E.T.S. de Ingenieros en Informática

Universidad Politécnica de Madrid

Algoritmos y Estructura de Datos

Autores: Pablo Nogueira Iglesias
Guillermo Román Díez

Grado en Ingeniería Informática
Grado en Matemáticas e Informática
Doble Grado en Ingeniería Informática y ADE

2 de Septiembre de 2020

Índice general

Importante

- Lo primero de todo decir que ESTO NO SON UNOS APUNTES COMPLETOS. Se trata de un *guion* elaborado por Pablo Nogueira y Guillermo Román, con los temas a tratar en las clases de teoría de la asignatura *Algoritmos y Estructuras de Datos*. Incluye referencias bibliográficas, definiciones y explicaciones útiles, ejemplos de código para analizar y discutir en clase, etc.
- El guión no contiene todo el material docente necesario para superar la asignatura. Son fundamentales las explicaciones y el desarrollo de código en clase, las transparencias, y las librerías de código descargables desde el Moodle.

1. Abstracción de Datos

1.1 Características del lenguaje Java para la abstracción de datos

1.1.1 Conceptos de programación imperativa que hay que tener claros

Comandos

Los comandos o mandatos afectan el estado del programa, es decir, el flujo de control o los valores de las variables. Son comandos:

- Las expresiones seguidas de ";". Por ejemplo: la asignación, la invocación de un método, la creación de un objeto con **new**, expresiones de incremento y decremento, etc. Algunas expresiones cambian el estado del programa.
- Declaraciones e inicializaciones de variables — no confundir estas últimas con asignaciones.
  La inicialización por defecto es sólo para atributos de clase (la hace el constructor por defecto de la clase). No hay inicialización por defecto para variables locales a un método o a un bloque. Éstas tienen que tener valor obligatoriamente antes de usarse. Para ello pueden inicializarse o asignarse antes de usarse en una expresión.
- Bloques: comienzan por "" y terminan con "". Se permiten bloques vacíos (aunque está feo en muchos casos) y anidados (*nesting*).
- Flujo de control:
  - Condicional: if-then, if-then[-else], switch. (¿Qué es if-else-if?)
  - Bucles: while, do-while, for.
  Equivalencia "while" y "do-while":

```
INIT                        INIT
while (COND) {              do {
  BODY                        BODY
}                           } while (COND);
```

  El "for" es un tipo de "while":

```
                              INIT
for (INIT ; COND ; POST) {    while (COND) {
  BODY                          BODY
}                               POST
                              }
```

- Otros: return, throw, try-catch-finally, assert.
  Todos ellos pueden anidarse, pero después de algunos no tiene sentido poner sentencias (el compilador se queja si las pones).
- Mandato vacío (";") (se puede usar en bucles)

Las variables tienen un tipo (*type*) y un ámbito (*scope*). El tipo se indica al declarar la variable. El ámbito es el trozo del programa en el que la variable es visible. Concepto importante: "shadowing" (ensombrecido). El lenguaje Java tiene unas reglas de ámbito y visibilidad establecidas. Algunos compiladores o entornos integrados como Eclipse pueden ser más restrictivos.

Ejercicio 1.1 Determina el ámbito de las distintas declaraciones de la variable 'x':

```
public class C {
  int x;

  public C(int x) { x = x; }

  public void m1(int x) {
    for (int x = 1; x < 10; x++) {
       x++;
    }
    if (x < 0) {
      int x = 1;
      this.x = x;
    }
  }

  public void m2(int c) {
    x = c;
  }
}
```

Entornos como "Eclipse" pueden protestar sin razón por algo como ésto:

```
int x; // no está incializada

if (2 == 1 + 1) {
  x = 1;
}

System.out.println(x);  // Eclipse se queja aquí
```

Expresiones

Las expresiones se evalúan a un valor de un tipo y se pueden asignar en variables que sean del mismo tipo. Son expresiones los literales, las constantes (entre ellas **`this`**, **`super`**, **`null`**), las variables (atributos de clase, parámetros de métodos, variables locales a un bloque), operadores aplicados a argumentos, la invocación de un método, el upcasting, etc. Por ejemplo, el siguiente fragmento es una expresión booleana:

```
(a = b = 1 + c * y.scale() - ((Automovil)Vehiculo).getPrice()) != 0
```

Hay dos familias de tipos: **primitivos** (o básicos) y **referencias** (arrays, clases, interfaces, enumerados).

- Las variables de *tipo básico* almacenan directamente el valor al que son inicializadas o asignadas.
- Las variables de *tipo referencia* almacenan una referencia o puntero al (es decir, la dirección de memoria del) objeto o array, cuyos datos están situados en otra parte de la memoria.

Operadores

Los **operadores** pueden ser aritméticos, lógicos (estrictos o en cortocircuito), el operador condicional, el operador de asignación, el operador "." de acceso a un campo de un objeto (atributo o método), el operador de acceso a un elemento de un "array", el operador **`new`**, **`instanceof`**, etc.

Recordad que los **operadores lógicos** evaluados en **cortocircuito** son diferentes a los operadores de la lógica convencional. `A && B` no es la conjunción de los predicados 'A' y 'B'. Su significado es "si A es cierto entonces el valor de `A && B` es el valor de B, pero si A es falso entonces el valor de `A && B` es falso". Más resumido:

`A && B` = "si A entonces B pero si no A entonces no A (y B o no B)"
`A || B` = "si A entonces A (y B o no B) pero si no A entonces B"

Entre otras cosas, esto permite en `&&` usar A como condición preliminar a la comprobación de B, y en `||` usar !A como condición preliminar a la comprobación de B.

Por ejemplo, si 'v' es null entonces 'v[i]' lanza 'NullPointerException' pero estas dos expresiones no lanzan la excepción. La primera es falsa si 'v' es null mientras que la segunda es cierta.

```
v != null && v[i] == 0
```

```
v == null || v[i] == 0
```

Veremos más abajo las leyes de *de Morgan* en cortocircuito.

Los operadores tienen una **precedencia** y una **asociación** (no confundir con *asociatividad* que es la propiedad de que la asociación del operador no afecta el resultado). Conviene conocer la tabla de precedencia y asociación de Java para en caso de que no haya paréntesis explícitos que indiquen dicha precedencia o asociación. La expresión anterior es equivalente a la siguiente según las tablas de precedencia y asociación de Java y los paréntesis son por tanto innecesarios, pero si ayudan a la comprensión del código es mejor ponerlos:

```
(a = (b = ((1 + (c * (y.scale())))-((Automovil)Vehiculo).getPrice()))) != 0
```

Ejercicio 1.2 Dibuja las variables como cajas y dibuja dentro los valores que toman durante la ejecución.

```
public static void main(String [] args) {
  int v1 = 3;
  int v2 = 5;
  Object o = new Object ();
  Object o2 = new Object ();

  v3 = v1;
  v1 = v2 = 6;
  v1 = 6 = v2;    // Error

  o = o2; // que pasa con o2? y el objeto al que apuntaba?
}
```

■

A continuación vemos un ejemplo de código para entender la asginación de variables de tipo **int**.

```
public static void main(String [] args) {
  int v1 = 3;
  int v2 = v1;

  if (v1 == v2) { // cierto
    System.out.println(''Las variables almacenan el mismo valor'');
  }
  v2 = 3;

  if (v1 == v2) { // cierto
    System.out.println(''Las variables almacenan el mismo valor'');
  }
  v2 = 4;

  if (v1 == v2) { // falso
    System.out.println(''Las variables almacenan el mismo valor'');
  }
}
```

Arrays

Los arrays, que también llamaremos "vectores" en el guión. (No confundirlos con los 'Vector<E>' de la Java Collection Framework.)

Ejercicio 1.3 Dibuja con cajas y flechas las variables y los arrays que resultan de estas declaraciones.

```
int [] a;

int [] a2 = null;

int [] b = new int[20];

b[0] = 7;
```

```
int [] c = new int[getIntegerFromUser()];

String [] d = new String[10];   // no hay ningún String todavía

d[0] = new String(''Hola'');

int [] e = new int[0];

int [] f = { };

int [] g = { 7, 9, 8 };

Clase [] h = new SubClase[5];
```

Es fundamental conocer la condición de parada o de continuación de un bucle y utilizarla para hacer código de mejor calidad. Aplicar leyes de *de Morgan*, aprovechar el cortocircuito, etc. Por ejemplo, el método indica si el array de enteros 'arr' tiene un 0.

```
public static boolean hayCero(int arr []) {
  int i;
  for (i = 0; i < arr.length && arr[i] != 0; i++)
    ; // El bucle no ejecuta nada
  return i < arr.length;
}
```

No es necesario que el bucle tenga ninguna sentencia. El bucle termina cuando el índice 'i' está fuera de rango (es mayor o igual que el tamaño del array) o bien cuando está en rango pero se encuentra el primer cero. Ambas condiciones son excluyentes.

```
i >= arr.length || i < arr.length && arr[i] == 0
```

La condición de terminación es la negada de la de continuación del bucle, la una se obtiene de la otra usando las **leyes de *de Morgan* en cortocircuito**:

```
! (A && B)          = !A || A && !B
! (!A || A && ! B)  = A && B
```

La condición `i < arr.length` es verdadera en el segundo caso donde también es verdadero que se ha encontrado el cero. Por tanto basta con devolver el valor de verdad de esa expresión.

Se prefiere el bucle anterior al siguiente que usa una variable booleana innecesaria. Se puede comenzar usando la variable booleana pero es preferible sustituir variables booleanas por las expresiones que significan. El uso de variables booleanas puede ser peligroso (al confundirlas) y hacer menos legible el programa. Este código sería una opción razonable:

```
boolean encontrado = false;
for (int i = 0; i < arr.length && !encontrado; i++) {
  encontrado = arr[i] == 0;
}
return encontrado;
```

Lo siguiente es equivalente pero algo peor:

```
boolean encontrado = false;
for (int i = 0; i < arr.length && !encontrado; i++) {
  if (arr[i] == 0) {
    encontrado = true;
  }
}
return encontrado;
```

Lo siguiente todavía peor:

```
boolean encontrado = false;
for (int i = 0; i < arr.length && !encontrado; i++) {
  if (arr[i] == 0) {
```

```
        encontrado = true;
      }
      else {
        encontrado = false
      }
    }
    return encontrado;
```

Y lo siguiente es directamente incorrecto:

```
    boolean encontrado = false;
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] == 0) {
        encontrado = true;
      }
      else {
        encontrado = false
      }
    }
    return encontrado;
```

No es recomendable el uso de **`break`**, **`continue`**, o **`return`** dentro de bucles porque entonces la condición de salida del bucle no es la negada de la condición de continuación. Además dificulta la legibilidad del código.

```
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] == 0) {
        return true;
      }
    }
    return false;
```

Puede no respetarse la invariante del bucle (la propiedad que se cumple en cada paso del bucle) y hay que leer todo el bloque para determinar todas las condiciones de salida que no están en la condición de terminación. Se complica la comprensión y el mantenimiento del código. En este ejemplo el código del bucle es corto pero hay bucles muy grandes. En la práctica no tiene por qué ser necesario leer el bloque. Dentro del bucle pueden modificarse muchas variables del programa, pero una vez terminado el bucle, se puede escribir junto con la negada de la condición de continuación la condición que se cumple sobre todas las variables afectadas por el bucle. El programador no necesita saber qué hace el bucle sino qué condiciones se cumplen antes y después.

En general, podríamos decir que la **estructura "típica"** de un método podría ser algo así:

```
  // método que busca m en array
  // PRE: m > 0 (por poner alguna)
  public boolean metodo (int m, int arr[]) {
    if (m <0) {  // Evaluamos la/s precondición/es del método
       throw new IllegalArgumentException ("m no es positivo");
    }

    if (arr == null || arr.length == 0) {  // Caso/s frontera
       return false;
    }

    int i = 0;
    while (i < arr.length && arr[i] != m) {
       i++;
    }

    return i < arr.length;
  }
```

Paso de parámetros

El paso de parámetros en **Java** siempre es **por valor**, es decir, se realiza una copia del valor de la variable que se pasa como parámetro por cada una de las llamadas a un método. El comportamiento es diferente dependiendo del tipo de la variable.

El siguiente fragmento de código intercambia el valor de dos variables utilizando una variable auxiliar:

```
public static void main(String [] args) {
  int x = 5;
  int y = 4;

  int tmp = x;    // codigo del intercambio ("swapping")
  x = y;
  y = tmp;
}
```

El siguiente método que pretende reutilizar el código del intercambio no funciona:

```
public static void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

public static void main(String [] args) {
  int x = 5;
  int y = 4;

  this.swap(x,y); // no funciona

  int z = 7;
  int k = 8;

  this.swap(z,k); // no funciona
}
```

Sin embargo, este método sí que intercambia correctamente los valores de `array[0]` y `array[1]`:

```
public static void swap(int array[]) {
  int tmp = array[0];
  array[0] = array[1];
  array[1] = tmp;
}
```

Esto se debe a que swap realmente no recibe una copia del array completo, sino que lo que está recibiendo es una copia del *puntero* al array, con lo que el contenido del array se modifica correctamente.

Esto puede producir algunos problemas típicos:

```
public static void creaObjeto(Object o) {
  o = new Object ();
}

void main (String [] args) {
  Object o = null;
  creaObjeto(o);
  // o apunta al nuevo objeto?
}
```

Clases, objetos y variables

Las clases definen los métodos y atributos (”fields”) en el argot de Java, que tendrán los objetos (instancias) de la clase. Por ejemplo:

```
public class Vehiculo {
  protected int ancho;
  protected int largo;
  protected int alto;

  /** Constructor por defecto */
  public Vehiculo() { ancho = largo = alto = 0; }

  public Vehiculo(int a, int l, int h) {
    ancho  = a;
    largo  = l;
```

```
    alto   = h;
  }

  /** Constructor de copia */
  public Vehiculo(Vehiculo v) {
    ancho = v.ancho;
    largo = v.largo;
    alto  = v.alto;
  }

  /** ''Getter'' */
  public int getDimension() { return ancho * largo * alto; }

  /** ''Setter'' */
  public void setAncho(int a) { ancho = a; }

  public boolean equals(Object o) {
    if (o == this) return true;
    if (o instanceof Vehiculo) {
      Vehiculo v = (Vehiculo) o; // downcasting
      return ancho == v.ancho && largo == v.largo && alto == v.alto;
    } else return false;  // tambien si o == null
  }
}
```

Los *objetos* (instancias) se crean en tiempo de ejecución con **new**. Las *variables* de tipo clase referencian objetos. Se puede cambiar el objeto al que referencia una variable mediante una asignación de ésta a otro objeto. El recolector de basura recogerá los objetos que no son referenciados por ninguna variable o posición de array, etc.

Ejercicio 1.4 Dibuja las variables, los objetos, y las flechas (referencias) entre variables y objetos que resultan de ejecutar el siguiente código:

```
public static void main(String [] args) {
  Vehiculo v1 = new Vehiculo(10,10,10);
  Vehiculo v2 = new Vehiculo(20,20,20);
  Vehiculo v3;

  v3 = v1;
  v1 = v2;
  v3 = new Vehiculo(10,10,10);
}
```

■

Ahora vamo a ver un ejemplo con objetos, variables, referencias, e igualdad:

```
public static void main(String [] args) {
  Vehiculo v1 = new Vehiculo(10,10,10);
  Vehiculo v2 = v1;

  if (v1.equals(v2))  // cierto
    System.out.println(''Los Vehiculos son iguales usando equals'');

  if (v1 == v2)       // cierto
    System.out.println(''Las variables referencian el mismo objeto'');

  v2 = new Vehiculo(10,10,10);

  if (v1.equals(v2))  // cierto
    System.out.println(''Los Vehiculos son iguales usando equals'');

  if (v1 == v2)       // falso
    System.out.println(''Las variables referencian el mismo objeto'');
}
```

En el argot técnico, se habla de **identidad** y **estado** de un objeto. Las variables referencian la identidad (generalmente la dirección de memoria donde está el objeto) mientras que los valores de los atributos (contenido) del objeto determinan su estado.

El método **`equals`** se usa para comparar el estado mientras que el operador **==** se usa para comparar identidades. Para expresiones de tipo primitivo sólo puede usarse '=='. Véase `http://www.javapractices.com/topic/TopicAction.do?Id=17` saber más sobre la dificultad de programar el método `equals`. El siguiente fragmento de código intercambia el valor de dos variables referencia utilizando una variable auxiliar:

```
Vehiculo x = new Vehiculo(10,10,10);
Vehiculo y = new Vehiculo(20,20,20);

Vehiculo tmp = x;    // intercambio (''swapping'')
x = y;
y = tmp;
```

El siguiente método que pretende reutilizar el código del intercambio no funciona:

```
public static void swap(Vehiculo a, Vehiculo b) {
  Vehiculo tmp = a;
  a = b;
  b = tmp;
}

public static void main(String [] args) {
  Vehiculo x = new Vehiculo(10,10,10);
  Vehiculo y = new Vehiculo(20,20,20);

  this.swap(x,y); // no funciona
}
```

Clases envoltorio

Clases envoltorio ("wrapper classes") 'Integer', 'Character', 'Boolean', etc, y sus conversiones automáticas ("autoboxing", "boxing", "unboxing"). Mira qué curioso lo que pasa en este `main`:

```
public static void main(String[] args) {
    Integer a, b;

    a = b = 3;
    System.out.println(''Case 1: `` + (a == b ? ``equal'' : ``different''));

    a = 3;
    b = 3;
    System.out.println(''Case 2: `` + (a == b ? ``equal'' : ``different''));

    a = b = 300;
    System.out.println(''Case 3: `` + (a == b ? ``equal'' : ``different''));

    a = 300;
    b = 300;
    System.out.println(''Case 4: `` + (a == b ? ``equal'' : ``different''));
}
```

Salida por pantalla:

```
Case 1: equal
Case 2: equal
Case 3: equal
Case 4: different
```

El puntero `this`

La palabra reservada **`this`** es una constante, no puede modificarse. Para recordar qué es **`this`** analiza éstos fragmentos de código, algunos de los cuales compilan pero son incorrectos y otros directamente ni compilan.

Ejercicio 1.5 ¿Cuál de estos fragmentos son incorrectos?

```
public class C {
  public int x;

  public C(int x) {
    this.x = x;
  }
}

public class C {
  public int x;

  public C(int y) {
    x = y;
  }
}

public class C {
  public int x;

  public C(int x) {
    x = x;
  }
}

public class C {
  public int x;

  public C(int x) {
    this.x = this.x;
  }
}

public class C {
  public int x;

  public C(C c2) {
    this = c2;
  }
}
```

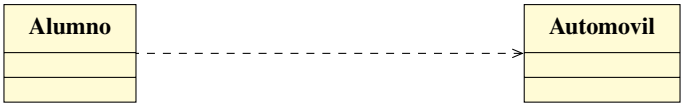
■

Composición: relación “tiene-un”

```
public class Automovil { ... }

public class Alumno {
  private Automovil a;
  ...
}
```

Un objeto de clase ’Alumno’ “tiene-un” objeto de clase `Automovil` como atributo. En el ejemplo anterior el atributo es privado, únicamente se puede acceder a él dentro de la clase Alumno.

Diagrama de clases que usaremos en éste guión:

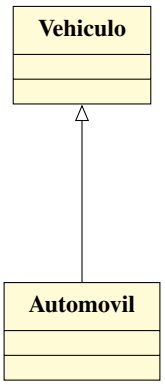


Herencia (”inheritance”): relación “es-un”

Veamos un ejemplo típico de herencia

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
```

Diagrama de clases UML:



Un objeto de clase `Automovil` también "es-un" objeto de clase 'Vehiculo'. La palabra clave "extends" dice que la clase `Vehiculo` extiende la clase 'Vehiculo', la hace más específica aportando nuevos atributos o métodos (o también sobrescribiendo métodos).

La herencia permite reusar código: un objeto de la clase `Vehiculo` tiene todos los atributos y métodos de la clase 'Vehiculo'

Hay muchos aspectos técnicos: subclase vs subtipo, leyes semánticas, monotonía, sobrescritura, reemplazo vs refinamiento, Principio de Sustitución de Liskov, co-varianza y contra-varianza, etc. Para más información, leed los libros de [Bud91] y [Mey88].

Upcasting

El upcasting es un mecanismo *de compilación* para convertir el tipo ("cast") yendo hacia arriba ("up") en la jerarquía de clases. La conversión la hace el compilador automáticamente. No es necesario que el programador escriba una expresión de "casting". Más concretamente:

- Se puede asignar un objeto de una subclase a una variable de tipo superclase:

  ```
  Vehiculo v = new Automovil();  // upcasting
  ```

  El tipo de una variable en tiempo de compilación es aquel con el que la variable ha sido declarada. En este ejemplo el tipo de 'v' es 'Vehiculo'.

  Pero en tiempo de ejecución, al crearse el objeto con 'new' la variable 'v' referencia un objeto de clase `Vehiculo`.

  La línea compila porque el compilador ha realizado la **conversión de tipo** de forma automática en la asignación: a la derecha del '=' se tiene una expresión que evalúa a un objeto de tipo `Vehiculo` y a la izquierda se tiene una variable de tipo 'Vehiculo'.

  Para el compilador, la variable 'v' es y será de tipo 'Vehiculo'. El **objeto no se ha convertido**. De hecho, no existe pues se crea en ejecución.

  En tiempo de ejecución todo objeto almacena internamente una representación de su tipo.
- La variable 'v' es una *variable polimórfica*: puede referenciar objetos de clase 'Vehiculo' así como objetos de cualquiera de las subclases de 'Vehiculo'.
- También se puede hacer "upcasting" en el paso de parámetros. Por ejemplo, el siguiente método puede invocarse pasándole como parámetro un objeto de clase `Vehiculo`:

  ```
  public void método(Vehiculo v) { ... }
  ```
- Un método puede devolver un objeto de la subclase cuando su tipo de retorno es la superclase:

  ```
  public Vehiculo método() { return new Automovil(); }
  ```
- Se pueden combinar éstos dos aspectos. Supongamos que disponemos de los siguientes métodos:

  ```
  public Automovil m1() { return new Automovil(); }
  public Vehiculo  m2() { return new Automovil(); }
  public void      m3(Vehiculo v) { ... }
  ```

Los siguientes son todos "upcastings" válidos (asumid que 'o' es un objeto de la clase donde se definen los métodos 'm1', 'm2' y 'm3' anteriores):

```
Vehiculo v1 = o.m1();
Vehiculo v2 = o.m2();
o.m3(new Automovil());
```

Y ¿para qué sirve el "upcasting"? Tiene varios usos, pero el uso principal tiene que ver con el enlazado dinámico.

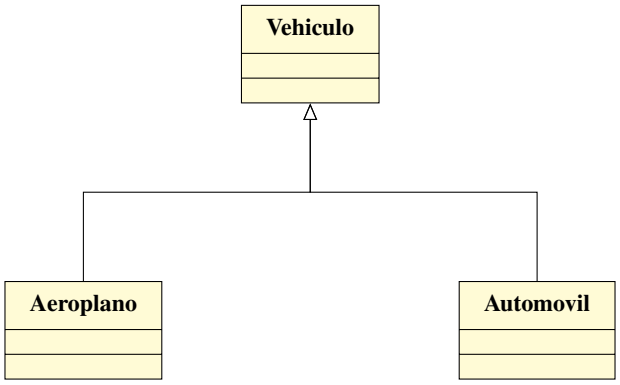
Upcasting

El upcasting es un Mecanismo *de compilación* para convertir el tipo ("cast") yendo hacia abajo ("down") en la jerarquía de clases. La conversión debe especificarla explícitamente el programador anotando el tipo deseado entre paréntesis (una expresión de "casting").

Supongamos que tenemos:

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
public class Aeroplano extends Vehiculo { ... }
```

Diagrama de clases:



Veamos:

```
Vehiculo v = new Automovil(); // upcasting

Automovil a1 = v;  // no compila
```

Para convertir el tipo en tiempo de compilación deberíamos tener la garantía de que **en tiempo de ejecución** la variable 'v' referencia un objeto de clase `Vehiculo` y no de clase 'Aeroplano'.

En este ejemplo se da el caso, pero en general en **tiempo de compilación** el compilador no puede determinar el tipo del objeto al que referenciará una variable **en tiempo de ejecución**. Más concretamente, un compilador no puede para todos los casos determinar los valores que tendrán las variables en tiempo de ejecución para todos los posible programas. ¡Para eso hay que ejecutar los programas! Esto es así variables de cualquier tipo.

Es responsabilidad del programador indicar la conversión explícitamente. El "downcasting" es decirle al compilador "tranquilo, sé lo que hago".

Si el programador se equivoca, puesto que el compilador no puede ayudarle, descubrirá su error en tiempo de ejecución.

Algunos ejemplos de downcasting:

```
Vehiculo v   = new Automovil(); // upcasting

Automovil a2 = (Automovil) v;   // downcasting, compila y correcto

Aeroplano p2 = (Aeroplano) v;   // downcasting, compila pero incorrecto
```

Al ejecutar el programa el último "casting" lanzará la excepción 'ClassCastException'.

Java ofrece **`instanceof`** al programador para averiguar el "Run-Time Type Information" (RTTI), es decir, para preguntar el tipo en tiempo de ejecución del objeto. Como hemos dicho antes, los objetos almacenan una representación de su tipo.

```
Vehiculo v = new Automovil(); // upcasting

if (v instanceof Automovil)

  Automovil a = (Automovil) v;

else if (v instanceof Aeroplano)

  Aeroplano p = (Aeroplano) v;
```

Pero **instanceof** tiene dos *limitaciones*:

1. No fuerza a que la pregunta y la acción estén coordinadas. El siguiente código erróneo compila con consecuencias desastrosas al ejecutar:

```
Vehiculo v = new Automovil(); // upcasting

if (v instanceof Automovil)
  Aeroplano p = (Aeroplano) v;
```

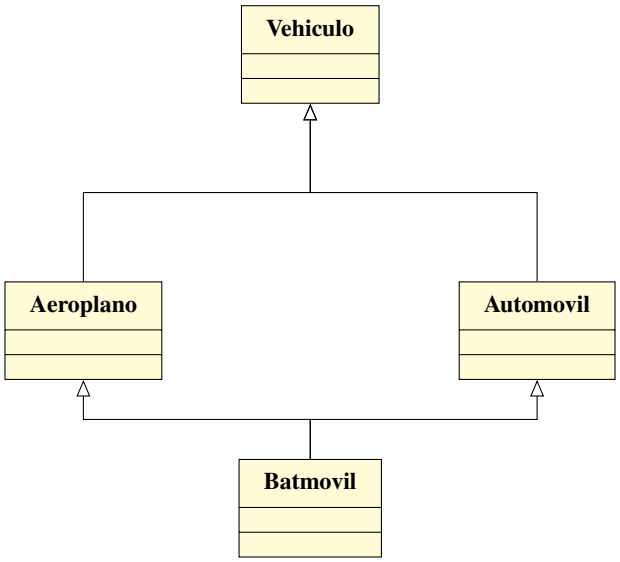
   En otros lenguajes sí se fuerza. Por ejemplo C++ ofrece un método genérico "dynamic cast" que recibe una clase y un objeto y pregunta si el objeto es una instancia de la clase. En caso positivo devuelve el objeto tras un casting a dicha clase. En caso negativo, devuelve null.
2. Se pueden crear modularmente (sin tener que recompilar las clases ya compiladas) nuevas subclases de la superclase. Pero el código donde está "instanceof" no puede extenderse modularmente para tener en cuenta esos nuevos casos. Por cada nueva clase hay que editar el código, añadir nuevas ramas "if" y recompilar. Este problema se llama "Expression Problem" y hay soluciones, algunas muy buenas en otros lenguajes.

Herencia múltiple y "Diamond of Death"

Supongamos que tenemos la siguiente herencia:

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
public class Aeroplano extends Vehiculo { ... }
public class Batmovil  extends Automovil, Aeroplano { ... }
  // La ultima linea no compila en Java
```

Se produce el "Diamond of Death":



Las clases `Vehiculo` y 'Aeroplano' heredarían los atributos y métodos públicos y protegidos de 'Vehiculo'. ¿De quién los hereda 'Batmóvil' el atributo 'ancho' común a 'Vehiculo', `Vehiculo` y 'Aeroplano'? Hay una ambigüedad que debe resolverse.

Java no permite herencia múltiple Solución de Java: se introducen los **interfaces** para permitir herencia múltiple. Los interfaces no definene atributos ni código, unicamente nombres de métodos. Los veremos posteriormente.

Esta es la motivación principal de los interfaces, y no la separación entre especificación e implementación para TADs, aunque hay una relación débil, como veremos más adelante.

Enlazado dinámico y sobrescritura

La segunda utilidad es poder invocar un método sobre una variable polimórfica de forma que el método seleccionado dependa del tipo del objeto que referencia la variable en tiempo de ejecución. Un ejemplo:

```
public static void main (String [] args) {
  Vehiculo v[] = new Vehiculo[3];
  v[0] = new Automovil();
  v[1] = new Aeroplano();
  v[2] = new Barco();

  for (int i = 0; i < v.length; i++)
    v[i].display();    // dibuja el objeto en pantalla
}
```

El objetivo es poder utilizar una variable de tipo superclase que referencie objetos de subclases (por tanto una variable polimórfica) e invocar un método *sobrescrito* en las subclases de forma que el comportamiento se adecúe a lo esperado para las subclases.

El ejemplo asume que el método `display` está definido en la clase 'Vehiculo'. Supongamos que lo que dibuja es un cubo del tamaño de las dimensiones dadas por los atributos 'ancho', etc.

Si las clases `Automovil`, `Aeroplano` y `Barco` no sobrescriben el método `display` para dibujar respectivamente un Automovil, un aeroplano y un barco, entonces en tiempo de ejecución se invocará el método `display` heredado de `Vehiculo`. Es decir, se dibujarán tres cubo. Pero si todas sobrescriben adecuadamente el método `display` entonces se dibujará respectivamente un Automovil, un aeroplano y un barco.

Si se añaden nuevas subclases a `Vehiculo` entonces se podrían almacenar en el vector objetos de las nuevas subclases. Pero habría que modificar las inicializaciones para almacenar los objetos y habría que recompilar. Como en el caso de **`instanceof`**, se pueden añadir nuevas clases de forma modular pero el código ya escrito no puede extenderse de forma modular—sin utilizar trucos, claro.

Terminología

- **Sobrescritura** (*overriding*): Cuando una subclase redefine un método o atributo de una superclase. (Debe usarse la anotación `@Override` de Java sobre las cabeceras de los atributos y métodos sobrescritos.)
- **Sobrecarga**: Hay que diferenciar la sobrescritura (*overriding*) de la sobrecarga ("overloading"). No son lo mismo y es fundamental comprender la diferencia entre ambas. La sobrecarga implica tener más de un método con el mismo nombre, pero con distintos parámetros y para esto no es necesario que haya herencia.
- **Enlazado dinámico** (*dynamic dispatching*): Dada una variable que referencia un objeto, a través de la cual se accede a un atributo público o se invoca un método del objeto, qué atributo se accede y qué método se invoca está determinado por el tipo **en tiempo de ejecución** del objeto al que referencia la variable, no por el tipo con que ésta se declara en tiempo de compilación. Esto es así para atributos o métodos sobrescritos.
  Es decir, en

  ```
  Vehiculo v = new Automovil();
  v.display();
  ```

  El método 'display' invocado será el de la clase `Vehiculo` si dicho método se herada de 'Vehiculo' y se sobrescribe en `Vehiculo`.
  Este concepto no tiene nada que ver con otro de similar nombre: el enlazado dinámico de librerías (*dynamic linking*).
- **Enlazado dinámico** no es la traducción más precisa del Inglés. Por un lado tenemos *dynamic dispatching* (elegir la implementación de un método en tiempo de ejecución) y por otro *dynamic binding* (también llamado *late binding*: asociar un nombre en tiempo de ejecución). "Bind" se suele traducir por "enlazar o asociar". En Java, la asociación del nombre del método se hace en tiempo de compilación pero la elección de la implementación del método se hace en tiempo de ejecución.
  Más concretamente, la asociación de un método a una variable polimórfica se fija en tiempo de compilación. Por ejemplo, `v.display()` compila si el método `display` está definido en la clase `Vehiculo`. En tiempo de ejecución se elige la implementación del método 'display' de la clase del objeto al que referencia 'v'.

Enlazado dinámico

Notad que no hay enlazado dinámico sin sobrescritura

```
public class Vehiculo {
  public void display() { ... }
}
public class Automovil extends Vehiculo {
  public int matricula() { ... }  // método que no esta en 'Vehiculo'
}


Vehiculo v = new Automovil(); // upcasting
v.display();     // compila, invoca el método 'display' heredado.
v.matricula();   // no compila: Vehiculo no tiene un método 'matricula'

Automovil a = new Automovil();
a.display();    // compila, invoca el método 'display' heredado.
a.matricula();  // compila, invoca el método 'matricula' de Vehiculo
```

En `v.display()` la variable 'v' almacena un `Vehiculo`. Por tanto se invoca al objeto `Automovil` el método heredado de 'Vehiculo'. El comportamiento será el mismo que si en 'v' hubiera un objeto de clase `Vehiculo`: se dibujará un cubo.

