que las complementan
  - Google Guava:
    - http://code.google.com/p/guava-libraries/
  - Otras:
    - http://java-source.net/open-source/collection-libraries

Estructuras de datos concurrentes

- Introducción
- Valores atómicos concurrentes
- Estructuras de datos en Java
- **Estructuras de datos sincronizadas**
- Estructuras de datos concurrentes
- Colas (*Queues*)

Estructuras de datos sincronizadas

code URJC

- En el *Java Collections Framework* existen implementaciones de las colecciones denominadas **colecciones sincronizadas** (*synchronized collections*)
- Estas implementaciones están diseñadas para que se puedan **compartir entre varios hilos**
- Tienen todos sus **métodos** bajo **exclusión mútua** (**sincronizados**)
- Ningún hilo podrá ejecutar ningún método mientras otro hilo esté ejecutando otro método (o el mismo)

Estructuras de datos sincronizadas

- 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 evuelven a la colección original

Estructuras de datos sincronizadas

- Ejemplo de creación de **colecciones sincronizadas**

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

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

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

Estructuras de datos sincronizadas

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

```
public class SynchronizedCollectionsSample {

  private List<String> sharedList;

  private void process(int num) {
    for (int i = 0; i < 5; i++) {
        sharedList.add("H"+num+"_I"+i);
    }
  }

  public exec() {

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

    //Create threads and wait for it to finish

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

}
```

Instrucciones atómicas

- ¿El siguiente código es 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);
}
```

Instrucciones atómicas

- ¿El siguiente código es 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 consultar el 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

- 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 en un **mapa sincronizado** si no existen ya antes

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

Instrucciones atómicas

- 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 (que no causan interferencias)
- 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

- Introducción
- Valores atómicos concurrentes
- Estructuras de datos en Java
- Estructuras de datos sincronizadas
- **Estructuras de datos concurrentes**
- Colas (*Queues*)
- Streams

Estructuras de datos concurrentes

- Las **colecciones concurrentes** son implementaciones de las colecciones diseñadas para compartirse entre hilos
- Para ciertos casos, son **más eficientes** que las **colecciones sincronizadas**

Estructuras de datos concurrentes

- Algunas de las colecciones concurrentes **más usadas:**
  - **Map:** ConcurrentHashMap
  - **List:** CopyOnWriteArrayList
  - **Set:** CopyOnWriteArraySet , set basado en ConcurrentHashMap

ConcurrentHashMap

- La clase **ConcurrentHashMap** está diseñada para permitir acceso concurrente de lectura y escritura
- Es mucho más **eficiente** que el mapa sincronizado y se **recomienda su uso siempre**

Threads	ConcurrentHashMap	Mapa sincronizado
1	1.00	1.03
4	5.58	78.23
8	13.21	163.48
32	57.27	778.41

http://www.ibm.com/developerworks/library/j-jtp07233/

ConcurrentHashMap

- En las colecciones concurrentes **no se puede usar la sincronización** para implementar operaciones atómicas de **grano grueso**
- Para solucionarlo, el interfaz **ConcurrentMap** proporciona las operaciones atómicas de grano grueso más habituales como **métodos**

ConcurrentHashMap

- **Métodos de ConcurrentMap**
  - **V putIfAbsent(K key, V value):** Si la clave especificada no está asociada con ningún valor, se asocia con el valor indicado. Se devuelve el antiguo valor si existía.
  - **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
  - **Muchos más…**

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

ConcurrentHashMap

- Asociar un valor a una clave sólo si no existe esa clave en el **mapa**

synchronizedMap

```
synchronized (map) {
    if (!map.containsKey(key)){
        map.put(key, value);
    } else {
        // Ya existe la clave
    }
}
```

ConcurrentHashMap

```
Object old = map.putIfAbsent(key, value);
if(old != null){
  // Ya existe la clave
}
```

CopyOnWriteArrayList

- **CopyOnWriteArrayList** es una implementación concurrente de una lista
- Es **más eficiente** que la lista sincronizada si hay **muchas lecturas** y muy pocas escrituras
- Las colecciones **copia-al-escribir** (*copy-on-write*) no bloquean los hilos que leen.
- Cuando un hilo quiere escribir, se crea una copia interna de los elementos (**muy costoso**), por eso sólo se usan con **pocas escrituras**.

Estructuras de datos concurrentes

- **Colecciones concurrentes para conjuntos**
  - La clase **CopyOnWriteArraySet** es similar a CopyOnWriteArrayList pero para conjuntos
  - Siempre se puede usar un **ConcurrentMap** como si fuera un **conjunto**, considerando las **claves** (*keys*) del map como los valores del conjunto (y usando cualquier valor)

Estructuras de datos concurrentes

- **Colecciones concurrentes para conjuntos**
  - Si necesitamos ver ese mapa como un conjunto, podemos "recubrirlo"

