Estructuras de datos y algoritmos

UNIVERSIDAD DE CANTABRIA

1. *Introducción*
2. Estructuras de datos lineales
3. Estructuras de datos jerárquicas
4. Grafos y caminos
5. Implementación de listas, colas, y pilas
6. Implementación de mapas, árboles, y grafos

UNIVERSIDAD DE CANTABRIA

1. Introducción

- 1.1 Estructuras de datos abstractas
- 1.2 Eficiencia de las estructuras de datos
- 1.3 Interfaces y herencia múltiple
- 1.4 Estructuras de datos genéricas
- 1.5 Colecciones
- 1.6 Iteradores
- 1.7 Relaciones de igualdad y orden

1.1 Estructuras de datos abstractas

UNIVERSIDAD DE CANTABRIA

Una estructura o tipo de datos abstracto (ADT) es una clase o módulo de programa que contiene:

- **datos privados, con una determinada estructura**
- **un conjunto de métodos públicos para manejar esos datos**

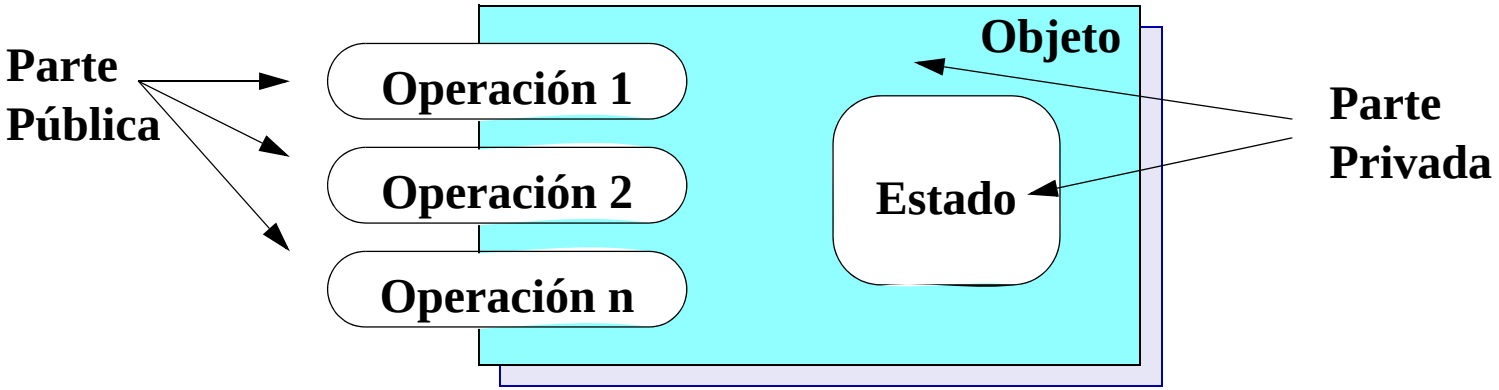
El conjunto de operaciones permite el uso de la estructura de datos sin conocer los detalles de su implementación

- **los programas que usan la clase son *independientes* de la forma en la que éste se implementa**
- **no es necesario conocer los detalles internos del tipo de datos ni de su implementación**

Encapsulamiento

Se dice que la clase *encapsula* el tipo de datos junto a sus operaciones, ocultando los detalles internos

- **consiguen la *reutilización* de código**
- **para muy diversas aplicaciones**