```
public class Vehiculo {
  public void display() { ... }
}
public class Automovil extends Vehiculo {
  @Override
    public void display() { ... }   // sobrescribe el de 'Vehiculo'

  public int matricula() { ... }  // método que no esta en 'Vehiculo'
}
```

Suponemos que el método `display` de `Automovil` se sobrescribe para que dibuje un `Automovil`.

```
Vehiculo v = new Automovil(); // upcasting
v.display();   // compila, invoca el método sobrescrito, no el heredado.
v.matricula(); // no compila, 'Vehiculo' no tiene un método 'matricula'

Automovil a = new Automovil();
a.display();    // compila, invoca el método sobrescrito, no el heredado.
a.matricula();  // compila, invoca el método 'matricula' de Automovil
```

En `v.display()` la variable 'v' almacena un `Automovil`. Por tanto se invoca al objeto `Automovil` su método `display` sobrescrito. La invocación `v.display()` dibujará el `Automovil`.

El comportamiento no es el mismo que si 'v' referenciara un `Vehiculo` pues en este caso se dibujaría un cubo. El comportamiento es el mismo que el de la invocación `a.display()` posterior.

Repetimos: La idea es utilizar una variable de tipo superclase a la que se asignan objetos de subclases. A esa variable se le invoca un método común sobrescrito en las subclases.

¿Se podría conseguir lo mismo usando "downcastings"?

Uno puede preguntarse si con "downcastings" sería suficiente y no haría falta el enlazado dinámico.

```
public class Vehiculo {
  public void display() { ... }
}
public class Automovil extends Vehiculo {
  @Override
    public void display() { ... }
}
public class Aeroplano extends Vehiculo {
  @Override
    public void display() { ... }
}
```

La sobrescritura es necesaria para que el compilador acepte invocar el método 'display' a una variable de tipo `Vehiculo`.

Podríamos pensar que con "downcastings" es suficiente:

```
public static void main (String [] args) {
  Vehiculo v[] = new Vehiculo[2];
  v[0] = new Automovil();
  v[1] = new Aeroplano();

  ((Automovil)v[0]).display();  // dibuja un Automovil
  ((Aeroplano)v[1]).display();  // dibuja un aeroplano
}
```

El problema es que esto no nos permite escribir código uniforme. Por ejemplo no podemos invocar `display` en un bucle:

```
public static void main (String [] args) {
  Vehiculo v[] = new Vehiculo[2];
  v[0] = new Automovil();
  v[1] = new Aeroplano();

  for (int i = 0; i < v.length; i++)
    ((QUE_CASTING_PONGO_AQUI?)v[i]).display();
}
```

El enlazado dinámico permite no tener que utilizar "downcastings".

La herencia y el enlazado dinámico tienen algunos peligros. El siguiente ejemplo ilustra el comportamiento del enlazado dinámico que puede llevar a confusión si no se conoce. Se tiene una clase 'Padre' con un método 'ricosuave' que invoca los métodos 'rico' y 'suave' de dicha clase. (Los alumnos nacidos después de 1990 probablemente no reconocerán la alusión a la "curiosa" canción del peculiar rapero Gerardo Mejía.) La clase 'Hijo' extiende la clase padre sobrescribiendo todos los métodos, refinándolos. La clase 'Main' crea objetos de clase 'Padre' y de clase 'Hijo'. La variable 'phijo' es de tipo 'Padre' pero referencia en ejecución un objeto de clase 'Hijo'.

```
public class Padre {

  public void rico() {
    System.out.print(''Rico. ``);
  }

  public void suave() {
    System.out.print(''Suave.'');
  }

  public void ricosuave() {
    this.rico();
    this.suave();
  }
}

public class Hijo extends Padre {

  @Override
    public void rico() {
    System.out.print(''Muy ``);
    super.rico();
  }

  @Override
    public void suave() {
    System.out.print(''Muy ``);
    super.suave();
  }

  @Override
    public void ricosuave() {
```

```
    System.out.print(''Muy ``);
    super.ricosuave();
  }
}

public class Main {
  public static void main(String [] args) {
    Padre padre = new Padre();
    padre.ricosuave();
    System.out.println();

    Hijo hijo  = new Hijo();
    hijo.ricosuave();
    System.out.println();

    Padre phijo = new Hijo();
    phijo.ricosuave();
    System.out.println();
  }
}
```

La salida por pantalla **no** es:

```
Rico. Suave.
Muy Rico. Suave.
Muy Rico. Suave.
```

La salida por pantalla es:

```
Rico. Suave.
Muy Muy Rico. Muy Suave.
Muy Muy Rico. Muy Suave.
```

El comportamiento para 'hijo' y 'phijo' es el mismo, como cabe esperar por el enlazado dinámico.

Esta es la traza de ejecución de la invocación hijo.ricosuave():

- se muestra "Muy" por pantalla. - se invoca super.ricosuave() - se invoca this.rico() pero 'this' es el objeto al que referencia 'hijo' es decir, se invoca el método 'rico' de 'Hijo'. - se muestra "Muy" por pantalla. - se invoca super.rico() - se muestra "Rico" por pantalla. - se invoca this.suave() pero 'this' es el objeto al que referencia 'hijo' es decir, se invoca el método 'suave' de 'Hijo'. - se muestra "Muy" por pantalla. - se invoca super.suave() - se muestra "Suave" por pantalla.

Herencia y sobrecarga

Decimos que un atributo está sobrecargado cuando aparece declarado con distinto tipo en la misma clase. En esta asignatura no usaremos atributos sobrecargados. Un método está sobrecargado cuando aparece declarado en la misma clase con algún tipo distinto en sus parámetros. Por ejemplo:

```
public class Vehiculo {
  ...
  public int  display() { ... }
  public void display(Window w)  { ... }
}
```

La sobrecarga de atributos y métodos se resuelve en tiempo de compilación. Para los métodos, la sobrecarga se resuelve en función del tipo de los parámetros del método.

Cuando el programador escribe 'v.display(w)' con 'v' de tipo `Vehiculo` o subclase de `Vehiculo` entonces el compilador determina que tiene que invocarse el segundo método si 'w' es de clase 'Window' o de cualquier subclase de 'Window'.

Si no puede resolverse la sobrecarga porque hay ambigüedad, entonces el el compilador indica el problema y el programa no compila correctamente.

La sobrescritura es una especie de sobrecarga que se resuelve trivialmente mediante el ámbito y el enlazado dinámico.

```
public class Vehiculo {
  public void display(Window w)  { ... }
}
```

```
public class Automovil extends Vehiculo {
  @Override
    public void display(Window w) { ... }
  public int  display() { ... }
}
```

Cuando el programador escribe 'v.display(w)', qué método se invoca depende del tipo del objeto al que referencia 'v' en tiempo de ejecución.

Ejercicio 1.6 ¿Qué pasa si `Automovil` no sobrescribe el método 'void display(Window w)'? ■

Ejercicio 1.7 ¿Qué pasa con la invocación 'v.display()' si 'v' es de tipo `Vehiculo` pero referencia un objeto `Automovil`? ■

La interacción entre sobrecarga y herencia complica la resolución de la sobrecarga:

```
public class Vehiculo {
  public int  display(Vehiculo v)   { ... }
  public void display(Automovil a)  { ... }
}
```

¿Cómo se resuelve la sobrecarga en los siguientes casos (suponiendo que `Automovil` no sobrescribe ni sobrecarga `display`)?

```
Vehiculo  v = new Vehiculo();
Automovil a = new Automovil();
v.display(v);
v.display(a);

Vehiculo  v = new Automovil();
Automovil a = new Automovil();
v.display(v);
v.display(a);
```

■ **Ejemplo 1.1** Ejemplo de interacción entre sobrecarga y sobrescritura:

```
public class Vehiculo {
  public void chocar(Vehiculo v) { ... }
}

public class Automovil extends Vehiculo {
  public void chocar(Automovil a) { ... }
}

public class Main {
  public static void main (String args []) {

    Vehiculo v  = new Automovil();  /* upcasting */
    Automovil a = new Automovil();

    v.chocar(a);  // compila y
                  // se invoca chocar heredado de \lstinline{Vehiculo}!
  }
}
```

El método 'chocar' no está 'sobrescrito' en `Automovil`, está sobrecargado, pues `Automovil` hereda 'chocar' de `Vehiculo` y tiene por tanto dos métodos `chocar` que toman parámetros de distinto tipo. ■

La sobrecarga se resuelve en tiempo de compilación: 'v' es de tipo `Vehiculo` (aunque referencia un objeto de tipo `Automovil`). El compilador determina que se invoca el 'chocar' de `Vehiculo`, que en tiempo de ejecución es el 'chocar' que el objeto `Automovil` hereda de `Vehiculo`.

Ejercicio 1.8 Probar y explicar lo que sucede al cambiar el código del 'main' anterior según distintas combinaciones:

```
Vehiculo v  = new Vehiculo();
Automovil a = new Automovil();
v.chocar(a);
a.chocar(v);

Vehiculo v1 = new Vehiculo();
Vehiculo v2 = new Automovil();
v1.chocar(v2);
v2.chocar(v1);

Vehiculo v1 = new Automovil();
Vehiculo v2 = new Automovil();
v1.chocar(v2);
v2.chocar(v1);

Automovil a = new Automovil();
Vehiculo  v = new Automovil();
v.chocar(a);
a.chocar(v);

Automovil a1 = new Automovil();
Automovil a2 = new Automovil();
a1.chocar(a2);
a2.chocar(a1);

Vehiculo v1 = new Automovil();
Vehiculo v2 = new Automovil();
v1.chocar((Automovil)v2);
v2.chocar(v1);
```

Hay lenguajes en los que la herencia lleva pareja la "covarianza" del tipo de los parámetros de los métodos, que van hacia abajo en la jerarquía de interfaces/clases cuando éstos van hacia abajo. En Java no hay covarianza, pero si la hubiera:

```
public class Vehiculo {
  public void chocar(Vehiculo v) { ... }
}

public class Automovil extends Vehiculo {
  /* hereda 'void chocar(Automovil v)'
   * con el tipo del parametro 'v' yendo hacia abajo en la jerarquia
   */

  public void chocar(Automovil v) { ... }
  /* ya no seria una sobrecarga sino una sobrescritura */
}

public class Main {
  public static void main (String args []) {

    Vehiculo v  = new Automovil();  /* upcasting */
    Automovil a = new Automovil();

    v.chocar(a);  // compila, enlazado dinamico, invoca el sobrescrito
  }
}
```

Ejercicio 1.9 ¿Permite Java la covarianza en el tipo del valor de retorno de un método?

```
public class Vehiculo {
   public Vehiculo m() { ... }
}

public class Automovil {
   public Automovil m() { ... }
}
```

El método 'm' devuelve un objeto de tipo diferente, ¿se trata de sobrecarga, sobrescritura, puede el "upcasting" tener algo que ver aquí?

Pista: ¿se toma en cuenta el tipo del valor de retorno del método en la resolución de la sobrecarga? ▪

1.2 Introducción a los TADs y los algoritmos

Empecemos con algunas definiciones:

- Tipo Abstracto de Datos (TAD): [GTG14] "a systematic way of organizing and accessing data". Se trata de un concepto anterior e independiente del concepto de objeto.
- Algoritmos [GTG14]: "a step-by-step procedure for performing some task in a finite amount of time". Ejemplos de algoritmos son las operaciones con o sobre TADs.
- Los lenguajes de programación ofrecen mecanismos (p.ej., interfaces, clases, objetos, herencia, genéricos, etc) para implementar TADs.

Vamos a ver un ejemplo motivador. Tenemos una clase para números complejos implementados mediante representación cartesiana.

```
public class ComplejoCart {
  private double real;
  private double imaginaria;

  public ComplejoCart(double re, double im) {
    this.real = re;
    this.imaginaria = im;
  }

  public ComplejoCart(double re) {  // real a complejo
    this.real = re;
    imaginaria = 0.0;
  }

  public ComplejoCart(ComplejoCart c) { // constructor de copia
    real = c.getReal();
    imaginaria = c.getImaginaria();
  }

  // ''getters''

  public double getReal() { return real; }

  public double getImaginaria() { return imaginaria; }

  public double getModulo() {
    return ... // Aquí va una fórmula sobre 'real' e 'imaginaria'
               // para obtener el módulo.
  }
  public double getArgumento() {
    return ... // Aquí va una fórmula sobre 'real' e 'imaginaria'
               // para obtener el ángulo.
  }

  public ComplejoCart suma(ComplejoCart c) {
    return new ComplejoCart(this.real + c.getReal(), this.imaginaria + c.getImaginaria()) ;
```

```
  }

  public ComplejoCart resta(ComplejoCart c) {
    return new ComplejoCart(this.real - c.getReal(), this.imaginaria - c.getImaginaria()) ;
  }

  public String toString() {
    return new String(''(('' + this.real + ``, `` + this.imaginaria + ``)'');
  }

  public boolean equals(Object o) {
    if (o == this) return true;
    if (o instanceof ComplejoCart) {
      return this.real == c.getReal() && this.imaginaria == c.getImaginaria();
      ComplejoCart c = (ComplejoCart) o; // downcasting
    } else
      return false;
  }
}
```

Los métodos 'suma' y 'resta' no modifican el objeto sobre el que se invocan, devuelven un objeto nuevo. Esta es la forma natural de operar con valores matemáticos:

```
public static void main(String [] args) {
  ComplejoCart c1 = new ComplejoCart(1.0, 2.0);
  ComplejoCart c2 = new ComplejoCart(2.0, 3.0);

  ComplejoCart c3 = c1.suma(c2);

  System.out.println(c3.toString());
}
```

Acorde con la implementación del método 'suma', 'c1' no queda modificado al invocarle el método 'suma' sino que el método devuelve un nuevo objeto de clase `ComplejoCart` con la suma de los atributos de 'c1' y 'c2'. En el ejemplo hemos "guardado" (la referencia a) dicho objeto en otra variable 'c3'.

Observad además que no hay métodos "setters" para modificar el estado (los atributos) del objeto.

Los objetos cuyo estado no se modifica se llaman **objetos inmutables**. Las entidades matemáticas como los números se suelen implementar como objetos inmutables. Leed la discusión en los libros [Bud91], [Mey88] y [Blo08] sobre las ventajas e inconvenientes de los mismos. Otros ejemplos de objetos inmutables son los de clase envoltorio (Integer, Boolean, etc.) y los de clase 'String'.

Los TADs no tienen por qué consistir de objetos inmutables. En esta sección estamos usando un ejemplo con objetos inmutables por comenzar por algo sencillo, pero el concepto de TAD es independiente de si los objetos que lo implementan son inmutables o no.

Hemos usado los "getters" en 'c.getReal()' y 'c.getImaginaria()' en el código, pero podríamos haber accedido a los atributos del objeto referenciado por 'c' directamente pues es de clase `ComplejoCart`:

```
public ComplejoCart suma(ComplejoCart c) {
  return new ComplejoCart(this.re + c.re, this.im + c.im) ;
}
```

Problemas:

El código fuente revela la implementación. El cliente de la clase (en este ejemplo, el programador del método 'main') no está interesado en conocer los detalles de implementación sino en crear objetos `ComplejoCart` e invocarles métodos. Si le damos la clase pre-compilada en un ejecutable entonces el cliente no sabe qué métodos tiene que usar porque están en código binario. Habría que ofrecerle algún tipo de documentación (p.ej., Javadoc) aparte.

Si deseamos cambiar a una implementación con representación polar de complejos mediante módulo y ángulo entonces tenemos que reescribir la clase entera, o definir otra `ComplejoPolar`. Este sería el esquema de la implementación polar:

```
public class ComplejoPolar {
  private double modulo;
  private double argumento;
```

```
  public ComplejoPolar(double mod, double ang) {
    this.modulo = mod;
    this.angumento = ang;
  }

  ... // aquí van más constructores

  public double getReal() { mod * Math.cos(ang) ; }
  public double getImaginaria() { mod * Math.sin(ang) ; }
  public double getModulo() { return mod; }
  public double getArgumento() { return ang; }

  public ComplejoPolar suma(ComplejoPolar c) {
    // la suma tiene un código diferente también.
  }

  ... // etc
}
```

¡Cambia toda la implementación! Sin embargo el "cliente" de la clase (el código de 'main') cambiaría ligeramente:

```
public static void main(String [] args) {
  ComplejoPolar c1 = new ComplejoPolar(...);  // nuevos constructores
  ComplejoPolar c2 = new ComplejoPolar(...);

  ComplejoPolar c3 = c1.suma(c2);

  System.out.println(c3.toString());
}
```

El cambio sería todavía menor si cada clase ofreciera métodos para transformar un cartesiano en polar y viceversa.

Ejercicio 1.10 ¿Ves en este ejemplo la desventaja de tener en Java los nombres de los constructores de una clase sobrecargados? ▪

No se pueden combinar implementaciones diferentes. Sería cómodo poder sumar un objeto `ComplejoPolar` a un objeto 'ComplejoCartesiano' pues ambos son números complejos aunque cambie la representación.

Clases, interfaces, herencia y TADs

En Java un TAD consiste en una jerarquía de interfaces y clases. En un TAD hecho en Java, los interfaces especifican ("qué") y las clases implementan ("cómo").

Veamos un ejemplo de interfaz para definir un número complejo:

```
public interface Complejo {

  /**  Devuelve la parte real */
  public double getReal();

  /** Devuelve la parte imaginaria */
  public double getImaginaria();

  /** Devuelve el módulo */
  public double getModulo();

  /** Devuelve el argumento */
  public double getArgumento();

  /** Devuelve un nuevo Complejo haciendo la suma con 'c' */
  public Complejo suma(Complejo c);

  /** Devuelve un nuevo Complejo haciendo la resta con 'c' */
  public Complejo resta(Complejo c);
```

```
}
```

La semántica (es decir, la descripción precisa de qué debe hacer cada método) debe documentarse, bien informalmente mediante comentarios en lenguaje natural, en Javadoc, o mejor todavía, usando una notación formal.

El interfaz no incluye constructores, ni métodos heredados de 'Object' como 'toString', 'equals', etc. En Java estos métodos residen en las clases que implementan el interfaz.

Los interfaces de Java se introdujeron para la herencia múltiple y su uso como interfaces de TADs es ad-hoc. Pero es lo que hay.

Como hay dos formas de representar complejos, el interfaz ofrece "getters" para cada posible representación. Una razón de ser de los "getters" es ésta. No es necesario tener "getters" para clases con una representación única. Es válido acceder a los atributos directamente sin usar "getters" si la implementación de la clase no va a cambiar. (Sí, a pesar de lo que os hayan dicho por ahí.)

Las clases deben declarar que implementan el interfaz:

```
public class ComplejoCart implements Complejo {

  ... // los atributos, constructores, y ''getters'' no cambian.

  public Complejo suma(Complejo c) {  // Complejo es un interfaz
    return new ComplejoCart(this.re + c.getReal(), this.im + c.getImaginaria()) ;
  }

  // ... tampoco cambia toString

  public boolean equals(Object o) {
    if (o == this) return true;
    if (o instanceof Complejo) {
      Complejo c = (Complejo) o;       // downcasting al interfaz
      return this.re == c.getReal() && this.im == c.getImaginaria();
    } else return false;
  }
}
```

Los métodos 'suma' y 'resta' toman como parámetro un objeto **de cualquier clase que implementa el interfaz `Complejo`**. Por tanto, la implementación de dichos métodos debe tener ésto en cuenta. Por ejemplo, el parámetro 'c' del método 'suma' o bien es null (no hay que olvidarse nunca de null, en nuestro ejemplo que se lance `NullPointerException` es razonable) o bien referencia un objeto de una clase que implementa el interfaz Complejo. Por otro lado, el método 'suma' devuelve un objeto de una clase que implementa el interfaz Complejo.

De igual manera, una declaración 'Complejo c1' lo que dice es que 'c1' almacenará o bien null o bien una referencia a un un objeto de una clase que implementa el interfaz Complejo. Recordad:

```
Complejo c1 = new Complejo();   suspenso
```

No tiene sentido y no compila.

Ventajas de usar "getters" sobre variables de tipo interfaz:

Gracias al uso de "getters", tanto el constructor de copia, como el método `suma` y el método `equals` funcionan cuando 'c' es un objeto **de cualquier clase que implemente el interfaz**. Observad que en `equals` el "downcasting" se hace de `Object` a `Complejo`, que es un interfaz, no una clase.

Ejemplo de uso por el "cliente" del TAD, esta vez usando el interfaz y las dos implementaciones:

```
public static void main(String [] args) {
  Complejo c1 = new ComplejoCart(1.0, 2.0);
  Complejo c2 = new ComplejoPolar(8.4, 60.0);

  Complejo c3 = c1.suma(c2);

  System.out.println(c3.toString());

  System.out.println(c2.suma(c1).toString());
}
```

Las variables que utilizamos **son de tipo interfaz** pero **en tiempo de ejecución referencian objetos de clases que implementan el interfaz**.

La "librería" o paquete de números complejos consta de un interfaz y dos clases. El cliente de la librería sólo tiene que conocer cuáles son y qué hacen los métodos del interfaz, así como los nombres de las clases y los constructores. El resto de detalles de implementación no son necesarios.

Observad que `c1.suma(c2)` es un objeto de clase `ComplejoCart` mientras que `c2.suma(c1)` es un objeto de tipo `ComplejoPolar`.

Hay que tener cuidado al diseñar TADs con múltiples representaciones. Observad que en el `main` anterior `c1.suma(c2)` es un objeto de clase `ComplejoCart` mientras que 'c2.suma(c1)' es un objeto de clase `ComplejoPolar`. Debe cumplirse que 'c1.equals(c2)' da el mismo valor de verdad que 'c2.equals(c1)' aunque tengan distinta representación. Aquí se reafirma el rol de los "getters" y que los cambios de representación sean biyectivos.

Acceso via "downcasting" a la clase que implementa el interfaz

En ocasiones es necesario acceder directamente a los atributos de la clase que implementa el interfaz en vez de utilizar los "getters" del interfaz. Explicaremos más adelante cuándo es necesario hacer ésto. De momento sólo mostramos cómo hacerlo.

Volvamos a la clase `ComplejoCart`. En la siguiente versión modificamos el código del método 'suma' para que sólo sume con objetos de la clase `ComplejoCart`, accediendo directamente a los atributos de la case vía "downcasting":

```
public class ComplejoCart implements Complejo {

  ...

  public Complejo suma(Complejo c) throws IllegalArgumentException {
    if (c instanceof ComplejoCart)

       return new ComplejoCart( this.re + ((ComplejoCart)c).re
                              , this.im + ((ComplejoCart)c).im );

    else

       throw new IllegalArgumentException();
  }

}
```

El acceso a la implementación puede complicar o imposibilitar al cliente el inter-operar con múltiples representaciones.

Resumen de ideas de este tema de introducción

- Un TAD consiste en uno o más interfaces y clases. Los interfaces especifican ("qué") y las clases implementan ("cómo").
- El "cliente" del TAD sólo tiene que conocer los interfaces, para qué sirven, qué hacen los métodos y con qué eficiencia. También debe conocer el nombre y los constructores de las clases que implementan los interfaces.
- El "cliente" NO tiene que conocer los atributos ni el código de los métodos de las clases. De hecho, puede recibir el código pre-compilado en un ejecutable.
- En lo posible, los interfaces y clases deben estar diseñados para poder combinar objetos de clases distintas. El código de las clases debe invocar sobre los objetos los métodos recogidos en el interfaz.
- Idealmente, el "implementador" de las clases puede cambiar los detalles de éstas sin que el código del "cliente" se vea afectado, ya que el cliente utiliza el interfaz.

Conceptos y terminología

- **Abstracción**: "Separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción." (Diccionario de la Real Academia Española, 22ª edición)
- **Abstracción funcional**: atender a la función (funcionalidad, propósito, "el qué") ignorando la estructura/representación interna ("el cómo").
- La función se describe mediante un **interfaz** (p.ej., un interfaz de Java) y una especificación del comportamiento (semántica o contrato) que suele darse bien en lenguaje natural, bien en una notación semiformal (p.ej., Javadoc), bien en una notación formal (p.ej., lógica de orden superior).
- La estructura/representación interna se describe en una **implementación** (p.ej., una clase de Java) que realiza la funcionalidad acorde al contrato.

- **Encapsulación** (ocultación de información): sólo es necesario conocer la función, la estructura puede quedar oculta.
  Ejemplo: el cliente de un TAD sólo necesita conocer los interfaces y los nombres y constructores de las clases que implementan los interfaces. Los detalles de implementación de las clases pueden quedar ocultos.
- **Independencia de la representación**: la misma función puede ser realizada por diferentes representaciones. Por ejemplo, el interfaz puede ser implementado por distintas clases.
- **Modularidad**: un módulo o cápsula realiza una funcionalidad. Debe tener **alta cohesión** (ser autocontenido) y **bajo acoplamiento** (no depender de cambios en otros módulos).
- **Abstracción de datos**: mecanismos para abstraer datos. Típicamente: tipos de datos, clases, paquetes, etc.
- **Abstracción de control**: macros, subrutinas, programación estructurada (condicionales, bucles, etc), procedimientos y funciones, objetos y métodos, mecanismos de sincronización y comunicación en concurrencia, etc.
- La definición matemática más aceptada de **TAD** es la de álgebra, en particular la de álgebra no libre: conjuntos de elementos que son manipulados en términos de operaciones sobre el conjunto las cuales cumplen leyes descritas mediante ecuaciones condicionales.
  Por ejemplo, los enteros son un TAD. Son un conjunto de datos cuya representación interna en la máquina no conocemos y que tienen una serie de operaciones (suma, resta, división, etc.) donde cada una cumple propiedades (conmutativa, distributiva, etc.).

Idealmente, la independencia de la implementación implica bajo acoplamiento: se debería poder cambiar la implementación sin afectar el código cliente que usa los métodos del TAD. El interfaz sirve de "contrato" entre el cliente y el implementador del TAD.

Pero esto no es siempre así. Hay interfaces que por los parámetros que toman los métodos sugieren una implementación concreta y un cambio implica cambiar el interfaz: "interface coupling".

Además, en Java no basta con conocer el interfaz, hay que conocer las clases y los constructores para crear los objetos. Si se cambia de implementación entonces hay que recompilar el código donde se crean los objetos. Ejemplo:

```
public class Main {
  public static void main (String[] args) {
    Complejo c = new ComplejoCart(3.32, 5.21);
    Complejo s = c.suma(c);
  }
}
```

Para cambiar a representación polar hay que cambiar el constructor `ComplejoCart` por `ComplejoPolar` y recompilar.

Existen patrones de diseño para solucionar este problema, como el patrón factoría. La idea es de nuevo usar encapsulación. Se puede crear una clase intermedia que hace las veces de "interfaz" para los constructores.

Idealmente, los TADs tienen alta cohesión, bajo acoplamiento, y encajan con el diseño modular y el refinamiento progresivo (se pueden posponer decisiones de implementación durante el proceso de diseño). En la práctica, hay factores pragmáticos como la eficiencia que pueden requerir un cambio de implementación que afecta al interfaz o incluso requerir la sustitución de un TAD por otro. El software evoluciona. Hay métodos estadísticos y probabilísticos para la elección de la implementación adecuada de un TAD basándose en el análisis de las operaciones del TAD que el "cliente" utiliza en su código.

2. Listas Indexadas

Los TADs contenedores son aquellos que se diseñan para insertar (añadir, almacenar), buscar, y eliminar (borrar) datos (elementos) de forma eficiente. En el argot callejero: “meter”, “sacar”, “encontrar”. Una lista es un TAD contenedor que consiste en una secuencia lineal de elementos. Por ejemplo, una lista de la compra escrita con un editor de textos. Se puede insertar un elemento en cualquier posición de la lista, por ejemplo entre otros dos elementos. La lista no tiene por qué estar ordenada.

En programación, las listas se suelen escribir horizontalmente de izquierda a derecha. Los elementos están ordenados linealmente según su posición de izquierda a derecha: primer elemento, segundo, etc. La *posición* de un elemento cambia al insertar y al borrar elementos y se pueden insertar y borrar elementos en cualquier posición. La búsqueda de elementos suele ser secuencial o lineal: de izquierda a derecha o de derecha a izquierda. No hay acceso aleatorio como en los arrays.

De forma teórica, el tamaño de la lista no está acotado y podría aumentar indefinidamente. En la práctica el tamaño está acotado por la memoria disponible del computador. Puesto que el computador no tiene memoria infinita, podrá producirse una excepción de insuficiente memoria al invocar un método de inserción.

A continuación vamos a repasar el interfaz `IndexedList<E>` que se vio en la asignatura Programación II y que se trata de una lista que utilia un índice númerico para el acceso a los elementos de la lista.

2.1 Interfaz `IndexedList`

```
public interface IndexedList<E> extends Iterable<E> {

  public void add(int index, E e) throws IndexOutOfBoundsException;

  public E get(int index) throws IndexOutOfBoundsException;

  public boolean isEmpty();

  public int size();

  public E set(int index, E e) throws IndexOutOfBoundsException;

  public int indexOf(E search);

  public E removeElementAt(int i) throws IndexOutOfBoundsException;

  public boolean remove(E element);

  public Object [] toArray();
}
```

El interfaz `IndexedList<E>` dispone de los siguientes métodos:

- Dos métodos interrogadores `size` que devuelve el tamaño y `isEmpty` que devuelve **`true`** si el tamaño de la lista es 0
- Un método de inserción `add` y otro de modificación `set` que cambia el elemento en una cierta posición
- Dos métodos de borrado `remove`, que borra un elemento, y `removeElementAt`, que borra el elemento que ocupa una cierta posición
- dos métodos de consulta, `get`, que accede por índice a un elemento y `indexOf`, que devuelve el índice de un cierto elemento

Los métodos que reciben un índice evalúan si el índice recibido se encuentra dentro del rango del array y en caso contrario lanzan la excepción `IndexOutOfBoundsException`.

■ **Ejemplo 2.1** El siguiente método código utiliza algunos métodos de la lista

```
IndexedList<Integer> list = new ArrayIndexedList<>();

list.add(0, 4);              // [4]
list.add(1, 5);              // [4,5]
list.add(0, 1);              // [1,4,5]
list.add(1, 8);              // [1,8,4,5]

show(list);

System.out.println("La posicion del 4 es: " + list.indexOf(4));

System.out.println("La posicion del 14 es: " + list.indexOf(14));

System.out.println("El que ocupa la posición 3 es el: " + list.get(3));

System.out.println("Ahora borramos");

list.remove(4);              // [1,8,5]
list.removeElementAt(0);     // [8,5]

show(list);

System.out.println("Size: "+ list.size());
System.out.println("IsEmpty: " + list.isEmpty());

}
```

Y utiliza el método `show`, que imprime todos los elementos de una lista y su correspondiente índice:

```
public static <E> void show(IndexedList<E> list) {
  for (int i = 0; i < list.size(); i ++) {
    System.out.println("[" + i + "]:" + list.get(i));
  }
}
```

Este otro método busca en una lista indexada si un elemento está en la lista. Obsérvese con cuidado el método `eqNull` por si la lista contiene elementos **`null`** y así evitamos la temida `NullPointerException`.

