TEMA 1

Estructuras de Datos y Abstracción

Asignatura: *Estructuras de Datos*
Grado: *Ingeniería en Informática en Tecnologías de la Información*

Pedro Hernández Arauzo
phernandez@uniovi.es

Sobre la asignatura

La representación de la información es fundamental en la ciencia de la computación. El primer propósito de la mayoría de los programas de ordenador no es realizar cálculos sino recuperar y almacenar información, por lo general, tan rápido como sea posible. Por esta razón, el estudio de las estructuras de datos y de los algoritmos que permiten manipularlas es crítico en la ciencia de la computación.

En esta asignatura se muestra cómo representar **información estructurada**, a diseñar algoritmos eficientes u operaciones para manipular dicha información y a cómo diseñar las aplicaciones (programas más o menos complejos) que requieran utilizar la información.

Tabla de contenido

En este primer tema se presentan aspectos relevantes entorno a las estructuras de datos. Éstos posteriormente se utilizarán de forma asidua en el resto de temas de la asignatura, temas en los que se verán las formas más habituales de estructurar la información.

Los conceptos que se enseñan en este primer tema facilitan al programador la construcción de software reutilizable, comprensible, confiable y fácil de mantener. Estos conceptos se pueden aplicar a programas desarrollados bajo cualquier lenguaje de programación, con independencia de que éste proporcione suficiente soporte o no a los mismos. La única diferencia es que, en el primer caso el propio lenguaje de programación dispondrá de herramientas que faciliten la correcta aplicación de dichos conceptos, mientras que en el segundo la tarea será más engorrosa y toda la responsabilidad de su correcta aplicación recaerá directamente en el programador.

1 Abstracción

En el desarrollo de nuestra comprensión de fenómenos complejos la herramienta más potente disponible por el intelecto humano es la abstracción. La abstracción resulta de un reconocimiento de similitudes entre objetos, situaciones o procesos del mundo real y la decisión de concentrarse en estas similitudes e ignorar inicialmente las diferencias (O. J. DHAL, 1972).

El diccionario de la Real Academia Española (RAE) define el término **abstracción** como **acción y efecto de abstraer o abstraerse**. A su vez define abstraer como separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia o noción. Y utilizado como verbo intransitivo: *pensar es olvidar diferencias, es generalizar, abstraer*.

A modo de ejemplo, para definir una mesa es irrelevante su forma (cuadrada, redonda, ovalada, etc.), su color e incluso el material con el que está construida. Una mesa es un mueble compuesto de un tablero horizontal liso y sostenido a la altura conveniente, generalmente por una o varias patas, para diferentes usos, como escribir, comer, etc. (definición de la RAE).

La relevancia de los detalles dependerá del contexto, en un nivel inferior pueden ser relevantes otras características. Por tanto, se trata de un proceso jerárquico. Siguiendo con el ejemplo de la mesa, la mayor parte de las características de ésta que se han considerado irrelevantes en su definición, o todas ellas, si son importantes si nuestra intención es comprar una mesa.

Así, desde hace mucho tiempo hasta hoy, la abstracción resulta un medio y un instrumento para pensar y explicar, siendo indispensable en el razonamiento, el conocimiento y la interrelación entre estos.

> La **abstracción** permite simplificar el análisis y resolución de un problema separando las características que son relevantes de aquellas que no lo son.
>
> Forma parte de la capacidad intelectual y se considera como un elemento clave para la investigación científica, la deducción y la lógica.

El concepto de abstracción aparece en distintas disciplinas o campos del saber: filosofía, psicología, **informática**, investigación científica, el arte, etc. En particular, el uso de la abstracción en informática permite poner el énfasis en el *¿qué hace?* en lugar de en *¿cómo lo hace?* Para ello se introduce el principio de **ocultación de información**, entendido éste como la omisión intencionada de detalles de implementación (*¿cómo lo hace?*) tras una interfaz simple *(¿qué hace?*). De esta forma es posible proteger el software de cambios que puedan producirse en la implementación de ciertas partes de código o, de forma equivalente, permite continuar el

desarrollo de aplicaciones software con independencia de otras partes de código que todavía puedan estar por implementar o estar implementándose.

En el campo de la programación los lenguajes de alto nivel han supuesto una simplificación importante para el programador: *los programas se realizan mediante construcciones del lenguaje de alto nivel en lugar de utilizar las secuencias de código máquina en que éstas se traducirán de forma automátic*a. Pero, a pesar del avance que han supuesto los lenguajes de alto nivel, la abstracción que éstos aportan resulta insuficiente para diseñar programas cada vez más complejos. Por esta razón los paradigmas y lenguajes de programación establecen un uso creciente de la abstracción facilitando herramientas para que el programador construya sus propias abstracciones reutilizables, resultando imprescindible comprender y utilizar con agilidad los *mecanismos y tipos de abstracción* disponibles.

Históricamente el primer tipo de abstracción que aparece es la *abstracción funcional* (o *procedimental*) que permite separar el propósito de una función de su implementación. Para poder utilizar ésta en un programa (*invocar o llamar a la función*), lo importante es saber qué hace la función y resulta irrelevante conocer el algoritmo (o el código) de su definición; es decir, el cómo lo hace.

Las funciones son un tipo de abstracción sobradamente conocido y ampliamente utilizado incluso desde los primeros pasos en el aprendizaje de la programación, por lo que nos referiremos inicialmente a éste para identificar los mecanismos o métodos de abstracción existentes, los cuales también se utilizarán en otros tipos de abstracción.

1.1 Métodos de abstracción

Separando la definición de una función de su invocación, un lenguaje de programación permite dos importantes métodos o mecanismos de abstracción: la **abstracción por parametrización** y la **abstracción por especificación**.

1.1.1 Abstracción por parametrización

La **abstracción por parametrización** nos permite representar un conjunto potencialmente infinito de cálculos distintos mediante un simple programa de texto que es una abstracción de todos ellos. Así, en el caso de la función de la Figura 1 su invocación únicamente permite obtener el mayor valor de cierto vector de enteros `v[0..n-1]`, mientras que en el caso de la función de la Figura 2 su invocación permite obtener el mayor de los valores de cualquier vector de enteros que se utilice como argumento.

```
int max() {
   int result = v[0];
   for (int e: v) {
      if (result < e) {
         result = e;
      }
   }
   return result;
}
```

Figura 1. Función sin parámetros

```
int max(int[] v) {
   int result = v[0];
   for (int e: v) {
      if (result < e) {
         result = e;
      }
   }
   return result;
}
```

Figura 2. Función con parámetros

Este mecanismo de abstracción es posible generalizarlo aún más, generalización que es indispensable para su aplicación en otro tipo de abstracciones. Para ello es suficiente con darse cuenta que la segunda versión de la función (con parámetros) puede ser igualmente válida para cualquier vector con independencia del tipo de dato de sus elementos. Por ejemplo, el código

de una función para obtener la mayor componente de un vector de caracteres sería idéntico, salvo por las declaraciones de tipo, sólo habría que sustituir las declaraciones de tipo `int` por `char` (ver Figura 3).

```
char max(char[] v) {
   char result = v[0];
   for (char e: v) {
      if (result < e) {
         result = e;
      }
   }
   return result;
}
```

Figura 3. Máximo de un vector de caracteres

```
T max(T[] v) {
   T result = v[0];
   for (T e: v) {
      if (result < e) {
         result = e;
      }
   }
   return result;
}
```

Figura 4. Función con parámetros de tipos de datos

Por tanto, si también se utiliza un parámetro como tipo de dato de los elementos del vector `v`, es posible dar una versión más genérica de la función `max` (más reutilizable). Ahora, en lugar de decir que la función retorna un entero (`int`) o un carácter (`char`) y que el parámetro de la función es un vector de enteros o de caracteres, se utilizará en su lugar un *parámetro de tipo de dato*, `T` (ver Figura 4).