Por ejemplo, las listas o colas que estudiamos el año pasado

- **aunque necesitamos saber la eficiencia de los métodos**

1.2 Eficiencia de las estructuras de datos

UNIVERSIDAD DE CANTABRIA

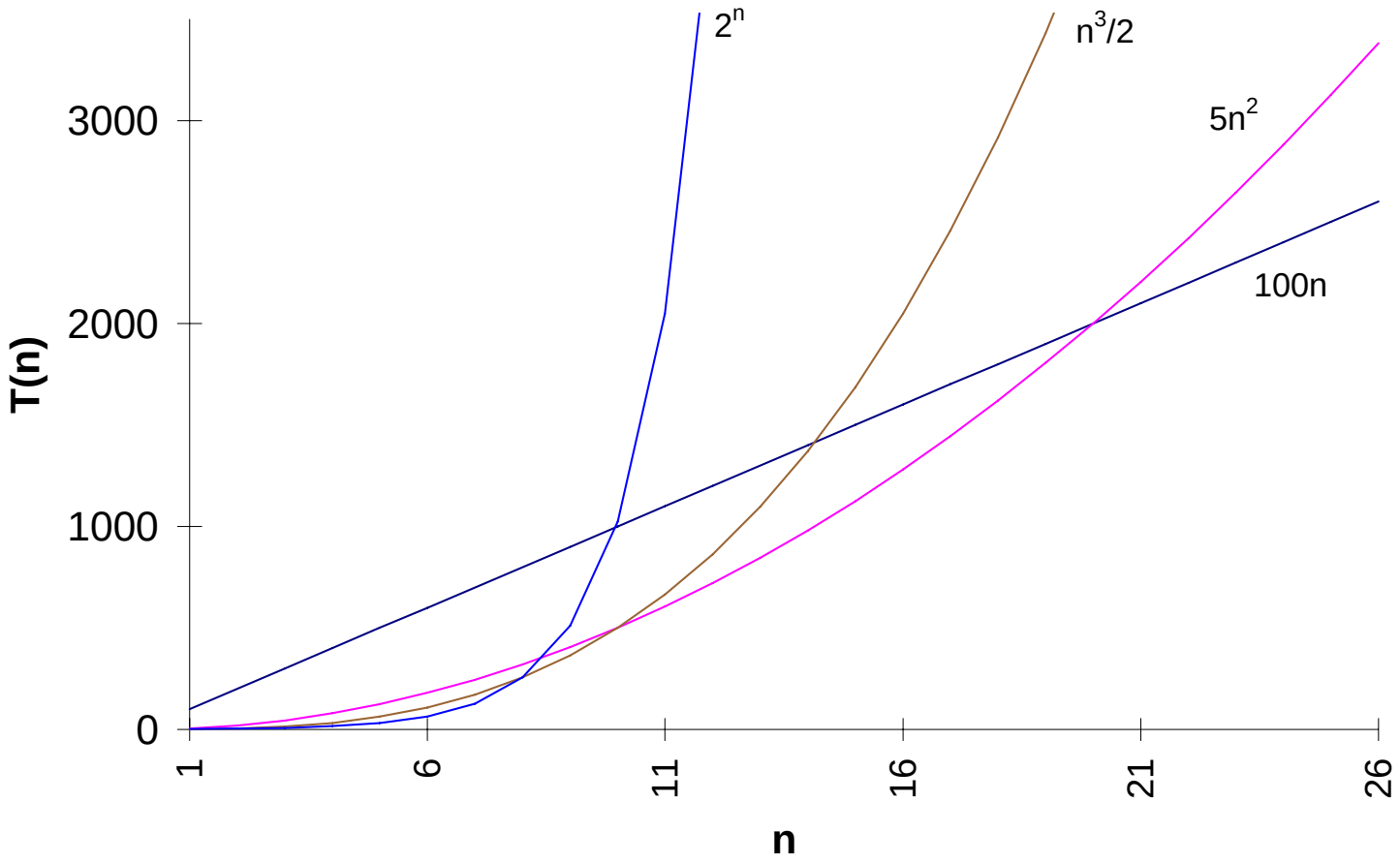
Entre los criterios a tener en cuenta al diseñar o elegir un algoritmo están su complejidad, y su tiempo de ejecución

El tiempo de ejecución depende de factores variados y, muy en particular, del tamaño del problema

El tiempo de ejecución puede ser:

- ***exacto*: es muy difícil de predecir; normalmente sólo se puede saber midiéndolo**
- ***predicción* del ritmo de crecimiento del tiempo de ejecución con respecto al tamaño del problema**

La "tiranía" del ritmo de crecimiento



La "tiranía" del ritmo de crecimiento (cont.)

Función	n=25	n=50
$100n$	2.5 seg	5.0 seg
$5n^2$	3.12 seg	12.5 seg
$n^3/2$	7.81 seg	62.5 seg
2^n	33554.43 seg	35677 años

La notación O(n)

UNIVERSIDAD DE CANTABRIA

El tiempo de ejecución depende no sólo de la cantidad de datos (n) sino también de cuáles son los datos; por ello distinguimos:

- tiempo de peor caso: T(n)
- tiempo promedio: $T_{avg}(n)$

Para expresar los ritmos de crecimiento se suele usar la notación O(n):

- decimos que T(n) es O(f(n)) si existen constantes c y n_0 tales que $T(n) \leq c \cdot f(n)$ para todo $n \geq n_0$

La notación O(n) muestra una cota superior al ritmo de crecimiento de un algoritmo

La notación O(n) (cont.)

UNIVERSIDAD DE CANTABRIA

El tiempo en la notación O(n) es relativo

- **Las unidades de T(n) son inespecíficas**

Por ejemplo, la función $T(n) = 3n^3+2n^2$ es $O(n^3)$.

- **Basta hacer c=5 y n_o=0 para comprobarlo,**

$$3n^3+2n^2 \leq 5n^3$$

En definitiva, cuando decimos que T(n) es O(f(n)), esto significa que f(n) es el límite de la velocidad de crecimiento de T(n)

La notación $\Omega(n)$

UNIVERSIDAD DE CANTABRIA

También se puede expresar un límite inferior al ritmo de crecimiento de T(n) mediante la notación $\Omega(n)$:

- **decimos que T(n) es $\Omega(g(n))$ si existe una constante c tal que $T(n) \geq c.g(n)$ para un número infinito de valores de n**

Por ejemplo, $T(n) = n^3 + 2n^2$ es $\Omega(n^3)$

- **Basta probar para c=1 y n>0.**

Recordar siempre que tanto O(n) como $\Omega(n)$ son medidas relativas, no absolutas

- **Por ejemplo, supongamos dos algoritmos cuyos tiempos de ejecución son $O(n^2)$ y $O(n^3)$**
- **¿Qué ocurre si las constantes son $100n^2$ y $5n^3$?**

Cálculo del tiempo de ejecución de un programa

UNIVERSIDAD DE CANTABRIA

Regla de las sumas:

- si T1(n) es O(f(n)) y T2(n) es O(g(n)), entonces
- T1(n)+T2(n) es O(max(f(n),g(n)))

Es decir, que la suma de los tiempos de ejecución de dos algoritmos tiene un ritmo de crecimiento igual al del máximo de los dos

Por ejemplo, para tiempos polinómicos $T(n)=an^p+bn^{p-1}+...$

- entonces T(n) es $O(n^p)$

Cálculo del tiempo de ejecución de un programa (cont.)