```
public static <E> boolean member (IndexedList<E> list, E elem) {
  int i = 0;
  while (i < list.size() && !eqNull(list.get(i),elem)) {
    i++;
  }
  return i < list.size();
}

public static boolean eqNull (Object o1, Object o2) {
  return o1 == o2 || o1!= null && o1.equals(o2);
}
```

■

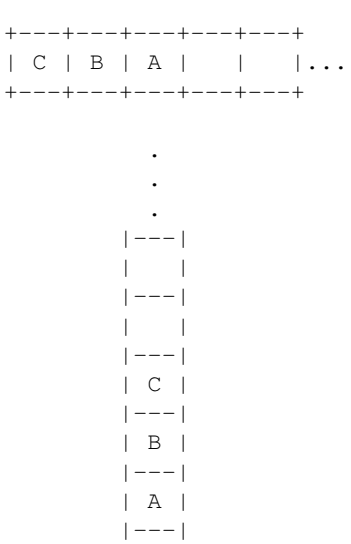
3. Pilas (LIFO) y Colas (FIFO)

Las pilas o LIFOs (*stacks*) y las colas FIFO (*queues*) son TADs fundamentales que a pesar de su sencillez tienen innumerables aplicaciones, desde la propia ejecución de los programas hasta su uso en navegadores web, dispositivos móviles, sistemas operativos, hardware, etc.

3.1 Pila - LIFO (Last In First Out)

Una pila (*stack*) o LIFO es un TAD contenedor que consiste en una secuencia lineal de elementos de forma que el último elemento apilado ("push") es el primero en ser desapilado ("pop"). Por ejemplos, pensad en una pila de platos, ¿cuál es el primer plato que vas a sacar?

Escribiremos las pilas horizontalmente de izquierda a derecha, siendo el primer elemento apilado el más a la izquierda. También se suelen escribir verticalmente, siendo el primer elemento apilado el más profundo: Por ejemplo en una pila apilamos A, después B y finalmente C:



No existe una operación de búsqueda. Sólo se puede apilar, desapilar, o devolver el elemento en la cima de la pila (`top`). Igual que en el caso de las listas, el tamaño de la pila está acotado por la memoria disponible del computador. Podrá producirse una excepción de insuficiente memoria al invocar el método `push`, lo que en el caso de la pila se conoce como "desbordamiento de pila" o "stack overflow". (Ahora ya sabéis de dónde le viene el nombre a vuestra página web de consulta preferente, después de Google creo, en cuestiones de programación.)

Veremos dos formas de implementar pilas: usando listas de posiciones y usando arrays.

Interfaz `LIFO<E>`

El código del interfaz `'LIFO<E>'` es el siguiente:

```
public interface LIFO<E> extends Iterable<E> {

  public int size();

  public boolean isEmpty();

  public E top() throws EmptyStackException;

  public E pop();

  public void push(E elem);

  public Object [] toArray();

  public E [] toArray(E[] a);
}
```

Algunos comentarios sobre los métodos del interfaz:

- Interrogadores: `size`, `isEmpty`.
- Inserción: `push` apila un elemento.
- Borrado: `pop` desapila un elemento y lo devuelve.
- Acceso ("getters"): `top` devuelve el elemento en la cima de la pila. El comportamiento de `top` es equivalente a desapilar ('pop') y volver a a pilar ('push') el elemento.

■ **Ejemplo 3.1** El siguiente método indica si en una cadena de caracteres que contiene una expresión de Java los paréntesis están equilibrados. El método recorre la cadena. Los paréntesis izquierdos se apilan. Al encontrar un paréntesis derecho la pila debe no ser vacía y se desapila el paréntesis izquierdo que le corresponde. Al terminar de recorrer la cadena la pila debe ser vacía.

Por ejemplo, dada la cadena "((3 * 5) + (8 * (5 - 1))":

```
                                                                pila
((3 * 5) + (8 * (5 - 1))
^                                                               (

((3 * 5) + (8 * (5 - 1))                                        ( (
 ^

((3 * 5) + (8 * (5 - 1))                                        (
       ^

((3 * 5) + (8 * (5 - 1))                                        ( (
           ^

((3 * 5) + (8 * (5 - 1))                                        ( ( (
               ^

((3 * 5) + (8 * (5 - 1))                                        ( (
                     ^

((3 * 5) + (8 * (5 - 1))                                        (
                      ^

((3 * 5) + (8 * (5 - 1))
                       ^
```

El código del método:

```
public static boolean balance(String str) {
  LIFO<Character> lifo = new .... ;
  boolean error = false;
```

```
        for (int i = 0; i < str.length() && !error; i++)
          switch (str.charAt(i)) {

            case '(': lifo.push('(');   // para equilibrar con ')'
                      break;

            case ')': if (lifo.isEmpty()) error = true;
                      else lifo.pop();
                      break;

            default:  break;
          }
        return !error && lifo.isEmpty();
      }
```

Observad que no importa el elemento que se apila. Lo que importa es si la pila está o no vacía al encontrar un paréntesis derecho y al terminar de recorrer la cadena. Se podría haber apilado un asterisco, un entero, etc.

▪

3.2 Colas FIFO (First In First Out)

Una cola ("queue") FIFO es un TAD contenedor que consiste en una secuencia lineal de elementos de forma que el primer elemento "encolado" (*enqueue*) es el primero en ser "desencolado" (*dequeue*), ..., y el último elemento encolado es el último en ser desencolado. (Sí, aquí estamos haciendo un uso anglicista y espurio de la palabras .[en]colarz "desencolar".) Pensad en la cola de la parada del bus. El primero en llegar es el primero en subir ... y el último en llegar es el último en subir.

Escribiremos las colas horizontalmente de izquierda a derecha, siendo el primer elemento encolado el más a la izquierda. Cola en la que se encola A, después B y finalmente C:

```
+---+---+---+---+---+
| A | B | C |   |   |
+---+---+---+---+---+
```

y el estado de la cola al desencolar:

```
+---+---+---+---+---+
| B | C |   |   |   |
+---+---+---+---+---+
```

Al igual que pasa con las pilas, No existe una operación de búsqueda. Sólo se puede encolar, desencolar, o devolver el elemento a desencolar ('first'). Al igual que con las listas y las pilas, el tamaño de la cola está acotado por la memoria del computador. (Por razones estéticas no diremos que hay "desbordamiento de cola" cuando la cola esté llena.)

Veremos dos formas de implementar colas: usando listas de posiciones y usando arrays.

Interfaz FIFO<E>

El interfaz `FIFO<E>` es el siguiente:

```
public interface FIFO<E> extends Iterable<E> {

  public int size();

  public boolean isEmpty();

  public E first() throws EmptyFIFOException;

  public void enqueue(E elem);

  public E dequeue() throws EmptyFIFOException;

  public Object [] toArray();

  public E [] toArray(E[] a);
}
```

Algunos comentarios sobre el interfaz:

- El interfaz `FIFO<E>` extiende `Iterable<E>`:
- Interrogadores: `size`, `isEmpty`.
- Inserción: `enqueue` encola un elemento.
- Borrado: `dequeue` desencola un elemento.
- Acceso ("getters"): `first` devuelve el primer elemento a desencolar de la cola. El comportamiento de `first` NO es equivalente a desencolar y volver a encolar el elemento. Esto sólo se cumple cuando el elemento es el único de la cola.
- Conversión: el método `toArray` devuelve un vector (array) con los elementos de la cola. Devuelve un vector de 'Object'. El método `toPositionList` devuelve una lista de posiciones con los elementos de la cola. Siguiendo la convención descrita anteriormente el primer elemento del vector y de la lista será el primer elemento desencolado, etc.

■ **Ejemplo 3.2** Supongamos que la variable `fifo` referencia una cola FIFO de objetos de clase `Process` de procesos del sistema operativo. La clase `Process` ofrece entre otros los métodos `run` y `suspend`. El método `run` permite ejecutar un proceso durante un número máximo de milisegundos indicados como parámetro. El proceso puede parar su ejecución antes de ese tiempo, bien porque ha terminado, bien porque ha recibido alguna interrupción o está a la espera de un evento, etc. El método `run` devuelve un número negativo si el proceso ha terminado. El método `suspend` salva el estado de un proceso parado que no ha terminado de forma que pueda reanudarse su ejecución en la próxima invocación a `run`. El siguiente bucle ejecuta los procesos con planificación circular o "round robin".

```
while(!fifo.isEmpy()) {
    Process p = fifo.first();
    fifo.dequeue();
    int status = p.run(20);
    if (status < 0) {
      p.suspend();
      fifo.enqueue(p);
    }
}
```

Este ejemplo es meramente ilustrativo. En un sistema operativo real hay comunicación y sincronización entre procesos y la planificación es más sofisticada.

■

4. Complejidad

Nos referimos a la **complejidad** de un programa como una medida de su tiempo de ejecución o del uso de memoria. Sinónimos: eficiencia, coste. Podemos hablar de diferentes otros tipos de complejidad, pero en la asignatura hablaremos de dos tipos:

- Complejidad **temporal**: complejidad del tiempo de ejecución.
- Complejidad **espacial**: complejidad del espacio de memoria utilizado.

La complejidad en tiempo y la complejidad en espacio suelen estar reñidas: se ahorra tiempo a costa de usar más espacio, y se ahorra espacio a costa de tardar más tiempo. Es frecuente poder almacenar cálculos temporales para no volverlos a hacer, ganas velocidad, pero, a cambio, gastas más memoria.

La complejidad en tiempo o en espacio de un programa se calcula en *función del tamaño o de los valores de los datos de entrada* (abreviado "tamaño de la entrada", "input size"). No se trata de todos los datos que toma o usa el programa. Se trata del tamaño de aquellos datos que son utilizados por el programa para resolver el problema y que tienen un impacto en el coste en tiempo o en espacio.

Al crecer el tamaño de la entrada la complejidad del programa o bien se mantiene constante o bien crece en función de dicho tamaño. No puede decrecer, incluso si se usa más espacio, porque se introduce el problema de buscar en el

Por n método que indica si un elemento está o no en un array. El "tamaño de la entrada" en este caso es el tamaño del array donde se hace la búsqueda. Cuanto más grande el tamaño del array más tarda el método en encontrar el elemento. El *tamaño del elemento* no es importante, incluso si se trata de un dato de gran tamaño en bits, pues sólo afecta la complejidad en tiempo total en un valor constante, es decir, sólo suma un valor constante al valor de complejidad total.

▪ **Ejemplo 4.1** Un método que toma un elemento 'elem' y dos arrays 'arr1' y 'arr2' tal que 'arr2.length >= arr1.length'. El método busca 'elem' en 'arr1'.

- Si no está entonces termina y devuelve null.
- Si está, sea 'i' la posición de 'arr1' tal que 'elem == arr1[i]', entonces el método devuelve el elemento en 'arr2[i]'.

Aquí el "tamaño de la entrada" es 'arr1.length' pues el segundo array no es usado en la búsqueda. Cuanto más grande es el tamaño de 'arr1' más tarda el método en encontrar el elemento. El tamaño de 'arr2' no afecta el tiempo, pues el acceso directo a un array se asume que tiene complejidad en tiempo constante (es constante respecto al tamaño del array ▪

No sólo el tamaño de la entrada es relevante, también la *distribución de los datos*. Por ejemplo, el método que busca un elemento en un array no tarda lo mismo si cada vez pasamos arrays más grandes pero:

- El elemento a buscar es siempre el primero.
- El elemento a buscar está por el medio, o a veces está el primero y a veces en medio y a veces está el último o no está (y consideramos estos casos como equiprobables).
- El elemento a buscar está siempre el último o no está.
- El elemento a buscar está más veces el primero que en otra posición.

Podemos decir que se estudia la complejidad en *caso mejor*, *caso (pro)medio*, *caso peor*, y *caso amortizado*.

El estudio del *caso peor* nos da unos resultados más precisos y con menor variabilidad que los estudios de casos medios y experimentales. Los programas eficientes en el caso peor lo serán también en los otros casos. Para ello hay que considerar cuál es la entrada que daría el caso peor. En el ejemplo anterior, la peor entrada son arrays donde el elemento está siempre el último o no está, pues el programa siempre recorre el array completo.

Para hablar de complejidad habitualmente nos referimos a la **Complejidad asintótica**: idealmente el tamaño de la entrada podría ir creciendo indefinidamente (p.ej., se podrían ir pasando arrays más y más grandes a los métodos anteriores). La complejidad por tanto se calcula como el tiempo o espacio requerido en función del tamaño de la entrada cuando éste *tiende a infinito*. De esta forma, al hacerlo cuando tiene a infinito estamos ignorando los valores constantes o los valores nos significativos para la complejidad. Por ejemplo, si tenemos un método de complejidad n^3+n, en el infinito podemos ignorar n y quedarnos sólo con n^3. Para ello, la notación $O()$ nos permitirá dar una medida de complejidad.

Podemos decir que hay dos maneras de calcular la complejidad:

1. **Análisis experimental**: Se escribe el programa en un lenguaje de programación concreto. Requiere análisis estadístico (con buen muestreo de datos de entrada) y probabilístico (distribución de datos). Los resultados dependen del *entorno de ejecución* (lenguaje de programación, compilador, sistema operativo, arquitectura, entorno dinámico, etc.)
2. **Análisis teórico**: Se estudian los programas independientemente del entorno de ejecución. El lenguaje utilizado puede ser pseudocódigo, una notación que abstrae (alto nivel) las particularidades de un *paradigma* de programación.
   Se asume que la complejidad de las operaciones del lenguaje pseudocódigo es proporcional en una constante a la complejidad real de su implementación en un lenguaje de programación concreto y ejecutándose en entorno concreto. Por tanto, la complejidad del programa en pseudocódigo es proporcional en una constante a la complejidad de su implementación. Dicho de otro modo, los factores a tener en cuenta en una implementación real afectan la complejidad teórica en un factor constante.
   De esta forma, podemos estudiar programas en un entorno de ejecución concreto (p.ej., programa escrito en Java compilado con Eclipse y ejecutándose en las Aulas Informáticas) asumiendo que la complejidad del entorno es una función constante del programa en pseudocódigo.

En la práctica se necesita tanto el análisis teórico como el experimental: Ïn theory there is no difference between theory and practice. In practice there is."(Yogi Berra)

En lo tocante a TADs, las medidas de complejidad están asociadas a los métodos de las clases (implementaciones), no a los interfaces. Un interfaz describe simplemente la cabecera del método (el qué) y no la implementación (el cómo).

Pero un interfaz puede exigir que ciertos métodos se implementen con una complejidad determinada, forzando así una representación y familia de algoritmos para el TAD. Finalmente, la elección de métodos en los interfaces puede sugerir o descartar ciertas implementaciones.

Complejidad asintótica

Hablamos de complejidad asintótica cuando ignoramos los valores constantes de las expresiones. Algunas de las funciones matemáticas más habituales que representan la complejidad son las siguientes:

- Función **constante**: $f(n)=c$ con c constante, p.ej., $f(n)=10$.
- Función **polinómica**: $f(n)=c_1*n^{e_1}+...+c_m*n^{e_m}$ donde c_i (coeficientes) y e_i (exponentes) son enteros, p.ej.,
  $f(n)=2*n^2+7*n-20$.
  El grado del polinomio se define como el exponente de mayor valor. Por ejemplo, el polinomio anterior tiene grado 2.
  Entre las funciones polinomiales encontramos la lineal (grado 1), cuadrática (grado 2) y cúbica (grado 3).
- Función **logarítmica**, será casi siempre en base 2, por tanto en vez de $log_2(n)$ escribiremos $log(n)$.
- Función **exponencial**: $f(n)=c^n$ con c constante, p.ej., $f(n)=2^n$.
  La funciones $log(n)$ y 2^n son inversas: $log_2(x)=y <=> 2^y=x$.
  - $log_2(2^x)=x$
  - $2^{log_2(x)}=x$
- Función **n-log(n)**: $f(n)=n*log_2(n)$
  - Sumas aritméticas y geométricas.
  - Razones de crecimiento.
  - Funciones techo ('ceiling') y suelo ('floor').

Veamos algunas cifras comparativas:

$Funcin$	$n = 32$	$n = 64$	$n = 128$
$log_2(n)$	5	6	7
n	32	64	128
$n * log_2(n)$	160	384	896
n^2	1,024	4,096	16,384
n^3	32,768	262,144	2,097,152
2^n	4,294,967,296	18,446,744,073,709,551,616	(no cabe)

Para hacernos una idea del tamaño:

$$2^n \text{ para } n = 128 : 340{,}282{,}366{,}920{,}938{,}463{,}463{,}374{,}607{,}431{,}768{,}211{,}456$$

y una más:

$$n^n \text{ para } n = 32 : 1{,}461{,}501{,}637{,}330{,}902{,}918{,}203{,}684{,}832{,}716{,}283{,}019{,}655{,}932{,}542{,}976$$

Podemos hablar de *problemas intratables*: polinomiales a partir de un 'n' medio, exponenciales a partir de un 'n' relativamente bajo, etc.

Notación $O()$

La intuición de la función $O()$ es establecer la complejidad de una función de 'n' indicando otra función proporcional que la acota asintóticamente (ignorando los valores contantes).

Definicion 4.0.1 — Notación $O()$. Sean f y g dos funciones de naturales a reales. Se dice que $f(n)$ *es del orden de* $g(n)$, o $f(n)$ *es* $O(g(n))$, si existe una constante real $c > 0$ y una constante natural $n_0 >= 1$ tal que $f(n) <= c * g(n)$ para $n >= n_0$

De forma intuitiva podemos de que a partir de un valor n_0 de la entrada, la función $c * g(n)$ acota los valores de la función f(n) cuando n tiende a infinito. Nótese que $O(0)$ no tiene sentido.

Una escala de complejidad de menor a mayor:

1. Constante: $O(1)$
2. Logarítmica: $O(log(n))$
3. Lineal: $O(n)$
4. N-Log-N: $O(n * log(n))$
5. Cuadrática $O(n^2)$
6. Cúbica: $O(n^3)$
7. Polinomial: $O(n^m)$ de orden m
8. Exponencial: $O(2^n)..O(m^n)$

Algunos comentarios sobre las funciones:

- Todas las funciones constantes son $O(1)$: Sea $f(n) = c$ función constante. Entonces $f(n) <= c * g(n)$ con $g(n) = 1$ y $n_0 = 1$.
- Todas las funciones polimórficas son $O(n^{e_1})$ donde e_1 es el exponente de mayor grado: Sea $f(n) = c_1 * n^{e_1} + ... + c_m * n^{e_m}$ tal que e_1 es el exponente de mayor grado. Entonces existe un coeficiente c y un n_0 tal que $f(n) <= c * n^{e_1}$ para $n >= n_0$. Es decir, multiplicando n^{e_1} un número c de veces podemos hacer el resultado mayor que lo que vale el resto del polinomio $c_2 * n^{e_2} + ... + c_m * n^{e_m}$, pues n^{e_1} crece mucho más rápido que $n^{e_2}, ..., n^{e_m}$.
- En las funciones polinomiales, **se ignoran factores constantes y de orden inferior**. Esto viene justificado por el algebra de O():
- $O(c * f(n))$ es $O(f(n))$
- $O(f(n) + g(n))$ es $O(f(n))$ si $g(n)$ es $O(f(n))$
- $O(f(n) + g(n))$ es $O(g(n))$ si $f(n)$ es $O(g(n))$

- **Ejemplo 4.2** Veamos algunos ejemplos de métodos con diferentes complejidades:

```
public static long contador = 0;

public static void logn (int n) {
  for (int i = n; i > 0; i = i/2) {
    contador ++;
```

```
  }
}

public static void lineal (int n) {
  for (int i = 0; i < n; i ++) {
    contador ++;
  }
}

public static void nlogn (int n) {
  for (int i = 0; i < n; i ++) {
    for (int j = i; j > 0; j = j / 2) {
      contador ++;
    }
  }
}

public static void cuadratica (int n) {
  for (int i = 0; i < n; i ++) {
    for (int j = 0; j < n; j ++) {
      contador ++;
    }
  }
}

public static void cubica (int n) {
  for (int i = 0; i < n; i ++) {
    for (int j = 0; j < n; j ++) {
      for (int k = 0; k < n; k ++) {
        contador ++;
      }
    }
  }
}

public static void polinomica4 (int n) {
  for (int i = 0; i < n; i ++) {
    for (int j = 0; j < n; j ++) {
      for (int k = 0; k < n; k ++) {
        for (int l = 0; l < n; l ++) {
          contador ++;
        }
      }
    }
  }
}

public static void  exponencial (int n) {
  contador ++;
  if (n < 0) return ;
  exponencial(n-2);
  exponencial(n-1);
}

public static void exponencialN (int n) {
  contador ++;
  if (n < 0) return;
  for(int i = 0; i < n; i ++) {
    exponencialN(n-1);
  }
}
```

■

5. Ordenación

5.1 Órdenes Totales

En este tema estudiaremos el concepto matemático de **orden total**, también llamado **orden lineal**.

Vamos a empezar con una perspectiva teórica. Desde la teoría de conjuntos, una entrada es un par perteneciente al producto cartesiano de un conjunto K de claves y un conjunto V de valores: (k,v)' está en $K\ x\ V$, donde:

$$K\ x\ V = (k,v) \mid k \in K \wedge v \in V$$

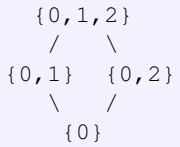
Relación de **orden total computable** $\leq$ sobre el conjunto de claves K.

1. Hay una relación de igualdad computable (existe un programa que la computa) en K que permite determinar si una clave es igual o no a otra. (El requisito de computabilidad es obvio pero importante, pues según el famoso resultado de Alan Turing, existen funciones de naturales a naturales que no son computables, es decir, no existe un programa que compute sus valores.)
   La relación de igualdad es una relación de equivalencia, es decir, cumple las siguientes reglas:
   - Reflexiva: para toda clave k se cumple k = k.
   - Simétrica: para toda k_1 y k_2, si k_1 = k_2 entonces k_2 = k_1.
   - Transitiva: para toda k_1, k_2 y k_3, si k_1 = k_2 y $k_2 = k_3$ entonces $k_1 = k_3$.
2. Hay una relación total computable (hay un programa que la computa) que satisface las siguientes reglas:
   - Totalidad: para toda k_1 y k_2, o bien k_1 <= k_2 o bien k_2 <= k_1, una de ellas debe ser cierta.
   - Antisimétrica: para toda k_1 y k_2, si k_1 <= k_2 y k_2 <= k_1 entonces $k_1 = k_2$.
   - Transitiva: para toda k_1, k_2 y k_3, si $k_1 <= k_2$ y $k_2 <= k_3$ entonces $k_1 <= k_3$.

   Totalidad y antisimétrica implican reflexiva.

Una relación computable entre dos elementos tiene asociado un programa que toma los dos elementos y devuelve un booleano que indica si el par está o no en la relación. El programa computa una relación de orden total si cumple las propiedades matemáticas de una relación de orden total. Algunos ejemplos de órdenes totales pueden ser: los números naturales con $\leq$ o las cadenas de caracteres con orden lexicográfico.

▪ **Ejemplo 5.1** Ejemplo de orden **parcial**: dados dos conjuntos S_1 y S_2, definimos la relación $\leq$ de la siguiente manera: $S_1 \leq S_2$ si y sólo si para todo x en S_1 se tiene que x está en S_2. Dicha relación es una relación de orden pues cumple "reflexiva", "antisimétrica" y "transitiva", pero no cumple "totalidad" y por lo tanto es un orden parcial:



Por ejemplo con esto, sabemos:

- $\{0\} \leq \{0,2\}$ es cierto y que $\{0,2\} \leq \{0\}$ es falso.
- Pero $\{0,1\} \leq \{0,2\}$ es falso y $\{0,2\} \leq \{0,1\}$ es falso.

■

Un orden parcial define un retículo. Un orden total define una recta.

Todo orden total lleva asociado un *orden total estricto* $<$ que se define como $k_1 < k_2$ si y sólo si $k_1 \leq k_2$ y no $k_1 = k_2$. Se puede definir un orden estricto sobre un orden total porque tenemos igualdad.

En la práctica se desea tener conjuntos K acotados en los que hay una **clave mínima** y o bien una **clave máxima** o bien un **supremo**:

- Clave mínima: si para toda k se tiene $k_{min} \leq k$ entonces k_{min} es una clave mínima.
- Clave máxima: si para toda k se tiene $k \leq k_{max}$ entonces k_{max} es una clave máxima.
- Las claves mínimas y máximas son únicas porque el orden es total.
- Supremo: se toma la unión de K con un conjunto formado por un único elemento sup, se extiende la relación de orden a esa unión, y se postula que $k \leq sup$ para toda k.

5.2 Interfaces `Comparable` y `Comparator`

Ambos interfaces se utilizan para definir órdenes totales en Java. La diferencia está en cómo y para qué se usan.

5.2.1 Interfaz `java.lang.Comparable`

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

El interfaz `Comparable<T>` establece un orden *natural* entre los objetos de una clase. La implementación de `Comparable<T>` por la clase 'T' hace que ésta tenga que implementar el método `compareTo` para comparar dos objetos de 'T'. Ese método de comparación *queda fijo para todos los objetos de 'T' y se le llama orden natural*.

Dados 't1' y 't2' objetos de 'T', la invocación `t1.compareTo(t2)` debe devolver un entero 'i' tal que:

$i < 0$	si t1 es menor que t2.
$i == 0$	si t1 es igual que t2.
$i > 0$	si t2 es menor que t1.

El comparador debe ser **consistente con `equals`** cuando se cumple que `t1.equals(t2)`, entonces `t1.compare(t2) == 0`. De la misma forma, si `t1.equals(t2)` también se debe cumplir la expresión `t1.compare(t2) == t2.compare(t1)`.

■ **Ejemplo 5.2** La clase `String` de Java implementa `Comparable<String>` y por tanto implementa el método 'compareTo' que se puede usar para comparar cadenas de caracteres. El método de comparación elegido computa el orden lexicográfico de las cadenas.

```
public class String implements Comparable<String> {
  public int compareTo(String s) {
    /* Comparación lexicográfica entre la cadena de este objeto y la
     * cadena 's' pasada como argumento
     */
    ...  // código aquí que no nos interesa ahora
  }
}
```

■

■ **Ejemplo 5.3** Ahora un ejemplo de uso:

```
public static void main(String [] s) {
  String s1 = "Terabyte";   /* 10^12 bytes.  */
  String s2 = "Terapeuta";  /* 10^12 peutas. */

  int r = s1.compareTo(s2);
```

```
  if (r < 0)
    System.out.println(s1 + " es menor que " + s2);
  else if (r == 0)
    System.out.println(s1 + " es igual que " + s2);
  else
    System.out.println(s1 + " es mayor que " + s2);
}
```

■

■ **Ejemplo 5.4** El siguiente método indica si una clave está en una lista de entradas, el código asume que `Comparable<K>` está implementado por la clase 'K' de las claves:

```
public static <K,V> boolean search(K k, Entry<K,V> [] array) {
  boolean found = false;
  int i = 0;
  while (i < array.length && !found) {
    found = array[i].getKey().compareTo(k) == 0 ;
    i ++;
  }
  return found;
}
```

■

5.2.2 Interfaz `java.util.Comparator`

El interfaz `Comparator<T>` permite implementar objetos que son "operadores" de comparación. El interfaz `Comparator<T>` se define:

```
public interface Comparator<T> {
   public int compare(T t1, T t2);
}
```

Los objetos de aquellas clases que implementen el interfaz `Comparator<T>` implementan un método de comparación `compare` para objetos de la clase 'T'. Dados 't1' y 't2' objetos de 'T' y 'c' un objeto comparador, la invocación `c.compare(t1,t2)` debe devolver un entero 'i' tal que:

$$\begin{array}{ll} i < 0 & \text{si t1 es menor que t2.} \\ i == 0 & \text{si t1 es igual que t2.} \\ i > 0 & \text{si t2 es menor que t1.} \end{array}$$

El comparador debe ser **consistente con `equals`** cuando se cumple que `t1.equals(t2)`, entonces `c.compare(t1,t2) == 0`. De la misma forma, si `t1.equals(t2)` también se debe cumplir la expresión `c.compare(t1,t2) == c.compare(t2,t1)`.

Se pueden usar diferentes objetos comparadores para comparar objetos de clase 'T'. Por ejemplo: podemos definir varios comparadores para la clase `String`: comparador lexicográfico, tamaño, coste de compresión bajo zip, etc.

■ **Ejemplo 5.5** Una clase de comparadores de cadenas de caracteres que las comparan según su longitud:

```
public class SizeStringComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
}
```

■

▪ **Ejemplo 5.6** Una clase de comparadores de cadenas de caracteres que las comparan según el orden lexicográfico:

```
public class LexicographicStringComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    int minLength = Math.min(s1.length(),s2.length());
    for (int i = 0; i < minLength && s1.charAt(i) == s2.charAt(i); i++) ;
    if (i < minLength) {
      return s1.charAt(i) < s2.charAt(i) ? -1 : 1 ;
    }
    else {
      return s1.length() - s2.length();
    }
  }
}
```

▪

▪ **Ejemplo 5.7** Un ejemplo sencillo que usa ambos comparadores:

```
  public static void main(String [] s) {
    String  s1 = "Terabyte";
    String  s2 = "Terapeuta";
    SizeStringComparator          sc = new SizeStringComparator();
    LexicographicStringComparator lc = new LexicographicStringComparator();
    int r;

    r = lc.compare(s1,s2);
    if (r < 0)
      System.out.println(s1 + " es menor que " + s2);
    else if (r == 0)
      System.out.println(s1 + " es igual que " + s2);
    else
      System.out.println(s1 + " es mayor que " + s2);

    r = sc.compare(s1,s2);
    if (r < 0)
      System.out.println(s1 + " es menor que " + s2);
    else if (r == 0)
      System.out.println(s1 + " es igual que " + s2);
    else
      System.out.println(s1 + " es mayor que " + s2);
  }
```

▪

▪ **Ejemplo 5.8** El siguiente método indica si una clave está en una lista de entradas, el código recibe un comparador para la clase 'K' de las claves:

```
  public static <K,V> boolean search(K k, Entry<K,V> [] array, Comparator<K> c){
    boolean found = false;
    int i = 0;
    while (i < array.length && !found) {
      found = c.compare(array[i].getKey(),k) == 0 ;
      i++;
    }
    return found;
  }
```

▪

■ **Ejemplo 5.9** Un comparador lexicográfico de puntos en dos dimensiones:

```
public class Lexicographic2DPointComparator implements Comparator<2DPoint> {
  private int xa, ya, xb, yb;
  public int compare(2DPoint a, 2DPoint b) {
    return a.getX() != b.getX() ? b.getX() - a.getX() : b.getY() - a.getY();
  }
}
```

■

`Comparator<T>` a partir de `Comparable<T>`

Si un objeto es de una clase que implementa el interfaz `Comparable<T>` entonces se puede crear un objeto `Comparator<T>` por defecto para dicha clase:

```
import java.util.Comparator;
import java.lang.Comparable;

public class DefaultComparator<T> implements Comparator<T> {
  public int compare(T a, T b) throws UnsupportedOperationException {
    if (a instanceof Comparable<?>) {
       return ((Comparable<T>) a).compareTo(b);
    }
    else {
       throw new UnsupportedOperationException();
    }
  }
}
```

Si 'T' implementa `Comparable<T>` entonces dispone de 'compareTo' que puede usarse para comparar. En otro caso se lanza la excepción.