```
ConcurrentMap<String,Boolean> map = new ConcurrentHashMap<>();
Set<String> names = Collections.newSetFromMap(map);
```

  - Si queremos usar las operaciones atómicas de grano grueso, lo tendremos que hacer sobre el mapa concurrente

Ejercicio 4

- 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 4

```
public class FindDuplicates {

   private Map<String,String> duplicates = new HashMap<>();

   public 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 void exec() {
      findDuplicates(new File("X:\\Dir"));
   }
}
```

Estructuras de datos concurrentes

- Introducción
- Valores atómicos concurrentes
- Estructuras de datos en Java
- Estructuras de datos sincronizadas
- Estructuras de datos concurrentes
- **Colas (*Queues*)**
- Streams

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, por prioridades… y depende de la implementación

Colas (*Queues*)

- Proporciona métodos para **insertar**, **extraer** e **examinar** elementos (sin extraerlos)
- Los métodos se proporcionan de dos formas
  - Métodos que **lanzan excepciones** si **fallan**
  - Métodos que **devuelven false** o **null** si **fallan**

	Lanza una excepción	Devuelve false o null
Insertar	add(e)	offer(e)
Extraer	remove()	poll()
Examinar	element()	peek()

Colas (*Queues*)

- **Insertar**
  - Los métodos **offer(e)** y **add(e)** insertan un elemento si la cola no está llena
  - Si la cola está llena:
    - **offer(e)** devuelve false
    - **add(e)** lanza una excepción no chequeada

Colas (*Queues*)

- **Extraer**
  - Los métodos **poll()** y **remove()** extraen elementos de la cabeza si la cola no está vacía