En las abstracciones funcionales los parámetros de tipo se resuelven en la propia invocación de la función, dependiendo del tipo de dato concreto de sus argumentos. En el caso de la función `max`, según el tipo de dato de los elementos del vector que se pase como argumento en la llamada a ésta (ver Figura 5).

```
int[] a = {10, 2 , -3, 24, 15};
char[] b = {'f', 'a', 'm', 'x', 'c' };
. . .
m1 = max(a);  // T se sustituye por int
m2 = max(b);  // T se sustituye por char
```

Figura 5. Llamadas a la función `max`

Aunque en otros lenguajes de programación el parámetro de tipo `T` podría ser sustituido por un tipo primitivo del lenguaje (por ejemplo, **`int`** o **`char`**), en Java sólo se puede sustituir por tipos de objetos (`Integer` o `Character`).

El uso de parámetros de tipos de datos en abstracciones funcionales impone ciertas restricciones, o condicionantes, a los tipos de datos reales que sustituirán a los parámetros de tipo en el momento de la llamada o bien a la propia definición de la función. Así, en el caso de la función `max` sólo es posible invocar a ésta con vectores cuyas componentes sean de un tipo que soporte la operación `<` o bien, en su defecto, que ésta se pueda definir (o establecer) de alguna forma para los valores de dicho tipo de dato, por ejemplo, proporcionando una función de comparación o parámetro equivalente a ésta (ver Figura 6).

Más adelante se incidirá en este tipo de funciones que tienen algún parámetro que a su vez es una función o que retornan una función y, en particular, cómo se definen e invocan estas funciones en el lenguaje de programación Java.

```
T max(T[] v, boolean mayor(T e1, T e2)) {
   T result = v[0];
   for (T e: v) {
      if (comp(result, e)) {
         result = e;
      }
   }
   return result;
}
```

Figura 6. Función con comparador de valores de tipo `T`

1.1.2 Abstracción por especificación

La abstracción por especificación nos permite abstraernos de los cálculos descritos por el cuerpo de una función para centrarnos en el fin para el cual fue diseñada. La especificación de una abstracción funcional consta de una parte *sintáctica* y otra *semántica*. La parte sintáctica incluye el nombre de la función, los parámetros y sus tipos, así como del tipo de retorno de ésta. La parte semántica establece el comportamiento de la función y puede darse formalmente mediante asertos (predicados lógicos ciertos) de entrada y salida, o bien, mediante el lenguaje natural.

La ventaja de utilizar una especificación formal basada en la lógica de predicados es que es breve y precisa. Este tipo de especificación se basa en considerar un algoritmo como una caja negra en la que sólo es posible observar sus parámetros de entrada y salida. La ejecución del algoritmo comienza con un *estado inicial* válido dado por los parámetros de entrada y, tras un cierto tiempo, termina en un *estado final* en el que los parámetros de salida contienen los resultados esperados.

Si S representa la función a especificar (parte sintáctica de la especificación), P es un predicado que tiene como variables libres los parámetros de entrada de S (*precondición*) y Q es un predicado que tiene como variables libres los parámetros de entrada y salida de S (*postcondición*), la notación

```
{P} S {Q}
```

representa la **especificación formal de *S*** que ha de leerse de la forma siguiente: *Si S comienza su ejecución en un estado que satisface la precondición P, S termina y lo hace en un estado que satisface la postcondición Q.*

Así, por ejemplo, la especificación formal de la función `max` que como parámetro de entrada tiene un vector `v[0..n-1]` de elementos de un tipo `T` cualquiera (parámetro de tipo de dato) y retorna el mayor de todos ellos, sería la indicada en la Figura 7.

$\{(n > 0) \land (\forall i: 0 \leq i < n: v[i] = V_i)\}$
```
T max(T[] v);
```
$\{\exists k: 0 \leq k < n: (\max(v) = v[k]) \land (\forall i: 0 \leq i < n: v[i] \leq v[k])\}$

Figura 7. Especificación formal de la función `max`

La alternativa a la especificación formal es utilizar el lenguaje natural, pero el problema de éste es que suele resultar ambiguo con cierta frecuencia y, por tanto, este tipo de especificación no es lo suficientemente precisa como para responder a cualquier pregunta sobre un algoritmo sin recurrir a su implementación, que es precisamente lo que se quiere evitar.

La *especificación mediante **cláusulas*** permite especificar una función utilizando el lenguaje natural de forma suficientemente precisa, ya que cada cláusula establece un único aspecto relevante de la especificación mediante una frase breve.

Las cláusulas que se pueden utilizar en la especificación son las siguientes:

1. ***Necesita***: Indica cómo deben ser los argumentos en la invocación de la función.
2. ***Produce***: Indica qué es lo que retorna una función al invocarla.
3. ***Modifica***: Indica que en la invocación de la función alguna de sus entradas (argumentos) se modifica y cómo es dicha modificación.
4. ***Efecto***: Indica un efecto lateral en la invocación a una función.

5. ***Excepción***: Indica que al invocar a la función con unos valores concretos de las entradas se produce una modificación, un resultado, o un efecto lateral excepcional.
6. ***Error***: Indica que al invocar a la función con unos valores concretos de las entradas se produce un error.