UNIVERSIDAD DE CANTABRIA

Regla de los productos:

- **si T1(n) es O(f(n)) y T2(n) es O(g(n)), entonces**
- **T1(n)·T2(n) es O((f(n)·g(n)))**

Es decir, que el producto de los tiempos de ejecución de dos algoritmos (p.e. cuando uno está anidado en el otro), tiene un ritmo de crecimiento igual al producto de los dos

Por ejemplo si T(n) es O(c·f(n)) entonces también es O(f(n)), ya que c es O(1)

Ritmos de crecimiento más habituales

UNIVERSIDAD DE CANTABRIA

1. O(1), o constante
2. O(log(n)), o logarítmico
3. O(n), o lineal
4. O(n·log(n))
5. $O(n^2)$, o cuadrático
6. $O(n^x)$, o polinómico
7. $O(2^n)$, o exponencial

Ritmos de crecimiento más habituales

n	1	log(n)	n·log(n)	n^2
128	1	7	896	16384
256	1	8	2048	65536
512	1	9	4608	262144
1024	1	10	10240	1048576
2048	1	11	22528	4194304
4096	1	12	49152	16777216
8192	1	13	106496	67108864
16384	1	14	229376	268435456
32768	1	15	491520	1073741824

Pistas para el análisis

UNIVERSIDAD DE CANTABRIA

Instrucciones *simples* (asignación y op. aritméticas): O(1)

***Secuencia* de instrucciones simples: O(max(1,1,1)) = O(1)**

Instrucción *condicional*:

- **si es un "if" simple y no se conoce el valor de la condición, se supone que la parte condicional se ejecuta siempre**
- **si es un "if" con parte "else" y no se conoce el valor de la condición, se elige la que más tarde de las dos partes**

***Bucle*: número de veces, por lo que tarden sus instrucciones**

Procedimientos *recursivos*: número de veces que se llama a sí mismo, por lo que tarda cada vez

Ejemplo de análisis

Analizar el tiempo de ejecución del siguiente algoritmo:

```
    método Burbuja (entero[1..n] A) es
        var entero temporal;
        fvar
(1)     para i desde 1 hasta n-1 hacer
(2)         para j desde n hasta i+1 hacer
(3)             si A(j-1) > A(j) entonces
                    // intercambia A(j-1) y A(j)
(4)                 temporal := A(j-1);
(5)                 A(j-1) := A(j);
(6)                 A(j) := temporal;
                fsi;
            fpara;
        fpara;
    fmétodo;
```

Ejemplo (cont.)

N es el nº de elementos a ordenar. Vamos a analizar el programa desde el interior hacia el exterior

- **Cada instrucción de asignación es O(1), independiente de la dimensión de entrada**
  - **(4), (5) y (6) son O(1) y por la regla de las sumas, la secuencia es O(max(1,1,1))=O(1)**
- **Si suponemos el peor caso en el "if", por la regla de las sumas el grupo de instrucciones (3)-(6) toma un tiempo O(1)**
- **El lazo que comienza en la línea (2) y abarca hasta la línea (6) tiene un cuerpo que toma un tiempo de la forma O(1) en cada iteración, y como toma n-i iteraciones, el tiempo gastado en ese lazo es O((n-i)·1) que es O(n-i).**

Ejemplo (cont.)

- El último lazo se ejecuta n-1 veces, de forma que el tiempo total de ejecución esta limitado superiormente por:

$$\sum_{i=1}^{n-1} (n-i) = \frac{1}{2}n(n-1) = \frac{n^2}{2} - \frac{n}{2}$$

- que es $O(n^2)$

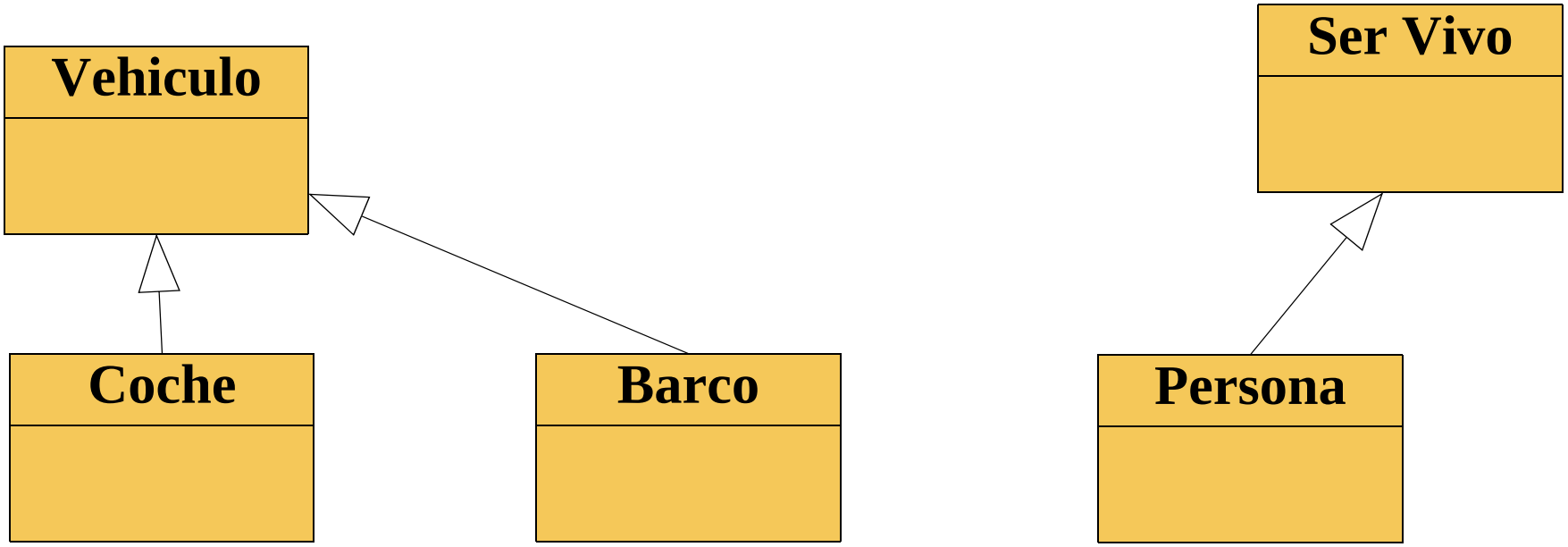
Ejemplo de análisis recursivo

```
método Factorial (Entero n) retorna Entero es
comienzo
(1)         if n <= 1 entonces
(2)              retorna 1;
            si no
(3)              retorna n*Factorial(n-1);
            fin if;
fin Factorial;
```

