Sesión 2:
Tipos de datos y colecciones

Índice

Tipos de datos y colecciones

- Introducción
- Enumeraciones e iteradores
- Colecciones
- Wrappers de tipos básicos
- Otras clases útiles

Introducción

- **Java proporciona un amplio conjunto de clases útiles para desarrollar aplicaciones**
- **Podemos encontrar este conjunto de clases (API) en:**

  http://java.sun.com/j2se/1.5.0/docs/api/

  http://java.sun.com/javase/6/docs/api/

- **En esta sesión veremos algunos grupos de ellas:**
  - **Clases útiles para crear y gestionar diferentes tipos de colecciones**
  - **Clases para recorrer, ordenar y manipular las colecciones**
  - **Otras clases útiles para desarrollar aplicaciones**

Tipos de datos y colecciones

- Introducción
- Enumeraciones e iteradores
- Colecciones
- Wrappers de tipos básicos
- Otras clases útiles

Enumeraciones e iteradores

- **Las enumeraciones y los iteradores no son tipos de datos en sí, sino objetos útiles a la hora de recorrer diferentes tipos de colecciones**
- **Con las *enumeraciones* podremos recorrer secuencialmente los elementos de una colección, para sacar sus valores, modificarlos, etc**
- **Con los *iteradores* podremos, además de lo anterior, eliminar elementos de una colección, con los métodos que proporciona para ello.**

Enumeraciones

- La interfaz *Enumeration* permite consultar secuencialmente los elementos de una colección
- Para recorrer secuencialmente los elementos de la colección utilizaremos su método *nextElement*:

```
Object item = enum.nextElement();
```

- Para comprobar si quedan más elementos que recorrer, utilizamos el método *hasMoreElements*:

```
if (enum.hasMoreElements()) ...
```

Enumeraciones

- **Con lo anterior, un bucle completo típico para recorrer una colección utilizando su enumeración de elementos sería:**

```
// Coger la enumeración
Enumeration enum = coleccion.elements();
while (enum.hasMoreElements())
{
    Object item = enum.nextElement();
    ...// Convertir item al objeto adecuado y
       // hacer con él lo que convenga
}
```

Iteradores

- La interfaz *Iterator* permite iterar secuencialmente sobre los elementos de una colección
- Para recorrer secuencialmente los elementos de la colección utilizaremos su método *next*:

```
Object item = iter.next();
```

- Para comprobar si quedan más elementos que recorrer, utilizamos el método *hasNext*:

```
if (iter.hasNext()) ...
```

- Para eliminar el elemento de la posición actual del iterador, utilizamos su método *remove*:

```
iter.remove();
```

Iteradores

- **Con lo anterior, un bucle completo típico para recorrer una colección utilizando su iterador sería:**

```
// Coger el iterador
Iterator iter = coleccion.iterator();
while (iter.hasNext())
{
    Object item = iter.next();
    ...// Convertir item al objeto adecuado y
       // hacer con él lo que convenga, por ejemplo
    iter.remove();
}
```

Tipos de datos y colecciones

- Introducción
- Enumeraciones e iteradores
- Colecciones
- Wrappers de tipos básicos
- Otras clases útiles

Colecciones

- En el paquete *java.util*
- Representan grupos de objetos, llamados elementos
- Podemos encontrar de distintos tipos, según si sus elementos están ordenados, si permiten repetir elementos, etc
- La interfaz *Collection* define el esqueleto que deben tener todos los tipos de colecciones
- Por tanto, todos tendrán métodos generales como:
  - `boolean add(Object o)`
  - `boolean remove(Object o)`
  - `boolean contains(Object o)`
  - `void clear()`
  - `boolean isEmpty()`
  - `Iterator iterator()`
  - `int size()`
  - `Object[] toArray()`

Listas de elementos

- La interfaz *List* hereda de *Collection* para definir elementos propios de una colección tipo *lista*, es decir, colecciones donde los elementos tienen un orden (posición en la lista)
- Así, tendremos otros nuevos métodos, además de los de *Collection*:
  - `void add(int posicion, Object o)`
  - `Object get(int indice)`
  - `int indexOf(Object o)`
  - `Object remove(int indice)`
  - `Object set(int indice, Object o)`