Las cláusulas se suelen proporcionar también entre llaves, como los asertos, después de la parte sintáctica de la especificación, tal y como se puede ver en la Figura 8 para la función `max`. A este tipo de especificación se la denomina *cuasi-formal*.

```
T max(T[] v);
{ Necesita: Un vector v de elementos de tipo T de dimensión mayor
            que cero. Los valores del tipo T deben ser comparables.
   Produce: El mayor elemento del vector v dado.                    }
```

Figura 8. Especificación de la función `max` *mediante cláusulas*

Dado que las cláusulas que se utilizan en la especificación pueden ser las indicadas u otras similares y que un aspecto importante en el desarrollo de software es la documentación, posiblemente sea preferible especificar las funciones utilizando directamente una herramienta de documentación de programas como: *doxygen*, *doc++* o *Javadoc*. Estas herramientas utilizan una sintaxis parecida, pueden documentar (o especificar) una función mediante sus propias cláusulas (o directivas), las cuales se incluyen en comentarios del código de los programas, y dan soporte para varios lenguajes de programación. De esta forma se tienen una especificación equivalente a la especificación cuasi-formal, con la ventaja adicional de poder generar la documentación de la función (su especificación) en varios formatos (HTML, PDF, TeX, Word, etc.).

```
/**
 * Retorna el mayor elemento del vector especificado. Los valores
 * del tipo T de los elementos del vector deben ser comparables.
 * @param v el vector proporcionado
 * @param <T> parámetro de tipo de los elementos del vector
 * @return el mayor elemento del vector
 */
public static <T> T max(T[] v);
```

Figura 9. Especificación de la función `max` *mediante JavaDoc*

En este documento los ejemplos se presentan en Java y éste es el lenguaje de programación que se utiliza en las prácticas de la asignatura, por eso para realizar la especificación de las funciones se utilizará la herramienta de documentación *Javadoc* (ver Figura 9). Además, ésta es una herramienta ya conocida por el estudiante al haberse utilizado en las asignaturas de programación del primer curso de la titulación. La documentación generada para la función `max` en formato HTML tiene un aspecto similar al mostrado en la Figura 10.

```
public static <T> T max(T[] v)
El mayor elemento del vector especificado. Los valores del tipo T deben
ser comparables.
Type Parameters:
    T - parámetro de tipo de los elementos del vector
Parameters:
    v - el vector
Returns:
    el mayor elemento del vector v
```

Figura 10. Documentación HTML de la función `max`

1.2 Tipos de abstracciones

La abstracción mediante parámetros y la abstracción por especificación son herramientas útiles y potentes para la construcción de programas y pueden utilizarse en tres tipos de abstracciones: *abstracción funcional, abstracción de datos* y *abstracción de iteración*. Existe, además, un cuarto tipo de abstracción: la *jerarquía de tipos*.

Los tipos de abstracción son los siguientes:

- **Abstracción funcional**: permite extender un lenguaje de programación añadiendo a éste nuevas operaciones.
- **Abstracción de datos**: permite extender un lenguaje de programación añadiendo a éste nuevos tipos de datos. Consta de un conjunto de valores (u objetos) y un conjunto de operaciones (o métodos) que caracterizan su comportamiento.
- **Abstracción de iteración**: permite iterar sobre los elementos de una colección ignorando el detalle de cómo se obtienen éstos.
- **Jerarquía de tipos**: permite abstraerse de tipos de datos individuales a conjuntos de tipos relacionados. Todos los tipos de un mismo conjunto tienen operaciones en común definidas en un *supertipo* que será un antecesor de todos ellos, los denominados *subtipos*.

2 Abstracción de datos

2.1 Conceptos previos

En los lenguajes de programación un **tipo de dato** es un *atributo* de una parte de los datos que indica a los ordenadores (o al programador) ciertas características sobre la información (o datos) que se van a procesar. Esto incluye el imponer restricciones a los datos como qué valores pueden tomar y qué operaciones se pueden realizar. Los tipos de datos comunes que se pueden encontrar en cualquier lenguaje de programación, son: enteros, caracteres, números de coma flotante (decimales), cadenas alfanuméricas, fechas, horas, etc.

Un tipo de dato también se puede ver como una limitación impuesta en la interpretación de los datos en un *sistema de tipos*, describiendo la representación, interpretación y la estructura de los valores u objetos almacenados en la memoria del ordenador.

Un *enlace* es una asociación, por ejemplo, entre un atributo y una entidad, o entre una operación y un símbolo.

> Un **tipo de dato** define un conjunto de valores y un conjunto de operaciones que permiten su manipulación.
>
> Los *tipos datos* se asocian a las variables de un programa y al momento en que se realiza esta asociación (o enlace) se le denomina **tiempo de enlace**, éste puede ser *estático* o *dinámico*. Un enlace es **estático** si ocurre antes del tiemplo de ejecución de un programa y no cambia durante la ejecución de éste. Mientras que un enlace es **dinámico** si se produce en tiempo de ejecución o puede cambiar durante la ejecución del programa.

Esto es válido tanto para lenguajes de programación imperativos, como para lenguajes de programación declarativos. Si bien, en los lenguajes puramente funcionales y lógicos (declarativos) las variables están ligadas a expresiones y mantienen su valor durante todo su *tiempo de vida* para garantizar la *transparencia referencial* (ver apartado 7).

El *tipo estático* (más eficiente) necesita ser especificado mediante una declaración explícita del tipo de las variables o implícita (por la primera aparición de la variable en el programa). Además, el tipo se puede *inferir*, obtener a partir del contexto de una referencia, lo que permite al programador reducir el número de anotaciones de tipo en los programas.

Si el *tipo es dinámico* (más flexible) el tipo de dato de las variables no está predefinido y el sistema de tipificación gestionará éstas en función de la naturaleza de cada valor asociado.

2.2 Definición de tipos de datos

En el desarrollo de software es necesario organizar la información que se va a procesar (los datos) y, por lo general, los tipos de datos comunes disponibles en cualquier lenguaje de programación resultan insuficiente para ello. Es necesario, por tanto, combinar estos tipos de datos comunes (y quizá otros tipos de datos definidos por el programador) de forma estructurada, y más o menos compleja, para definir nuevos tipos de datos. Surge así la construcción de *tipos de datos abstractos* que capturan el modo en que pensamos sobre la información.

> Un **tipo de dato abstracto** (TDA) define un conjunto de valores y un conjunto de operaciones que proporcionan la única manera de manipularlos.

Para poder utilizar el tipo de dato en las aplicaciones que lo requieran (clientes del tipo de dato) es necesario proporcionar la *especificación de TDA*; es decir, debe darse respuesta a las siguientes preguntas:

1. ¿Qué valores define el TDA?
2. ¿Cuáles son las operaciones del TDA?
3. ¿Qué hacen las operaciones o cómo se comporta los valores del tipo al ser invocadas?

Las respuestas a estas preguntas constituyen también una especificación para una implementación correcta del tipo de dato, pero debe observarse que el TDA no indica cómo debe almacenarse la información, ni cómo deben implementarse las operaciones. Ambos detalles son irrelevantes para usar el TDA en los clientes y sólo son importantes a un nivel inferior, cuando se implementa el tipo de dato.

Los **tipos de datos abstractos** proporcionan una **interfaz** (operaciones disponibles) y **semántica** (qué hacen las operaciones) bien definida, lo que permite diseñar las aplicaciones clientes en términos de éstos independientemente de su implementación.

Finalmente, debe proporcionarse una implementación para el TDA, para ello es necesario considerar cómo se representará la información en la memoria del ordenador. El modo en que la información se organice estructuralmente en la memoria determinará la eficiencia de las distintas operaciones que permiten manipular la información.

La organización estructural de la información de un TDA en la memoria del ordenador se conoce como **representación** o **estructura de datos**, y a la forma en que se implementan las operaciones en términos de ésta como **algoritmos**.

Los clientes del tipo de dato abstracto accederán a la información (datos) mediante las operaciones públicas de la interfaz ignorando los detalles de la implementación del TDA que es susceptible de cambios.