  - El orden de extracción depende de la política de ordenación de la cola (FIFO, Prioridad…)
  - Si la cola está vacía:
    - **poll()** devuelve null
    - **remove()** lanza una excepción no chequeada

Colas (*Queues*)

- **Examinar**
  - Los métodos **peek()** y **element()** devuelven, pero no eliminan, la cabeza de la cola, si la cola no está vacía
  - Si la cola está vacía:
    - **peek()** devuelve null
    - **element()** lanza una excepción no chequeada

Colas (*Queues*)

- Las colas se utilizan habitualmente para implementar esquemas de **productores consumidores** en programación **concurrente**
- Para ello es necesario que los métodos para extraer y añadir sean **bloqueantes**
- El interfaz de colas (**Queue**) no define métodos **bloqueantes**
- 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():** Bloquea hasta que se pueda realizar la operación
  - **poll(time, unit):** Bloquea y devuelve **null** si no se realiza la operación en el tiempo indicado

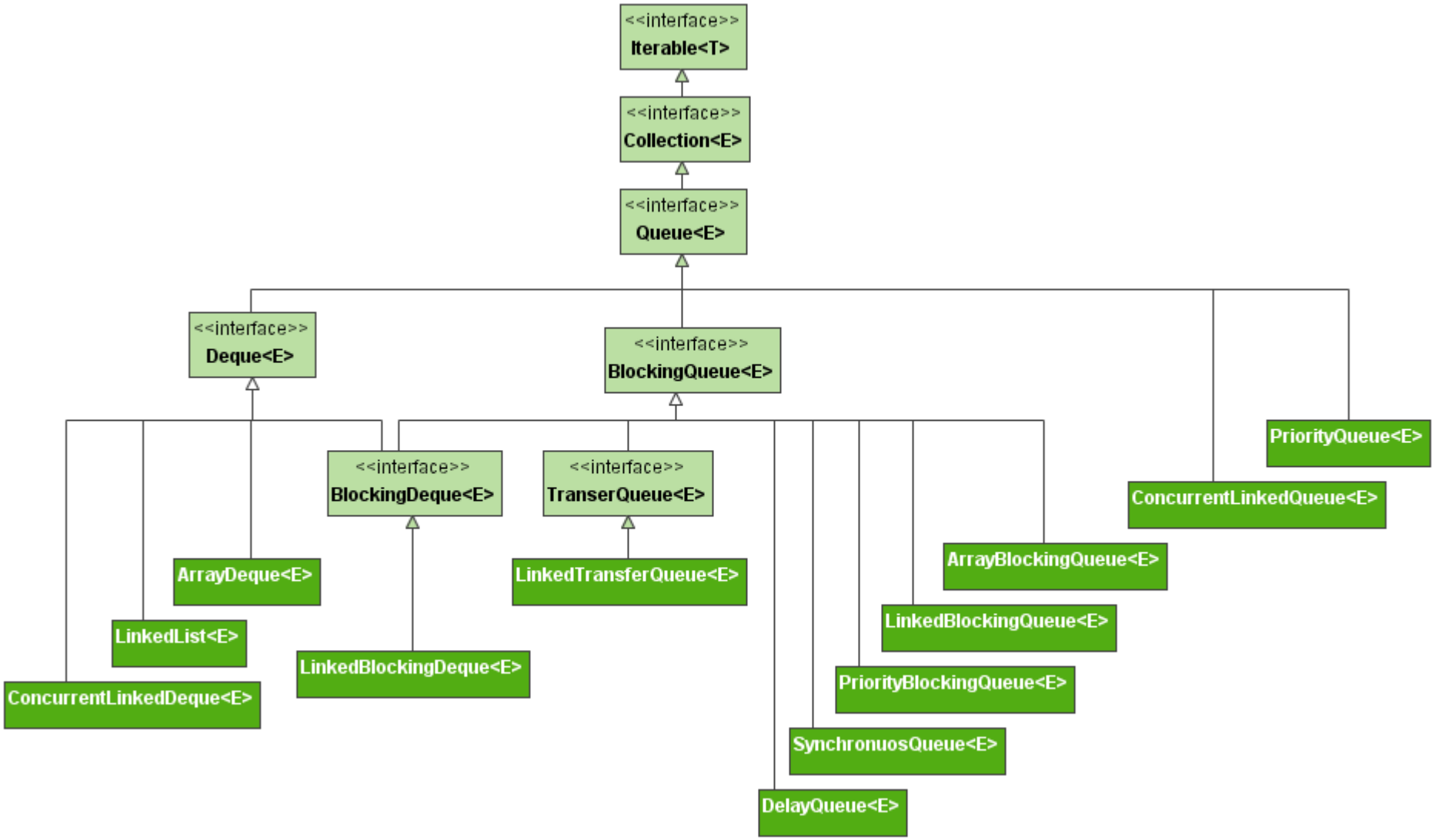
Colas bloqueantes (*BlockingQueue*)

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

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
- El interfaz Deque **hereda de Queue**

ESTRUCTURAS DE DATOS CONCURRENTES



Implementaciones de Queue

- **Queue**
  - **PriorityQueue**
    - Ordenación basada en prioridades
    - No concurrente (no *thread-safe*)
  - **ConcurrentLinkedQueue**
    - Ordención 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*)** antes de que estén disponibles para la extracción

Implementaciones

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

Uso de las colas

- Las colas se utilizan para esquemas de productores-consumidores