La función se llama a sí misma n veces, y cada ejecución es O(1), luego en definitiva factorial es O(n)

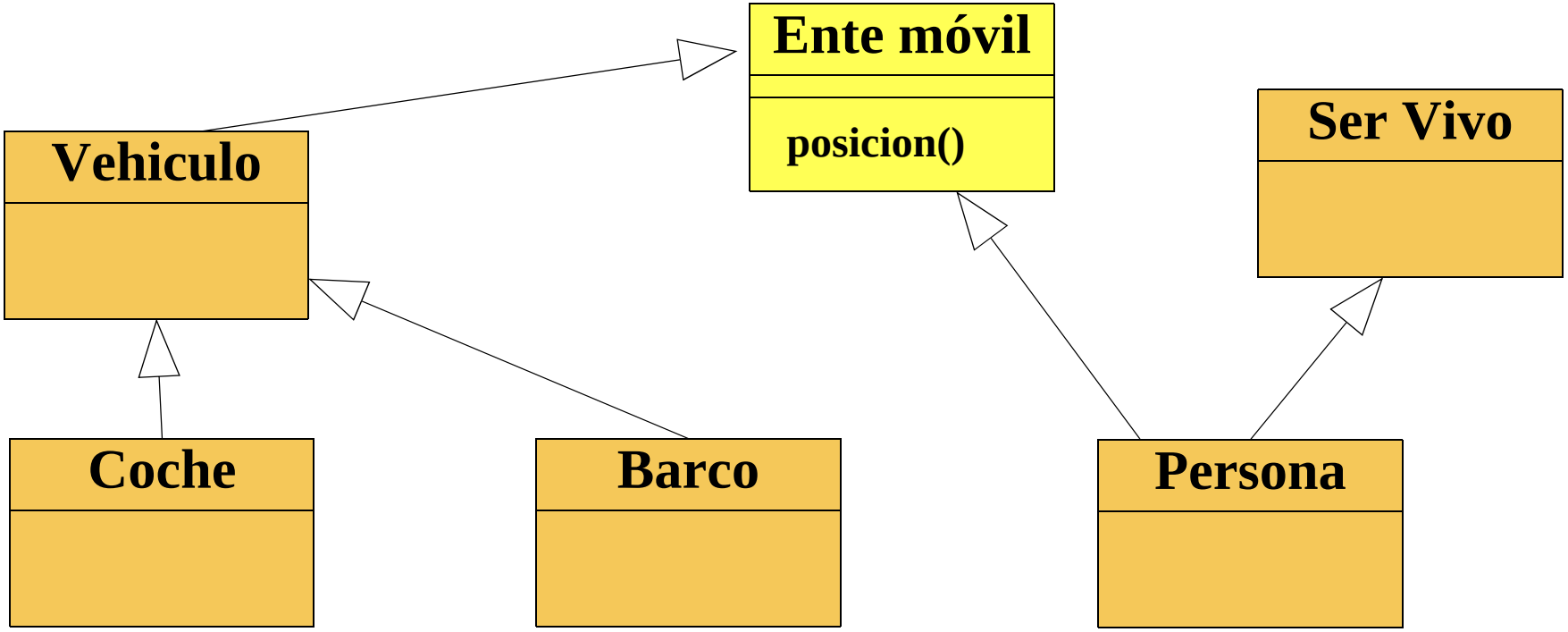
1.3 Interfaces y Herencia Múltiple

En el curso pasado hablamos de la extensión de clases por herencia:



Herencia múltiple

Podemos imaginar que podemos querer heredar de varias jerarquías, con herencia múltiple



Herencia Múltiple

UNIVERSIDAD DE CANTABRIA

La herencia múltiple sin restricciones presenta anomalías

- **atributos del mismo nombre**
- **métodos iguales con distinta implementación**

Es fácil equivocarse

- **se permite en C++**

En Java para la herencia múltiple se usa el mecanismo conocido como *Interfaz*

Interfaz Java

UNIVERSIDAD DE CANTABRIA

Representa una clase abstracta de la que no se pueden crear objetos

- **se usa sólo para herencia (incluida la herencia múltiple)**
- **no tiene estado**
  - **los atributos sólo son públicos y finales (constantes)**
- **los métodos son abstractos**
  - **no tienen instrucciones, sólo cabecera**
  - **están pensados para ser redefinidos**

La mejor forma de escribir en Java un ADT es con una *interfaz*

- **luego la extenderemos con implementaciones concretas escritas como *clases* Java**