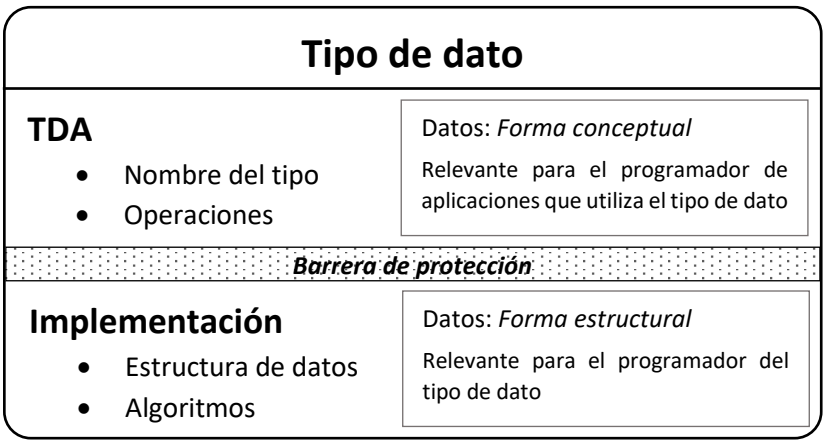

Figura 11. Definición de tipos de datos

Para facilitar la creación de nuevos tipos de datos y controlar su correcto uso en los programas de aplicación, los lenguajes de programación que soportan TDAs disponen de mecanismos para *ocultar la información* (restringir el acceso a los datos por medio de las operaciones) y *encapsular la información* (datos y código de las operaciones). Por tanto, separan el TDA de su implementación y sus mecanismos de protección impiden que los clientes del tipo puedan acceder a los detalles de implementación, tal y como se representa en la Figura 11.

2.2.1 Clasificación de los tipos de datos

Los tipos de datos se pueden clasificar según su estructura de datos, o bien, según la clase de operaciones básicas que soporta. La clasificación dada a continuación es válida tanto para los tipos de datos comunes que se pueden encontrar en los lenguajes de programación, como para los tipos de datos abstractos.

Atendiendo a su estructura, los tipos de datos se pueden clasificar en:

- *Tipos de datos simples*: cambian su valor, pero no su estructura. El espacio de almacenamiento se mantiene constante, por ejemplo, booleanos, enteros, caracteres, reales, subrangos, etc.
- *Tipos de datos contenedores*: cambian su valor y su estructura (colecciones de elementos en número variable), por ejemplo: listas, pilas, colas, árboles, conjuntos, grafos, etc.

Atendiendo al tipo de operaciones soportadas, los tipos de datos se pueden clasificar como:

- *Tipos de datos no modificables*: sus casos no se pueden modificar (pueden crearse y destruirse, pero no existen operaciones de modificación). La representación de estos tipos puede ser modificable o no.
- *Tipos de datos modificables*: sus casos se pueden modificar (existen operaciones de modificación). La representación de estos tipos debe ser modificable.

2.2.2 Especificación de tipos de datos abstractos

Al igual que la especificación de una función, la especificación de un tipo de dato abstracto puede darse formalmente, mediante un conjunto de axiomas en forma de ecuaciones, o bien, puede darse utilizando el lenguaje natural.

Ambas formas de especificación presentan las mismas ventajas e inconvenientes que se vieron para la especificación de funciones. La especificación formal es más precisa y concisa que la especificación mediante el lenguaje natural y ésta última puede resultar ambigua.

La especificación de un TDA utilizando el lenguaje natural está constituida, principalmente, por la especificación cuasi-formal de cada una de sus operaciones (función o procedimiento), especificación que ya se trató en el apartado 1.1.2. En este caso, tanto la interfaz como la semántica del TDA se corresponden, respectivamente, con el conjunto de las partes sintácticas y de las partes semánticas de la especificación de sus operaciones.

Sin embargo, la especificación formal de un TDA no se basa en la especificación formal de cada una de sus operaciones, en su lugar, se utiliza una especificación algebraica. En ésta, la interfaz del TDA se establece mediante las *signaturas*[1] (o perfiles) de las operaciones y la semántica del TDA mediante un conjunto de axiomas en forma de ecuaciones. Estas ecuaciones son igualdades que indican como operan cada una de las operaciones del TDA con respecto a otras operaciones del mismo tipo y son válidas para todos los valores de éste.

2.2.2.1 Especificación cuasi-formal

La especificación cuasi-formal consta de una cabecera con el nombre del tipo de dato y de tres secciones principales: *sección de declaración de tipos*, *sección de características* y *sección de operaciones*.

[1] La signatura o firma de un método o una función define su entrada y su salida. Incluye el nombre de la función o método, el tipo retornado y el tipo de sus parámetros.

- La *sección de declaración de tipos* define los diferentes tipos de datos que va a utilizar el TDA, éstos pueden ser tipos predefinidos de un lenguaje de programación u otros TDAs.
- La *sección de características* define los valores del tipo y si éstos son modificables o no. La definición puede darse mediante algún modelo para los valores, es decir, describiendo estos mediante otros que previsiblemente son comprensibles para el lector de la especificación.
- La *sección de operaciones* contiene una especificación cuasi-formal para cada operación.

Por ejemplo, se podría dar la siguiente especificación cuasi-formal para el tipo de dato abstracto `Racional`:

```
Tipo Racional
    Declaración de tipos Racional, Entero
    Características
        El valor abstracto de una instancia del tipo será un
        número racional. Las instancias no son modificables.
    Operaciones
        Racional crearRacional(int n, int d);
        { Necesita:  dos valores enteros, n y d.
          Produce:   el número racional n/d.
          Excepción: si d es cero.              }
        int numerador(Racional r);
        { Necesita:  un racional r.
          Produce:   el numerador de r. }
        int denominador(Racional r);
        { Necesita:  un racional r.
          Produce:   el denominador de r. }
        void simplificar(Racional r);
        { Necesita:  un racional r.
          Modifica:  el racional r a otro equivalente irreducible. }
        Racional sumar(Racional r1, Racional r2);
        { Necesita:  dos racionales r1 y r2.
          Produce:   la suma de los racionales dados. }
fin Racional
```

Figura 12. Especificación cuasi-formal del TDA Racional

2.2.2.2 Especificación algebraica

La especificación algebraica consta de una cabecera con el nombre del tipo de dato y tres secciones principales*: sección de declaración de tipos*, *sección de declaración de sintaxis* y *sección de axiomas*.

La principal ventaja de esta especificación es que es completamente independiente de cualquier representación y no impone característica alguna en la implementación del tipo de dato. No ocurre lo mismo con una especificación basada en el lenguaje natural, ya que la parte sintáctica de la especificación de las operaciones fija de antemano si el tipo de dato será modificable o no. Así, en la especificación del apartado previo la operación `Simplificar` modifica el valor de un racional y, en consecuencia, toda implementación dará lugar a un tipo de dato modificable.

La sintaxis de la especificación algebraica se asemeja a la de cualquier lenguaje de programación, pero sólo se permiten las siguientes características: variables libres, expresiones `si-entonces-si_no`, expresiones booleanas y recursión.