```
public class ProdConsQueue {

   private BlockingQueue<Integer> queue
      = new ArrayBlockingQueue<>(10);

   public void productor() throws InterruptedException{
      for (int i = 0; i < 20; i++) {
         Thread.sleep((long)(Math.random() * 500));
         queue.put(i);
      }
   }

   public void consumidor() throws InterruptedException{
      while (true) {
         int data = queue.take();
         Thread.sleep((long)(Math.random() * 500));
         System.out.println(data + " ");
      }
   }

   public void exec() {
      //Crear hilos productores y consumidores
   }
}
```

Ejercicio 5

- Se pretende simular mediante un programa concurrente el comportamiento de una **Fábrica de coches**.
- La fábrica tiene dos tipos de elementos:
  - **Máquinas:** Que generan piezas de un determinado tipo.
  - **Robots:** Que ensamblan todas las piezas que forman el coche.
- Para hacer el programa más genérico, se utilizan las siguientes constantes:
  - **NUM_ROBOTS:** Número de robots
  - **NUM_TIPOS_PIEZAS:** Tipos diferentes de piezas que se necesitan para montar un coche. Cada pieza tiene asociado su tipo y es un valor entre 0 y NUM_TIPOS_PIEZAS-1

Ejercicio 5

- **Máquinas**
  - Las máquinas **fabrican** piezas y las **almacenan**. Este proceso se repite indefinidamente.
  - Las piezas se simulan como un **valor aleatorio de tipo double**
  - Los métodos que se simulan estas operaciones son: **fabricarPieza(…)** y **almacenarPieza(…)**
  - Cada máquina sólo genera piezas del **mismo tipo**. Hay una máquina por cada tipo de pieza.
  - El almacén donde se guardan las piezas es **común** para todas las máquinas.
  - El almacén de piezas puede albergar un máximo de piezas de cada tipo (identificado por la constante **MAX_PIEZAS**)

Ejercicio 5

- **Robots**
  - Los robots **recogen piezas** del almacén y **las montan** para crear los coches. Este proceso se repite indefinidamente.
  - Los métodos que simulan estas operaciones son: **recogerPieza(…)** y **montarPieza(…)**
  - Las piezas deben recogerse en **orden (de 1 a NUM_TIPOS_PIEZAS)** para que se puedan montar adecuadamente.

Ejercicio 5

- **Requisitos**
  - Las máquinas pueden fabricar piezas y los robots pueden montar piezas en el producto sin sufrir ninguna interferencia entre ellos
  - El programa debe implementarse de forma que ni las máquinas ni los robots interfieran entre sí de forma inadecuada al acceder al almacén

Estructuras de datos concurrentes

- Introducción
- Valores atómicos concurrentes
- Estructuras de datos en Java
- Estructuras de datos sincronizadas
- Estructuras de datos concurrentes
- Colas (*Queues*)
- **Streams**

Streams

- Los **streams** (flujos) permiten procesar colecciones de elementos con un estilo **funcional (declarativo)**
- **Ventajas** frente al estilo actual:
  - Más **compacto** y de **alto nivel**
  - Fácilmente **paralelizable**
  - No se necesita tener todos los elementos en **memoria** para empezar a procesar

Streams

- Tenemos la clase **Person** y la lista **people**

```
public class Person {

  private String name;
  private int age;

  // constructors
  // getters / setters
}
```

```
List<Person> people = new ArrayList<>();
```

Streams

- Se quiere calcular la edad media

```
int sum = 0;
int average = 0;
for (Person person : people) {
  sum += person.getAge();
}

if (!list.isEmpty()) {
  average = sum / list.size();
}
```

Streams

- Se quiere calcular la edad media de los menores de 20

```
int sum = 0;
int n = 0;
int average = 0;
for (Person person :people){
 if (person.getAge() > 20){
  n++;
  sum += person.getAge();
 }
}
if (n > 0){
 average = sum /n ;
}
```

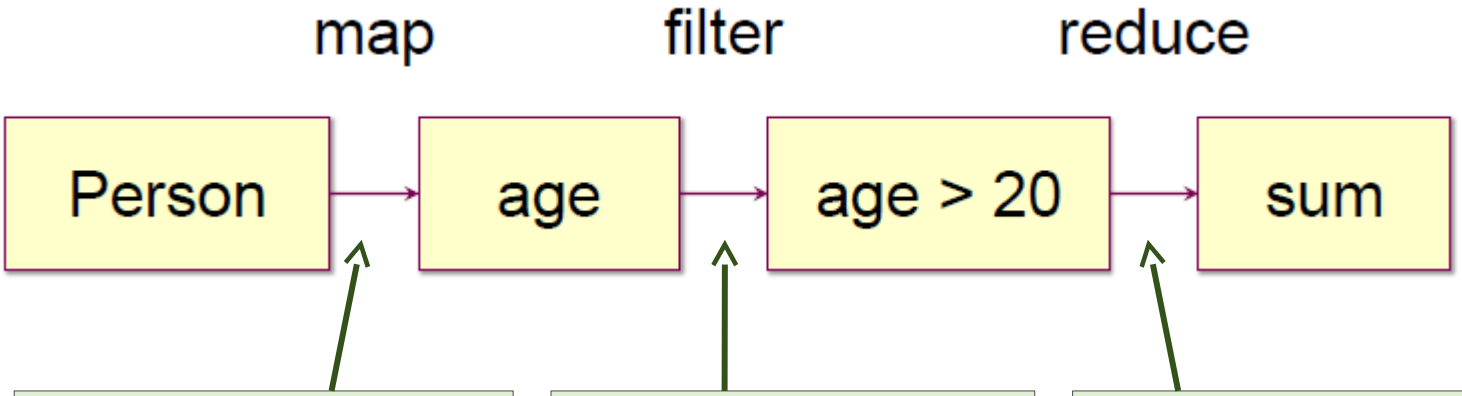
Streams

- La **programación imperativa** es muy ***verbosa*** porque se indica **qué** se quiere y **cómo** se consigue
- La **programación declarativa** es mucho más compacta porque basta indicar **qué** se quiere