Tipos de listas

- ***ArrayList*: implementa una lista de elementos mediante un array de tamaño variable**
  - ***NO sincronizado***
- ***Vector*: existe desde las primeras versiones de Java, después de acomodó al marco de colecciones implementando la interfaz *List*.**
  - ***Similar a ArrayList, pero SINCRONIZADO. Tiene métodos anteriores a la interfaz List:***
  - `void addElement(Object o) / boolean removeElement(Object o)`
  - `void insertElementAt(Object o, int posicion)`
  - `void removeElementAt(Object o, int posicion)`
  - `Object elementAt(int posicion)`
  - `void setElementAt(Object o, int posicion)`
  - `int size()`
- ***LinkedList*: lista doblemente enlazada. Util para simular pilas o colas**
  - `void addFirst(Object o) / void addLast(Object o)`
  - `Object getFirst() / Object getLast()`
  - `Object removeFirst() / Object removeLast()`

Conjuntos

- **Grupos de elementos donde no hay repetidos**
- **Consideramos dos objetos de una clase iguales si su método *equals* los da como iguales (*o1.equals(o2)* es *true*)**
- **Los conjuntos se definen en la interfaz *Set*, que, como *List*, también hereda de *Collection***
- **El método *add* definido en *Collection* devolvía un *booleano*, que en este caso permitirá saber si se insertó el elemento en el conjunto, o no (porque ya existía)**

Tipos de conjuntos

- ***HashSet*: los objetos del conjunto se almacenan en una tabla hash.**
  - ***El coste de inserción, borrado y modificación suele ser constante***
  - ***La iteración es más costosa, y el orden puede diferir del orden de inserción***
- ***LinkedHashSet*: como la anterior, pero la tabla hash tiene los elementos enlazados, lo que facilita la iteración**
- ***TreeSet*: guarda los elementos en un árbol**
  - ***El coste de las operaciones es logarítmico***

Mapas

- No forman parte del marco de colecciones
- Se definen en la interfaz *Map*, y sirven para relacionar un conjunto de claves (*keys*) con sus respectivos valores
- Tanto la clave como el valor pueden ser cualquier objeto
  - `Object get(Object clave)`
  - `Object put(Object clave, Object valor)`
  - `Object remove(Object clave)`
  - `Set keySet()`
  - `int size()`

Tipos de mapas

- ***HashMap*: Utiliza una tabla hash para almacenar los pares *clave=valor*.**
  - **Las operaciones básicas (*get* y *put*) se harán en tiempo constante si la dispersión es adecuada**
  - **La iteración es más costosa, y el orden puede diferir del orden de inserción**
- ***Hashtable*: como la anterior, pero SINCRONIZADA. Como *Vector*, está desde las primeras versiones de Java**
  - `Enumeration keys()`
- ***TreeMap*: utiliza un árbol para implementar el mapa**
  - ***El coste de las operaciones es logarítmico***
  - ***Los elementos están ordenados ascendentemente por clave***

Algoritmos

- **La clase *Collections* dispone de una serie de métodos útiles para operaciones tediosas, como ordenar una colección, hacer una búsqueda binaria, sacar su valor máximo, etc**
  - `static void sort(List lista)`
  - `static int binarySearch(List lista, Object objeto)`
  - `static Object max(Collection col)`
  - `...`

Tipos de datos y colecciones

- Introducción
- Enumeraciones e iteradores
- Colecciones
- Wrappers de tipos básicos
- Otras clases útiles

Wrappers

- Los tipos simples (*int*, *char*, *float*, *double*, etc) no pueden incluirse directamente en colecciones, ya que éstas esperan subtipos de *Object* en sus métodos
- Para poderlos incluir, se tienen unas clases auxiliares, llamadas *wrappers*, para cada tipo básico, que lo convierten en objeto complejo
- Estas clases son, respectivamente, *Integer*, *Character*, *Float*, *Double*, etc.
- Encapsulan al tipo simple y ofrecen métodos útiles para poder trabajar con ellos

Wrappers

- **Si quisiéramos incluir un entero en un *ArrayList*, lo podríamos hacer así:**

```
int a;
ArrayList al = new ArrayList();
al.add(new Integer(a));
```