Extensión de interfaces

UNIVERSIDAD DE CANTABRIA

Una clase puede

- **extender a una sola clase**
- **implementar varias interfaces**

Una interfaz puede

- **extender a otra interfaz**

Especificación de una interfaz

UNIVERSIDAD DE CANTABRIA

Es como una clase, pero usa la palabra `interface`

- los métodos son públicos, pero no se pone la palabra `public`

Ejemplo

```
public interface EnteMovil
{
    double posicion();
}
```

Implementación de la interfaz

```
public class Persona extends SerVivo
    implements EnteMovil
{
    public Persona(String nombre)
    {
        super(nombre);
    }

    /** Implementación del método posicion
     *  definido en la interfaz EnteMovil
     */
    public double posicion() {
        return pos;
    }
}
```

Uso de la interfaz

UNIVERSIDAD DE CANTABRIA

Uso polimórfico del método `posicion`, para cualquier objeto de una clase que implemente la interfaz `EnteMovil`

```
public static void prueba(EnteMovil e) {
    System.out.println
       ("Posicion="+e.posicion());
}
```

1.4 Estructuras de datos genéricas

UNIVERSIDAD DE CANTABRIA

La abstracción por genericidad se implementa en Java mediante módulos genéricos

- permiten hacer que un módulo sea independiente de una determinada clase de objetos
- esta clase queda indeterminada
- ejemplo de módulo genérico: una lista de objetos de una clase indeterminada

Al crear un objeto a partir de una clase genérica

- es preciso indicar la clase que quedó indeterminada mediante un parámetro genérico
- por ejemplo, podemos decir que los objetos de la lista son de la clase `Alumno`

Módulos genéricos (cont.)

UNIVERSIDAD DE CANTABRIA

Al usar el objeto

- puede usarse con objetos de la clase del parámetro genérico o de sus subclases
- por ejemplo, podemos almacenar en la lista, usando sus métodos, objetos de la clase `Alumno`, o de sus subclases

Declaración de módulos genéricos

```
public class Nombre <T> {...}
```

T es el parámetro genérico

Ejemplo

UNIVERSIDAD DE CANTABRIA

```
/**
 * Almacén de un objeto cualquiera
 *  (del tipo T indeterminado)
 */
public class Almacen <T>
{
    private T contenido;
    private boolean hayAlgo;

    /**Constructor que deja el almacén vacío */
    public Almacen()
    {
        hayAlgo=false;
    }
```

Ejemplo

```
/** Indica si el almacén tiene o no dato */
public boolean hayAlgo()
{
   return hayAlgo;
}

/** Meter un dato en el almacén */
public void almacena(T dato)
{
   contenido=dato;
   hayAlgo=true;
}
```

Ejemplo

```
    /** Sacar un dato del almacén */

    public T saca() throws NoHay {
      if (hayAlgo) {
         hayAlgo=false;
         return contenido;
      } else {
         throw new NoHay();
      }
    }
}
```

Uso de un módulo genérico

UNIVERSIDAD DE CANTABRIA

Para crear objetos a partir de una clase genérica hay que indicar cuál es el parámetro genérico

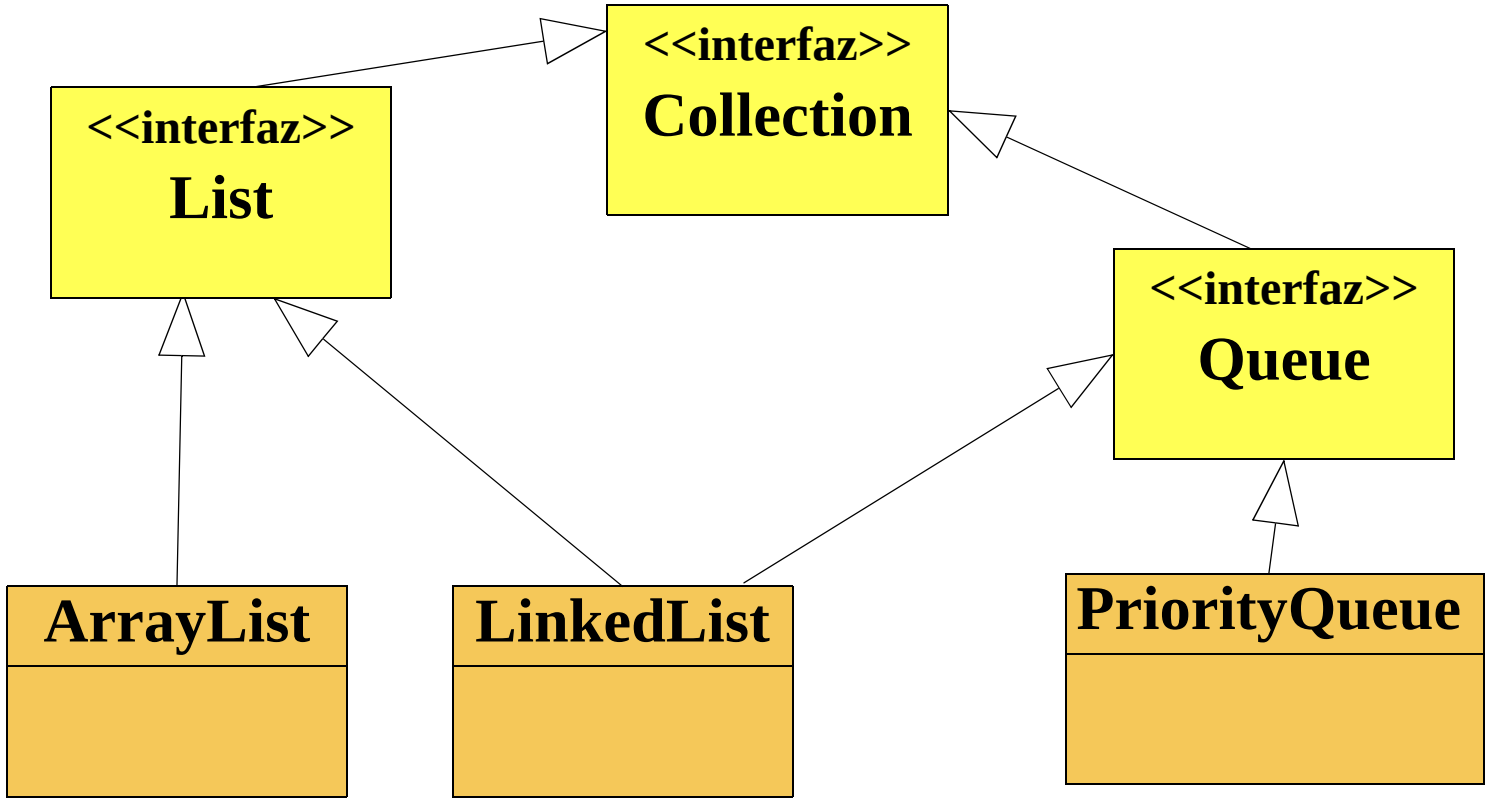
```
Almacen<Persona> alPer= new Almacen<Persona>();
Almacen<Animal>  alAni= new Almacen<Animal>();

Persona p= new Persona(...);
Animal  a= new Animal(...);

alPer.almacena(p);
alAni.almacena(a);
Persona sacado= alPer.saca();
```