```
select
avg(age)
from Person
where age > 20
```

Sentencia SQL equivalente

Streams



Mapeo (*mapping*):

De cada elemento de la lista original obtiene un nuevo valor para la lista resultante.

Se devuelve el mismo número de elementos

Filtrado:

Filtra los elementos.

Se devuelven menos elementos de los originales

Reducción:

Se agregan todos los elementos en uno solo:

Ejemplos: media, mínimo, máximo, suma...

Streams

- Los ***streams*** (**flujos**) permiten aplicar operaciones de mapeo, filtrado y reducción a colecciones de datos
- Un stream es **una secuencia de elementos** que sólo pueden **procesarse una vez**
- Puede **generarse** de forma dinámica (no tiene que estar toda la secuencia en memoria)
- Esto permite **implementaciones muy eficientes** de las operaciones **sin** crear estructuras de **datos intermedias**

Streams

- La forma más habitual de crear un **stream** es partiendo de una **colección**

Creación del stream

```
int sumOver20 = persons.stream()
   .map(Person::getAge)
   .filter(age -> age > 20)
   .reduce(0, Integer::sum);
```

Streams

- También se pueden crear de forma literal:

```
Stream<String> s =
Stream.of("one","two","three");

Stream<String> s2 = Stream.empty();
```

- Partiendo de un array:

```
String[] numbers = {"one", "two", "three"};

Stream<String> s = Arrays.stream(numbers);
```

Streams

- O desde algunos métodos de la API:

```
IntStream chars = “Hola”.chars();

Stream<String> lines = lineNumberReader.lines();

IntStream nums = random.ints();

Stream s3 = Stream.concat(stream1, stream2);

Stream<Integer> pares = Stream.iterate(0,n->n+2);