```
  public static void main (String [] arg) {
    DefaultComparator<String> c = new DefaultComparator<String>();
    String s1 = new String("aa");
    String s2 = new String("bb");
    System.out.println(c.compare(s1,s2));

    DefaultComparator<Object> o = new DefaultComparator<Object>();
    try {
      System.out.println(o.compare(new Object(), new Object()));
    } catch (UnsupportedOperationException e) {
      System.out.println("Object no implementa Comparable<Object>");
    }
  }
```

El problema de este comparador por defecto es que la excepción se lanza al invocarse 'compare'. Sería mejor que la excepción se lanzase al crear el objeto comparador. Para hacer bien esto hay que usar trucos de Java, en particular el API de reflexión que nos permite pasar a un constructor un valor en tiempo de ejecución que representa el tipo 'T' de un elemento. El constructor puede entonces comprobar si ese tipo implementa `Comparable<?>` y lanzar la excepción en caso negativo.

6. Listas de Posiciones

Ya hemos visto un tipo de lista, la lista indexada `IndexedList<E>` en el Tema 2, en la que, para acceder a los elementos, se utiliza un índice numérico. En la asignatura veremos otro posible TAD de tipo lista: las **listas de posiciones**, y veremos los detalles de una implementación mediante cadenas de objetos nodo doblemente enlazados ("doubly-linked"), es decir, cada nodo que compone la lista tiene un puntero al siguiente elemento y puntero al elemento anterior. Las listas de posiciones no utilizan un índice numérico para el acceso a los elementos sino que utilizan como medio de acceso el nodo en el que están cotenidos los elementos de la lista. La lista se puede recorrer solicitando el siguiente/anterior nodo de la lista hasta que ya no haya nodos pendientes de recorrer porque estémos en la última/primera posición.

¿Qué operaciones definen una lista?

De forma similar a lo descrito para los números complejos en el Tema 1, el TAD de listas vendrá dado por uno (o más) interfaces que definen la funcionalidad de la lista, y por una (o más) clases que implementan el interfaz. Primero nos centramos en el interfaz y en el uso del mismo, ilustrando así la importancia y los beneficios de la abstracción de datos: escribiremos código que únicamente *usa* el interfaz de listas de posiciones sin saber realmente cómo se implementan dichas listas. Podemos decir que este código sería un "cliente" del TAD lista:

De acuerdo a lo visto en Programación II, observamos que la inserción, búsqueda y borrado de elementos usando índices puede ser un poco diferente al uso de un array:

```
list.insert(0, "vais");         // ["vais"]

list.insert(1, "a");            // ["vais", "a"]

list.insert(2, "aprobar");      // ["vais", "a", "aprobar"]

list.insert(0, "no");           // ["no", "vais", "a", "aprobar"]

list.set(3, "suspender");       // ["no", "vais", "a", "suspender"]

list.remove(2);                 // ["no", "vais", "suspender"]

list.set(1, "quiero");          // ["no", "quiero", "suspender"]
```

Los métodos 'insert', 'remove' y 'set' tienen que posicionarse en el nodo con número correspondiente, y el programador debe recordar y llevar actualizada la correspondencia entre números y nodos. Además está el tema del control de rango, p.ej., si hacemos `list.set(200,"Fuera de rango")` tenemos un problema.

Nuestro objetivo es poder insertar y borrar los nodos directamente sin comprometer la abstracción de datos, es decir, sin que el cliente de la lista tenga que conocer cómo se implementa un nodo ni cómo se enlaza en la lista.

6.1 Los interfaces: Position y PositionList

Interfaz `Position<E>`

Una `Position` es una abstracción de un nodo, es decir, el interfaz `Position<E>` ofrece un único método `element` que debe devolver el elemento almacenado en el nodo. Cualquier otra cosa no le interesa al cliente de la lista. Otra cosa es la implementación del interfaz, que veremos más adelante.

```
public interface Position<E> {
  public E element ();
}
```

Este interfaz se utilizará tanto para listas de posiciones como para otros TADs que veremos más adelante.

Interfaz `PositionList<E>`

El interfaz `PositionList<E>` utiliza el interfaz `Position<E>` para abstraer la clase de los nodos: los objeto nodo de la lista serán objetos de una clase que implemente `osition<E>`. De una posición únicamente podemos obtener el elemento que almacena. Recordad que `Position` (posición) significa "nodo abstracto".

Nótese que el interfaz `PositionList<E>` extiende `Iterable<E>`: de momento ignoramos ésto, ya volveremos sobre ello en el tema de Iteradores.

```
public interface PositionList<E> extends Iterable<E> {

  public int size();

  public boolean isEmpty();

  public Position<E> first();

  public Position<E> last();

  public Position<E> next(Position<E> p) throws IllegalArgumentException;

  public Position<E> prev(Position<E> p) throws IllegalArgumentException;

  public void addFirst(E elem);

  public void addLast(E elem);

  public void addBefore(Position<E> p, E elem) throws IllegalArgumentException;

  public void addAfter(Position<E> p, E elem) throws IllegalArgumentException;

  public E remove(Position<E> p) throws IllegalArgumentException;

  public E set(Position<E> p, E elem) throws IllegalArgumentException;

  public Object [] toArray();

  public E [] toArray(E[] a);
}
```

La documentación completa sobre los métodos la podéis encontrar en el javadoc de la clase `PositionList`. A modo de resumen, algunos comentarios sobre los métodos:

- Interrogadores: `size`, `isEmpty`.
- Acceso ("getters"): `first`, `last`, `next` y `prev`. Permiten moverse por las posiciones de la lista. Devuelven 'null' si no existe el nodo al que se pretende acceder. De esta forma se puede usar null como valor centinela en bucles.
- Inserción: `addFirst`, `addLast`, `addBefore` y `addAfter`.
- Borrado: `remove` borra una posición, dejando las otras posiciones "conectadas".
- Modificación: `set` modifica el elemento de una posición.
- Conversión: el método `toArray` devuelve un vector (array) con los elementos de la lista en el mismo orden posicional de izquierda a derecha. Devuelve un vector de 'Object' (mucho cuidado con los arrays de genéricos, para tratarlos Java tiene "serios" problemas).
- La excepción 'IllegalArgumentException' será lanzada cuando la posición pasada al método sea null o no sea una posición de la lista (p.ej. cuando sea un nodo de otra lista).

Una forma de entender el uso del interfaz es ver una lista como un objeto al que le podemos pedir cajas (con `first` o `last` de las cuales sólo podemos sacar el elemento que contienen (método `element` del interfaz `Position<E>`). Tenemos que darle al objeto lista la caja que tenemos para que nos de la siguiente o la anterior. También tenemos que darle la caja para cambiar el elemento. El interfaz no especifica si la inserción conlleva *copia superficial* ("shallow copy") o *copia profunda* ("deep copy") de los elementos. Con la copia superficial se insertan (las referencias a) los objetos elemento en la lista, mientras que con la copia profunda se insertan (las referencias a) copias de los objetos elemento en la lista. Como norma general las estructuras de datos utilizan la copia superficial. Todos los interfaces que veremos en la asignatura asumen copia superficial, dejando al usuario del interfaz la responsabilidad de hacer copias profundas. Para poder hacer copia profunda los objetos elemento deben tener constructor de copia (o el método 'clone', si bien el uso de dicho método es cuestionado, ved por ejemplo:

`http://www.javapractices.com/topic/TopicAction.do?Id=71`).

El uso de copia superficial puede ser problemático ya que la referencia al objeto elemento podría estar compartida. A este problema se le llama *aliasing*.

Por ejemplo, supongamos que insertamos un mismo contacto en dos listas distintas.

```
Contacto c = new Contacto("LuisMi");

list1.addFirst(c);
list2.addFirst(c);

c.setName("LuisMa");
```

Ambas listas comparten el mismo elemento que ha quedado modificado desde "fuera"de la lista.

Además, el elemento puede modificarse desde "dentro"de la primera lista quedando también modificado en la segunda:

```
Contacto c = new Contacto("LuisMi");

list1.addFirst(c);
list2.addFirst(c);

list1.first().element().setName("Luisito");
```

Para evitar alias el usuario del interfaz debe hacer copia profunda explícita con el constructor de copia:

```
Contacto c = new Contacto("LuisMi");

list1.addFirst(new Contacto(c));
list2.addFirst(new Contacto(c));

c.setName("LuisMa");
list1.first().element().setName("Luisito")
```

Ahora 'c' referencia un objeto "LuisMa", el primer elemento de 'list1' referencia un objeto "Luisitoz el primer elemento de 'list2' referencia un objeto "LuisMi".

Ejemplo de uso

Ahora actuaremos como clientes que quieren usar el TAD de listas, es decir, únicamente disponemos del interfaz `Position<E>` y de `PositionList<E>` que documenta lo que hace cada método. Asumimos que disponemos de una clase que implementa el interfaz. De momento desconocemos los detalles de dicha clase (los veremos más adelante), y como clientes del interfaz, realmente no nos interesan. Nos basta con conocer los constructores. Se nos dice que hay un constructor que crea una lista vacía (sin nodos). Podemos empezar a escribir código sin saber nada acerca de los nodos o la implementación, salvo que tenemos a nuestra disposición una clase `NodePositionList<E>` con un constructor que construye una lista vacía.

```
PositionList<String> list = new NodePositionList();  // lista vacia

Position<String> cursor;          // cursor a nodos

list.addFirst("vais");            // ["vais"]

list.addLast("a");                // ["vais", "a"]
```

```
list.addLast("aprobar");          // ["vais", "a", "aprobar"]

list.addFirst("no");              // ["no", "vais", "a", "aprobar"]

cursor = list.last();

list.set(cursor, "suspender");    // ["no", "vais", "a", "suspender"]

cursor = list.prev(cursor);

list.remove(cursor);              // ["no", "vais", "suspender"]

cursor = list.prev(list.last());

list.set(cursor, "quiero");       // ["no", "quiero", "suspender"]
```

El programador tiene que seguir llevando cuenta de dónde está cada elemento, pero no es necesario recordar la correspondencia entre nodos e índices. Además, los métodos no tienen que empezar a operar desde el comienzo o el fin de la lista sino desde la posición dada.

Recorridos de listas de posiciones

■ **Ejemplo 6.1** Mostar por pantalla todos los elementos de una lista. Observad el uso de **`null`** como posición *centinela* en la condición del bucle. Observad que usamos *métodos genéricos* en los ejemplos.

```
// con bule while

public static <E> void show(PositionList<E> list) {
  Position<E> cursor = list.first();
  while (cursor != null) {
    System.out.println(cursor.element());
    cursor = list.next(cursor);
  }
}

// con bucle for
public static <E> void show(PositionList<E> list) {
  for (Position<E> cursor = list.first();
       cursor != null;
       cursor = list.next(cursor))
    System.out.println(cursor.element());
}
```

Nota: éstos métodos son estáticos y genéricos porque no están dentro de la clase que implementa el interfaz.

■

Consideraciones generales

- Se recorren las listas de forma secuencial usando bucles y nodos cursor.
- La inicialización consistirá en hacer que el cursor referencie la primera (o la última) posición de la lista. Si la lista está vacía entonces el cursor será null.
- La condición de parada (la contraria de la de continuación) del bucle la marca el problema, pero suele incluir la condición de control de rango: el cursor debe ser distinto de null.
- El incremento consistirá en avanzar el cursor a la siguiente (o anterior) posición.
- Frecuentemente se usan iteradores para recorrer listas, pero para entender cómo funcionan tenemos que aprender a recorrer listas mediante cursores. Un iterador es una abstracción de un cursor.
- Cuidado: en ocasiones `cursor.element().equals(x)` puede ser preferible a `x.equals(cursor.element())`. Hay que tener en cuenta si la lista puede contener elementos null, y si los elementos a comparar (los 'x') pueden ser null. Si la lista no contiene elementos null entonces 'cursor.element().equals(x)' es seguro. En cualquier cas esto sería seguro:

```
cursor.element() != null && cursor.element().equals(x)
```

▪ **Ejemplo 6.2** Insertar un elemento antes de la primera aparición de otro. El siguiente método toma como argumento una lista 'list' y dos elementos 'a' y 'e'. El método añade a la lista el elemento 'a' antes de la primera aparición en la lista del elemento 'e'. Si 'e' no está en la lista entonces el método deja la lista intacta:

```
public static <E> void addBeforeElement(PositionList<E> list, E e, E a) {
  Position<E> cursor;
  for (cursor = list.first();
       cursor != null && !cursor.element().equals(e);
       cursor = list.next(cursor))
       ; // No hay cuerpo del bucle

      // cursor == null || cursor != null && cursor.element().equals(e);
      if (cursor != null) list.addBefore(cursor,a);
   }
}
```

▪

▪ **Ejemplo 6.3** Borrar todas las apariciones de un elemento dado. Recordad algo importante: *antes de borrar hay que avanzar.*

```
public static <E> void deleteAll(E elem, PositionList<E> list) {
  Position<E> cursor = list.first();
  while (cursor != null) {
    if (cursor.element().equals(elem)) {
      Position<E> borrar = cursor;
      cursor = list.next(cursor);       // avanza
      list.remove(borrar);              // borra
    } else
      cursor = list.next(cursor);       // avanza
  }
}
```

Variante que invoca `list.next()` una vez:

```
public static <E> void deleteAll(E elem, PositionList<E> list) {
  Position<E> cursor = list.first();
  while (cursor != null) {
    Position<E> nxt = list.next(cursor);  // avanza
    if (cursor.element().equals(elem))
      list.remove(cursor);                // borra
    cursor = nxt;
  }
}
```

Algunas variantes del condicional anterior que NO FUNCIONAN:

```
if (cursor.element().equals(elem)) {
   list.remove(cursor);
   cursor = list.next(cursor);      // IllegalArgumentException
} ...
```

```
if (cursor.element().equals(elem)) {
   cursor = list.next(cursor);
   list.remove(list.prev(cursor));  // IllegalArgumentException
} ...
```

```
if (cursor.element().equals(elem)) {
  Position<E> borrar = cursor;
  list.remove(cursor);
```

```
    cursor = list.next(borrar);       // IllegalArgumentException
  } ...
```

■

■ **Ejemplo 6.4** El peligro de usar contadores de elementos: usar un contador de elementos recorridos no es necesario y puede ser peligroso dependiendo del problema a resolver:

```
// no funciona
public static <E> void show(PositionList<E> list) {
  for (int i = 0; i < list.size(); i++)
    System.out.println(cursor.element());
}


// redundante
public static <E> void show(PositionList<E> list) {
  Position<E> cursor = list.first();
  for (int i = 0; i < list.size(); i++) {
    System.out.println(cursor.element());
    cursor = list.next(cursor);
  }
}

// redundante y no funciona
public static <E> void deleteAll(E elem, PositionList<E> list) {
  Position<E> cursor = list.first();
  int i = 0;
  while (i < list.size()) {
    Position<E> nxt = list.next(cursor);
    if (cursor.element().equals(elem))
      list.remove(cursor);
    cursor = nxt;
    i++;
  }
}
```

■

■ **Ejemplo 6.5** Intercalar los elementos de dos listas hasta que acabe la más pequeña. El siguiente método toma como parámetros dos listas y devuelve una nueva lista con los elementos de las dos listas intercalados hasta terminar de recorrer la lista más pequeña. Si alguno de los parámetros es null o referencia una lista vacía entonces el método devuelve una nueva lista vacía.

```
public static <E> PositionList<E> shuffleShorter(PositionList<E> list1,
                                                  PositionList<E> list2) {

  PositionList<E> result = new NodePositionList<E>();

  if (list1 == null || list2 == null || list1.isEmpty() || list2.isEmpty())
    return result;

  Position<E> cursor1 = list1.first();
  Position<E> cursor2 = list2.first();

  while (cursor1 != null && cursor2 != null) {
    result.addLast(cursor1.element());
    result.addLast(cursor2.element());
    cursor1 = list1.next(cursor1);
    cursor2 = list2.next(cursor2);
  }
```

```
    return result;
  }
```

■

Ejercicio 6.1 Modificar el código para que añada al final de la lista resultado los elementos sobrantes de la lista de mayor tamaño. Hay que modificar también la condición frontera: si una lista es null o vacía y la otra no entonces el método debe devolver una copia de la lista no null y no vacía. ■

■ **Ejemplo 6.6** Invertir los elementos de una lista. El siguiente método toma una lista y devuelve una nueva lista con los elementos de la primera en orden inverso. Si la lista parámetro es null entonces el método devuelve null. Si la lista parámetro es vacía entonces el método devuelve una nueva lista vacía.

```
public static <E> PositionList<E> reverseList(PositionList<E> list) {
  if (list == null) return null;
  PositionList<E> result = new NodePositionList<E>();
  if (list.isEmpty()) return result;

  Position<E> cursor = list.first();
  while (cursor != null) {
    result.addFirst(cursor.element());
    cursor = list.next(cursor);
  }

  /* Otra posibilidad es recorrer la lista de derecha a izquerda:

  Position<E> cursor = list.last();
  while (cursor != null) {
    result.addLast(cursor.element());
    cursor = list.prev(cursor);
  }

  */

  /* Podemos en ambos casos usar un bucle for, por ejemplo:

     for (Position<E> cursor = list.last();
          cursor != null;
          cursor = list.prev(cursor))
       result.addLast(cursor.element());
  */

  return result;
}
```

■

■ **Ejemplo 6.7** El siguiente método invierte los elementos de la lista parámetro directamente sin devolver una nueva lista. Usa los métodos `remove` y `addFirst` y por tanto borra nodos (no todos) de la lista y crea nuevos nodos. Recorre la lista desde el final.

```
public static <E> void reverseInPlace(PositionList<E> list) {
  if (list != null && !list.isEmpty() && list.size() > 1) {
    Position<E> cursor = list.last();
    Position<E> end    = list.first();
    while (cursor != end) {
      Position<E> prv = list.prev(cursor);
      E element       = list.remove(cursor);
```

```
          list.addFirst(element);
          cursor = prv;
        }
      }
    }
```

La condición frontera `list.size() > 1` se añade por claridad, pero el código funciona igualmente sin ella.

Si modificamos el código anterior para recorrer la lista desde el principio el resultado es erróneo, tanto si se usa `addFirst` como `addLast`.

```
public static <E> void reverseInPlace(PositionList<E> list) {
  if (list != null && !list.isEmpty() && list.size() > 1) {
    Position<E> cursor = list.first();      // MAL
    Position<E> end    = list.last();
    while (cursor != end) {
      Position<E> nxt = list.next(cursor);  // MAL
      E element       = list.remove(cursor);
      list.addLast(element);                // MAL
      cursor = nxt;
    }
  }
}
```

Una versión correcta que recorre la lista desde el principio. Usa 'addAfter' sobre el último nodo.

```
public static <E> void reverseInPlace(PositionList<E> list) {
  if (list != null && list.isEmpty() && list.size() > 1) {
    Position<E> cursor = list.first();
    Position<E> end    = list.last();
    while (cursor != end) {
      Position<E> nxt = list.next(cursor);
      E element = list.remove(cursor);
      list.addAfter(end, element);
      cursor = nxt;
    }
  }
}
```

Una versión alternativa que surge de observar en la solución anterior que el nodo a borrar (el que referencia `cursor`) siempre es el primero, y que despúes del borrado la variable `cursor` referencia el objeto que referencia 'nxt', que siempre es el primero. Por tanto `cursor` y `nxt` no son necesarios y se puede usar directamente `list.first()`.

```
public static <E> void reverseInPlace(PositionList<E> list) {
  if (list != null && list.isEmpty() && list.size() > 1) {
    Position<E> end = list.last();
    while (list.first() != end) {
      list.addAfter(end, list.first().element());
      list.remove(list.first());
    }
  }
}
```

Esta versión invierte los elementos de la lista sin crear nodos nuevos. Modifica los elementos de los nodos existentes.

```
public static <E> void reverseSwap(PositionList<E> list) {
  if (list != null && list.isEmpty() && list.size() > 1) {
    Position<E> start = list.first();
    Position<E> end   = list.last();
```

```
        while (start != end) {
          // swap elements
          E tmp = start.element();
          list.set(start,end.element());
          list.set(end,tmp);
          // move
          if (list.next(start) == end)
            start = end;
          else {
            start = list.next(start);
            end   = list.prev(end);
          }
        }
      }
    }
```

■

■ **Ejemplo 6.8** Borrar un cierto rango de valores de una lista. El siguiente método borra de una lista desde el nodo número 'i' hasta el nodo número 'j'. Los nodos se numeran de 0 a list.size()-1.

```
  public static <E> void removeRange(PositionList<E> list,
                                     int i,
                                     int j)
    throws IllegalArgumentException {

    if (! (0 <= i && i <= j && j < list.size()) )
        throw new IllegalArgumentException("Invalid range");

    if (list != null && !list.isEmpty()) {
      Position<E> cursor = list.first();
      int pos = 0;
      while (pos != i) {  // posicionarse
        cursor = list.next(cursor);
        pos++;
      }
      while (pos <= j) {  // borrar
        Position<E> nxt = list.next(cursor);
        list.remove(cursor);
        cursor = nxt;
        pos++;
     }
    }
  }
```

La siguiente variación se posiciona y borra en un único bucle:

```
  public static <E> void removeRange(PositionList<E> list,
                                     int i,
                                     int j)
    throws IllegalArgumentException {

    if (! (0 <= i && i <= j && j < list.size()) )
        throw new IllegalArgumentException("Invalid range");

    if (list != null && !list.isEmpty()) {
      Position<E> cursor = list.first();
      int pos = 0;
      while (pos <= j) {
        Position<E> nxt = list.next(cursor);
        if (pos >= i) list.remove(cursor);
        cursor = nxt;
        pos++;
```

```
        }
      }
    }
```

■

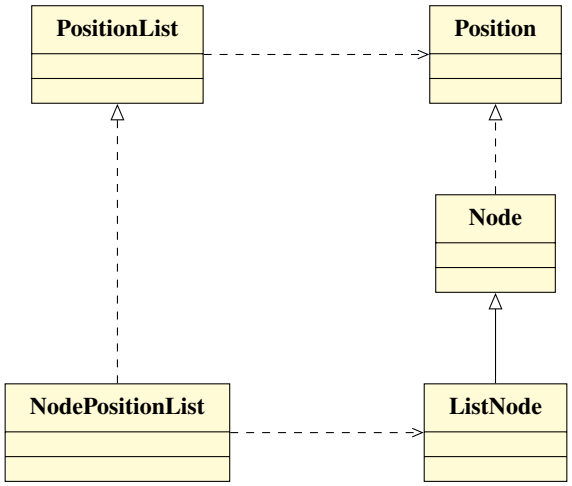
■ **Ejemplo 6.9** Cortar la lista a un cierto tamaño. El siguiente método modifica una lista para quedarse con los 'size' primeros nodos. Si 'size' es negativo deja la lista intacta. Si 'size' es 0 entonces deja la lista vacía. Si size es mayor que el tamaño de la lista entonces deja la lista intacta. El método recorre la lista desde el final para dejar los 'size' primeros nodos en la lista.

```
public static <E> void trimToSize(PositionList<E> list, int size) {

  if (list != null && !list.isEmpty() &&
      size >= 0 && size < list.size()) {

    Position<E> cursor = list.last();
    size = list.size() - size;
    while (size > 0) {
      Position<E> prv = list.prev(cursor);
      list.remove(cursor);
      cursor = prv;
      size--;
    }
  }
}
```

■

6.2 Implementación mediante cadenas de nodos doblemente enlazados

El diagrama de clases correspondiente a la implementación sería:



Clase `Node<E>` y `ListNode<E>`

La clase `Node<E>` implementa los objetos nodo para múltiples estructuras de datos. Implementa el interfaz `Position<E>` y por tanto provee código para el método 'element' declarado en dicho interfaz. Para su uso en listas se ha implementado la clase `ListNode<E>` que extiende de la clase `Node<E>`. El código fuente de `Node<E>`, `ListNode<E>` y de `NodePositionList<E>` se encuentra disponible en el Moodle de la asignatura.

Entre la clase `Node<E,O>` y `ListNode<E>` tienen los siguientes atributos y métodos:

- `owner` es un objeto que identificará unívocamente la lista a la que pertenece el nodo. Está declarado como 'final', lo que significa que una vez inicializado su valor por el constructor ya no podrá modificarse.
- `prev` y `next` son respectivamente referencias al nodo previo y al nodo siguiente en la lista.
- `elem` referencia el elemento que se almacena en el nodo.

Los métodos y constructores de `Node<E>`:

- La clase provee un constructor y también "gettersz "setters"para 'prev', 'next' y 'elem'. (El método `element` es el "getter"de 'elem'.)
- Intencionalmente, no hay constructor de copia ni se sobrescribe `equals`.
- No hay "setter"para `owner` ya que este atributo no puede modificarse al ser declarado como **`final`**.
- No hay "getter"para `owner`. Solamente un objeto nodo puede determinar usando el método `kinOf` si otro objeto nodo tiene el mismo 'owner' ("kin" significa "pariente" en Inglés). Por ejemplo, dado un nodo 'n1' distinto de null y un nodo 'n2', la invocación `n1.kinOf(n2)` indicará si 'n1' y 'n2' tienen el mismo `owner`.
- Los "setters" `setPrev` y `setNext` de los atributos `prev` y `next` pueden tomar como parámetros válidos tanto el valor null como una referencia a un nodo con el mismo `owner`. En otro caso lanzan la excepción `IllegalArgumentException`. (¿Qué pasa si el nodo parámetro es 'this'?).

Como veremos en la clase `NodePositionList<E>`, permitir **`null`** tiene sentido si queremos borrar un nodo de la lista desenlazándolo de su nodo previo y de su nodo siguiente poniendo 'next' y 'prev' a null. Por otro lado el constructor también permite que los parámetros que se le pasan sean null. Uno de los constructores de la clase `NodePositionList<E>` tiene que crear un nodo con un atributo temporalmente a null para poder enlazar los nodos centinela.

Clase `NodePositionList<E>`

Tiene tres atributos:

- `size`, que almacena el tamaño o número de nodos de la lista, representado mediante un entero.
- Dos nodos centinela, `header` y `trailer`, que no almacenan elementos y son usados para marcar el principio y el fin de la lista.

El propósito de los nodos centinela es que al insertar/borrar siempre haya un nodo previo y otro siguiente al que enlazar/desenlazar. El código de inserción/borrado (métodos `addFirst`, `addBefore`, `remove`, etc.) resulta más sencillo y uniforme que si tuviéramos referencias al primer y último nodo útiles. Por ejemplo, cuando se inserta un nodo por primera vez, el nodo previo es el `header` y el nodo siguiente es el siguiente al 'header' que en esta ocasión es el nodo `trailer`. Si inmediatamente después se inserta tro nodo al principio de la lista, de nuevo el nodo previo es el 'header' y el nodo siguiente es el siguiente al 'header' que en esta ocasión es el nodo insertado anteriormente.

El tamaño de la lista *está acotado por la memoria* disponible, en concreto, por la memoria dinámica disponible para crear nuevos objetos nodo en los métodos de inserción 'addFirst', etc.

Idealmente el tipo del atributo 'size' debería ser concordante con la memoria dinámica disponible. Hemos considerado que 'int' es suficiente. En Java 'int' es un entero de 32 bits con valor máximo 2.147.483.647. Para los propósitos de esta asignatura es suficiente. También está disponible el tipo 'long' de enteros de 64 bits cuyo valor máximo es 9.223.372.036.854.775.807. La memoria dinámica disponible depende de la arquitectura y de otros factores circunstanciales como el número de procesos ejecutándose, etc. La correspondencia entre 'size' y la memoria dinámica (número de nodos que podemos insertar) no es una cuestión meramente teórica. Los enteros de tamaño fijo puede sufrir desbordamiento ("overflow"). Por ejemplo:

```
int i = 2147483647;
System.out.println(++i);
```

el valor mostrado por consola es -2147483648.

Por tanto, la inserción con éxito (hay memoria disponible) de un elemento en una lista con 2147483647 elementos conllevaría a un valor negativo de `size`. Esta problemática afecta a otras estructuras de datos. No volveremos a incidir en ella pero tenedla presente.

Algunos comentarios sobre los métodos de `NodePositionList<E>`:

- El primer constructor crea una lista vacía que consiste en 'header' y 'trailer' encadenados.
- El segundo constructor construye la lista tomando los elementos de un array. Usa el "for-each"de java que veremos en detalle en el tema de [[*Iteradores]].
- El tercer constructor (constructor de copia) toma los elementos (¡no los nodos!) de otra lista. No se copian los elementos, se copian sus referencias, es decir, hace "shallow copy". Los constructores segundo y tercero no comprueban si el vector o la lista es null ya que en ese caso sí debe lanzarse `NullPointerException`.
- El primer constructor y los métodos de inserción 'addFirst', 'addLast', 'addBefore' y 'addAfter' son los únicos que crean nodos. Al crearlos los marcan como pertenecientes a la lista. En concreto, le pasan al constructor de nodos un entero generado por un método privado 'iayf' (Ï am your father") que encapsula la forma de generar el entero con que marcar los nodos. (Si quisiéramos cambiar el método de marcado sólo tenemos que modificar el método 'iayf'.)
- En la implementación actual, el método `iayf` llama al método `hashCode` de la superclase de `NodePositionList`, esto es, de `Object`. Según la especificación del lenguaje Java, el método `hashCode` de `Object` genera un entero único para cada objeto de clase `Object`. Por tanto, el valor de `owner` de un nodo será un entero que identifica unívocamente el objeto lista al que pertenece. Observad que `NodePositionList<E>` podría

sobrescribir el método `hashCode` sin problema porque `iayf` invoca explícitamente el `hashCode` de `Object` usando **`super`**. Como apunte sobre `hashcode` decir que el método 'hashCode' de un objeto arbitrario debe generar un entero que puedan usar las tablas de dispersión ("hash tables") como valor de dispersión ("hash") del objeto.

`http://eclipsesource.com/blogs/2012/09/04/the-3-things-you-should-know-about-hashcode/.`

Incidiremos sobre esto más adelante en el tema de tablas de dispersión.

- El método privado `checkNode` comprueba si una posición es realmente un nodo válido de la lista, es decir:
  1. No es 'null'.
  2. Es de clase 'Node<E>'.
  3. Es un nodo de la lista (se comprueba haciendo 'kinOf' con el 'header').
  4. Está encadenado a un nodo previo y a un nodo siguiente.
- El método `iterator`, los métodos sobrescritos `toString` e `equals`, así como el método `toArray` los explicaremos cuando veamos el tema de Iteradores. Simplemente observamos que el método `equals` compara **`this`** con cualquier objeto 'o' que implemente el interfaz `PositionList<E>`, el cual no tiene que ser de clase 'NodePositionList' necesariamente. El concepto de iterador nos permitirá implementar este tipo de comparación de igualdad general en otros TADs donde los métodos ofrecidos por del interfaz no son suficientes para recorrerlo. En el caso de las listas se podría haber implementado el `equals` general usando el cursor como en 'toString' porque el interfaz `PositionList<E>` ofrece métodos para recorrer la lista.

Complejidad de los métodos de la clase NodePositionList<E>

Nos interesa la complejidad (coste) en tiempo (de ejecución) y espacio (consumo de memoria) en el "caso peor" (el peor escenario posible). A modo de recordatorio:

- O(1): la complejidad en tiempo del método en el caso peor se mantiene constante independientemente de lo que aumente el tamaño de la lista
- O(n): la complejidad en tiempo del método en el caso peor es linealmente proporcional al tamaño 'n' de la lista. Es decir, si la lista tiene 'n' elementos entonces la complejidad es 'n'. Si el tamaño es 'n+1' entonces la complejidad es 'n+1', etc.

La complejidad temporal en el caso peor de todos los métodos (excepto los de iteración) es O(1). Por ejemplo, en la inserción y el borrado se le proporciona al los métodos 'addAfter', 'addBefore' y 'remove' el nodo con el elemento a insertar. Los métodos únicamente tienen que modificar los enlaces de los nodos, cuyo coste es constante respecto de (o "no depende de") el tamaño de la lista. Los métodos 'addFirst' y 'addLast' toman elementos, no nodos, pero crear un objeto nodo también tiene complejidad constante.

Ejercicio 6.2 Dado un elemento, si no se tiene su posición entonces habría que buscarla. Añadir a la clase e implementar el método:

```
public Position<E> getPos(E e)
```

que debe devolver la primera posición en la lista que contiene el elemento 'e', o null si el elemento no está en la lista. ¿Cuál sería la complejidad en el caso peor de dicho método? ■

Listas de la Java Collections Framework

La Java Collection Framework (JCF) es una librería de TADs estándar de Java

`https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`

Los interfaces importantes de la JCF son `Collection<E>` y `Map<K,V>`. A resaltar:

- Usado en muchas aplicaciones reales.
- El código fuente de las clases no está disponible y por tanto no podemos usarlo para docencia.
- Se ofrecen muchos métodos útiles: conversión a array y viceversa, reordenación ('sort', 'shuffle', 'reverse', etc), computación con segmentos, búsqueda por patrones, etc.
- `Collection<E>` extiende `Iterable<E>` pero 'Map<K,V>' no. Veremos las consecuencias de ésto en el tema de Iteradores.

Tradicionalmente los TADs (incluidos las listas) se usan para insertar, buscar y borrar elementos directamente o a través de índices, no se usan abstracciones como 'Position<E>'. Sin embargo, 'Position<E>' es un TAD útil que permite obtener una buena complejidad y además nos permite estudiar la implementación mediante cadenas de nodos

doblemente enlazados. La Java Collection Framework ofrece varios TADs lista. Ninguno usa 'Position<E>'. Se usan índices y/o elementos. Los interfaces de lista extienden `Collection<E>` que a su vez extiende `Iterable<E>`.

- Interfaces: `List<E>` (usa índices) y `ListIterator<E>`. Hay además clases abstractas intermedias: `AbstractList<E>`, `AbstractSequentialList<E>`, etc.
- Implementaciones: `ArrayList<E>`, `LinkedList<E>`, y otras más como pueden ser `AttributeList`, `CopyOnWriteArrayList`, `RoleList`, `RoleUnresolvedList`, `Vector`, etc.

Las dos más utilizadas serían: (1) `ArrayList<E>`, que implementa listas usando vectores cuyo tamaño puede "modificarse" (realmente se crea un vector nuevo del tamaño deseado y se copian elementos); (2) la clase `LinkedList<E>`, que es parecida a `NodePositionList<E>`, pero se usan directamente elementos (no `Position<E>`) y se permite el uso de índices, puesto que hereda de `List<E>`.

7. Iteradores

Muchos programas tienen que iterar sobre los elementos de un TAD). Por ejemplo, con las listas de posiciones, los programas iteran linealmente sobre la lista: posicionándose en el nodo 'first' (o el 'last'), avanzando dentro de un bucle con 'next' (o retrocediendo con 'prev'), y usando 'null' como valor centinela del bucle, junto con otras posibles condiciones. Los elementos se obtienen invocando el método 'element' sobre el objeto nodo correspondiente. Recordad lo visto en los recorridos de listas de posiciones. Por lo leneral una iteración sobre un TAD se realiza mediante bucles. Se invocan los métodos del interfaz del TAD para obtener elementos y moverse. (Esto es posible si hay suficientes métodos observadores que permitan realizar la iteración.)

Concepto de iterador

Ese código tiene varios problemas: (1) Es específico para listas de posiciones porque se usan datos y métodos específicos de su interfaz: cursor de tipo 'Position<E>', métodos 'first' y 'next', y uso de 'null' como centinela. No se puede adaptar el código fácilmente para iterar sobre otros TADs como, por ejemplo, colas FIFO; (2) el programador tiene que declarar el cursor, inicializarlo adecuadamente, e incrementarlo adecuadamente. Hay varias oportunidades para cometer errores y generar una 'IllegalArgumentException'.

Se puede abstraer el cursor y tener código fácilmente reusable para otros TADs convirtiendo el cursor en un TAD llamado Iterador.* Un objeto iterador permite iterar *linealmente* (uno a uno) los elementos de *otro* objeto TAD que almacena elementos, de forma que la iteración no se realiza usando los métodos del interfaz del TAD sino usando los métodos del interfaz del iterador y se puede reutilizar el código para iterar otros TADs.

Mostramos un ejemplo con 'show':

```
public static <E> void show(PositionList<E> list) {

  Position<E> cursor = list.first(); // Inicializacion.

  while (cursor != null) {                  // Bucle mientras el cursor
                                            // referencie a donde hay
                                            // un elemento.