Además, se restringe el uso de abstracciones funcionales a funciones sin efectos laterales. Por tanto, en este formalismo no se permite realizar asignaciones a variables, ni utilizar sentencias iterativas. Estas restricciones, que son las mismas que se dan en un lenguaje de programación puramente funcional, permiten que la semántica de las operaciones se puede axiomatizar fácilmente.

Así, por ejemplo, la especificación algebraica del TDA `Racional` sería la siguiente:

```
Tipo Racional
   declaración de tipos Racional, Entero
   sintaxis
      crearRacional(Entero,Entero) → Racional ∪ {Indefinido}
      sumar(Racional,Racional) → Racional
      simplificar(Racional) → Racional
      numerador(Racional) → Entero
      denominador(Racional) → Entero
   para todo r1,r2 ∈ Racional, n,d ∈ Entero
      crearRacional(n,0) = Indefinido
      numerador(crearRacional(n,d)) = n
      denominador(crearRacional(n,d)) = d
      simplificar(crearRacional(n,d)) = crearRacional(n/m.c.d(n,d),
                                                      d/m.c.d(n,d))
      sumar(r1, r2) = crearRacional(numerador(r1) * denominador(r2) +
                                    numerador(r2) + denominador(r1),
                                    denominador(r1) * denominador(r2))
   fin para
fin Racional
```

Figura 13. Especificación algebraica del TDA Racional

2.2.3 Operaciones de una abstracción de datos

2.2.3.1 Clasificación de las operaciones

Las operaciones de un tipo de dato abstracto (TDA) se pueden clasificar en **operaciones básicas** y **operaciones no básicas**. Las primeras son todas aquellas operaciones que sólo se pueden implementar una vez elegida la *representación* (o *estructura datos*), mientras que las segundas son operaciones que pueden y deben implementarse en base a otras operaciones del tipo de dato, evitando así el acceso directo a la representación. Por ejemplo, para la especificación del TDA `Racional` de la Figura 12 las operaciones básicas son: `crearRacional`, `numerador`, `denominador` y `simplificar`, el resto son operaciones no básicas.

A su vez, las *operaciones básicas* de un tipo de dato abstracto se pueden clasificar según cómo se comportan en: *constructoras*, *observadoras* o *modificadoras*. Además, si el lenguaje de programación utilizado no dispone de un sistema de recuperación automática de memoria, también habría operaciones básicas *destructoras* (por ejemplo, en C++).

Las *operaciones constructoras* crean un nuevo valor (u objeto) del TDA. Éstas proporcionan a los programas cliente del TDA la capacidad de generar de forma dinámica valores (u objetos) del tipo de dato. Por ejemplo, la operación `crearRacional` del TDA `Racional`.

Las *operaciones observadoras* retornan un valor (u objeto) que no pertenece al TDA. Éstas permiten a los programas clientes del TDA consultar propiedades (o determinar características)

de los valores (u objetos) del tipo de dato. Por ejemplo, la operación `numerador` del TDA `Racional`.

Las *operaciones modificadoras* generan un nuevo valor (o modifican un objeto) del TDA partiendo de un valor (u objeto) del tipo de dato y, posiblemente, de valores (u objetos) de otro tipo. Por ejemplo, la operación `simplificar` del TDA `Racional`.

Las *operaciones destructoras*, si existen, permiten recuperar la memoria de las estructuras de datos de los valores (u objetos) del TDA creadas mediante las operaciones constructoras.

2.2.4 Implementación de tipos de datos abstractos

Para implementar un tipo de dato abstracto (TDA) en primer lugar debe elegirse la representación o estructura de datos y entonces implementar los algoritmos de las operaciones del tipo en términos de ésta. En el caso del `TDA Racional`, una posibilidad sería utilizar como representación dos enteros, uno para el numerador (`num`) y otro para el denominador (`den`). En la Figura 14 se muestra la representación para el `TDA Racional` y se adjunta el código de un par de operaciones de éste.

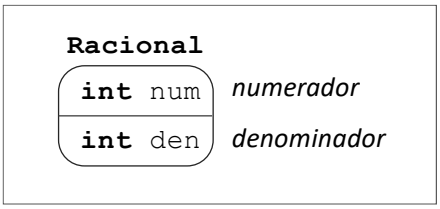

Figura 14. Representación para el TDA Racional

```
Racional crearRacional(int n, int d) {
   if (d == 0) {
      throw new RuntimeException();
   }
   return new Racional(n, d);
}

int numerador(Racional r) {
   return r.num;
}
```

Aunque seguramente la estructura de datos de la Figura 14 sea la más habitual para representar un racional, no es la única opción. Una forma alternativa de representar un racional, sería utilizar un almacenamiento común para todos los racionales que se creen y gestionar este almacenamiento. Para ello se utilizará un vector de un tamaño dado y en cada componente del mismo se almacenará la información de un racional, un registro de dos campos: `num` para el numerador y `den` para el denominador (ver Figura 15).

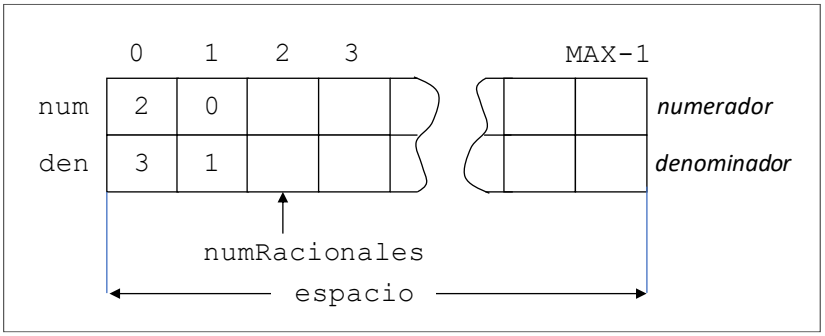

Figura 15. Representación alternativa para el TDA Racional

A continuación, se muestra parte de una clase que implementa el TDA `Racional` para la estructura de datos de la Figura 15. Todos los campos que se refieren al espacio común de almacenamiento son *campos de clase* (estáticos), el único *campo de instancia* (`rep`) es la forma en la que aquí se representa un racional: *por el índice de la componente del vector que contiene*

su información, un registro que contiene dos enteros. Este registro es una instancia de una clase interna y protegida.