Stream<Double> rand =
Stream.generate(Math::random);
```

Streams

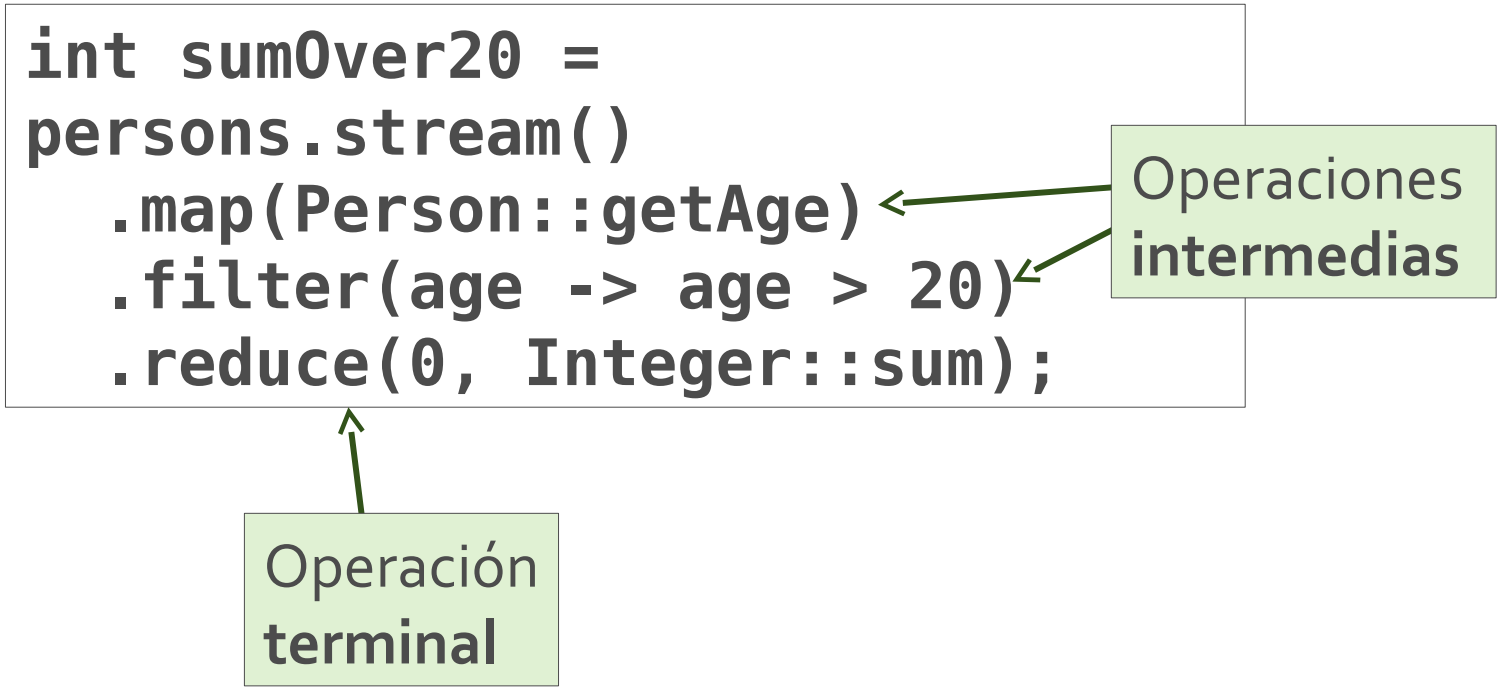
Stream	Colección
Secuencia de elementos	Elementos que se pueden procesar en secuencia o con acceso directo (get)
Se pueden calcular según se van procesando	Tienen que estar en memoria antes de procesarse
Información volátil que sólo se puede procesar una vez	Estructuras de datos en memoria
Tamaño infinito	Tamaño finito
Tiene versiones eficientes para tipos primitivos (IntStream, DoubleStream, LongStream)	No tiene versiones para tipos primitivos

Streams

- **Operaciones** que se pueden realizar sobre un stream:
  - 1) Operaciones **intermedias**
    - Se procesan de forma perezosa (sólo si son necesarias)
    - Puede haber varias (incluso del mismo tipo)
    - Ejemplos: map, filter, skip,…
  - 2) Operaciones **terminales**
    - Inician la computación y devuelven un objeto, un valor, una lista o nada…
    - Sólo puede haber una al final
    - Una vez aplicada, el stream no se puede reutilizar
    - Ejemplos: sum, find, min, toArray…

Streams

- **Operaciones** que se pueden realizar sobre un stream:



Streams

- **Operaciones intermedias:**
  - **filter**: quita algunos elementos
  - **map**: por cada elemento obtiene un nuevo valor:
  - **sorted**: ordena
  - **peek**: aplica una operación a cada elemento
  - **distinct**: filtra dejando los distintos
  - **limit**: limita el número de elementos
  - **skip**: ignora los primeros elementos
  - **range:** devuelve un rango de los elementos (from, to)

Streams

- **Operaciones terminales en Stream<T>:**
  - **count:** Cuenta los elementos.
  - **min, max:** Obtiene el mínimo el y máximo
  - **anyMatch, allMatch, noneMatch():** Indica si se cumple (o no) el criterio.
  - **findFirst, findAny:** Devuelve el elemento que cumpla el criterio
  - **mapToInt:** Convierte a IntStream
  - **toArray:** Devuelve el contenido como array
  - **forEach, forEachOrdered:** Ejecuta por cada elemento
  - **reduce:** Mecanismo genérico de reducción de todos los elementos
  - **collect:** Mecanismo genérico para "recolectar" los elementos

Streams

- **Operaciones terminales en streams numéricos (IntStream, LongStream, DoubleStream):**
  - **average():** Calcula la media
  - **sum():** Suma los elementos
  - **summaryStatistics():** Calcula estadísticas de los datos (media, cuenta, min, max, sum)

Streams

- **Operación terminal Collect (Recolectar)**
  - La operación **collect** es un mecanismo genérico para implementar operaciones terminales
  - Se puede implementar **cualquier** política de recolección de los elementos:
    - Devolver una estructura de datos (List, Set, Map…)
    - Fusionar todos los elementos en un String
    - Obtener un valor (suma, media, min, max…)
  - La clase **Collectors** tiene métodos estáticos para crear una infinidad de **recolectores**