    System.out.println(cursor.element());   // Se hace algo con el elemento.

    cursor = list.next(cursor);             // Se avanza el cursor.
  }
}
```

He aquí una forma de abstraer el cursor en un objeto iterador:

```
public static <E> void show(PositionList<E> list) {

  Iterator<E> it = list.iterator();    // La lista nos da un objeto iterador
                                       // (un cursor) ya inicializado.

  while (it.hasNext()) {               // Bucle mientras el cursor
```

```
                                    // referencie a donde hay un elemento.

      System.out.println(it.next());   // Se hace algo con el elemento
                                       // y queda el cursor avanzado.
   }
}
```

El método `it.next()` devuelve el elemento accesible desde el cursor (no devuelve un nodo), y además dejará el cursor avanzado, lo que constituye un *efecto secundario* o *side effect* parecido a un post-incremento del índice en la iteración de un vector:

```
   int i = 0;                          // Inicialización

   while (i < arr.length) {            // Bucle mientras el índice
                                       // está en rango

     System.out.println(arr[i++]);     // Se hace algo con el elemento
                                       // y queda el índice avanzado.
   }
```

En la práctica, el objeto iterador puede mantener un cursor como atributo, pero habrá casos en los que no será necesario. El cursor podrá referenciar el elemento directamente, o un objeto que contiene un elemento, p.ej., el nodo de una lista, etc. Por ello he escrito "el cursor referencia a donde hay un elemento". Ahora el código con el objeto iterador se puede adaptar fácilmente a otros TADs que implementen el iterfaz `Iterable<E>`. Lo único que cambia es que se le pide el iterador a otro TAD en vez de a una lista, el resto permanece igual.

```
   public static <E> void show(Iterable<E> tad) {

     Iterator<E> it = tad.iterator();    // El TAD nos da un objeto iterador
                                         // (un cursor) ya inicializado.

     while (it.hasNext()) {              // Bucle mientras el cursor
                                         // referencie a donde hay un elemento.

        System.out.println(it.next());   // Se hace algo con el elemento
                                         // y queda el cursor avanzado.

   }
```

Otra forma de escribir un bucle con iteradores usando un **`for`**:

```
   for (Iterator<E> it = list.iterator(); it.hasNext();  ) {
     System.out.println(it.next());
   }
```

No hay incremento en el bucle porque el método 'next' realiza el incremento como efecto secundario.

No deben usarse métodos modificadores (inserción o borrado) del interfaz del TAD durante la iteración (por ejemplo, añadir elementos al mismo tiempo que iteramos) pues podría dejarse el iterador en estado inconsistente.

Interfaces `Iterator<T>` e `Iterable<T>`

El interfaz `Iterator<E>` declara los métodos que debe implementar todo objeto iterador. El interfaz está en `java.util.Iterator`.

```
public interface Iterator<E> {
  public E next() ;            /* obligatorio implementarlo */
  public boolean hasNext() ;   /* obligatorio implementarlo */
  public void remove() ;       /* no obligatorio y problemático! */
}
```

El interfaz 'Iterable<E>' declara un método 'iterator' que debe devolver un objeto iterador. Este interfaz es la pieza que nos permitirá asociar iteradores a un TAD. El interfaz está en 'java.lang.Iterable'.

```
public interface Iterable<E> {
  public Iterator<E> iterator() ;
}
```

Cuidado con el significado de los métodos del iterador:

- Los nombres de los métodos `hasNext` y `next` suelen despistar.
- NO debe interpretarse `hasNext` como "¿hay siguiente al cursor?".
- NO debe interpretarse `next` como "dame el siguente elemento al cursor".
- El método `hasNext` indica si el cursor del iterador referencia un elemento (o si referencia un objeto que contiene un elemento, p.ej., un nodo de una lista).

Una interpretación válida es: "*¿referencia el cursor algo donde hay un elemento, el cual todavía no te he pedido con 'next'?*". Usando la metáfora de una frutería, cuando el frutero dice "¿hay siguiente?" si hay alguien entonces levanta la mano el que está primero, cuando atiende al primero (`next`) y vuelve a decir "¿hay siguiente?", levanta la mano el que estaba segundo, etc. En una secuencia de elementos, el primer "siguiente" es el primer elemento. Otra posible interpretación válida de `hasNext` podría ser "¿es el cursor distinto de null?".

El método `next` guarda el elemento al que se llega a través del cursor, avanza el cursor (o lo deja a null si no se puede avanzar) y devuelve el elemento guardado. Una interpretación válida sería "dame el elemento del cursor y deja el cursor avanzado (o a null si no se puede avanzar más)". También hay un método `remove`, que borra el elemento que devolvió el `next` que se ha ejecutado inmediatamente antes. Los métodos `hasNext` y `next` se usan en bucles, donde `hasNext` es la condición centinela que debe comprobarse antes de invocar `next`. El método `remove` puede invocarse si antes (en el tiempo) se ha invocado un `next`.

Desde el cursor no puede alcanzarse ningún elemento del TAD cuando éste es vacío, ni cuando `next` devuelve el último elemento de la iteración. En este último caso `next` deja el cursor a null pues no hay elemento siguiente.

■ **Ejemplo 7.1** Tenemos un TAD con 2 elementos 'A' y 'B', Al crearse el objeto iterador, el cursor referencia el primer elemento porque el TAD no es vacío.

```
{A,B}
 ^
 |
cursor

hasNext() devuelve true.
next()    avanza el cursor a B y devuelve A.


{A,B}
   ^
   |
 cursor

hasNext() devuelve true.
next()    avanza el cursor a null y devuelve B.

{A,B}

 cursor -> null

hasNext() devuelve false.
next()    lanza NoSuchElementException.
```

Al crearse el objeto iterador:

- Si el TAD está vacío:

```
hasNext() devuelve false.
next()    lanza NoSuchElementException.
```

- Si el TAD no está vacío:

```
hasNext() devuelve true.
next()    avanza el cursor al segundo elemento (si hay) o a null (si no
          hay) y devuelve el primer elemento.
```

■

Se pueden usar varios iteradores simultáneamente sobre un mismo TAD:

```
for (Iterator<E> it1 = list.iterator(); it1.hasNext(); ) {
  E e = it1.next();
  for (Iterator<E> it2 = list.iterator(); it2.hasNext(); it2.next()){
    System.out.println(e);
  }
}
```

Método `remove` del iterador

El método `remove` del iterador debe borrar del TAD el elemento que ha sido devuelto por el último `next`. El método `remove` debe lanzar `IllegalStateException` cuando se invoca pero no se invocó ningún `next` antes en el tiempo (no necesariamente en la línea anterior).

```
if (!it.hasNext())
  it.remove();      // Incorrecto, no hay elementos.

if (it.hasNext())
  it.remove();      // Incorrecto, no hay 'next' previo.

while (it.hasNext()) {
  it.next();
  it.remove();      // Correcto
}

it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.remove();        // Correcto si no falló el 'next' anterior

it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.remove()         // Correcto si no falló el 'next' anterior
it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.remove();        // Correcto si no falló el 'next' anterior

it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.remove();        // Correcto si no falló el 'next' anterior

it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
.                                                          ^
.                   // Mucho código aquí sin ningún 'remove'       |
.                                                          |
it.remove();        // Correcto si no falló el 'next' anterior ----/

it.remove();        // Incorrecto, no hay 'next' previo.
```

Si usamos un iterador sobre un TAD, sólo está permitido borrar con el `remove` **del iterador**.

Ejemplos de uso de iteradores

■ **Ejemplo 7.2** Método que devuelve el décimo elemento de una lista (contando desde 1)

```
public static <E> E tenth(PositionList<E> list)
throws NoSuchElementException {
  if (list.size() < 10) throw new NoSuchElementException();
  else {
    Iterator<E> it = list.iterator();
    for (int i = 1; i < 10; i++) {
      it.next();
    }
    return it.next();
  }
}
```

■

■ **Ejemplo 7.3** Método que borra todos los elementos de una lista

```
public static <E> void deleteAll(PositionList<E> list) {
  Iterator<E> it = list.iterator();
  while (it.hasNext()) {
    it.next();
    it.remove();
  }
}
```

Recordad que el método `remove` debe invocarse después de `next` y que `remove` borra el nodo donde está el elemento que ha devuelto el `next` anterior. ■

■ **Ejemplo 7.4** Método que devuelve la suma de los elementos de una lista de enteros. Vamos a ver dos versiones:
(1) Versión que asume que los elementos nunca son null:

```
public static int sumaElems(PositionList<Integer> list) {
  Iterator<E> it = list.iterator();
  int suma = 0;
  while (it.hasNext()) {
    suma += it.next();
  }
  return suma;
}
```

(2) Versión que asume que los elementos pueden ser null. Se necesita una variable donde guardar el elemento:

```
public static int sumaElems(PositionList<Integer> list) {
  Iterator<E> it = list.iterator();
  int suma = 0;
  while (it.hasNext()) {
    Integer e = it.next();
    if (e != null) suma += e;
  }
  return suma;
}
```

■

■ **Ejemplo 7.5** Método que devuelve el máximo elemento de una lista de enteros asumiendo que los no son null.

```
public static int maximum(PositionList<Integer> list)
throws EmptyListException {
  if (!it.hasNext()) throw new EmptyListException();
  int m = it.next();
  while (it.hasNext()) {
    m = Math.max(m, it.next());
  }
  return m;
}
```

■

■ **Ejemplo 7.6** Método que indica si un elemento no null está en una lista
(1) Versión sin iterador:

```
public static <E> boolean member(E e, PositionList<E> list) {
  Position<E> cursor = list.first();
  while (cursor != null && !e.equals(cursor.element())) {
    cursor = list.next(cursor);
  }
```

```
    // cursor == null || cursor != null && e.equals(cursor.element())
    return cursor != null;
  }
```

(2) La siguiente implementación con iterador es **errónea**:

```
  public static <E> boolean member(E e, PositionList<E> list) {
    Iterator<E> it = list.iterator();
    while (it.hasNext() && !e.equals(it.next()))
      ;                    // el "incremento" del cursor lo hace 'next'

    // !it.hasNext() || it.hasNext() && e.equals(it.next())
    return it.hasNext();
  }
```

Para saber por qué es errónea ejecutad el código en papel sobre una lista con un único nodo con elemento igual a 'e'. Recordad que 'it.next()' produce un efecto secundario: también avanza el cursor.

(3) La siguiente implementación correcta utiliza una variable booleana 'found' para guardar el estado de la última comparación.

```
  public static <E> boolean member(E e, PositionList<E> list) {
    Iterator<E> it = list.iterator();
    boolean found = false;
    while ( it.hasNext() && !(found = e.equals(it.next())) )
      ;

    // !it.hasNext() || it.hasNext() && found
    return found;
  }
```

■

■ **Ejemplo 7.7** Método que elimina de una lista la primera aparición del un elemento. Para simplificar asumimos que el elemento no es null.

```
  public static <E> void borra(E e, PositionList<E> list) {
     Iterator<E> it = list.iterator();
     boolean found = false;
     while ( it.hasNext() && !(found = e.equals(it.next())) )
       ;
     if (found) it.remove();
  }
```

■

■ **Ejemplo 7.8** Método que elimina todas las apariciones de ese elemento no null

```
  public static <E> void borra(E e, PositionList<E> list) {
     Iterator<E> it = list.iterator();
     while (it.hasNext()) {
        if (e.equals(it.next())) {
          it.remove();
        }
     }
  }
```

Ahora una solución que contempla el caso de que los elementos sean null

```
  public static <E> void borra(E e, PositionList<E> list) {
    Iterator<E> it = list.iterator();
    while (it.hasNext()) {
      E elem = it.next();
      if (e == null && elem == null || e != null && e.equals(elem)) {
```

```
            it.remove();
          }
        }
      }
```

■

■ **Ejemplo 7.9** Método que indica si una lista es sublista de otra

```
public static <E> boolean sublist(PositionList<E> l1, PositionList<E> l2) {
  if (l1 == null || l2 == null) return false;
  if (l1 == l2) return true;
  Iterator<E> it1 = l1.iterator();
  boolean res = false;
  while ( it1.hasNext() && (res = this.member(it1.next(),l2)) )
    ;
  return res;
}
```

■

■ **Ejemplo 7.10** Método que indica si dos listas son iguales. Recordad que dos listas son iguales si contienen los mismos elementos de izquierda a derecha.

(1) Versión con varias variables auxiliares:

```
public static <E> boolean iguales (PositionList<E> list1,
                                   PositionList<E> list2) {
  if (list1 == list2) return true;
  // El caso anterior incluye list1 == null && list2 == null
  if (list1 == null || list2 == null || list1.size() != list2.size())
    return false;

  Iterator<E> it1 = list1.iterator();
  Iterator<E> it2 = list2.iterator();
  E e1, e2;
  boolean iguales = true;

  while (it1.hasNext() && iguales) {
    e1 = it1.next();
    e2 = it2.next();
    iguales = e1 == null && e2 == null || e1 != null && e1.equals(e2);
  }

  return iguales;
}
```

(2) Creamos un método auxiliar para comparar elementos:

```
private static <E> boolean igualesElem(E e1, E e2) {
  return e1 == null && e2 == null || e1 != null && e1.equals(e2);
}

public static <E> boolean iguales (PositionList<E> list1,
                                   PositionList<E> list2) {
  .
  .
  .
  Iterator<E> it1 = list1.iterator();
  Iterator<E> it2 = list2.iterator();
  E e1, e2;
  boolean iguales = true;
```

```
    while (it1.hasNext() && iguales) {
      e1      = it1.next();
      e2      = it2.next();
      iguales = igualesElem(e1,e2);
    }

    return iguales;
  }
```

(3) Eliminamos las variables elemento y subimos la comparación al bucle:

```
  public static <E> boolean iguales (PositionList<E> list1,
                                     PositionList<E> list2) {
    .
    .
    .
    Iterator<E> it1 = list1.iterator();
    Iterator<E> it2 = list2.iterator();
    boolean iguales = true;

    while (it1.hasNext() && (iguales = igualesElem(it1.next(),it2.next())))
      ;
    return iguales;
  }
```

No podemos eliminar la variable booleana, pues ésta guarda el estado de la última comparación. Necesitamos dicho estado porque `next` avanza el cursor. El siguiente código NO funciona:

```
  while (it1.hasNext() && igualesElem(it1.next(),it2.next()))
    ;
  // !it1.hasNext() || it1.hasNext && !igualesElem(it1.next(),it2.next())
  return !it1.hasNext();
```

Este tampoco:

```
  while (it1.hasNext() && igualesElem(it1.next(),it2.next()))
    ;
  // !it1.hasNext() || it1.hasNext && !igualesElem(it1.next(),it2.next())
  return igualesElem(it1.next(),it2.next());
```

Probadlo con dos listas que contienen un único elemento distinto el uno del otro.

■

¿Cuándo y cómo usar iteradores?

Los iteradores se usan para iterar sobre TADs que son colecciones de elementos. No todos los TADs serán colecciones de elementos y por tanto no todos serán iterables por iteradores. También hay TADs que son colecciones de elementos pero dada su naturaleza no son iterables (p.ej., conjuntos no acotados que no tienen "getters" para obtener elementos). Y si son iterables es con restricciones, como que el iterador puede ser no-determinista: recorre los elementos de forma arbitraria según cómo estén disponibles en el objeto que implementa del TAD. Pero entonces la computación que se realice con los elementos iterados debe dar el mismo resultado para cualquier posible permutación de todos los elementos. Por ejemplo, la suma de elementos de un conjunto de enteros daría el mismo resultado para cualquier permutación. Sin embargo mostrar elementos por pantalla daría resultados diferentes.

Para usar iteradores el problema debe requerir únicamente el acceso a elementos (método `next`) y/o borrar el último elemento devuelto por el iterador (método `remove`). No se pueden usar iteradores si hay que borrar un elemento que no sea el último devuelto por `next` (que es el elemento que borra el `remove` del iterador), o si es necesario acceder, por ejemplo, a los nodos que se recorren, puesto que `next` sólo devuelve el elemento y no tenemos acceso al nodo.

Ejercicio 7.1 Para ver cuando "no" usar iteradores, intentad programar con un iterador el 'addBeforeElement' visto en el Ejemplo 6.2, es decir, un método que inserta un elemento "a" justo antes que el elemento "e":

```
<E> void addBeforeElement(PositionList<E> list, E e, E a) {
```

■

El interfaz `Iterator<E>` sólo permite avanzar y borrar. Hay más interfaces en la JCF que extienden este interfaz con métodos para retroceder, reset, etc. En la JCF los interfaces de lista extienden `Collection<E>` que a su vez extiende `Iterable<E>`. También hay mecanismo "fail-fast" para múltiple iteradores cuando se modifica la colección durante la iteración [GTG14].

Iteración con bucle "for-each"

El bucle *for-each* es una abstracción de código que permite recorrer cualquier estructura de datos iterable de forma homogénea.

Vamos a reescribir 'show' una vez más, pero en vez de:

```
public static <E> void show(PositionList<E> list) {
  Iterator<E> it = list.iterator();
  while (it.hasNext())
    System.out.println(it.next());
}
```

ahora usamos un *for-each*:

```
public static <E> void show(PositionList<E> list) {
  for (E e : list)
    System.out.println(e);
}
```

El *for-each* tiene una notación parecida a la cuantificación universal ($\forall$) de la lógica: "para todo elemento 'e' de tipo 'E' en el *iterable* 'list' ejecuta 'System.out.println(e)"'.

La sintaxis del *for-each* es la siguiente:

```
for (type elem : expr) { stmts }
```

Donde:

- La variable `elem` tiene tipo 'type' y no aparece en la expresión 'expr'.
- La expresión `expr` tiene tipo `Iterable<T>` o tipo "array de T", con T un subtipo de `type`. Es decir, se puede usar for-each con arrays o con objetos iterables (objetos de clases que implementan el interfaz `Iterable<E>`.)
- Dentro de `stmt` no se tiene acceso al iterador (a una variable que referencie el objeto iterador), sólo al elemento `elem`. No se pueden invocar `next`, `hasNext`, ni `remove`. Los dos primeros son invocados por el bucle de forma automática.

El *for-each* se recorre el TAD iterable **por completo** "for-each = para todo elemento", etc. No debe usarse *for-each* para recorridos que pueden ser parciales. Por ejemplo, no debe usarse *for-each* para indicar si un elemento está en un TAD pues el recorrido debe terminar al encontrarse el elemento.

El compilador traduce el *for-each* a un "for.ºrdinario con iteradores:

```
for (Iterator<E> it = list.iterator(); it.hasNext(); ) {
  E e = it.next();
  System.out.println(e);
}
```

La variable `it` local al bucle la genera el propio compilador.

Ejemplo 7.11 Suma de elementos de una lista de enteros, los elementos pueden ser null:

```
public static int sumaElems(PositionList<Integer> list) {
  int suma = 0;
  for (Integer e : list)
    if (e != null)
      suma += e;
  return suma;
}
```

■

Por ejemploe, dos de los constructores de la clase `NodePositionList` están escritos con *for-each*.

Ejemplo 7.12 Reescribimos el `toString` de la clase `NodePositionList` usando *for-each*:

```
public String toString() {
  String s = "[";
```

```
      for (E e : this)) {
        if (e == null)
          s += "null";
        else
          s += e.toString();
        if (cursor != last())
          s += ", ";
      }
      s += "]";
      return s;
    }
```

■

De manera informal podríamos verlo de la siguiente forma:

1. *Cuando* `expr` *es de tipo* `Iterable<T>` el bucle se lee como "ejecuta 'stmt' *para todo* elemento `elem` del iterable `expr`". En este caso el bucle *for-each* es una abreviatura de:

   ```
   for (Iterator<type> it = expr.iterator(); it.hasNext(); ) {
     type elem = it.next();
     stmts
   }
   ```

   donde 'it' es una variable oculta nueva que no aparece en 'stmts'. Nótese que 'elem' se declara después de usarse 'expr' y por eso 'elem' no puede ocurrir en 'expr'. Si no hay elementos para iterar ('hasNext' devuelve falso) no se ejecuta el código del *for-each*. La variable 'elem' puede declararse localmente dentro del bloque del bucle 'for'. También se puede expresar el *for-each* con un bucle 'while':

   ```
   {
     Iterator<type> it = expr.iterator();
     type elem;
     while (it.hasNext()) {
       elem = it.next();
       stmts
     }
   }
   ```

   java

2. *Cuando* `expr` *es de tipo array de T* entonces el bucle *for-each* es una abreviatura de:

   ```
   for (int j = 0; j < expr.length; j++) {
     type elem = v[j];
     stmts
   }
   ```

   Donde `j` es una nueva variable que no aparece en 'stmts'.

■ **Ejemplo 7.13** Sumar los elementos de un vector:

```
public int sumaArray(int [] v) {
  int suma = 0;
  for (int e : v)
    suma += e;
  return suma;
}
```

■

■ **Ejemplo 7.14** Ejemplo de MAL uso: mostrar los 'n' primeros elementos de un array de caracteres 'v'. Usando *for-each* se recorre innecesariamente todo el array:

```
for (char c : v) {
  if (n > 0) {
    System.out.println(c);
    n--;
  }
}
```

Usar 'break' es un error de estilo y va contra la razón de ser de *for-each* que sirve para recorrer todo el vector:

```
for (char c : v) {
  if (n == 0) break;          // Suspenso
  if (n > 0) {
    System.out.println(c);
    n--;
  }
}
```

La solución idónea es no usar el *for-each*. ▪

Cómo implementar iteradores

Hay dos alternativas para implementar iteradores sobre TADs:

1. El objeto iterador itera ("mueve el cursor") **usando los métodos del interfaz del TAD** para obtener el siguiente elemento. De esta forma el iterador puede usarse para iterar sobre objetos de cualquier clase que implemente el interfaz. En otras palabras, dado el interfaz 'I', el iterador puede iterar sobre cualquier clase que implemente 'I' si usa únicamente métodos de 'I' para "mover" el cursor.
2. El objeto iterador itera ("mueve el cursor") **accediendo a los atributos de la clase que implementa el TAD**. Estos iteradores únicamente pueden usarse para iterar sobre objetos de las clases concretas. En otras palabras, dada la clase 'C' que implementa el interfaz 'I', el iterador definido para objetos de 'C' sólo puede usarse sobre objetos de 'C' y no sobre objetos de otras clases que implementan 'I'.

La primera alternativa es más deseable desde el punto de vista de abstracción y reusabilidad, pero puede no ser posible realizarla porque no hay suficientes métodos observadores en el interfaz, o porque la iteración usando métodos del interfaz es ineficiente.

Iteradores sobre interfaces A continuación damos una *receta* para implementar iteradores sobre TADs de forma que los iteradores invocan los métodos de los interfaces de los TADs.

1. El interfaz del TAD debe extender 'Iterable<E>' para indicar que las clases que implementen el interfaz tendrán que implementar el método 'iterator', el cual deberá devolver un objeto iterador (un objeto que implementa el interfaz 'Iterator<E>') para dicho TAD.

```
import java.util.Iterator;
public interface TAD<E> extends Iterable<E> {

  ... /* métodos del TAD */

  public Iterator<E> iterator() ;
}
```

   Recordad que el interfaz `PositionList<E>` extiende `Iterable<E>`.

2. Se implementa una clase iterador cuyos métodos usan los métodos del interfaz del TAD para "mover el cursor":

```
import java.util.Iterator;
public class TADIterator<E> implements Iterator<E> {
  TAD<E> tad;  /* el TAD sobre el que itera es un atributo */
  CursorTAD<E> cursor;

  public TADIterator(TAD<E> t) {
    tad = t;
    ... /* código aquí que inicializa el valor del cursor */
  }
  public boolean hasNext()      { /* código aquí */ }
  public E next()               { /* código aquí */ }
```

```
    public void remove()          { /* código aquí */ }
}
```

El constructor del objeto iterador toma como parámetro el objeto TAD sobre el que debe iterar. El tipo de 'cursor' depende del TAD concreto. Puede ser directamente un elemento:

```
E cursor;
```

También puede ser una referencia a un objeto que contiene un elemento (como en el caso de `PositionListIterator.java`, donde tenemos:

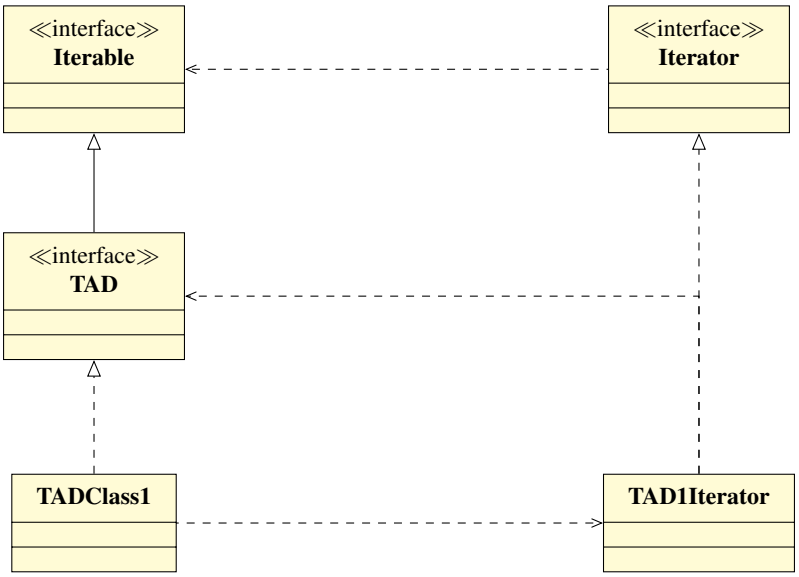
```
Position<E> cursor;
```

También puede ser un valor de algún tipo, a partir del cual se puede localizar el elemento en el TAD, etc.

3. Implementar el método 'iterator' en todas las clases que implementan el interfaz 'TAD<E>'. Dicho método devuelve un objeto iterador.