- **Si quisiéramos recuperar un entero de un *ArrayList*, lo podríamos hacer así:**

```
Integer entero = (Integer)(al.get(posicion));
int a = entero.intValue();
```

Tipos de datos y colecciones

- Introducción
- Enumeraciones e iteradores
- Colecciones
- Wrappers de tipos básicos
- Otras clases útiles

Object

- Clase base de todas las demás
- Es importante saber las dependencias (herencias, interfaces, etc) de una clase para saber las diferentes formas de instanciarla o referenciarla
- Por ejemplo, si tenemos:

```
public class MiClase extends Thread implements List
```

- Podremos crear un objeto *MiClase* de estas formas:

```
MiClase mc = new MiClase();
Thread t = new MiClase();
List l = new MiClase();
Object o = new MiClase();
```

Object: objetos diferentes

- **También es importante distinguir entre entidades independientes y referencias:**

```
MiClase mc1 = new MiClase();
MiClase mc2 = mc1;
// Es distinto a:
MiClase mc2 = (MiClase)(mc1.clone());
```

- **El método *clone* de cada objeto sirve para obtener una copia en memoria de un objeto con los mismos datos, pero con su propio espacio**
  - **Deberemos redefinir este método en las clases donde lo vayamos a usar, para asegurarnos de que se crea un nuevo objeto con los mismos campos y los mismos valores que el original**

Object: comparar objetos

- **Cuando queremos comparar dos objetos entre sí (por ejemplo, de la clase *MiClase*), no se hace así:**

```
if (mc1 == mc2) ...
```

- **Sino con su método *equals*:**

```
if (mc1.equals(mc2)) ...
```

  - **Deberemos redefinir este método en las clases donde lo vayamos a usar, para asegurarnos de que los objetos se comparan bien**
  - **Notar que la clase *String*, es un subtipo de *Object* por lo que para comparar cadenas...:**

```
if (cadena == “Hola”) ...       // NO
if (cadena.equals(“Hola”)) ... // SI
```

Object: representar en cadenas

- **Muchas veces queremos imprimir un objeto como cadena. Por ejemplo, si es un punto geométrico, sacar su coordenada X, una coma, y su coordenada Y**
- **La clase *Object* proporciona un método *toString* para definir cómo queremos que se imprima un objeto. Podremos redefinirlo a nuestro gusto**

```
public class Punto2D {
    ...
    public String toString()
    {
          System.out.println("("+x+","+y+")");
    }
}
...
Punto2D p = ...;
System.out.println(p);  // Sacará (x, y) del punto
```

Properties

- Esta clase es un tipo de *Hashtable* que almacena una serie de propiedades, cada una con un valor asociado
- Además, permite cargarlas o guardarlas en algún dispositivo (fichero)
- Algunos métodos interesantes:

```
Object setProperty(Object clave, Object valor)
Object getProperty(Object clave)
Object getProperty(Object clave, Object default)

void load(InputStream entrada)
void store(OutputStream salida, String cabecera)
```

System

- Ofrece métodos y campos útiles del sistema, como el ya conocido *System.out.println*
- Otros métodos interesantes de esta clase (todos estáticos):

```
void exit(int estado)
void gc()
long currentTimeMillis()
void arrayCopy(Object fuente, int pos_fuente,
               Object destino, int pos_destino,
               int numElementos)
```

Runtime

- **Toda aplicación Java tiene una instancia de la clase *Runtime*, que se comunica con el entorno donde se ejecuta**
- **Accediendo a este objeto, podremos, por ejemplo, ejecutar comandos externos**

```
Runtime rt = Runtime.getRuntime();
rt.exec("mkdir pepe");
```

Math y otras clases

- La clase *Math* proporciona una serie de métodos (estáticos) útiles para diferentes operaciones matemáticas (logaritmos, potencias, exponenciales, máximos, mínimos, etc)
- Otras clases útiles son la clase *Calendar* (para trabajar con fechas y horas), la clase *Currency* (para monedas), y la clase *Locale* (para situarnos en las características de fecha, hora y moneda de una región del mundo)