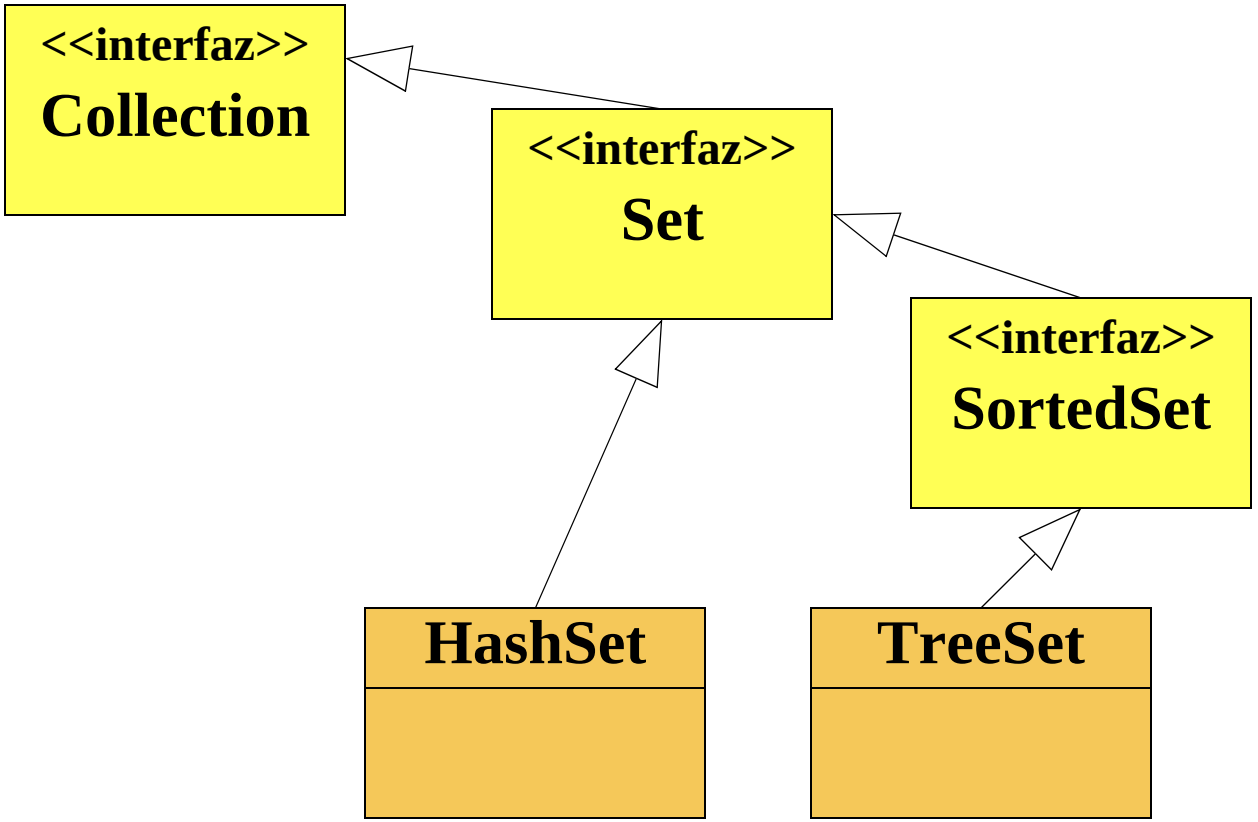
1.5 Colecciones

En Java hay diversas clases *genéricas* predefinidas para almacenar datos con diversas estructuras: *Java Collections*