```
import java.util.Iterator;
public class TADClass1<E> implements TAD<E> {
  ...
  public Iterator<E> iterator() { return new TADIterator<E>(this); }
}


import java.util.Iterator;
public class TADClass2<E> implements TAD<E> {
  ...
  public Iterator<E> iterator() { return new TADIterator<E>(this); }
}
```

El diagrama de clases sería:



Los interfaces y clases del diagrama deben ser declarados dentro de un paquete.
El fichero `PositionListIterator.java` contiene la implementación del iterador para el interfaz `PositionList<E>`:

- Esta clase implementa `Iterator<E>` para `PositionList<E>`.
- El cursor no referencia un elemento de tipo 'E' sino una posición `Position<E>` porque los métodos del interfaz `PositionList<E>` tales como `next` y `prev` (que se usan para "mover el cursor") toman como parámetros posiciones, no elementos.
- `NodePositionList<E>` implementa el método `iterator` y usa el iterador en el método `equals`.

Iteradores sobre clases: A continuación se describe otra posibilidad para implementar iteradores utilizando para ello la clase sobre la que se quiere iterar, en lugar de utilizar el interfaz. **El material de esta sección es opcional y no será evaluado en el examen de teoría.** Incluimos el material para los alumnos que desean profundizar en la materia.

1. Se puede iterar sobre un TAD declarando el método 'iterator()' como método propio de `TAD<E>` sin extender de `Iterable<E>`:

```
import java.util.Iterator;
public interface TAD<E> {
  ...
  public Iterator<E> iterator();
}
```

   Las clases que implementen el interfaz no tendrán que implementar 'iterator'. Sin embargo esto no es recomendable porque `TAD<E>` no es `Iterable<E>`. No podemos usar `TAD<E>` en contextos donde se espera un iterable, por ejemplo, no podemos hacer un *for-each* con el interfaz `TAD<E>`, aunque sí podríamos hacerlo con la implementación concreta del interfaz.
2. Se puede asociar un iterador a una clase en vez de a un interfaz:
   - Código del interfaz 'TAD<E>':

```
public interface TAD<E> /* no extiende Iterable<E> */ { ... }
```

   - Código de **una** clase:

```
public class TADClass1<E> implements TAD<E>, Iterable<E> {
    ...
    public Iterator<E> iterator() {
      return new TADClass1Iterator<E>(this);
    }
}
```

   - Código del iterador asociado a la clase `TADClass1<E>`:

```
public class TADClass1Iterator<E> implements Iterator<E> {
  TADClass1<E> tadClass1;
  CursorTAD<E> cursor;

  public TADClass1Iterator(TADClass1<E> t) {
    tadClass1 = t;
    ... /* código aquí que inicializa el valor del cursor */
  }

  public boolean hasNext()     { /* código aquí */ }
  public E next()              { /* código aquí */ }
  public void remove()         { /* código aquí */ }
}
```

   - El constructor del iterador toma como parámetro un objeto de la clase `TADClass1<E>`, no un objeto de tipo `TAD<E>`. Los métodos `hasNext` y `next` pueden implementarse o bien usando los métodos del interfaz `TAD<E>` (porque `TADClass1<E>` implementa los métodos de `TAD<E>`) o bien accediendo directamente a los atributos de `TADClass1<E>` si son públicos.
3. Finalmente, una variación de lo anterior es definir `TADClass1Iterator<E>` como una clase anidada no estática ("non-static") dentro de `TADClass1<E>`, pudiendo la primera acceder a los atributos privados de la segunda. Consultar en la documentación de Java los conceptos de clases *nested*, *inner*, *local*, y *anonymous*, así como la siguiente URL:
   `http://download.oracle.com/javase/tutorial/java/javaOO/nested.html`

IMPORTANTE: En los dos últimos casos (2 y 3) el iterador `TADClass1Iterator<E>` sólo puede usarse con objetos de clase `TADClass1<E>`. No puede usarse con objetos de otras clases que implementen `TAD<E>`. Esas clases deben tener su propio iterador.

Iteradores con alguna propiedad

Los TAD's vistos hasta ahora permiten recorrer los elementos contenidos en una estructura de datos siguiendo el orden "oficial" de la estructura, es decir, en una lista el iterador que se obtiene llamando al método `iterator()` del TAD recorre todos los elementos de la lista manteniendo el orden en el que se encuentran en la lista.

Pero también nos pueden interesar diferentes `vistas` del mismo TAD, por ejemplo, me puede interesar recorrer los elementos de la lista en sentido inverso o alterno, o me puede interesar recorrer únicamente los elementos que cumplan una cierta propiedad, p.e. lo que no sean null, o los positivos, ...

Esto se puede hacer creando nuevas clases que implementen el interfaz `Iterator<E>` proporcionándole al nuevo iterador las propiedades buscadadas. Por ejemplo el siguiente iterador recorrerá la `PositionList list` en orden inverso:

```
public class PositionListReverseIterator<E> implements Iterator<E> {

  private PositionList<E> list;
  private Position<E> cursor;

  public PositionListReverseIterator(PositionList<E> list) {
    if (list == null ) {
      throw new IllegalArgumentException ("La lista es null");
    }
    this.list   = list;
    this.cursor = list.last();
  }

  public boolean hasNext() {
    return cursor != null;
  }

  public E next() throws NoSuchElementException {
    if (cursor == null) throw new NoSuchElementException();

    E e = cursor.element();
    cursor = list.prev(cursor);
    return e;
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

La idea fundamental detrás de esto, sería, dada la misma estructura de datos, dependiendo del iterado utilizado, disponer de diferentes vistas o formas de recorrer la estructura. Hay que tener en cuenta que estos iteradores no se crean mediante el método iterator() en el TAD, sino que deben ser creados a mano teniendo una referencia a la estructura de datos:

```
PositionList<String> l = new NodePositionList<String>(...);
Iterator<String> it = new PositionListReverseIterator<String>(l);

// Devolverá los elementos en orden inverso
while(it.hasNext()) {
  System.out.println(it.next());
}
```

8. Recursividad

Un método es recursivo cuando se invoca a sí mismo en su propia definición. Un ejemplo típico y sencillo es el método que calcula el 'factorial' de un entero positivo. El factorial de un entero positivo n >0 se define como el valor de la expresión $n*(n-1)*...*1$. Para $n \leq 0$ el factorial se define como 1. Mostramos dos posibles soluciones iterativas. Ambas usan una variable acumulador, en la que se acumulan las multiplicaciones. La primera solución itera desde 'n' hasta '2'. La segunda desde '2' hasta 'n'.

```
public static int factorial(int n) {
  int r = 1;
  while (n > 1) {
    r = n * r;
    n--;
  }
  return r;
}

public static int factorial(int n) {
  int r = 1;
  for (int i = 2; i <= n; i++) {
    r = i * r;
  }
  return r;
}
```

Dejemos a un lado el problema de que en Java el tipo 'int' tiene un rango de valores limitado (probad qué pasa por ejemplo con `factorial(100)`). Dejemos también a un lado cómo solucionarlo (¿qué otros tipos primitivos o clases se podrían usar?).

Una definición recursiva es más intuitiva y elegante.

Observad que la expresión $(n-1)*(n-2)*...*1$ calcula `factorial(n-1)`. Por tanto se puede definir `factorial(n)` en función de `factorial(n-1)`:

```
n * (n-1) * (n-2) * ... * 1
    \_____________________/
          factorial(n-1)
\_____________________________/
       factorial(n)

factorial(n) = n * factorial(n-1)
```

En código:

```
public static int factorial(int n) {
  if (n <= 1) return 1;
  else        return n * factorial(n-1);
```

```
}
```

que queda más escueto usando una expresión condicional:

```
public static int factorial(int n) {
  return n <= 1 ? 1 : n * factorial(n-1);
}
```

Cualquier programa iterativo tiene uno equivalente recursivo y viceversa. Observad que en el ejemplo del factorial, la condición del 'if' en la versión recursiva es la negada de la condición del 'while' en la versión iterativa, y que llamarse recursivamente es una forma de iterar.

Este código tendría la siguiente ejecución para `factorial(4)`:

```
factorial(4)
4 *   factorial(3)
4 *   3 *  factorial(2)
4 *   3 *  2 *   factorial(1)
4 *   3 *  2 *   1
4 *   3 *  2
4 *   6
24
```

En el diagrama anterior, observamos que las llamadas recursivas se van *apilando*. Una vez se ha llegado al caso base, se calcula la multiplicación al retornar de la llamada recursiva, así hasta llegar a la primera invocación.

La recursión se suele implementar utilizando la pila de la memoria (donde se almacenan los registros de activación de los métodos) pero hay implementaciones más eficientes de la recursión como la transformación a estilo en paso de continuaciones (*continuation-passing style* o CPS) que deja los métodos recursivos con recursión de cola (*tail recursion*) que a su vez puede optimizarse en un bucle.

Mostramos cómo podría optimizar un compilador el método 'factorial'. Primero recordamos la versión original:

```
public static int factorial(int n) {
  if (n <= 1) return 1;
  else        return n * factorial(n-1);
}
```

Se invoca `factorial` recursivamente sobre 'n-1'. Cuando éste retorna el valor, se multiplica por 'n'. Sin embargo, se podría invocar `factorial` recursivamente pero pasándole además un parámetro extra que indique qué tiene que hacer cuando termine de computar el factorial de n-1. Es decir, se podría invocar 'factorial' diciéndole: (1) Computa el valor del factorial de n-1; (2) después multiplica ese valor por n y entonces retorna el resultado. De esta forma `factorial` tendría recursión de cola: la llamada recursiva sería el último mandato del método y no habría que multiplicar después.

Para que la recursión tenga sentido, la definición debe tener uno o más **casos base** (no recursivos) y debe haber un orden bien fundado en los valores de los argumentos, de forma que las (una o más) llamadas recursivas se realizan sobre valores anteriores en el orden, llegándose eventualmente a alguno de los casos base. En caso contrario el método recursivo no termina (es equivalente a un bucle infinito). En el ejemplo del factorial el orden bien fundado es el de los enteros positivos $(n > n-1 > ... > 1)$ y se llega al caso base.

Por ejmplo, sas siguientes variantes son incorrectas:

```
public static int factorial(int n) {
   return n * factorial(n-1);               // no hay caso base.
}

public static int factorial(int n) {
  if (n == 1) return 1;
  else        return n * factorial(n-2);    // no alcanza el caso base
}
```

El primer ejemplo es equivalente a un bucle infinito. Si el compilador implementa la recursión con una pila, puesto que el computador no tiene memoria infinita, eventualmente se producirá un error de desbordamiento de pila o *stack overflow*. Con la optimización de recursión de cola no habrá desbordamiento y se quedará colgado como en un bucle infinito.

Mostramos el código equivalente iterativo de los ejemplos anteriores:

```
public static int factorial(int n) {
 int r = 1;
 while (n != 1) {
   r = n * r;
   n -= 2;
 }
 return r;
}

public static int factorial(int n) {
 int r = 1;
 while (true) {
   r = n * r;
   n--;
 }
 return r;
}
```

■ **Ejemplo 8.1** El cálculo del máximo común divisor (también su versión iterativa):

```
public static int mcd(int n, int m) {
  if (m == 0) return n;
  else        return mcd(m, n % m);
}

public static int mcd(int n, int m) {
  while (m != 0) {
    int tmp = m;
    m       = n % m;
    n       = tmp;
  }
return n;
}
```

■

■ **Ejemplo 8.2** Ejemplo con más de un caso base y llamada recursiva (aquí hemos obviado los 'else'):

```
public static int fib(int n) {
  if (n < 0)  return 0;
  if (n <= 1) return n;   /* dos casos base */
  return fib(n-1) + fib(n-2);
}
```

■

■ **Ejemplo 8.3** Búsqueda binaria de un elemento en un vector ordenado. Para realizar la recursión necesitamos un método auxiliar que lleva cuenta del rango en el que se está buscando. La búsqueda se llama "binaria" porque en cada paso hay sólo dos (binario) opciones: o bien se busca en el rango desde `start` hasta el elemento `m-1` anterior al central, o bien se busca en el rango desde el elemento `m+1` siguiente al central hasta `end`. Al comienzo, `start = 0` y `end = arr.length-1`.

Versión iterativa:

```
public static boolean member(int elem, int [] arr) {
  if (arr == null   || arr.length == 0 ||
      elem < arr[0] || elem > arr[arr.length - 1])
    return false;
  int start = 0;
  int end   = arr.length - 1;
  int m;
```

```
    while (start <= end && arr[(m = (start + end) / 2)] != elem) {
      if (elem < arr[m])
        end   = m-1;
      else
        start = m+1;
    }
    // start > end || start <= end && arr[m] == elem
    return start <= end;
  }
```

Ahora la versión recursiva:

```
public static boolean member(int elem, int [] arr) {
  if (arr == null   || arr.length == 0 ||
      elem < arr[0] || elem > arr[arr.length - 1])  /* no es caso base */
    return false;
  else
    return memberRec(elem, arr, 0, arr.length-1);
}

private static boolean memberRec(int elem, int [] arr,
                                 int start, int end) {

  if (start > end)    return false;   /* caso base */
  int m = (start + end) / 2;
  if (elem == arr[m]) return true;    /* caso base */
  if (elem < arr[m])  return memberRec(elem, arr, start, m-1);
                      return memberRec(elem, arr, m+1, end);
}
```

El método principal no es recursivo, únicamente trata los casos frontera e invoca el método auxiliar (y por ello privado) recursivo que toma todos los parámetros necesarios para resolver el problema. Podemos simplificar el número de casos base en uno:

```
  private static boolean memberRec(int elem, int [] arr,
                                   int start, int end) {

    if (start >= end) return elem == arr[start];
    else {
      int m = (start + end) / 2;
      if (elem <= arr[m])
        return memberRec(elem, arr, start, m);
      else
        return memberRec(elem, arr, m+1, end);
    }
  }
```

Observad que al unificar los dos casos base, en la primera llamada recursiva `m` no puede decrementarse, de lo contrario el código sería incorrecto.

■

Recursión sobre listas de posiciones

Las listas se pueden definir de forma recursiva: una lista de posiciones puede definirse de forma recursiva como una secuencia de nodos que es o bien vacía (caso base) o bien está formada por un nodo seguido de una secuencia de nodos. Los métodos recursivos sobre listas explotarán esta definición.

■ **Ejemplo 8.4** El método `show` que muestra por pantalla los elementos de una lista

Versión iterativa:

```
 public static <E> void show(PositionList<E> list) {
    if (list != null) {
      Position<E> cursor = list.first();
      while (cursor != null) {
```

```
        System.out.println(cursor.element());
        cursor = list.next(cursor);
      }
    }
  }
```

Versión recursiva. El método principal no es recursivo, únicamente trata los casos frontera e invoca el método auxiliar (y por ello privado) recursivo. Los casos base se dan cuando el parámetro cursor es 'null' (caso en que la lista es vacía y caso en que se ha llegado al final de la lista).

```
public static <E> void show(PositionList<E> list) {
  if (list != null) showRec(list, list.first());
}

private static <E> void showRec(PositionList<E> list, Position<E> cursor) {
  if (cursor == null) return;
  else {
    System.out.println(cursor.element());
    showRec(list, list.next(cursor));
  }
}
```

Como el método recursivo no devuelve ningún valor, es habitual no hacer uso del 'return' explícito y dejar únicamente la rama que hace algo distinto de simplemente retornar.

```
private static <E> void showRec(PositionList<E> list, Position<E> cursor) {
  if (cursor != null) {
    System.out.println(cursor.element());
    showRec(list, list.next(cursor));
  }
}
```

Se puede subir el caso base de "lista vacía"del método recursivo auxiliar al método principal para ahorrarse una llamada recursiva:

```
public static <E> void show(PositionList<E> list) {
  if (list != null && !list.isEmpty())
    showRec(list, list.first());
}
```

■

■ **Ejemplo 8.5** El método `toString` de la clase `NodePositionList<E>`

Versión iterativa:

```
public String toString() {
  String s = "[";
  for (Position<E> cursor = first(); cursor != null; cursor = next(cursor)) {
    if (cursor.element() == null)
      s += "null";
    else
      s += cursor.element().toString();
    if (cursor != last()) s += ", ";
  }
  s += "]";
  return s;
}
```

Versión recursiva con método auxiliar privado:

```
public String toString() {
   return "[" + toStringRec(first()) + "]";
}

private String toStringRec(Position<E> cursor) {
```

```
    if (cursor == null) return "";
    String s;
    if (cursor.element() == null)
      s = "null";
    else
      s = cursor.element().toString();
    if (cursor != last())
      s += ", ";
    return s + toStringRec(next(cursor));
  }
```

Versión anterior con expresiones condicionales:

```
  public String toString() {
    return "[" + toStringRec(this.first()) + "]";
  }

  private String toStringRec(Position<E> cursor) {
    if (cursor == null) return "";
    return (cursor.element() == null ? "null" ? cursor.element().toString())
         + (cursor != last() ? ", " : "")
         + toStringRec(this.next(cursor));
  }
```

Versión recursiva con parámetro de acumulación:

```
  public String toString() {
    return toStringRec("[", first());
  }

  private String toStringRec(String acum, Position<E> cursor) {
    if (cursor == null) return acum + "]";
    if (cursor.element() == null)
      acum += "null";
    else
      acum += cursor.element().toString();
    if (cursor != last())
      acum += ", ";
    return toStringRec(acum, list.next(cursor));
  }
```

El código anterior puede simplificarse separando los casos de lista vacía y final de lista:

```
  public String toString() {
    if (isEmpty()) return "[]";
    else return toStringRec("[", first());
  }

  private String toStringRec(String acum, Position<E> cursor) {
    if (cursor.element() == null)
      acum += "null";
    else
      acum += cursor.element().toString();

    if (cursor == last())
      return acum + "]";
    else
      return toStringRec(acum + ",", list.next(cursor));
  }
```

■

■ **Ejemplo 8.6** Método que suma todos los elementos de una lista de enteros considerando que los elementos

pueden ser null:

```
public static int sumaElems(PositionList<Integer> list) {
  if (list == null) return 0;
  else return sumaRec(list, list.first());
}

private static int sumaRec( PositionList<Integer> list
                          , Position<Integer> cursor) {
  if (cursor == null)
    return 0;
  else if (cursor.element() == null)
    return sumaRec(list,list.next(cursor));
  else
    return cursor.element() + sumaRec(list, list.next(cursor));
}
```

Evitamos una llamada recursiva subiendo el caso base al método principal:

```
public static int sumaElems(PositionList<Integer> list) {
  if (list == null || list.isEmpty()) return 0;
  else return sumaRec(list, list.first());
}
```

También podemos simplificar el método recursivo:

```
private static int sumaRec( PositionList<Integer> list
                          , Position<Integer> cursor) {
  if (cursor == null) return 0;
  return (cursor.element() == null ? 0 : cursor.element())
         + sumaRec(list, list.next(cursor));
}
```

Otra versión que utiliza una variable auxiliar donde se acumula la suma (veremos el parecido con la versión con parámetro de acumulación de la siguiente sección):

```
private static int sumaRec( PositionList<Integer> list,
                            Position<Integer> cursor) {
  int r = 0;
  if (cursor != null)
    r = (cursor.element() == null ? 0 : cursor.element())
        + sumaRec(list, list.next(cursor));
  return r;
}
```

■

■ **Ejemplo 8.7** Método que suma con parámetro de acumulación (recursión de cola):

```
public static int sumaElems(PositionList<Integer> list) {
  if (list == null || list.isEmpty()) return 0;
  else return sumaRec(0, list, list.first());
}

private static int sumaRec( int r
                          , PositionList<Integer> list
                          , Position<Integer> cursor) {
  if (cursor == null)
    return r;
  else if (cursor.element() == null)
    return sumaRec(r, list, list.next(cursor));
  else
    return sumaRec(r + cursor.element(), list, list.next(cursor));
}
```

El código del método recursivo auxiliar puede simplificarse:

```
private static int sumaRec( int r
                          , PositionList<Integer> list
                          , Position<Integer> cursor) {
  if (cursor != null)
    r = sumaRec( r + (cursor.element() == null ? 0 : cursor.element())
               , list
               , list.next(cursor))
  return r;
}
```

■

■ **Ejemplo 8.8** Borrar una aparición de un elemento distinto de null

```
public static void borra(PositionList<E> list, E elem) {
  if (list != null) borraRec(list, elem, list.first());
}

private static void borraRec( PositionList<E> list,
                            , E elem
                            , Position<E> cursor) {
  if (cursor != null) {
    if (cursor.element() != null && cursor.element().equals(elem))
      list.remove(cursor);
    else
      borraRec(list, elem, list.next(cursor));
  }
}
```

■

■ **Ejemplo 8.9** Borrar todas las apariciones de un elemento distinto de null

```
public static void borra(PositionList<E> list, E elem) {
  if (list != null) borraRec(list, elem, list.first());
}

private static void borraRec( PositionList<E> list
                            , E elem
                            , Position<E> cursor) {
  if (cursor != null) {
    Position<E> aux = cursor;
    cursor = list.next(cursor);
    if (aux.element() != null && aux.element().equals(elem))
      list.remove(aux);
    borraRec(list, elem, cursor);
  }
}
```

Código alternativo del método recursivo:

```
private static void borraRec( PositionList<E> list
                            , E elem
                            , Position<E> cursor) {
  if (cursor != null) {
    Position<E> nxt = list.next(cursor);
    if (cursor.element() != null && cursor.element().equals(elem))
      list.remove(cursor);
    borraRec(list, elem, nxt);
  }
```

```
}
```

■

■ **Ejemplo 8.10** Inserción ordenada en una lista ordenada de enteros. Para simplificar asumimos que los elementos no son null:

```
public static void insertOrden(PositionList<Integer> list, E elem) {
  if (list != null) insertRec(list, elem, list.first());
}

private static void insertRec( PositionList<Integer> list
                             , E elem
                             , Position<E> cursor) {
  if (cursor == null)
    list.addLast(elem);
  else if (elem <= cursor.element())
    list.addBefore(cursor, elem);
  else
    insertRec(list, elem, list.next(cursor));
}
```

■

9. Funciones finitas y tablas de dispersión

9.1 Maps

Las funciones finitas o maps se usan para gestionar información (valores) a través de claves. La información puede ser compleja y no tener propiedades que permitan gestionarla rápidamente. Se usan claves que sí tendrán esas propiedades. La inserción, búsqueda y borrado se hará a través de las claves. *Map* en Inglés significa “correspondencia” o “función”. Nótese las diferencias con las as colas con prioridad, donde las claves se usaban como prioridades y el objetivo era obtener el elemento de mayor prioridad rápidamente. Un ejemplo, una ase de datos de alumnos. Se usa el NIF/NIE como clave para realizar búsquedas y borrados.

Conceptualmente, un *map* o “función finita” es un conjunto finito de entradas (pares clave-valor) que poseen una propiedad que podría expresarse informalmente en palabras de varias maneras equivalentes:

- Todas las entradas tienen claves distintas.
- No hay entradas con la misma clave.
- La clave asociada a un valor dado es única (esto a mí me suena mal).
- Si algunas entradas tienen claves iguales entonces los valores asociados también son iguales.

En matemáticas, el conjunto de entradas se denomina *grafo* o *extensión* de la función finita.

Como implementación inmediata: lista de entradas clave-valor sin claves repetidas o también llamadas .ªssociation lists". Las implementaciones más eficientes explotarán las propiedades de las claves: orden, codificación ("hash"), indexación, etc; podemos decir que la clave asociada a un objeto determina su “localización” en la estructura de datos.

Para poder implementar un *map* es necesario poder determinar que dos claves de tipo K son iguales: ¿Cómo se realiza esto en en Java?

1. Si K es una clase envoltura ("wrapper class") de un tipo primitivo (ejemplos: clases ’Integer’, ’Character’, ’Float’, etc.) entonces gracias al .ªutoboxing"se pueden usar los operadores de comparación ’==’, ’!=’, etc, siempre que estén definidos para el tipo primitivo que envuelven.
2. Si K es una clase, hereda de ’Object’ y puede implementar el método ’equals’. Si K es un interfaz, la clase que lo implementa puede implementar ’equals’.
3. Si K es una clase, puede implementar el interfaz ’Comparable<K>’. Si K es un interfaz, puede heredar (extender) el interfaz ’Comparable<K>’. La clase que implemente el interfaz K deberá implementar los metodos de ’Comparable<K>’.
4. Se puede implementar un objeto ’Comparator<K>’.

El Interfaz `Map<K,V>`

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

  public int size();

  public boolean isEmpty();

  public V put(K key, V value) throws InvalidKeyException;
```

```
  public V get(K key) throws InvalidKeyException;

  public V remove(K key) throws InvalidKeyException;

  public Iterator<K> keys();

  public Iterator<Entry<K,V>> entries();
}
```

El detalle del funcionamiento de los métodos se puede encontrar en XXX.

- El método `get(k)` devuelve el valor asociado a una clave "aplicando" la función finita a la clave k.
- El método `put(K)` inserta o reemplaza (si ya existía) el valor asociado a la clave k. En caso de que ya existiera devuelve el valor que hubiera almacenado.
- El método `remove(K)` borrar el valor asociado a una clave.
- Los tres métodos lanzan 'InvalidKeyException' y además usan 'null' para notificar que se ha producido una situación excepcional. El uso de 'null' tiene varias consecuencias:
- El valor 'null' tiene diversos significados que dependen del contexto. Por ejemplo, no podría haber entradas con valores null porque no se distinguirían de las entradas que no existen.
- El constructor de 'InvalidKeyException' toma una cadena de caracteres que indicará por qué la clave no es válida.
- Hay otra solución a la dicotomía entre excepción o centinela ('null') consiste en extender el "destino"de los métodos. No exploraremos esta solución por no estar extendida dentro del mundo Java.
- Los métodos `keySet`, `values` y `entrySet` devuelven respectivamente objetos iterables con las claves, valores y entradas de la función finita.

Ejemplo 9.1 Veamos un ejemplo de uso:

```
Map<Character,Integer> map = new HashTableMap<Character,Integer>();

map.put('a', 5); // Crea entrada <a,5>
map.put('b', 8); // Crea entrada <b,8>
map.put('c', 7); // Crea entrada <d,7>

map.put('a', 10); // Reemplaza <a,5> por <a,10>

System.out.println(map.get('a')); // 10
System.out.println(map.get('b')); // 8
System.out.println(map.get('c')); // 7

map.remove('c');
System.out.println(map.get('c')); // null

System.out.println(map.get('d')); // null
map.put('d', null);
System.out.println(map.get('d')); // null, existe?

// Recorremos Claves
System.out.println("*** Claves");
Iterator<Character> itk = map.keySet();
while(itk.hasNext()) {
    System.out.println(k + " " + map.get(itk.next()));
 }

// Recorremos entradas
System.out.println("*** Entradas");
Iterator<Character> ite = map.entrySet();
while(ite.hasNext()) {
    Entry<Character,Integer> e = it.next();
    System.out.println(e.getKey() + "--" + e.getValue());
 }
```

■

A Simple List-Based Map Implementation

Se usará una `NodePositionList<K,V>` de entradas desordenadas sin claves repetidas. La inserción, búsqueda y borrado tendrán complejidad $O(n)$ en caso peor. La inserción será $O(n)$ porque hay que comprobar si la clave de la entrada a insertar está en la lista. Se actualiza el valor en caso positivo o se inserta la entrada al final de la lista en caso negativo. Una función finita vacía no es representada por 'null'. La función finita vacía será construida por el constructor de la clase que implemente `Map<K,V>`, el cual devolverá un objeto, no null. Mostramos código Java de la clase con algunos métodos (falta por implementar `checkKey` y usar `InvalidKeyException`):

```
class MapList<K,V> implements Map<K,V> {
  public NodePositionList<Entry<K,V>> lista;

  public MapList() { lista = new NodePositionList<Entry<K,V>>(); }

  public int size() { return lista.size(); }

  public boolean isEmpty() { return lista.isEmpty(); }

  public V get(K k) {
    Iterator<Entry<K,V>> it = lista.iterator();
    boolean found = false;
    Entry<K,V> e;

    while (it.hasNext() && !found) {
      e = it.next();
      found = e.getKey().equals(k);
    }
    // En esta línea se cumple "!it.hasNext() || found"
    if (found) return e.getValue();
    else return null;
  }
  ...
  // resto de métodos
}
```

Un código de `get` más conciso:

```
public V get(K k) {
  Entry<K,V> e;
  for ( Iterator<Entry<K,V>> it = lista.iterator() ;
        it.hasNext() && !(e = it.next().element()).getKey().equals(k) ;
      ) ; // No tiene cuerpo
  return e.getKey().equals(k) ? e.getValue() : null ;
}
```

Si usamos los métodos del interfaz `PositionList<E>` nos queda un código más complicado. Por ejemplo:

```
public V get(k) {
  if (lista.isEmpty()) return null;
  else {
    boolean found = false ;
    boolean end   = false ;
    Position<Entry<K,V>> cursor = lista.first();
    do {
      found = cursor.element().getKey().equals(k);
      end   = cursor == lista.last();
      if (!end)
        cursor = lista.next(cursor);
    } while(!found && !end)
    // En esta línea se cumple 'found || end'
    if (found) return cursor.element().getValue();
    else return null
}
```

Con un iterador se abstrae todo lo relativo al uso del cursor, como hemos explicado en el tema de iteradores.

Para `put` y `remove` también hay que recorrer la lista y para ello puede usarse su iterador. El método `put` no puede insertar entradas con claves que ya están en la lista.

Hash tables

Hemos visto hasta ahora una implementación basada en listas desordenadas que tiene complejidad $O(n)$ para todas las operaciones. El objetivo es implementar funciones finitas de forma que la complejidad de la inserción, búsqueda y borrado sea *en el caso medio* $O(1)$. Para ello las claves deberán tener la propiedad de ser "dispersables", es decir, que exista una función, que llamaremos *hash* que se encargará de dispersar los posibles valore de la entrada en un cierto rango. Los valores deben poder ser cualquier objeto.

Para implementar tablas de dispersión necesitaríamos:

1. Un **array** T de tamaño N que llamamos "tablaz que almacenará las entradas.
2. Una **función de codificación** (*hash function*), $hash(k)$ cuyo objetivo es codificar las claves dentro del intervalo $0,..,K-1$.
3. Una **función de compresión/dispersión** f cuyo objetivo es comprimir el código de tamaño K obtenido en el punto anterior a tamaño N y dispersar las claves dentro del intervalo $0,..,N-1$, es decir, dentro del rango del array.

La ubicación en el array de la clave correspondiente se suele definir mediante la composición de ambas funciones $pos(k) = f(hash(k))$ El objetivo de la función $pos(k)$ es almacenar la entrada (k,v) en $T[h(k)]$. Un punto importante para no añadir complejidad es que que la complejidad de computar $pos(k)$ **debe ser** $O(1)$.

La función de codificación (*hash*) que aplica Java para una cierta clave se implementa en el método `hashcode`, que se encuentra en la clase `Object`. Dicho método, por defecto, utiliza devuelve la dirección de memoria del objeto como *hash*, pero, este método se puede sobrescribir, pero hay que hacerlo concierta precauciones (ver un poco más abajo).

Además, la implementación de la tabla hash requiere del uso de una función de compresión/dispersión. Algunas opciones para esta función son:

- La función de compresión/dispersión debe pasar del rango $i \in [0..K]$ devuelto por `hashCode` a un rango $[0..N]$ que es el tamaño del array
- La más sencilla sería el *"division method"*:

$$i \ mod \ N$$

  - N es un número primo
- El método *MAD* (*multiply-add-and-divide*):

$$[(a \cdot i + b) mod \ p] \ mod \ N$$

  - N es primo
  - $p > N$ y p es primo
  - a y b son números aleatorios en $[0..p-1]$ y $a > 0$

Después de aplicar estas funciones, puede ocurrir que dos objetos distintos coincidan en la misma posición del array, es decir, se puede producir una **colisión**. Esto se puede deber a que el método `hashcode` devuelva el mismo código para varios objetos o bien porque, aunque tengas distintos `hashcode`, la función de compresión devuelva la misma posición del array. Existen múltiples opciones para solucionar las colisiones (los detalles de cada una de ellas los podéis encotrar en [GTG14]):

- **Separate Chaining**: Cada uno de los elementos del array son listas y usar `equals` para buscar el elemento
  - Se puede usar un `Map` implementado con una lista
- Técnicas de **open addressing**
  - **Linear Probing**: Si la posición del array está ocupada, uso la siguiente posición (circularmente)
  - **Quadratic Probing**: Si hay colisión, calcula la siguiente opción usando una función $(i + j^2)$ *mod* N con $j = 0, 1, 2 \ldots$
  - **Double Hashing**: Si hay colisión, calcula la siguiente opción usando una función $(i + otrohash(j))$ *mod* N con $j = 0, 1, 2 \ldots$

Implementación de `equals` y `hashcode`

Como ya hemos visto, los objetos utilizados como clave on utilizados para saber si el objeto ya se encuentra en el `Map`. Las implementaciones de Java utilizarán el método `equals` para comprobar si una clave ya se encuentra en el Map o no en el momento de hacer un `put`. Si no se ha implementado el método `equals` en la clase que se utiliza como clave, se utilizará la comparación de la *identidad* (el `equals` de `Object`).

Para la implementación de una tabla de dispersión se debe utilizar una función de compresión. La clase `Object` tiene el método **`public int`** `hashcode ()` que devuelve un identificador de 32 bits para el objeto. Si no se sobrescribe este método, Java devuelve como identificador la dirección de memoria del objeto, es decir, únicamente dos referencias apuntando al mismo objeto, devolverían el mismo valor llamando a `hashcode`.

En caso de sobreescribir el método `equals` se DEBE sobreescribir el método `hashcode` de tal forma que dos objetos que sean iguales, deben devolver el mismo valor al llamar a `hashcode`. Este aspecto es clave a la hora de

utilizar `HashMap` en Java, ya que si dos objetos que consideramos iguales mediante el método `equals`, devuelven diferente valor al llamar a `hashcode` irían a posiciones diferentes de la tabla, con lo que se podrían introducir 2 elementos "iguales" en la tabla y el `HashMap` no lo detectaría. Concretamente, esto es lo que dice la documentación del método `hashcode` de la JVM:

> *Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`. The general contract of hashCode is:*
>
> - *Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer needs not remain consistent from one execution of an application to another execution of the same application.*
> - *If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.*
> - *It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.*

Podemos resumir las opciones en:

- Dados `o1` y `o2` usados como claves puede ocurrir:
  - `o1.hashCode()!=o2.hashCode() && !o1.equals(o2)`
    - OK. Son dos objetos distintos, dos entradas distintas en el map
  - `o1.hashCode()==o2.hashCode() && o1.equals(o2)`
    - OK. Una única entrada en el map
  - `o1.hashCode()==o2.hashCode() && !o1.equals(o2)`
    - Colisión. No hay problema, dos entradas distintas en el map
  - `o1.hashCode()!=o2.hashCode() && o1.equals(o2)`
    - Inconsistente. Si los objetos son iguales deberían tener el mismo `hashCode`
    - Se producen dos entradas distintas en el map, teniendo dos objetos iguales (de acuerdo al resultado de `equals`)

NOTA: Si el objeto implementa `Comparable`, también se debe conservar la coherencia con `compareTo`

10. Árboles

La definición recursiva típica es

Definicion 10.0.1 — árbol general. Un **árbol general** podría ser: un árbol general es o bien vacío o bien tiene dos componentes: (1) un nodo raíz que contiene un elemento, y (2) un conjunto de cero o más (sub)árboles hijos.

Recordad que en un conjunto no hay orden ni repetición de elementos. Un árbol está formado por nodos. Un nodo tiene un elemento y un conjunto de nodos que son la raíz de los subárboles hijos.

Los árboles fundamentalmente se utilizan para organizar datos de manera jerárquica. Los árboles también se utilizan en la implementación de otros TADs tales como colas, ficheros, funciones finitas con dominio ordenado ("ordered maps"), etc.

Una definición más formal la podemos encontrar en [Cor+01] (B.5), [AHU83] (p.231):

Definicion 10.0.2 — Árbol libre. Un *árbol libre* ("free tree") es un grafo conectado, no-dirigido y acíclico.

Definicion 10.0.3 — Árbol Ordinario. Un *árbol ordinario* ("rooted tree") es un árbol libre en el que se elige un nodo como raíz.

En la asignatura utilizaremos la definición de **árbol ordinario**.

Algo de terminología sobre árboles:

- **Raíz (*"root"*)**: nodo sin padre.
- **Nodo interno (*"internal node"*)**: nodo con al menos un hijo.
- **Nodo externo (*"external node"*)**: nodo sin hijos.
- **Nodo hoja (*"leaf node"*)**: sinónimo de nodo externo, usaremos estos dos nombres indistintamente.
- **Subárbol (*"subtree"*)**: árbol formado por el nodo considerado como raíz junto con todos sus descendientes.
- **Ancestro de un nodo**: un nodo 'w' es ancestro de 'v' si y sólo si 'w' es 'v' o 'w' es el padre de 'v' o 'w' es ancestro del padre de 'v'. Un nodo 'w' es un "ancestro propio" de 'v' si y sólo si 'w' es un ancestro de 'v' pero 'w' no es 'v'. (Incluir un nodo como ancestro de sí mismo permitirá definiciones concisas más adelante.)
- **Descendiente de un nodo** (la inversa de ancestro): 'v' es descendiente de 'w' si y sólo si 'w' es ancestro de 'v'.
- **Hermano (*"sibling"*) de un nodo**: nodo con el mismo padre.
- **Arista (*"edge"*) de un árbol**: par de nodos en relación padre-hijo o hijo-padre.
- **Grado (*"degree"*) de un nodo**: el número de hijos del nodo.
- **Grado (*"degree"*) de un árbol**: el máximo de los grados de todos los nodos.
- **Camino (*"path"*) de un árbol**: secuencia de nodos, sin repetir nodos, tal que cada nodo consecutivo forma una arista. La **longitud del camino** es el número de aristas.
- **Árbol ordenado (*"ordered tree"*)**: existe un orden lineal (total) definido para los hijos de cada nodo: primer hijo, segundo hijo, etc. Se visualiza dibujando los hijos en orden de izquierda a derecha bajo el padre.
- La **profundidad de un nodo (*"depth"*)** es la longitud del camino que va desde ese nodo hasta la raíz (o viceversa). La longitud del camino es cero si el nodo es la raíz. También de forma equivalente: la profundidad es el número de ancestros propios del nodo.

- La **altura de un nodo ("*height*")** es la longitud del mayor de todos los caminos que van desde el nodo hasta una hoja.
- **Altura de un árbol no vacío**: la altura de la raíz.
  Profundidad y altura se han definido como propiedades de nodos. Por tanto, según estas definiciones **no tiene sentido hablar de la profundidad o la altura de un árbol vacío** (sin nodos). La profundidad de un árbol podría definirse como la mayor de las profundidades de las hojas, pero ese valor es igual a la altura. De hecho, la altura de un árbol también se define como la máxima profundidad. La altura de un árbol de un sólo nodo sería 0.
  Nivel ('level'): conjunto de nodos con la misma profundidad. Así, tenemos desde el nivel 0 hasta el nivel 'h' donde 'h' es la altura del árbol.

10.1 Interfaz `Tree`

Estudiamos el interfaz `Tree<E>` porque es la base de los interfaces `GeneralTree<E>` y `BinaryTree<E>` de árboles binarios que veremos en más detalle posteriormente. El interfaz `Position<E>` que hemos usado en el capítulo de Listas de Posiciones (Capítulo 6) para abstraer la implementación de los nodos de una lista lo usa ahora `Tree<E>` para abstraer la implementación de los nodos de un árbol. Seguiremos usando la palabra "nodo" en sentido abstracto sin importarnos la clase concreta que lo implementa. Dicho interfaz en encuentra en la librería `aedlib.jar`.

`Tree<E>` es un interfaz pensado para trabajar directamente con posiciones (abstracciones de nodos). Los TADs de árboles suelen usarse en implementaciones de otros TADs y para ello se necesita poder trabajar directamente (y eficientemente) con nodos. Esto explica, en parte, que no es un interfaz recursivo. Los métodos trabajan con posiciones y no con árboles. Cada posición tendrá un elemento y un conjunto de posiciones (que pueden verse como posiciones raíz de subárboles).

El interfaz consta de los siguiente métodos (el detalle de los métodos se puede encontrar en la URL: `dd`)