```
ArrayList<String> l = stream.collect(Collectors.toList());
```

Streams

- Edad media

```
double avgAge = persons.stream()
  .collect(Collectors.averagingDouble(Person::getAge));
```

- Número de personas mayores de 20

```
int num = persons.stream().filter(p -> p.getAge() > 20)
  .collect(Collectors.counting());
```

- Persona más mayor (si existen)

```
Optional<Person> older = persons.stream()
.collect(Collectors.maxBy(Comparator.comparing(Person::getAge));
```

Streams

- Devolver los nombres en una lista

```
List<String> list = people.stream().map(Person::getName)
  .collect(Collectors.toList());
```

- Devolver los nombres sin repeticiones ordenados (TreeSet)

```
Set<String> set = people.stream().map(Person::getName)
  .collect(Collectors.toCollection(TreeSet::new));
```

- Devolver los nombres separados por comas

```
String joined = things.stream().map(Object::toString)
  .collect(Collectors.joining(", "));
```

Streams

- Suma de salarios de los empleados

```
int total = employees.stream()
  .collect(Collectors.summingInt(Employee::getSalary)));
```

- Agrupar por departamento

```
Map<Department, List<Employee>> byDept =
employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment));
```

- Suma de salarios por departamento

```
Map<Department, Integer> totalByDept = employees.stream()
   .collect(Collectors.groupingBy(
            Employee::getDepartment,
            Collectors.summingInt(Employee::getSalary)));
```

Streams

- Suma de salarios por departamento

```
Map<Department, Integer> totalByDept = employees.stream()
   .collect(
        Collectors.groupingBy(
            Employee::getDepartment,
            Collectors.summingInt(Employee::getSalary)
        )
    );
```

- Dividir los estudiantes entre aprobados y suspensos

```
Map<Boolean, List<Student>> passingFailing = students.stream()
   .collect(
       Collectors.partitioningBy(s->s.getGrade() >= PASS_THR)
    );
```

Procesamiento en paralelo de streams

- La principal ventaja de los streams es que especificamos **las operaciones** que queremos que se realicen, pero no especificamos **cómo deben realizarse**.
- **Por defecto** las operaciones se ejecutan de **forma secuencial** en el hilo de ejecución, pero podemos pedir que se ejecuten en **paralelo**

Procesamiento en paralelo de streams

- Partiendo de un stream se puede obtener su **stream paralelo**
- Todas las operaciones que se ejecutarán de forma automática en paralelo, **dividiendo** las tareas en **diferentes hilos**

```
Stream<String> s = nombres.parallelStream();
```

```
Stream<String> s = nombres.stream().parallel();
```

Procesamiento en paralelo de streams

- **Ejemplos de streams paralelos:**

```
List<Integer> even = numbers.stream()
      .filter(n -> n % 2 == 0)
      .sorted()
      .collect(Collectors.toList());
```

Secuencial
808ms

```
List<Integer> even =
numbers.parallelStream()
      .filter(n -> n % 2 == 0)
      .sorted()
      .collect(Collectors.toList());
```

Paralelo
507ms

Procesamiento en paralelo de streams

- Por defecto se usan tantos hilos como procesadores tiene el sistema, porque se asume que las operaciones no realizan **ninguna operación bloqueante**
- Si se quiere usar un número diferente de hilos:

```
List<Integer> even =
new ForkJoinPool(NUM_THREADS).submit(()->
   numbers.stream()
      .filter(n -> n % 2 == 0)
      .sorted()
      .collect(Collectors.toList()
).get();
```

Procesamiento en paralelo de streams

- **Cuidado al usar streams paralelos**
  - Paralelizar un algoritmo tiene un **coste de gestión** asociado (división del trabajo, gestión de los hilos, consolidación de resultados, etc…)
  - Para **pocos datos** o para ciertos tipos de algoritmos, **no merece la pena** paralelizar porque sería más lento.
  - Es importante **comparar** la versión secuencial y la paralela para unos datos de entrada habituales

https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html