Con respecto a los algoritmos de las operaciones, éstos se han implementado siguiendo estrictamente la especificación funcional proporcionada, razón por la cual todas las operaciones se declaran como métodos de clase y no como métodos de instancia. Además, los constructores están protegidos para que los clientes del tipo sólo puedan crear instancias mediante la operación `crearRacional`.

```
public class Racional {
    // espacio común de almacenamiento
    private static final int MAX = 100;
    private static Par[] espacio = new Par[MAX];
    private static int numRacionales = 0;
    // representación de un racional
    private int rep;
    private static final class Par { // registro de dos enteros
        int num; // numerador
        int den; // denominador
        Par (int n, int d) {
            num = d > 0 ? n : -n;  // signo en el numerador
            den = Math.abs(d);
        }
    }
    protected Racional() {}
    protected Racional (int n, int d) {
        this.rep = numRacionales++;
        espacio[this.rep] = new Par(n, d);
    }
    public static Racional crearRacional(int n, int d) {
        if (d == 0) {
            throw new RuntimeException("División por cero");
        }
        return new Racional(n, d);
    }
    public static int numerador(Racional r) {
        return espacio[r.rep].num;
    }
    ...
}
```

Figura 16. Implementación del TDA Racional

Obsérvese que en la implementación de un *tipo de dato abstracto* hay dos tipos de datos involucrados:

1. El tipo de dato abstracto, con una *interfaz* y *semántica* definidas por su especificación. Para el ejemplo, el tipo `Racional`.
2. El tipo de la representación, que denotaremos como *tipo* **`rep`**, utilizado para representar los valores del tipo de dato abstracto y sobre el cual se proporcionan los algoritmos de las operaciones del TDA. Para el ejemplo, el tipo `int`.

Ambos tipos están relacionados mediante una función parcial denominada *función de abstracción*. Esta función explica cómo ha de interpretarse la representación.

Para la implementación del TDA `Racional` dado en la Figura 16, la *función de abstracción* que relaciona el tipo de la representación con el tipo abstracto es la siguiente:

$$\begin{aligned} \boldsymbol{Abs}: rep &\rightarrow Racional \\ r &\rightarrow espacio[r].num/espacio[r].den \end{aligned}$$

y el *invariante de la representación*: $(espacio[r].den \neq 0) \wedge (0 \leq r < numRacionales < MAX)$

> Se denomina **función de abstracción** a la función parcial que relaciona los valores de *tipo* **`rep`** con los correspondientes valores del tipo de dato abstracto.
>
> La condición que identifica los valores de *tipo* **`rep`** que representan valores válidos del tipo de dato abstracto, se denomina **invariante de la representación**. Para estos valores de *tipo* **`rep`**, la *función de abstracción* está definida.

Por último, debe tenerse en cuenta que, a la hora de elegir la representación de la implementación de un TDA una estructura de datos puede soportar algoritmos rápidos para ciertas operaciones y lentos para otras, mientras que con otra estructura puede ocurrir justo lo contrario.

A modo de ejemplo, la decisión de mantener la información ordenada en la memoria puede hacer que una operación de búsqueda sea más eficiente y por el contrario hacer la operación de inserción más lenta, mientras que una información desordenada puede dar lugar a una operación de búsqueda más lenta y en cambio permitir una operación de inserción más rápida. Se evidencia, por tanto, que existen distintas implementaciones para un TDA (se puede representar con varias estructuras de datos) y unas serán mejores que otras para determinadas aplicaciones, raramente existirá una estructura de datos que sea mejor que otra para cualquier uso. En general, deberá elegirse la estructura de datos que permita realizar de forma eficiente las operaciones que la aplicación invocará con mayor frecuencia.

2.2.4.1 Implementación de operaciones

En el apartado 2.2.3.1 las operaciones de un tipo de dato abstracto se clasificaron en básicas y no básicas. Las primeras son las que sólo se pueden implementar una vez elegida la representación (necesitan acceder a ésta), mientras que las no básicas se pueden y deben implementar en base a otras operaciones del tipo. Así, el primer aspecto importante a destacar en la implementación de las operaciones de un TDA precisamente es éste.

> En la implementación de las operaciones de un TDA:
>
> 1. Únicamente las *operaciones básicas* deben acceder a la representación (o estructura de datos).
> 2. Las *operaciones no básicas* no deben acceder a la representación. Su implementación debe basarse, exclusivamente, en otras operaciones (básicas o no básicas) del propio tipo de dato y, si fuera necesario, en operaciones de otros tipos relacionados.

También es importante la decisión que se adopte sobre la representación de los nuevos valores del tipo de dato cuando éstos se modifiquen o se creen a partir de otros valores del mismo. Por ejemplo, en una operación de inserción de datos o en una operación de copia del valor, respectivamente.

Supóngase, por ejemplo, que el TDA `Racional` incluye la operación de copia, por sobrecarga de la operación `crearRacional`, que se especifica a continuación:

```
Racional crearRacional(Racional r);
{ Necesita: un racional r.
  Produce:  un número racional con el mismo
            numerador y denominador que r.  }
```

Figura 17. Especificación de la operación de copia de un racional

En este caso, habría que incorporar el código de esta operación a la implementación del TDA `Racional` dada en la Figura 16. Codificación que podría hacerse de dos formas:

```
Racional crearRacional(Racional r) {
   Racional temp = new Racional();
   temp.rep = r.rep;

   return temp;
}
```

Figura 18. Copia compartiendo representación

```
Racional crearRacional(Racional r) {
   int n = numerador(r);
   int d = denominador(r);

   return new Racional(n, d)
}
```

Figura 19. Copia sin compartir representación

En ambas propuestas se crea un nuevo racional que finalmente, cuando se retorna, tiene el mismo numerador y denominador que el racional que se pasa como argumento. Sin embargo, en el primer caso (Figura 18) la copia comparte información con el racional copiado porque ambos tienen la misma representación (el mismo índice a la componente del vector que contiene la información). Mientras que en el segundo caso (Figura 19) coincidirán numeradores y denominadores para la copia y el racional copiado, pero cada uno de ellos mantendrá esa información en su propio espacio de almacenamiento separado (sus representaciones difieren).

Para clarificar el comportamiento de las dos propuestas de implementación para la operación de copia, se verá a continuación cómo queda la estructura de datos tras realizar la secuencia de operaciones que se muestra en la Figura 20, supuesto que estas son las primeras instrucciones del programa cliente.

```
r1 = crearRacional(4, 6);
r2 = crearRacional(0, 1);
r3 = crearRacional(r1);
```

Figura 20. Programa que usa el tipo Racional

Para ambas propuestas, una vez creados los dos primeros racionales, `r1` y `r2`, el contenido de la estructura de datos sería exactamente el que se muestra en la Figura 21. En esta situación, el racional `r1` tiene el valor abstracto `4/6` y está representado por el `0` y el racional `r2` tiene el valor abstracto `0/1` y está representado por el `1`.

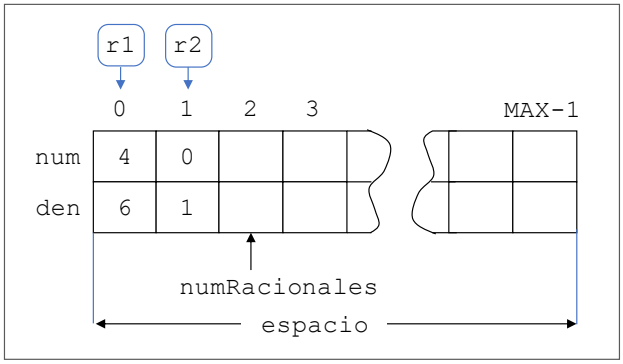

Figura 21. Cambios en la estructura de datos al crear racionales

A continuación, veremos cómo se modifica la estructura de datos para ambas propuestas de codificación de la operación de copia, al ejecutarse la última sentencia del cliente; es decir, al copiar el racional `r1` en `r3`.