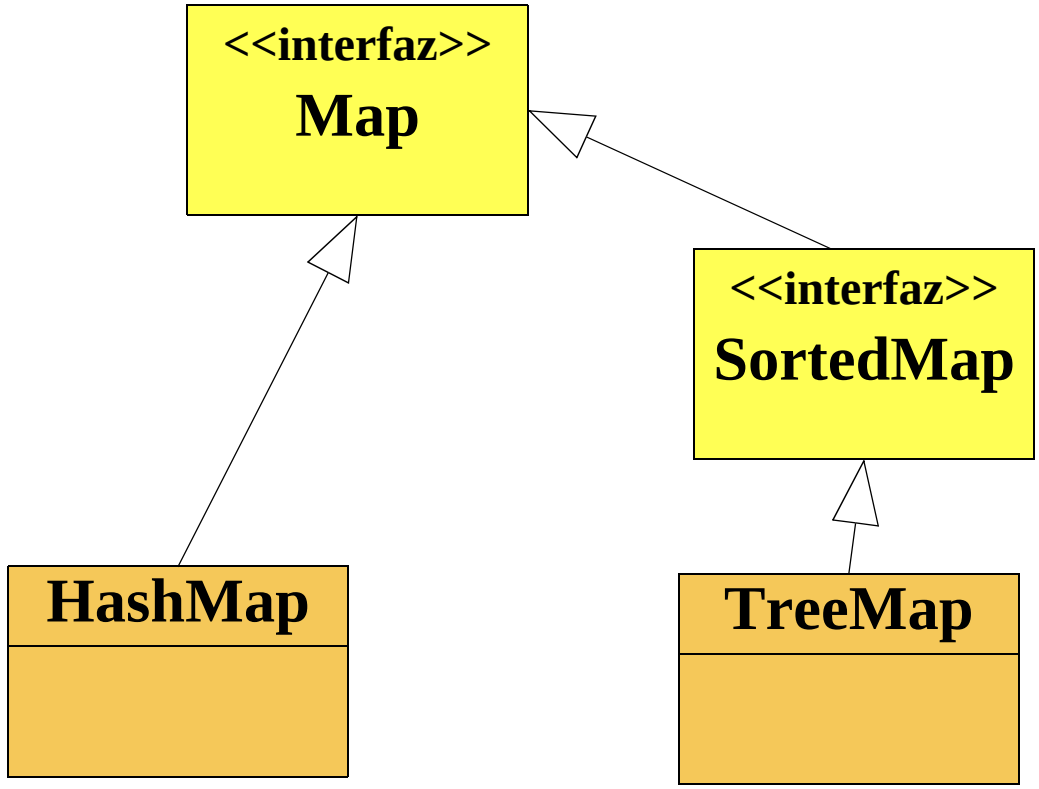
Colecciones (cont.)

Los conjuntos son otra extensión de la interfaz `Collection`



Colecciones (cont.)

Los mapas forman una jerarquía aparte dentro de las *Java Collections*



Colecciones (cont.)

Interfaz	Descripción
`Collection`	Colección de cero o más elementos, que se pueden añadir o quitar
`List`	Colección que admite elementos repetidos y en los que éstos tienen un orden establecido
`Queue`	Colección para el almacenamiento de datos intermedios, con operaciones adicionales de inserción y consulta
`Set`	Conjunto: no admite elementos repetidos
`SortedSet`	Conjunto en el que los elementos están en orden ascendente
`Map`	Contiene pares de valores {clave,valor}, para poder obtener un valor dada una clave
`SortedMap`	Mapa en el que las parejas se ordenan por sus claves

Las colecciones están en el paquete `java.util`

Colecciones (cont.)

UNIVERSIDAD DE CANTABRIA

Clase	Descripción
`ArrayList`	Tabla de tamaño que puede crecer. Antes se llamaba `Vector`
`LinkedList`	Lista de tamaño indefinido, con operaciones de inserción y eliminación que son O(1). También sirve para definir colas (FIFO) y pilas
`PriorityQueue`	Cola de prioridad, donde los elementos se almacenan ordenados según una relación de orden establecida al crearla
`HashSet`	Conjunto implementado con tablas de troceado (hash) que habitualmente tiene tiempos de inserción, eliminación y búsqueda que son O(1)
`TreeSet`	Conjunto implementado con árbol binario con operaciones de inserción, eliminación y búsqueda que son O(log n)
`HashMap`	Mapa con operaciones que habitualmente son O(1)
`TreeMap`	Mapa con operaciones que habitualmente son O(log n)

1.6 Iteradores

UNIVERSIDAD DE CANTABRIA

Un *iterador* definido sobre una colección nos permite visitar todos sus elementos uno por uno

- **permite implementar un algoritmo de *recorrido* o de *búsqueda***

El *iterador* se puede ver como una forma de recorrer una secuencia de elementos, disponiendo de un *cursor* o índice

- **el *cursor* está entre el elemento *previo* (el último visitado) y el *siguiente* (el que vamos a visitar), dentro del recorrido**
- **inicialmente no hay elemento *previo***
- **cuando se ha visitado el último elemento, no hay *siguiente***

La interfaz del iterador

```
public interface Iterator <T>
{
  /**
   * Retorna true si el iterador tiene siguiente
   * elemento
   */
    public boolean hasNext();

  /**
   * Retorna el siguiente elemento, y avanza el
   * cursor hacia adelante. Si no hay elemento
   * siguiente lanza NoSuchElementException
   */
  public T next();
```

La interfaz del iterador

UNIVERSIDAD DE CANTABRIA

```
   /**
    * Elimina el elemento previo de la colección
    * Lanza IllegalStateException si no hay
    * previo o si ya se había borrado desde la
    * ultima llamada a next()
    */
   public void remove();

}
```