```
public interface Tree<E> extends Iterable<E> {
    public int size();

    public boolean isEmpty();

    public E set(Position<E> p, E e) throws IllegalArgumentException;

    public Position<E> root();

    public Position<E> parent(Position<E> p) throws IllegalArgumentException;

    public boolean isInternal(Position<E> p);

    public boolean isExternal(Position<E> p);

    public boolean isRoot(Position<E> p);

    public Position<E> addRoot(E e) throws NonEmptyTreeException;

    public Iterator<E> iterator();

    public Iterable<Position<E>> children(Position<E> p);
}
```

El método 'children' devuelve un `Iterable<E>` (repasad el tema de Iteradores (Capítulo 7) con los nodos hijo del nodo. Si el árbol está ordenado entonces los nodos en el iterable estarán ordenado.

Algunos métodos lanzan excepciones cuando los nodos son inválidos:

- `NonEmptyTreeException` será lanzada por `addRoot` cuando se invoque dicho método sobre un árbol que ya tiene raíz.
- `IllegalArgumentException` será lanzada por los métodos que toman posiciones como argumento cuando la posición es inválida (p.ej., cuando no es un objeto de una clase que implementa los nodos del árbol).

El interfaz fundamentalmente consta de métodos observadores excepto los método modificadores `replace` y `addRoot` que permiten modificar el elemento almacenado en un nodo y añadir una raíz al árbol. El resto de métodos modificadores se añadirán en los interfaces que extienden el interfaz `Tree<E>`, es decir, en `GeneralTree<E>` y en `BinaryTree<E>`. A continuación veremos algunos ejemplos de uso de `Tree<E>`

▪ **Ejemplo 10.1** Método que indica si un nodo es ancestro de otro. La excepción será lanzada por 'isRoot' o por 'parent' si los nodos son inválidos. (En todos los ejemplos de este tema asumiremos que los parámetros nunca son null.)

```
public static <E> boolean ancestro(Tree<E> t, Position<E> w, Position<E> v)
     if (w == v) return true;
     if (tree.isRoot(v)) return false;
     return esAncestro (tree,w,tree.parent(v));
}
```

▪

▪ **Ejemplo 10.2** Método que indica si dos nodos de un árbol son hermanos:

```
public static <E> boolean isSibling(Tree<E> t, Position<E> w, Position<E> v) {
   if (w == v || t.isRoot(w) || t.isRoot(v))
     return false;
   else
    return t.parent(w) == t.parent(v);
}
```

Versión más complicada que comprueba si el segundo nodo está en el conjunto de hijos del padre del primero:

```
public static <E> boolean isSibling(Tree<E> t, Position<E> w, Position<E> v) {
  if (w == v || t.isRoot(w) || t.isRoot(v))
    return false;
  else {
    Iterator<Position<E>> it = t.children(t.parent(w)).iterator();
    boolean found = false;
    while (it.hasNext() && !(found = it.next() == v))
      ;
    return found;
  }
}
```

▪

▪ **Ejemplo 10.3** Método que devuelve una lista con los elementos que están en las hojas de un árbol:

```
public static <E> PositionList<E> leaves(Tree<E> t) {
  PositionList<E> r = new NodePositionList<E>();
  if (t.isEmpty()) return r;
  Iterator<Position<E>> it = t.positions().iterator();
  while (it.hasNext()) {
    Position<E> v = it.next();
    if (t.isExternal(v)) r.addLast(v.element());
  }
  return r;
}
```

Versión con "foreach":

```
public static <E> PositionList<E> leaves(Tree<E> t) {
  PositionList<E> r = new NodePositionList<E>();
  if (t.isEmpty()) return r;
  for (Position<E> v : t.positions())
    if (t.isExternal(v)) r.addLast(v.element());
  return r;
}
```

▪

▪ **Ejemplo 10.4** Método que calcula la profundidad de un nodo de un árbol dado. El método asume que el nodo es del árbol.

```
public static <E> int depth(Tree<E> tree, Position<E> v) {
  int c = 0;
  while (!tree.isRoot(v)) {
    c++;
    v = tree.parent(v);
  }
  return c;
}
```

Versión con for:

```
public static <E> int depth(Tree<E> tree, Position<E> v) {
  int c;
  for (c = 0; !tree.isRoot(v); c++, v = tree.parent(v))
    ;
  return c;
}
```

Versión recursiva:

```
public static <E> int depth(Tree<E> tree, Position<E> v) {
  if (tree.isRoot(v)) return 0;
  else                return 1 + depth(tree,tree.parent(v));
}
```

Versión en la que el método está implementado dentro de una clase que implementa el interfaz 'Tree<E>':

```
public int depth(Position<E> v) {
  if (this.isRoot(v)) return 0;
  else                return 1 + depth(this.parent(v));
}
```

▪

▪ **Ejemplo 10.5** Altura de un nodo. Primero vemos una versión recursiva:

```
public static <E> int height(Tree<E> tree, Position<E> v) {
  int h = 0;
  if (tree.isExternal(v)) return 0;

  for (Position<E> w : tree.children(v)) {
    h = Math.max(h, height(tree, w));
  }
  return 1 + h;
}
```

Versión en la que el método está implementado dentro de una clase que implementa el interfaz 'Tree<E>':

```
public int height(Position<E> v) {
  int h = 0;
  if (this.isExternal(v)) return 0;
  else {
    for (Position<E> w : this.children(v)) {
      h = Math.max(h, height(w));
    }
  }
  return 1 + h;
}
```

■

A continuación veremos ejemplos de los posibles recorridos de los nodos de un árbol. Veremos 2 posibles recorridos en profundidad, preorden y postorden, así como el recorrido en anchura (por niveles).

Primero veremos un esqueleto recursivo para hacer un recorrido en preorden:

```
void preorder(Tree<E> tree)
  if (tree.isEmpty()) {
    return;
  }
  preorder(tree,tree.root());
}

public static <E> void preorder(Tree<E> tree, Position<E> v)

  /* En esta línea se hace algo con v.element() */

  for (Position<E> w : tree.children(v)) {
    preorder(tree, w);  /* se recorren los hijos */
  }
}
```

■ **Ejemplo 10.6** Método que mprimime todos los nodos siguiendo el recorrido en preorden.

```
void printPreorder(Tree<E> tree)
  if (tree.isEmpty()) {
    return;
  }
  printPreorder(tree,tree.root());
}

public static <E> String printPreorder(Tree<E> tree, Position<E> v) {

  Systen.out.println(v.element());
  for (Position<E> w : tree.children(v)) {
     printPreorder(tree,w);
  }
}
```

■

Par aun recorrido en postorden, primero recorremos todos los hijos y luego visitamos el nodo. Un esqueleto de código sería:

```
public static void postorder(Tree<E> tree)
  if (tree.isEmpty()) {
    return;
  }
  postorder(tree,tree.root());
}

private static <E> void postorder(Tree<E> tree, Position<E> v)

  for (Position<E> w : tree.children(v)) {
    postorder(tree, w);  /* se recorren los hijos */
  }

  /* En esta línea se hace algo con v.element() */

}
```

■ **Ejemplo 10.7** Método que imprime todos los elementos de un árbol en postorden.

```
public static void printPostorder(Tree<E> tree)
  if (tree.isEmpty()) {
    return;
  }
  printPostorder(tree,tree.root());
}

private static <E> String printPostorder(Tree<E> tree, Position<E> v) {

  for (Position<E> w : tree.children(v)) {
     printPostorder(tree,w);
  }
  Systen.out.println(v.element());
}
```

■

El recorrido en anchura se realiza por niveles, es decir, primero se visitan todos los nodos de nivel 0, luego todos los nodos de nivel 1, nivel 2, ...

Para recorrer un árbol en anchura se necesita una cola FIFO en la que ir almacenando los nodos de izquierda a derecha por nivel. Este podría ser un esqueleto de código para este recorrido:

```
public static <E> void breadth(Tree<E> tree) {
    if (tree != null && !tree.isEmpty()) {
    FIFO<Position<E>> fifo = new FIFOList<Position<E>>();
    fifo.enqueue(tree.root());

    while (!fifo.isEmpty()) {
      Position<E> v = fifo.dequeue();

      // visit v.element();

      for (Position<E> w : tree.children(v)) {
        fifo.enqueue(w);
      }
    }
  }
}
```

■ **Ejemplo 10.8** Un posible versión recursiva pasaría la cola FIFO como parámetro de acumulación, pues es la estructura que se recorre para obtener el resultado:

```
public static <E> void breadth(Tree<E> tree) {
  if (tree != null && !tree.isEmpty()) {
    FIFO<Position<E>> fifo = new FIFOList<Position<E>>();
    fifo.enqueue(tree.root());
    breadthRec(tree,fifo);
}

private static <E> void breadthRec(Tree<E> tree, FIFO<Position<E>> v) {
  if (!fifo.isEmpty()) {
    Position<E> v = fifo.dequeue();

    // visit v.element();

    for (Position<E> w : tree.children(v)) {
      fifo.enqueue(w);
    }

    breadthRec(tree,fifo);

  }
}
```

■

■ **Ejemplo 10.9** Método que devuelve una lista con todas las hojas de un árbol

```
public static <E> PositionList<E> leaves(Tree<E> t) {
  PositionList<E> res = new NodePositionList<E>();
  if (t.isEmpty()) {
    return res;
  }
  leaves(t,t.root(),res);
  return res;
}

public static <E> void leaves(Tree<E> t,
                              Position<E> node,
                              PositionList<E> res) {

  if (t.isExternal(node)) {
    res.addLast(node.element());
  }
  for (Position<E> w: t.children(node)) {
    leaves(t,w,res);
  }
}
```

■

■ **Ejemplo 10.10** Método que calcula la altura de un cierto nodo y otro que calcula la suma de todos los elementos de un árbol de números:

```
public static <E> int height(Tree<E> tree, Position<E> v){

  if (tree.isExternal(v)) return 0;

  int h = 0;
  for (Position<E> w : tree.children(v)) {
    h = Math.max(h, height(tree, w));
  }
  return h + 1;
}
```

■

10.2 Interfaz `GeneralTree`

El interfaz `GeneralTree<E>` extiende el interfaz `Tree<E>` con métodos modificadores para permitir la creación del árbol.

```
public interface GeneralTree<E> extends Tree<E> {

    public Position<E> addChildFirst(Position<E> parentPos, E e);

    public Position<E> addChildLast(Position<E> parentPos, E e);

    public Position<E> insertSiblingBefore(Position<E> siblingPos, E e);

    public Position<E> insertSiblingAfter(Position<E> siblingPos, E e);

    public void remove(Position<E> p);
}
```

La documentación detallada de los métodos se encuentra en `dddd`. Obsérvese que los métodos son similares a las inserciones en una `PositionList<E>`. Lo más relevante es que el método `remove` borra TODO el súb-arbol

que tiene por debajo, no únicamente el nodo.

Por ejemplo, el siguiente código:

```
GeneralTree<Integer> tree = new LinkedGeneralTree<>();
tree.addRoot(1);
Position<Integer> n2 = tree.addChildLast(tree.root(), 2);
Position<Integer> n3 = tree.addChildLast(tree.root(), 3);

Position<Integer> n4 = tree.addChildLast(n2, 4);
Position<Integer> n6 = tree.addChildLast(n2, 6);
Position<Integer> n5 = tree.insertSiblingBefore(n6, 5);

Position<Integer> n7 = tree.addChildLast(n3, 7);
Position<Integer> n8 = tree.insertSiblingAfter(n7, 8);
Position<Integer> n9 = tree.addChildLast(n8, 9);
```

genera el árbol:

```
|-- 1
|   |-- 2
|   |   |-- 4
|   |   |-- 5
|   |   |-- 6
|   |-- 3
|       |-- 7
|       |-- 8
|           |-- 9
```

Después de hacer un `tree.remove(n3)`, el árbol resultante sería:

```
|-- 1
|   |-- 2
|   |   |-- 4
|   |   |-- 5
|   |   |-- 6
```

10.3 Interfaz `BinaryTree`

Un **árbol binario** es un caso especial de árbol general en el que todo nodo tiene como máximo 2 hijos, el hijo izquierdo ("*left child*") y el hijo derecho ("*right child*"), que están ordenados: el izquierdo precede al derecho.

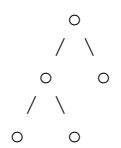
Definicion 10.3.1 — árbol binario. Un *árbol binario* es un árbol ordinario con grado 2

También lo podemos hacer de forma recursiva:

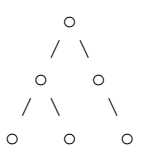
Definicion 10.3.2 Un árbol binario es o bien vacío o bien consiste en (1) un nodo raíz, (2) un (sub)árbol izquierdo, y (3) un (sub)árbol derecho.

Veamos algo más de terminología:

- Un **árbol binario propio** ("*proper binary tree*"), también llamado árbol estrictamente binario ("*strictly binary tree*"), también llamado *2-Tree*: todo nodo interno tiene 2 hijos. De forma equivalente: todo nodo (interno o externo) tiene o bien 0 o bien 2 hijos.



- Un **árbol binario impropio** ("*improper binary tree*"): es un árbol binario que no es propio.



El número de nodos de un árbol binario (sea propio o impropio) es como máximo $2^{h+1}-1$. El número de nodos internos de un árbol binario (sea propio o impropio) es como máximo 2^h-1.

- Un **árbol binario perfecto** ("*perfect binary tree*): árbol binario propio con el máximo número de nodos $2^{h+1}-1$.

```
    o         h         = 0
              2^{h+1}-1 = 1

    o         h         = 1
   / \        2^{h+1}-1 = 3
  o   o

    o         h         = 2
   / \        2^{h+1}-1 = 7
  o   o
 / \ / \
o  oo   o
```

Todas las hojas están en el mismo nivel y todos los nodos internos tienen dos hijos.
La altura de un árbol perfecto es O(log(n)).
Demostración: puesto que es perfecto, el número de nodos n es $2^{h+1}-1$.

```
  n = 2^{h+1}-1

<=> { por aritmética }

  n+1 = 2^{h+1}

<=> { tomando 'log' a ambos lados pues 'log' es una función }

 log(n+1) = log(2^{h+1})

<=> { por log(2^x) = x }

 log(n+1) = h+1

<=> { por aritmética }

 log(n+1) - 1 = h
```

Con este resultado podemos establecer que $h = O(log(n))$ encontrando la constante multiplicativa y el n_0.

El interfaz `BinaryTree<E>` extiende la funcionalidad de `Tree<E>` con métodos específicos de árboles binarios:

```
public interface BinaryTree<E> extends Tree<E> {

    public boolean hasLeft(Position<E> p);

    public boolean hasRight(Position<E> p);

    public Position<E> left(Position<E> p);

    public Position<E> right(Position<E> p);

    public Position<E> insertLeft(Position<E> parentPos, E e)
                                          throws NodeAlreadyExistsException;

    public Position<E> insertRight(Position<E> parentPos, E e)
                                          throws NodeAlreadyExistsException;

    public void remove(Position<E> p);
}
```

El funcionamiento detallado de cada método se encuentra en la XXX.

Dado que el interfaz `BinaryTree<E>` extiende el interfaz `Tree<E>` los métodos implementados anteriormente en los ejemplos de la Sección 10.1 funcionarían correctamente para árboles binarios. En cualquier caso, vamos a ver

algunos de estos métodos implementados con los métodos específicos de los árboles binarios.

▪ **Ejemplo 10.11** El método que calcula la profundidad de un nodo es idéntico al de árboles generales salvo que el parámetro 'tree' ahora sería de tipo `BinaryTree<E>`:

```
public static <E> int depth(BinaryTree<E> tree, Position<E> v) {
  if (tree.isRoot(v)) return 0;
  else return 1 + depth(tree, tree.parent(v));
}
```

▪

▪ **Ejemplo 10.12** El método que calcula la altura de un nodo también puede ser idéntico al de árboles generales, salvo en el tipo del parámetro 'tree', ya que se dispone del método `children` para árboles binarios heredado de `Tree<E>`. Mostramos otra alternativa que usa los métodos de `BinaryTree<E>`.

```
public static <E> int height(BinaryTree<E> tree, Position<E> v) {
  int hi = 0, hd = 0;
  if (tree.isExternal(v)) return 0;

  if (tree.hasLeft(v))  hi = height(tree, tree.left(v));
  if (tree.hasRight(v)) hd = height(tree, tree.right(v));
  return 1 + Math.max(hi,hd);
}
```

▪

Los recorridos en preorden y postorden también podrían hacerse con los métodos específicos de los `BinaryTree<E>`: El recorrido en preorden sería:

```
public void <E> preorder(BinaryTree<E> tree, Position<E> v)  {
  /* visit v.element() */
  if (t.hasLeft(v))  preorder(tree, tree.left(v));
  if (t.hasRight(v)) preorder(tree, tree.right(v));
}
```

El recorrido en preorden sería:

```
public void <E> postorder(BinaryTree<E> tree, Position<E> v)  {
  if (t.hasLeft(v))  postorder(tree, tree.left(v));
  if (t.hasRight(v)) postorder(tree, tree.right(v));
  /* visit v.element() */
}
```

En árboles binarios aparece un nuevo posible recorrido en profundidad, el recorrido **inorden**, que visita el nodo después de visitar toda la rama izquierda pero antes de visitar la rama derecha.

```
public void <E> inorder(BinaryTree<E> tree, Position<E> v)
throws InvalidPositionException {
  if (tree.hasLeft(v))  inorder(tree, tree.left(v));

  /* visit v.element() */

  if (tree.hasRight(v)) inorder(tree, tree.right(v));
}
```

El recorrido en anchura también sería idéntico al de árboles generales, salvo en el tipo del parámetro 'tree'. Mostramos otra alternativa que usa los métodos de 'BinaryTree<E>'.

```
public static <E> void breadth(BinaryTree<E> tree) {
  if (tree != null && !tree.isEmpty()) {
    FIFOQueue<Position<E>> fifo = new FIFOQueue<Position<E>>();
    fifo.enqueue(tree.root());

    while (!fifo.isEmpty()) {
      Position<E> v = fifo.dequeue();

      // visit v.element();

      if (tree.hasLeft(v))  fifo.enqueue(tree.left(v));
      if (tree.hasRight(v)) fifo.enqueue(tree.right(v));

    }
  }
}
```

■ **Ejemplo 10.13** Implementación del método `member` que busca un si un elemento está en el árbol.

```
public static <E> boolean member (BinaryTree<E> tree, E e) {
  return member(tree,tree.root(),e);
}

private static <E> boolean member (BinaryTree<E> tree, Position <E> n, E e) {
  if (n.element().equals(e)) return true;
  if (tree.isExternal(n)) return false;

  boolean found = false;

  if (tree.hasLeft(n)) {
    found = member(tree, tree.left(n),e);
  }
  if (!found && tree.hasRight(n)) {
    found = member(tree, tree.right(n),e);
  }
  return found;
}
```

■

La construcción de un árbol binario se puede hacer utilizando los métodos `insertLeft`, `insertRight` y `addRoot`. Al igual que en árboles generales, el método `remove(p)` borra todos los descendientes del nodo `p`.

```
LinkedBinaryTree<Integer> tree = new LinkedBinaryTree<Integer>();
tree.addRoot(1);
Position<Integer> left = tree.insertLeft(tree.root(), 2);
Position<Integer> right = tree.insertRight(tree.root(), 3);
Position<Integer> n4 = tree.insertLeft(left,4);
Position<Integer> n5 = tree.insertRight(left,5);
Position<Integer> n6 = tree.insertLeft(right,6);
```

producirá el árbol:

```
          /----- 4
   /----- 2
   |      \----- 5
  1
   |      /----- 6
   \----- 3
```

■ **Ejemplo 10.14** Los árboles binarios pueden servir para representar y evaluar expresiones aritméticas o booleanas. Por ejemplo el árbol:

```
LinkedBinaryTree<Character> tree = new LinkedBinaryTree<Character>();
tree.addRoot('&');
Position<Character> nl = tree.insertLeft(tree.root(),'|');
Position<Character> nr = tree.insertRight(tree.root(),'&');
tree.insertLeft(nl,'T');
tree.insertRight(nl,'F');
tree.insertLeft(nr,'T');
tree.insertRight(nr,'T');
```

```
          /----- T
 /----- &
 |      \----- T
&
 |      /----- F
 \----- |
          \----- T

 representa la expresión booleana: ((T|F)&(T&T))
```

y que se podrá imprimir y evaluar de forma recursiva con los siguientes métodos:

```
public static <E> void imprimirExpresion(BinaryTree<E> tree, Position<E> v) {
  if (!tree.isExternal(v)) System.out.print("(");

  if (tree.hasLeft(v)) {
    imprimirExpresion(tree, tree.left(v));
  }
  System.out.print(v.element());
  if (tree.hasRight(v)) {
    imprimirExpresion(tree, tree.right(v));
  }

  if (!tree.isExternal(v)) System.out.print(")");
}

public static boolean eval(BinaryTree<Character> expTree, Position<Character> v) {

  switch(v.element()) {
  case 'T': return true;
  case 'F': return false;
```

```
  case '|':
    return  eval(expTree,expTree.left(v)) || eval(expTree,expTree.right(v));
  case '&':
    return  eval(expTree,expTree.left(v)) && eval(expTree,expTree.right(v));
  default:
    throw new IllegalArgumentException("El arbol no contiene los elementos correctos " + v.eleme
  }
}
```

■

11. Colas con Prioridad y Montículos

11.1 Colas con Prioridad

En una cola FIFO ("First In First Out") el primer elemento en entrar (utilizando el método llamado típicamente 'enqueue') es el primero en salir (utilizando el método llamado típicamente 'first' o 'dequeue'). En las colas con prioridad el orden de salida viene determinado por la *prioridad* del elemento. El primer elemento en salir es el de mayor prioridad. Ejemplos:

- Cola del supermercado donde la prioridad del cliente es su pertenencia a algún colectivo (personas mayores, discapacidad, etc.)
- Sala de urgencias de un hospital donde la prioridad del paciente es la gravedad de la urgencia.
- Cola de procesos en memoria para ejecutar usada por el núcleo del sistema operativo, donde la prioridad del proceso es determinada por el propio sistema operativo, el usuario o el administrador.

Para los elementos de igual prioridad se sigue un esquema FIFO. A nivel *conceptual* puede verse una cola con prioridad como una estructura lineal en la que los elementos se almacenan en orden de prioridad. A nivel de *implementación* esto no tiene que ser así, lo único importante es que la operación de "desencolar" devuelva el elemento de mayor prioridad. Internamente los elementos pueden estar almacenados de otra forma.

La prioridad de un elemento vendrá dada por un objeto de una clase que representa prioridades. Llamamos **clave** (*key*) a dicho objeto y llamamos **valor** (*value*) al elemento. La asociación de una clave a un valor, o par **clave-valor**, lo llamamos **entrada** (*entry*). Las colas con prioridad que veremos almacenan entradas. En Java utilizaremos el interfaz `Entry<K,V>` para manejar las entradas.

Por convención: **la clave establece la prioridad inversamente: cuanto menor es la clave mayor la prioridad**. Este tipo de colas con prioridad se suele llamar "*min-max queue*" porque desencolar devuelve el elemento de menor clave (mayor prioridad). Podríamos haber seguido la convención opuesta de "*max-min queue*".

Las entradas son pares de una relación, es decir, de un subconjunto de un producto cartesiano:

- Dos o más entradas pueden tener la misma clave pero distintos valores. En otras palabras, valores distintos pueden tener la misma prioridad. Ejemplo: dos pacientes diferentes pueden tener dolencias de igual gravedad.
- Dos o más entradas pueden tener los mismos valores pero distintas claves. En otras palabras, el mismo valor puede tener distintas prioridades en distintas entradas. Ejemplo: el mismo paciente puede tener dos dolencias, una grave y otra menos grave.
- Puede modificarse la clave o el valor de una entrada. En el primer caso estamos diciendo que la prioridad de un valor puede cambiarse para esa entrada.

El interfaz `PriorityQueue<K,V>` tiene los siguientes métodos:

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

  public int size();

  public boolean isEmpty();

  public Entry<K,V> first() throws EmptyPriorityQueueException;

  public void enqueue(K key, V value) throws InvalidKeyException;
```

```
  public Entry<K,V> dequeue() throws EmptyPriorityQueueException;
}
```

El detalle de los métdos del interfaz se puede encontrar en XXX. Algumos comentarios sobre el interfaz:

- Los métodos `first` y `dequeue` devuelven objetos de una clase que implementa el interfaz `Entry<K,V>`.
- El método `first` devuelve el objeto entrada de menor clave (mayor prioridad) sin quitarlo de la cola. Es un método "observador".
- El método `dequeue` devuelve el objeto entrada de menor clave (mayor prioridad) y lo quita de la cola. Es un método "observador" con un efecto secundario "modificador".
- Excepciones:
  - `EmptyPriorityQueueException` se lanza cuando se intenta obtener o borrar la entrada de clave mínima en una cola vacía.
  - `InvalidKeyException` se lanza cuando la clave no es válida, por ejemplo si es null o no tiene igualdad o orden total definido para ella.

El interfaz `Entry<K,V>` tendrá la siguiente forma:

```
public interface Entry<K,V> {
  public K getKey();
  public V getValue();
}
```

No hay "setters", por tanto no es posible modificar una entrada a través de los métodos del interfaz.