En el primer caso, se crea el racional `r3` invocando el constructor por defecto, que no hace nada (no modifica la estructura de datos), y se asigna a su representación (`r3.rep`) la representación de `r1` (`r1.rep`), de forma que `r3` está representado, al igual que `r1`, por el `0` (ver Figura 22). Así, ambos racionales tienen el mismo valor abstracto (`4/6`) y la misma representación; es decir, están compartiendo la información (numerador y denominador).

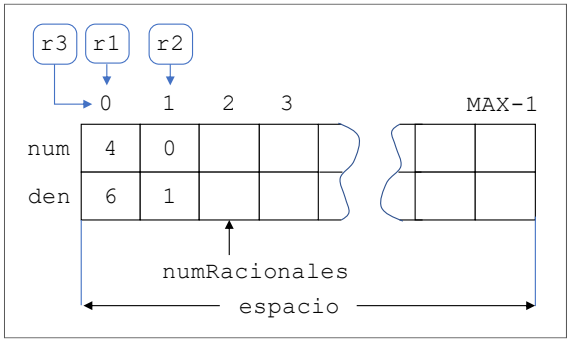

Figura 22. Copia con información compartida

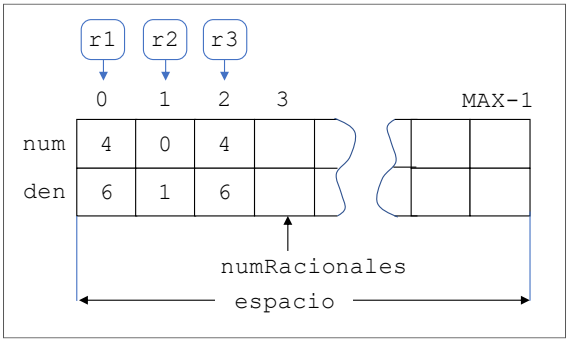

Figura 23. Copia con información no compartida

En el segundo caso, para crear el racional `r3` se invoca el constructor `Racional(n, d)` siendo `n` el numerador del racional `r1` y `d` su denominador. También aquí ambos racionales tendrán el mismo valor abstracto (`4/6`) pero ahora, al invocarse este constructor, la información del racional `r3` se almacenará en la primera posición libre del `espacio` común, la `2`, y este será el valor de su representación (ver Figura 23). Ambos racionales tienen el mismo valor abstracto, pero distinta representación, de forma que la información de cada uno de ellos está almacenada en una componente distinta del vector `espacio`.

Del análisis realizado se pueden extraer las siguientes conclusiones:

- En *los tipos de datos no modificables* la representación de un nuevo valor del tipo se puede compartir con la representación del valor del mismo tipo a partir del cual se creó, mejorando de esta forma su eficiencia espacial. En este caso, lo habitual es que *la implementación del constructor de copia se limite a una simple copia de la referencia a la estructura de datos que representa el valor a copiar*.
- Por el contrario, *los tipos de datos modificables* presentan inconvenientes cuando se comparte la representación porque una modificación en cualquiera de los valores afectará al otro. En este caso, se requiere *un constructor de copia que duplique toda la información (no que copie referencias) de la estructura de datos que representa el valor a copiar,* de forma que éste y el valor copiado no compartan información.

2.3 Abstracción de datos en LPOOs

Los lenguajes de programación orientados a objetos (LPOOs) ofrecen, entre otras características, un buen soporte para la *abstracción de datos*. En éstos una clase completamente abstracta (sin implementación) permite exponer las operaciones de un tipo de dato. Si además se incluye la semántica de las operaciones, por ejemplo, mediante comentarios que especifiquen éstas de manera cuasi-formal, entonces la clase completamente abstracta es la definición de un tipo de dato abstracto (TDA).

Debe tenerse en cuenta que las operaciones de la clase abstracta trabajarán sobre un objeto de una clase, mientras que en las especificaciones vistas hasta ahora (a*lgebraica* y *cuasi-formal*) las operaciones trabajan sobre los valores de un tipo de dato; es decir, estas últimas son especificaciones funcionales. Por este motivo el perfil de las operaciones en la clase completamente abstracta será ligeramente distinto.

Por otra parte, en algunos LPOOs las operaciones constructoras tienen el mismo nombre que el tipo de dato. En este caso, salvo que resulte conveniente disponer de otras operaciones constructoras (por ejemplo, en factorías de objetos), la clase abstracta carecerá de éstas.

2.3.1 Especificación de un TDA en Java

En Java, una **interfaz** (`interface`) es una clase completamente abstracta que sólo encapsula métodos abstractos. Por tanto, una interfaz es un mecanismo adecuado para definir un tipo de dato abstracto si, además, también se describe el comportamiento de cada una de las operaciones (su semántica). La especificación de las operaciones se realizará de manera informal, pero en este caso se utilizará la utilidad *Javadoc*. Esto nos permitirá tanto especificar las operaciones como generar la documentación de un TDA en distintos formatos. A modo de ejemplo, la Figura 24 muestra la definición del tipo de dato abstracto `Racional` en Java. Puede observarse que la especificación es similar a la proporcionada en el apartado 2.2.2.1, aunque hay algunas diferencias.

```
public interface Racional {

    /**
     * Retorna el racional n/d.
     * @param n el numerador del racional
     * @param d el denominador especificado
     * @return el racional n/d
     * @throws RuntimeException si d=0
     */
    Racional crearRacional(int n, int d);

    /**
     * El numerador de este racional.
     * @return si este número racional es n/d, produce el valor
     * n de su numerador
     */
    int numerador();

    /**
     * El denominador de este racional.
     * @return si este número racional es n/d, produce el valor
     * d de su denominador
     */
    int denominador();

    /**
     * Modifica este racional a otro equivalente e irreducible (opcional).
     * @throws UnsupportedOperationException si la operación no está
     * soportada para este racional
     */
    default void simplificar() {
        throw new UnsupportedOperationException();
    }
```

```
    /**
     * Retorna la suma de este racional con el especificado.
     * @param r el racional especificado
     * @return el racional suma de éste y del racional r
     */
    default Racional sumar(Racional r) {
        Racional s = crearRacional(
                this.numerador() * r.denominador()
                + this.denominador() * r.numerador(),
                this.denominador() * r.denominador());
        return s;
    }
}
```

Figura 24. Definición del tipo de dato abstracto `Racional` *en Java*

Lo primero que debe tenerse en cuenta es que a partir de la versión 8 de Java el cuerpo de cualquier interfaz puede contener *métodos por defecto*, definiciones de *métodos estáticos* y *constantes*, además de los *métodos abstractos*. Todos ellos son implícitamente públicos (se puede omitir el modificador de acceso `public`).