Uso del iterador

UNIVERSIDAD DE CANTABRIA

Las colecciones Java implementan la interfaz `Iterable`

- **por ella disponen del método `iterator`() que se usa para obtener un iterador sobre la colección**

Ejemplo:

```
LinkedList <MiClase> lista= ...;
...
Iterator <MiClase> iter=lista.iterator();
```

Recorrido con el iterador

UNIVERSIDAD DE CANTABRIA

Para hacer un *recorrido* usando el iterador

```
import java.util.*;
public class Ruta
{
    // lista de coordenadas
    private LinkedList<Coordenada> lista;

    /**
     * Constructor; crea la lista vacía
     */
    public Ruta(){
        lista=new LinkedList<Coordenada>();
    }
```

Recorrido con el iterador (cont.)

```
    /**
     * Recorre la lista mostrando en pantalla
     * sus elementos
     */
    public void muestra() {
        Iterator<Coordenada> iter=
            lista.iterator();
        while (iter.hasNext()) {
            Coordenada c=iter.next();
            System.out.println(c.aTexto());
        }
    }
    ...
}
```

Ejemplos

UNIVERSIDAD DE CANTABRIA

Codificar un método para buscar un elemento en una colección y borrarlo

Codificar un método para borrar todos los elementos de una colección mostrándolos en pantalla según se borran

El iterador de listas

La interfaz `ListIterator` permite iterar sobre los elementos de una *lista* pudiendo recorrerla hacia adelante (`next`()) o hacia atrás (`previous`())

Se obtiene con el método `listIterator`()

Ejemplo

```
LinkedList <MiClase> lista= ...;
...
ListIterator <MiClase> iter=
                          lista.listIterator();
```

La instrucción "for-each"

UNIVERSIDAD DE CANTABRIA

El bucle `for` tiene un modo especial para recorrer todos los elementos de un array o de una colección

```
// Colección de textos
Collection <String> c= ...;

// recorre todos los elementos de c
for (String s : c) {
   System.out.println(s);
}
```

Este bucle, internamente, usa el iterador de la colección.

1.7 Relaciones de igualdad y orden

UNIVERSIDAD DE CANTABRIA

Todos los objetos java disponen del método `equals`() para comparar dos objetos

- definido en la superclase `Object`

```
public boolean equals(Object o)
```

Por omisión, este método compara las referencias de los objetos

- podemos redefinirlo para que haga lo que necesitemos

Nota: Debe redefinirse de manera consistente el método de troceado `hashCode`(), que estudiaremos más adelante

- Los objetos iguales deben tener códigos de troceado iguales

Propiedades de la igualdad

UNIVERSIDAD DE CANTABRIA

- el método `equals()` debe ser
  - *reflexivo*: `a.equals(a)` es `true`
  - *conmutativo*: si `a.equals(b)`, `b.equals(a)`
  - *asociativo*: si `a.equals(b)` y `b.equals(c)`, `a.equals(c)`
  - *consistente* (debe dar siempre el mismo valor si los datos no cambian)
  - si `a` no es `null`, `a.equals(null)` es `false`

Ejemplo

```
public class Coordenada {
    // atributos privados
    private double lat,lon; // grados
    /**
     * Redefinimos la comparación, para
     * que compare los contenidos */
    public boolean equals(Object o) {
        if (o instanceof Coordenada) {
            Coordenada c= (Coordenada) o;
            return this.lat==c.lat &&
                   this.lon==c.lon;
        } else {
            return false;
        }
    }
```

Ordenación

UNIVERSIDAD DE CANTABRIA

Dados unos valores con dos operaciones de igualdad (==) y orden (<)

- **una relación de orden total es aquella que cumple para cualesquiera valores x, y, z:**
  - **o bien x==y, ó x<y, ó y<x**
  - **x==x**
  - **si x==y entonces y==x**
  - **si x==y e y==z, entonces x==z**
  - **si x<y e y<z, entonces x<z**

Podemos ordenar unos valores si y solo si existe una relación de orden total

Comparación

UNIVERSIDAD DE CANTABRIA

La interfaz `Comparable` especifica una operación de comparación `compareTo`() asociada a los objetos de las clases que implementan esta interfaz

- `x.compareTo(y)` retorna un entero:
  - 0 si `x` es igual a `y`
  - <0 si `x` es menor que `y`
  - >0 si `y` es menor `x`
- debe imponer una relación de orden total
- debe ser consistente con `equals`()

Ejemplo

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;
public class Alumno
        implements Comparable <Alumno>
{
    private String nombre;
    private String dni;

    public boolean equals(Object o) {
        if (o instanceof Alumno) {
            return this.dni.equals(((Alumno)o).dni);
        } else {
            return false;
        }
    }
```

Ejemplo (cont.)

```
    /**
     * Comparación
     */
    public int compareTo(Alumno a) {
        return this.dni.compareTo(a.dni);
    }

}
```

Observar la asimetría:

- `equals` tiene como parámetro un `Object`
- `compareTo` tiene como parámetro un `Alumno`

Comparadores

UNIVERSIDAD DE CANTABRIA

En ocasiones es preciso definir varias relaciones de orden para valores de una clase

- **Por ejemplo, ordenar una lista de personas por:**
  - **DNI**
  - **apellido**
  - **fecha de nacimiento**

Un objeto sólo puede tener un método `compareTo`()

Pero puede tener varios *Comparadores*, que implementan la interfaz `Comparator`