En una cola con prioridad puede usarse como clave un atributo del valor (p.ej., el precio de un libro), pero **debe tenerse un interfaz o clase para el tipo clave (no puede ser un tipo básico)**, pues es un parámetro genérico del interfaz `Entry<K,V>`, es decir, la prioridad tendría que ser un `Integer` y no un **`int`**. Además las claves debe poder ordenarse:

- Implementando el interfaz `Comparable`
- Disponer de un comparador que implemente el interfaz `Comparator<E>`

■ **Ejemplo 11.1** Ejemplo trivial de uso de colas con prioridad

```
PriorityQueue<Integer,String> cola = new SortedListPriorityQueue<Integer,String>();

Alumno alumno = new Alumno("Alumno de la ETSIINF");

cola.enqueue(1, "Programación II");
cola.enqueue(4, "Algorítmica Numérica");
cola.enqueue(3, "Lenguajes y Autómatas");
cola.enqueue(0, "AED");

while (cola.size() > 0) {
  alumno.estudiar(cola.dequeue());
}
}
```

¿Se podrían mostrar por pantalla los valores de `cola` sin usar `dequeue`? ■

Implementación de cola con prioridad mediante una lista

Una cola con prioridad se puede implementar usando una lista cuyos elementos son entradas. La clase que implementa la cola debe tener al menos dos atributos: la lista de entradas y el comparador de claves:

```
public class PositionListPriorityQueue<K,V> implements PriorityQueue<K,V> {

  private PositionList<Entry<K,V>> list;
  private Comparator<K> comp;

  public PositionListPriorityQueue(Comparator<K> comp) {
    this.list = new NodePositionList<Entry<K,V>>();
    this.comp  = comp;
  }
  ...
}
```

Se pueden dar más constructores y se puede intentar usar un comparador por defecto, es decir, definir `Comparator<T>` a partir de `Comparable<T>`. Para esto tenemos varias opciones:

- Una **lista desordenada**: se inserta con `addFirst` o `addLast` en tiempo constante. El método `first` recorre la lista entera para encontrar la entrada de menor clave. El método `dequeue` invoca `first` y además borra la entrada encontrada.
  Complejidad:

Método	Complejidad
`enqueue`	$O(1)$
`first`	$O(n)$
`dequeue`	$O(n)$

- **Lista desordenada con entrada mínima en cache**: se mantiene en un nuevo atributo la entrada con clave mínima para poder devolverla en O(1). El método `enqueue` actualiza el atributo en O(1):

```
public class PositionListPriorityQueue<K,V> implements PriorityQueue<K,V> {

     private PositionList<Entry<K,V>> list;
     private Comparator<K> comp;
     private Entry<K,V> minEntry;

     public PositionListPriorityQueue(Comparator<K> comp) {
       this.list    = new NodePositionList<Entry<K,V>>();
       this.comp     = comp;
       this.minEntry = null;
     }

     public Entry<K,V> first() throws EmptyPriorityQueueException {
       if (this.isEmpty()) throw new EmptyPriorityQueueException();
       return minEntry;
     }

     public Entry<K,V> enqueue(K k, V v) {
       Entry<K,V> e = new MyEntry<K,V>(k,v);
       if (this.isEmpty())
         minEntry = e;
       else if (comp.compare(e.getKey(),minEntry.getKey()) < 0)
         minEntry = e;
       ...  // resto del código de enqueue
     }
     ...
}
```

Método	Complejidad
`enqueue`	$O(1)$
`first`	$O(1)$
`dequeue`	$O(n)$

  Aunque `first` es constante, el borrado sigue teniendo O(n) porque hay que buscar la siguiente entrada más pequeña en la lista para actualizar el atributo `minEntry`. Cuando la lista es vacía, `minEntry` es null.
- **List ordenada**: se inserta en orden de forma que la entrada con clave mínima siempre está en el primer nodo de la lista. La clave mínima se obtiene con `first` y se puede borrar con `remove` del primer nodo en tiempo constante.
  Complejidad:

Método	Complejidad
`enqueue`	$O(n)$
`first`	$O(1)$
`dequeue`	$O(1)$

Ordenación con cola con prioridad

El siguiente método `pqSort` ordena una lista usando una cola con prioridad. El método toma como parámetro la lista a ordenar y el comparador de elementos que establece la relación de orden. El comparador se le pasa al constructor de la cola.

```
public static <E> void pqSort(PositionList<E> list, Comparator<E> c) {
  PriorityQueue<E,Object> pq = new SortedListPriorityQueue<E,Object>(c);

  while (list.size() > 0) {
    pq.enqueue(list.remove(list.first()),null);
  }
  while (pq.size() > 0) {
    list.addLast(pq.dequeue().getKey());
  }
}
```

En esta versión se usan los elementos de la lista como claves y se ignoran los valores. Ahora veamos una versión alternativa que usa los elementos como clave y valor:

```
public static <E> void pqSort(PositionList<E> list, Comparator<E> c) {
  PriorityQueue<E,E> pq = new SortedListPriorityQueue<E,E>(c);

  while (list.size() > 0) {
    E e = list.first().element();
    pq.enqueue(list.remove(list.first()),e);
  }

  while (pq.size() > 0)
    list.addLast(pq.dequeue().getValue());
}
```

Versión que ordena arrays:

```
public static <E> void pqSort(E [] v, Comparator<E> c) {
  PriorityQueue<E,E> pq = new SortedListPriorityQueue<E,E>(c);

  for (E e : v)
    pq.enqueue(e,e);

  for (int i = 0; pq.size() > 0; i++)
    v[i] = pq.dequeue().getValue();
}
```

La complejidad del método de ordenación depende de cómo se implemente la cola con prioridad.

1. Si la cola con prioridad se implementa mediante una *lista desordenada* tenemos el algoritmo *selection-sort* cuya complejidad es $O(n^2)$ con n el tamaño de la lista. Se llama así porque la complejidad está en `dequeue`, es decir, en seleccionar la entrada con clave mínima en la lista desordenada atributo de la cola.
2. Si la cola con prioridad se implementa mediante una *lista ordenada* tenemos el algoritmo *insertion-sort* cuya complejidad es $O(n^2)$ con n el tamaño de la lista. Se llama así porque la complejidad está en `enqueue`, es decir, en insertar la entrada de forma ordenada en la lista ordenada atributo de la cola.
3. Si la cola con prioridad se implementa usando un montículo o *heap* (que veremos a continuación) tenemos el algoritmo *heap-sort* cuya complejidad es $O(n*log(n))$ con n el tamaño de la lista a ordenar.

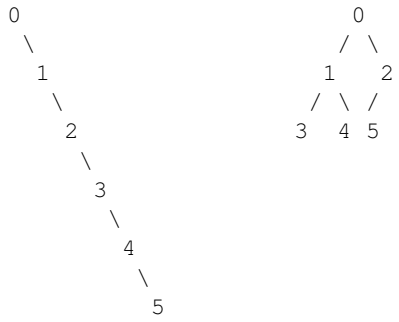
11.2 Montículos

El *montículo* (*heap*) es una implementación de colas con prioridad que satisface las siguientes complejidades:

Método	Complejidad
`enqueue`	$O(log(n))$
`first`	$O(1)$
`dequeue`	$O(log(n))$

Un montículo se describe usando árboles binarios, pero se implementa utilizando arrays. La idea fundamental es conseguir que la posición de una entrada en la cola con prioridad no dependa de la posición absoluta de su clave en el orden total de claves, como ocurre en una implementación de colas con prioridad mediante listas de posiciones. Se utiliza un árbol (casi)completo para particionar la entrada de forma exponencial y reducir así el número de comparaciones de forma logarítmica. En otras palabras, en vez de tener una cola formada por una única fila de

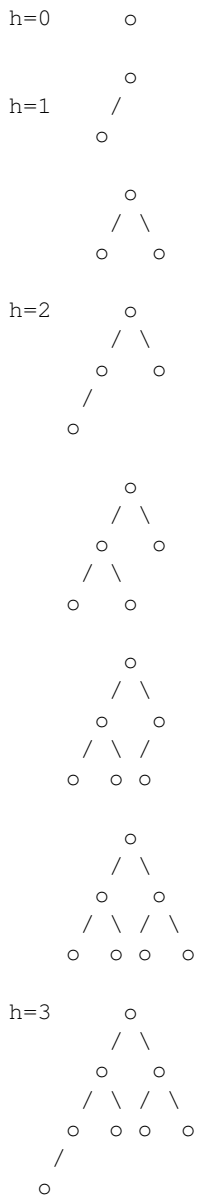
tamaño 'n', como sucede en la implementación mediante lista, tendremos muchas filas de tamaño 'log(n)' que formarán las ramas de un árbol binario.

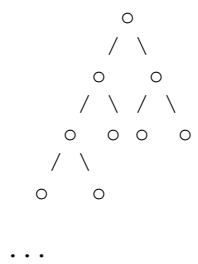


Arboles binarios (casi)completos

Repasad definiciones y terminología de árboles generales y binarios en el tema de árboles. En particular: nodo interno y externo, altura, nivel, árbol binario propio y árbol binario perfecto.

Una definición informal de *árbol (casi)completo* podría ser: árbol binario en el que se insertan elementos por niveles de izquierda a derecha. Por ejemplo:





...

Decimos que el árbol es (casi)completo porque puede estar completo o faltarle únicamente los nodos más a la derecha del último nivel. En [GTG14] llaman "árbol binario completo" (*complete binary tree*) a lo que aquí llamamos "árbol (casi)completo". Realmente no tiene sentido llamarlo completo. Otros autores definen árbol binario completo como sinónimo de árbol binario perfecto. La terminología no es estándar y es posible que textos diferentes llamen de forma diferente al mismo tipo de árbol.

Definicion 11.2.1 — nodo a la izquierda. Un nodo 'v' está a la izquierda de otro 'w' si 'v' aparece a la izquierda de 'w' en un recorrido en inorden.

Definicion 11.2.2 — árbol (casi)completo. Un *árbol (casi)completo*

1. Es perfecto hasta el nivel h-1, es decir, los niveles 0 a h-1 tienen el máximo número de nodos (2^n nodos en cada nivel n=0 hasta n=(h-1)).
2. En el nivel h-1 los nodos internos (tiene que haberlos) están a la izquierda de los nodos hoja (si los hay).
3. En el nivel h-1 hay como máximo un nodo interno con un único hijo izquierdo. Los nodos a la izquierda en ese nivel (si hay) del nodo con un único hijo izquierdo tienen dos hijos.

Definicion 11.2.3 — último nodo de un árbol (casi)completo. la hoja más a la derecha en el último nivel. Se trata del último nodo insertado.

La altura de un árbol (casi)completo es $log(n)$. El valor de la altura está acotado asintóticamente por el logaritmo en base dos del número de nodos almacenados en el árbol.

Montículos

Definicion 11.2.4 — montículo (*heap*). Arbol binario (casi)completo que almacena entradas en los nodos tal que para todo nodo distinto de la raíz, la clave de su entrada es mayor o igual que la clave de la entrada almacenada en el nodo padre.

En otras palabras: todos los caminos de la raíz a las hojas están ordenados de forma ascendiente. Como consecuencia de esto, la entrada de menor clave está almacenada en el nodo raíz. A esta propiedad se le llama *heap-order property*.

El nombre montículo viene de *amontonar* claves en orden ascendente. Se puede definir un montículo para que la clave mayor esté en la raíz. Basta con definir un comparador que invierta el orden. Estos montículos se llaman *max-heaps*. La definición vista arriba es para *min-heaps*.

Clase `Heap<K,V>`

- Atributos mínimos:
  - Un array de entradas.
  - El atributo 'size' que lleva cuenta del tamaño (número de nodos) del montículo.
  - Un comparador de claves.
- El constructor debe tomar un comparador para claves.
- El método `min` devolverá la entrada en el nodo raíz con complejidad $O(1)$.
- El método `enqueue`:
  1. Inserta la entrada al final del árbol, con complejidad O(1). El árbol resultado sigue siendo (casi)completo.
  2. Se comprueba la *heap-order property* entre el nuevo nodo y su padre. Si no se cumple entonces se intercambian las entradas entre el nodo nuevo y el padre. Se repite la operación con el nodo intercambiado y el padre correspondiente, así hasta que se cumpla la propiedad o hasta llegar a la raíz. A este proceso se le llama *up-heap bubbling*, abreviado *upheap*. Tiene complejidad $O(log(n))$ pues la altura de un árbol (casi)completo es $O(log(n))$. Por tanto, la complejidad de 'enqueue' es O(log(n)).
- El método `remove`:
  1. Debe eliminar la entrada de menor clave, pero dicha entrada está en el nodo raíz, que no puede eliminarse. En su lugar, se mueve la entrada del último nodo del árbol binario (casi)completo (el último nodo del

vector) a la raíz en O(1) y después se elimina el último nodo en O(1).

2. Se comprueba la "heap-order property.[en]tre la raíz y el hijo izquierdo (si la raíz tiene un hijo) o el hijo de *menor* clave (si la raíz tiene dos hijos), intercambiándose las entradas para satisfacer la propiedad. (Si se hubiese elegido el hijo de *mayor* clave entonces el intercambio podría violar la *heap-order property* con el hermano.) Se repite la operación con el hijo intercambiado y sus hijos si los hubiera hasta que se cumpla la "heap-order property.[o] se llegue a una hoja. A este proceso se le llama *down-heap bubbling*, abreviado *downheap*. Tiene complejidad $O(log(n))$ pues la altura de un árbol (casi)completo es $O(log(n))$. Por tanto, la complejidad de 'remove' es O(log(n)).

Heap Sort: ordenación por montículo

Recordad el método 'pqSort' visto en en. Sustituid la clase `SortedListPriorityQueue` por la clase `HeapPriorityQueue`. A este algoritmo se le llama *heap-sort*. Cada elemento de la lista se inserta en la cola con prioridad implementada mediante montículo y se elimina de la lista. Esta operación tiene coste $O(n*log(n))$. Después se borra el mínimo de la cola con prioridad y se inserta al final de la lista. Esta operación tiene coste $O(n*log(n))$. La complejidad total es $O(n*log(n))$.

Representación de árboles (casi)completos con arrays

La definición de árbol (casi)completo sugiere una representación del árbol mediante arrays. El árbol es perfecto en los niveles 0 a $h-1$ y en el nivel h se llena de izquierda a derecha con lo que se pueden meter todos los nodos *consecutivamente* en un array:

```
   a          [a,_,_,_,_,_,_,_,.....]
              0 1 2 3 4 5 6 7

   a          [a,b,_,_,_,_,_,_,.....]
  /           0 1 2 3 4 5 6 7
 b

   a          [a,b,c,_,_,_,_,_,.....]
  / \         0 1 2 3 4 5 6 7
 b   c

   a          [a,b,c,d,_,_,_,_,.....]
  / \         0 1 2 3 4 5 6 7
 b   c
/
d

   a          [a,b,c,d,e,_,_,_,.....]
  / \         0 1 2 3 4 5 6 7
 b   c
/ \
d   e

  ...

   a          [a,b,c,d,e,f,g,_,.....]
  / \         0 1 2 3 4 5 6 7
 b   c
/ \ / \
d  e f  g

   a          [a,b,c,d,e,f,g,h,.....]
  / \         0 1 2 3 4 5 6 7
 b   c
/ \ / \
d  e f  g
/
h
```

La lógica de índices sería:

- Sea 'size' el número de elementos almacenados en el array (que no es lo mismo que el tamaño del array).
- El montículo es vacío cuando 'size == 0'.

- La raíz debe ir en la posición 0.
- El último nodo ocupará el índice 'size - 1'
- Métodos de indexado para navegar por el "árbol" en el array:

```
int left(int i) { return 2*i+1;   }

int right(int i) { return 2*i+2;   }

int parent(int i) { return (i-1)/2; /* división entera */ }

boolean hasLeft(int i) { return i >= 0 && left(i)  < size;  }

boolean hasRight(int i) { return i >= 0 && right(i) < size;  }

boolean hasParent(int i) { return i > 0 && i < size; }
```

Usando la lógica de índices y las funciones sobre índices, la "heap-order property" puede expresarse así: para i tal que $0 < i$ y $i < size$ se cumple `v[i].getKey() >= v[parent(i)].getKey()`, donde 'v' es el array.

12. Grafos

Un grafo es una forma de representar relaciones entre pares de objetos. Un **grafo** está formado por objetos, conocidos como nodos o vértices y de una colección de conexiones entre vértices conocidas como aristas.

Definicion 12.0.1 — Grafo. Un grafo es un conjunto de objetos, llamados vértices (vertices), y una colección de aristas (edges), donde cada arista conecta dos vértices

Podemos ver un grafo como un conjunto de vértices $V = \{u, v, w, x \ldots\}$ y una colección de aristas $E = [(u,v),(u,x),\ldots]$. Nótese la diferencia entre "conjunto" de vértices frente a "colección" de aristas, es decir, los nodos no se repiten, pero sí que puede haber dos aristas que conecten los mismos nodos.

Los grafos son de aplicación en mútiples dominios: mapas, transporte, instalaciones eléctricas, redes de computadores, conexiones en redes sociales, ...

Las aristas que conectan los vértices (o nodos) de un grafo pueden ser de dos tipos

- *Aristas no dirigidas*: Decimos que una arista es no dirigida cuando el par (u,v) no está ordenado
  - La arista te lleva de u a v y de v a u
  - El par (u,v) sería lo mismo que el par (v,u)
- *Aristas dirigidas*: Decimos que una arista es dirigida cuando el par (u,v) está ordenado
  - La arista únicamente te lleva de u a v, pero no de v a u

Si todas las arista de un grafo son aristas no dirigidas, decimos que el grafo es no dirigido. Si hay alguna arista dirigida, el grafo es un grafo dirigido. Habitualmente se trabajará con un grafo completo dirigido o no dirigido, no es frecuente encontrar grafos que combinen ambos tipos de aristas.

Algunas definiciones de interés (hay muchas más) serían:

- Dos vértices son *adyacentes (adjacent)* si hay una arista que los conecta
- El *origen* y *destino* son los vértices inicial y final de una arista dirigida
- Un nodo puede tener *aristas salientes (outgoing edges)* que tienen como origen el nodo y *aristas entrantes (incoming edges)*, que tienen el nodo como destino
- El *grado* de un nodo es el número de aristas que entran y salen del nodo
  - Podemos distinguir entre el grado *entrante* y el grado *saliente*
- Un *camino (path)* es una sencuencia de vértices y aristas que empieza en y acaba en un vértice, de forma que cada arista del camino es adyacente con su vértice anterior y su vértice siguiente del camino
- Un *ciclo (cycle)* es un camino cuyo primer y último nodo son el mismo
- Un *camino simple* es un camino que no repite vértices (no contiene ciclos)
- Un *bosque* es un grafo sin ciclos

Como ya se ha comentado, es habitual que nos encontremos con grafos dirigidos y con grafos no dirigidos, pero no una mezcla de ambos. En la librería de la asignatura tendremos un interfaz específico para cada uno de ellos. Los métodos comunes del interfaz se encontrarán en el inbterfaz `Graph<V,E>`. Dicho interfaz dispone de dos genéricos para poder incluir información en los vértices (de tipo `V`) y en las aristas (de tipo `E`).

El interfaz `Graph<V,E>` contiene los siguientes métodos:

```
public interface Graph<V, E> {
  public int size();
  public boolean isEmpty();
```

```
  public int numVertices();
  public int numEdges();

  public int degree(Vertex<V> v) throws IllegalArgumentException;

  public Iterable<Vertex<V>> vertices();
  public Iterable<Edge<E>> edges();

  public V set(Vertex<V> p, V o) throws IllegalArgumentException;
  public E set(Edge<E> p, E o) throws IllegalArgumentException;

  public Vertex<V> insertVertex(V o);

  public V removeVertex(Vertex<V> v) throws IllegalArgumentException;
  public E removeEdge(Edge<E> e) throws IllegalArgumentException;
}
```

De dicho interfaz extienden los dos interfaces específicos para cada tipo de grafo: `UndirectedGraph<V,E>` para grafos no dirigidos y `DirectedGraph<V,E>` para grafos dirigidos.

Empezaremos viendo los grafos no dirigidos:

```
public interface UndirectedGraph<V,E> extends Graph<V,E> {
  public Iterable<Vertex<V>> endVertices(Edge<E> e) throws IllegalArgumentException;

  public Edge<E> insertUndirectedEdge(Vertex<V> u,
                                      Vertex<V> v, E o)
         throws IllegalArgumentException;

  public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
         throws IllegalArgumentException;

  public boolean areAdjacent(Vertex<V> u, Vertex<V> v)
         throws IllegalArgumentException;

  public Iterable<Edge<E>> edges(Vertex<V> v) throws IllegalArgumentException;
}
```

- `insertUndirectedEdge` permite crear aristas conectando los nodos `u` y `v` y asociar a la arista un objeto
- Dado un nodo y una arista, el método `opposite` devuelve el nodo que está "al otro lado de la arista"
- El método `areAdjacent` permite saber si dos nodos están conectados mediante alguna arista (son adyacentes)
- `edges` (sobrecargado) recibe un vértice y devuelve todas las aristas incidentes en él
- Todos los métodos que reciben un vértice o una arista como parámetro pueden lanzar `IllegalArgumentException`

La documentación completa del interfaz la podemos encontrar en:

`http://lml.ls.fi.upm.es/~entrega/aed/docs/aedlib/es/upm/aedlib/graph/UndirectedGraph.html`

A continuación veremos algunos ejemplos de uso de grafos no dirigidos:

▪ **Ejemplo 12.1** Ejemplo de recorrido en profundidad, es decir, avanzamos hasta que llegamos a un camino sin salida por un nodo no visitado. En este recorrido atravesamos todos los nodos alcanzables desde el vértice `n` y sin repetir nodos. Nótese que es necesario usar un conjunto para ir almacenando los nodoso ya atravesados para no pasar dos veces por los mismos nodos y entrar en una recursividad infinita (hasta StackOverflow):

```
  public static <V,E> void deepSearchNodes (UndirectedGraph<V, E> g,
  Vertex<V> n) {

    Set<Vertex<V>> visited = new HashTableMapSet<>();
    deepSearchNodes(g,n,visited);

  }

  public static <V,E> void deepSearchNodes (UndirectedGraph<V, E> g,
                                            Vertex<V> n,
```

```
                                          Set<Vertex<V>> visited ) {

  if  (visited.contains(n)) {
    return;
  }
  visited.add(n);

  for (Edge<E> e: g.edges(n)) {
    deepSearchNodes(g, g.opposite(n, e), visited);
  }
}
```

■

■ **Ejemplo 12.2** Ahora un recorrido en anchura, es decir, recorremos primero los nodos que están a 1 arista del nodo desde el que empezamos, luego los que están a 2 aristas... y así hasta recorrer todos los nodos.

```
public static <V,E> void breadthFirstSearch (UndirectedGraph<V, E> g,
                                             Vertex<V> n) {
  Set<Vertex<V>> visited = new HashTableMapSet<>();
  FIFO<Vertex<V>> pending = new FIFOList<>();
  pending.enqueue(n);

  while (!pending.isEmpty()) {
    Vertex<V> v = pending.dequeue();
    if (!visited.contains(v)) {
      visited.add(v);
      System.out.println("Visiting " + v);
      for (Edge<E> e: g.edges(v)) {
        pending.enqueue(g.opposite(v, e));
      }
    }
  }
}
```

■

■ **Ejemplo 12.3** Un método que devuelve **`true`** si se puede alcanzar el vértice `to` desde el vértice `from`:

```
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to) {
  Set<Vertex<V>> visited = new HashTableMapSet<>();
  return isReachable(g, from, to, visited);
}
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to,
                                         Set<Vertex<V>> visited ) {

  if (from == to) {
    return true;
  }
  if  (visited.contains(from)) {
    return false;
  }

  visited.add(from);
  boolean reachable = false;
  Iterator<Edge<E>> it = g.edges(from).iterator();
  while (it.hasNext() && !reachable) {
```

```
      reachable = isReachable(g, g.opposite(from, it.next()), to, visited);
    }
    return reachable;
  }
```

■

■ **Ejemplo 12.4** Dados dos nodos encontrar un camino de nodos conectados que permitan llegar desde `from` a `to`

```
  public static <V,E> PositionList<Vertex<V>> findOneSimplePath (UndirectedGraph<V, E> g,
                                                                 Vertex<V> from,
                                                                 Vertex<V> to) {

    Set<Vertex<V>> visited = new HashTableMapSet<>();
    PositionList<Vertex<V>> path = new NodePositionList<>();

    findOneSimplePath(g, from, to, visited, path);
    return path;
  }

  // It can be done by returning path
  public static <V,E> boolean findOneSimplePath (UndirectedGraph<V, E> g,
                                                 Vertex<V> from,
                                                 Vertex<V> to,
                                                 Set<Vertex<V>> visited,
                                                 PositionList<Vertex<V>> path) {


    if  (visited.contains(from)) {
      return false;
    }

    path.addLast(from);
    if (from == to) {
      return true;
    }

    visited.add(from);
    boolean found = false;
    Iterator<Edge<E>> it = g.edges(from).iterator();
    while (it.hasNext() && !found) {

      found = findOneSimplePath(g, g.opposite(from, it.next()), to, visited, path);
    }

    if (!found) {
      path.remove(path.last());
    }

    return found;
  }
```

■

■ **Ejemplo 12.5** Ahora un método que imprime todos los caminos (sólo los nodos) entre dos nodos. El coste computacional de este método es muy alto y únicamente funcionará con grafos de tamaño medio-bajo.

```
  public static <V,E> void findAllSimplePaths (UndirectedGraph<V, E> g,
                                               Vertex<V> from,
                                               Vertex<V> to) {
```

```
    Set<Vertex<V>> visited = new HashTableMapSet<>();
    PositionList<Vertex<V>> path = new NodePositionList<>();

    findAllSimplePaths(g, from, to, visited, path);

  }

  public static <V,E> void findAllSimplePaths (UndirectedGraph<V, E> g,
                                               Vertex<V> from,
                                               Vertex<V> to,
                                               Set<Vertex<V>> inPath,
                                               PositionList<Vertex<V>> path) {

    path.addLast(from);
    inPath.add(from);

    if (from == to) {
      System.out.println(path);
    }
    else {
      for (Edge<E> e: g.edges(from)) {
        if (!inPath.contains(g.opposite(from, e))) {
          findAllSimplePaths(g, g.opposite(from, e), to, inPath, path);
        }
      }
    }

    inPath.remove(from);
    path.remove(path.last()); // path.last() == from
  }
```

■

El interfaz `DirectedGraph<V,E>` representa grafos dirigidos y contiene los siguientes métodos:

```
public interface DirectedGraph<V,E> extends Graph<V,E> {

  public Vertex<V> startVertex(Edge<E> e) throws IllegalArgumentException;

  public Vertex<V> endVertex(Edge<E> e) throws IllegalArgumentException;

  public Edge<E> insertDirectedEdge(Vertex<V> from,
                            Vertex<V> to, E o) throws IllegalArgumentException;

  public Iterable<Edge<E>> outgoingEdges(Vertex<V> v)
       throws IllegalArgumentException;

  public Iterable<Edge<E>> incomingEdges(Vertex<V> v)
       throws IllegalArgumentException;

  public int inDegree(Vertex<V> v) throws IllegalArgumentException;

  public int outDegree(Vertex<V> v) throws IllegalArgumentException;
}
```

- `insertDirectedEdge` permite crear aristas dirigidas conectando los nodos `from` y `to` y asociar a la arista un objeto
- Dada una arista, los métodos `startVertex` y `endVertex` permiten obtener los nodos participantes en una arista
- Los métodos `outgoingEdges` y `incomingEdges` permiten obtener las aristas entrantes y salientes de un vértice
- `inDegree`, `outDegree` devuelven el número de entrantes o salientes de un vértice
- Todos los métodos que reciben un vértice o una arista como parámetro pueden lanzar `IllegalArgumentException`

`http://lml.ls.fi.upm.es/~entrega/aed/docs/aedlib/es/upm/aedlib/graph/`

DirectedGraph.html

A continuación vamos a ver algunos ejemplos de uso de grafos dirigidos.

IMPORTANTE: Nótese que son ejemplos similares a los grafos no dirigidos, pero las soluciones son un poco diferentes. Estas diferencias no se deben al interfaz utilizado, sino a formas diferentes de resolver los mismos problemas. Se podrían aplicar las soluciones anteriormente propuestas adaptándolas a grafos dirigidos, pero nos parece más ilustrativo proponer otras opciones para resolver problemas similares.

■ **Ejemplo 12.6** Método que devuelve un conjunto con los vértices visitables desde el vértice `startVertex`.

```
public static <V,E> Set<Vertex<V>> reachableVertices (DirectedGraph<V,E> g,
Vertex<V> startVertex) {

  Set<Vertex<V>> vertices = new HashTableMapSet<Vertex<V>>();
  visit(g, startVertex, vertices);
  return vertices;
}

private static <V,E> void visit(DirectedGraph<V,E> g,
Vertex<V> vertex,
Set<Vertex<V>> vertices) {
  if (vertices.contains(vertex)) {
    return;
  }
  vertices.add(vertex);
  for (Edge<E> edge : g.outgoingEdges(vertex)) {
    visit(g, g.endVertex(edge), vertices);

  }
}
```

■

■ **Ejemplo 12.7**

```
public static <V,E> PositionList<Transition<V,E>> findOnePath(DirectedGraph<V,E> g,
Vertex<V> from,
Vertex<V> to) {
  PositionList<Transition<V,E>> path =
  new NodePositionList<Transition<V,E>>();

  return findOnePath(g, from, to, path);
}

public static <V,E> PositionList<Transition<V,E>> findOnePath(DirectedGraph<V,E> g,
Vertex<V> from,
Vertex<V> to,
PositionList<Transition<V,E>> path) {

  if (from == to) return path;

  Iterator<Edge<E>> it = g.outgoingEdges(from).iterator();

  PositionList<Transition<V,E>> result = null;
  while (it.hasNext() && result == null) {
    Edge<E> e = it.next();
    Vertex<V> endVertex = g.endVertex(e);

    if (!endVertex.equals(g.startVertex(e))   &&
    !pathContainsVertex(path,endVertex)) {

      path.addLast(new Transition<>(from,e,endVertex));
```

```
        result = findOnePath(g,endVertex,to,path);

        if (result == null) {
          path.remove(path.last());
        }
      }
    }

    return result;
  }
```

Como en este caso queremos almacenar no sólo los nodos, sino que también queremos almacenar ls aristas por las que hemos pasado, utilizamos la clase `Transition` que contiene el nodo origen, el nodo destino y la arista que los conecta. Esto mismo se podría haber utilizado en la implementación de los ejemplos 12.4 y 12.5

```
public class Transition<V,E> {
  private Vertex<V> v1;
  private Edge<E> e;
  private Vertex<V> v2;

  public Transition(Vertex<V> v1, Edge<E> e, Vertex<V> v2) {
    this.v1 = v1;
    this.e = e;
    this.v2 = v2;
  }

  public Vertex<V> startVertex() {
    return v1;
  }

  public Edge<E> edge() {
    return e;
  }

  public Vertex<V> endVertex() {
    return v2;
  }

  public boolean equals(Object obj) {
    if (obj instanceof Transition<?,?>) {
      Transition<?,?> t = (Transition<?,?>) obj;
      return v1.equals(t.v1) && e.equals(t.e) && v2.equals(t.v2);
    } else return false;
  }

  public int hashCode() {
    return v1.hashCode() + e.hashCode() + v2.hashCode();
  }

  public String toString() {
    String edgeString = "";
    if (e.element() != null)
      edgeString = e.element().toString();

    return v1.element()+" -"+edgeString+"-> "+v2.element();
  }
}
```

También podemos usar la transición para el computo de todos los caminos. Algunas diferencias interesantes con respecto al código del ejemplo 12.5 es que se controlan los nodos ya visitados en el bucle y se añaden y se quitan elementos en el camino dentor del bucle en lugar de hacerlo al principio y al final del método. Asimismo, en ese método, en lugar de imprimir los caminos se están almacenando en un conjunto y devolviendo dicho

conjunt, que contiene listas de transiciones.

```
public static <V,E> Set<PositionList<Transition<V,E>>> findAllPaths(DirectedGraph<V,E> g,
                                                                    Vertex<V> from,
                                                                    Vertex<V> to) {
  PositionList<Transition<V,E>> path =
  new NodePositionList<Transition<V,E>>();

  Set<PositionList<Transition<V,E>>> allPaths =
  new HashTableMapSet<PositionList<Transition<V,E>>>();

  findAllPaths(g, from, to, path, allPaths);
  return allPaths;
}

public static <V,E> void findAllPaths(DirectedGraph<V,E> g,
                                      Vertex<V> from,
                                      Vertex<V> to,
                                      PositionList<Transition<V,E>> path,
                                      Set<PositionList<Transition<V,E>>>
                                      allPaths) {

  if (from == to) {
    allPaths.add(new NodePositionList<Transition<V,E>>(path));
    return;
  }

  for (Edge<E> e : g.outgoingEdges(from)) {
    Vertex<V> endVertex = g.endVertex(e);
    if (!endVertex.equals(g.startVertex(e)) && !pathContainsVertex(path,endVertex)) {

      path.addLast(new Transition<>(from,e,endVertex));
      findAllPaths(g,endVertex,to,path,allPaths);
      path.remove(path.last());

    }
  }
  return;
}


private static <V,E> boolean pathContainsVertex(PositionList<Transition<V,E>> path,
                                                Vertex<V> v) {
  for (Transition<V,E> transition : path) {
    if (v.equals(transition.startVertex()) ||
    v.equals(transition.endVertex())) {
      return true;   // Toma return dentro del bucle para terminar el curso :-)
    }
  }
  return false;
}
```

▪

Bibliografía

[AHU83] Alfred V. Aho, John E. Hopcroft y Jeffrey Ullman. *Data Structures and Algorithms*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0201000237.

[Blo08] Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. 2.ª edición. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0321356683, 9780321356680.

[Bud91] Timothy Budd. *An Introduction to Object-oriented Programming*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1991. ISBN: 0-201-54709-0.

[Cor+01] Thomas H. Cormen y col. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[GTG14] Michael T. Goodrich, Roberto Tamassia y Michael H. Goldwasser. *Data Structures and Algorithms in Java*. 6th. Wiley Publishing, 2014. ISBN: 1118771338, 9781118771334.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0136290493.