Tantos los métodos por defecto como los métodos estáticos están implementados en la interfaz y proporcionan herramientas útiles para las clases que la implementan. Los métodos por defecto van precedidos de la palabra reservada `default` en la cabecera de la función y permiten extender la funcionalidad de una nueva versión de una interfaz manteniendo la compatibilidad con versiones previas. Adicionalmente, dotan a Java de herencia múltiple de comportamiento, además de la herencia múltiple de tipos, ya que una clase puede implementar varias interfaces. Sin embargo, la herencia de estado es simple, sólo se puede extender otra clase (no varias).

En lo que respecta al perfil de las operaciones, aquí se está considerando el tipo de dato como un conjunto de objetos, en lugar de un conjunto de valores y, en consecuencia, el acceso a las operaciones se realiza mediante la notación punto aplicada a un objeto del tipo. Esta es la razón por la que cambia el perfil de la mayor parte de las operaciones, pasando a tener un parámetro de tipo `Racional` menos con respecto a las operaciones de una especificación funcional.

En la interfaz dada hay una operación constructora, la operación `crearRacional`, porque así estaba especificado inicialmente en el TDA `Racional`. Este método, al tratarse de un constructor, sería más útil como función estática o método de otro tipo distinto (de creación de objetos), dado que no parece excesivamente útil un método de creación de objetos que requiera un objeto del propio tipo para ser invocado. Está forma de construcción de objetos lleva al uso de *patrones de diseño*[2] creacionales y éstos se verán en una asignatura del segundo semestre. A pesar de todo, y aunque existan mejores soluciones, se mantendrá la operación `crearRacional` como parte de la interfaz, ya que en caso contrario la definición de cualquier *operación no básica* que requiera invocar un constructor quedaría necesariamente relegada a su implementación. Así ocurriría, por ejemplo, con la operación `sumar` (ver Figura 24).

> El incluir métodos constructores en la interfaz permite establecer la definición de *operaciones no básicas* que requieran invocar un constructor para su definición y que, en otro caso, sólo se podrían definir en su implementación.

[2] El término *patrón de diseño* se utiliza para referirse a una solución efectiva y reutilizable de un problema que se repite con frecuencia en el desarrollo de software.

Para finalizar, se debe tener presente que en Java no se suelen proporcionar interfaces separadas para cada variante del TDA (tipos de datos modificables o no modificables).

> Las operaciones modificadoras se incluyen siempre en la interfaz, independientemente de la característica de mutabilidad de los tipos de datos, y se designan como *operaciones opcionales*. Para éstas se proporciona una implementación por defecto (`default`) que lanza la excepción `UnsupportedOperationException`.

Las implementaciones de la interfaz pueden optar por soportar o no las operaciones opcionales y son éstas las responsables de documentar que operaciones opcionales soporta. Si un programa cliente invoca alguna operación opcional que no esté soportada, se lanza la excepción `UnsupportedOperationException` que es la implementación por defecto (`default`) de las operaciones modificadoras. Por ejemplo, así es como se proporciona la operación modificadora `simplificar` en la interfaz `Racional` de la Figura 24.

2.3.2 Implementación de un TDA en Java

En Java la implementación de un tipo de dato abstracto es una clase que implementa la interfaz correspondiente. Si el tipo de dato es *no modificable* todas las operaciones opcionales de la interfaz, que son operaciones modificadoras, se mantienen como operaciones no soportadas (no hay que redefinirlas). Por el contrario, si el tipo de dato es *modificable* habrá que redefinir alguna de las operaciones opcionales de la interfaz.

Obsérvese que la característica de mutabilidad de un tipo de dato indica si éste dispone o no de operaciones modificadoras en la interfaz del TDA y, en el caso particular de una interfaz Java, si dispone o no de *operaciones modificadoras soportadas*. En este sentido, un error muy común es considerar que cualquier clase que contenga una operación no constructora que modifique el estado (el área de datos de la clase) proporciona un tipo de dato modificable. En realidad, esto dependerá del modificador de acceso de la operación, si la operación es pública (`public`) efectivamente así es, pero si la operación está protegida (`protected` o `private`) no influye en la característica de mutabilidad del tipo de dato. Dicho de otra forma, una clase que implemente un tipo de dato *no modificable* puede incorporar operaciones protegidas que modifiquen el estado, pero lo que no puede tener son operaciones públicas que modifiquen éste.

2.3.2.1 Operaciones adicionales

En el apartado 2.2.4.1 ya se establecieron algunas consideraciones generales sobre la implementación de las operaciones de un TDA. Una de las cuales afectaba al constructor de copia que, habitualmente, se proporciona en cualquier implementación de éste.

Adicionalmente, en Java hay que tener en cuenta las operaciones que están especificadas para el objeto `Object`, ya que al ser éste la raíz de la jerarquía de objetos todos los subtipos las heredan. Interesa, por tanto, saber si es necesario o no redefinir alguna de las operaciones heredadas de `Object`, con el fin de que se sigan cumpliendo sus especificaciones (el contrato). En particular, hay cuatro de estas operaciones que tienen especial interés: `toString`, `clone`, `equals` y `hashCode`.

A continuación, se proporciona el tipo de dato `RacionalImp` que es una implementación de la interfaz `Racional` de la Figura 24 y, posteriormente, se analizará la implementación de las operaciones que se heredan de la clase `Object`. Por coherencia con otros tipos numéricos de Java el tipo de dato será no modificable y como área de datos (representación o estado de un objeto) se utilizarán dos enteros: `num` para el numerador y `den` para el denominador.

```java
/**
 * Función de abstracción: Abs: rep → Racional
 *                              r → r.num/r.den
 * Invariante de la representación: r.den ≠ 0
 */
public class RacionalImp implements Racional {
    private int num;  // numerador
    private int den;  // denominador

    /**
     * Crea el racional 0/1.
     */
    public RacionalImp() {
        this(0, 1);
    }

    /**
     * Crea el racional de numerador y denominador especificados.
     * @param num el numerador de este racional
     * @param den el denominador de este racional
     * @throws RuntimeException si den=0
     */
    public RacionalImp(int num, int den) {
        if (den == 0) {
            throw new RuntimeException();
        }

        this.num = den > 0 ? num : -num;  // signo en el numerador
        this.den = Math.abs(den);
    }

    /**
     * Crea un racional copia del especificado.
     * @param r el racional a copiar
     * @throws NullPointerException si r es null
     */
    public RacionalImp(Racional r) {
        if (r == null) {
            throw new NullPointerException();
        }

        if (r instanceof RacionalImp) {
            RacionalImp temp = (RacionalImp) r;
            this.num = temp.num;
            this.den = temp.den;
        } else {
            this.num = r.numerador();
            this.den = r.denominador();
        }
    }

    /**
     * @see estDatos.Racional#crearRacional(int, int)
     */
    @Override
    public Racional crearRacional(int n, int d) {
        return new RacionalImp(n, d);
    }
```

```
    /**
     * @see estDatos.Racional#numerador()
     */
    @Override
    public int numerador() {
        return num;
    }

    /**
     * @see estDatos.Racional#denominador()
     */
    @Override
    public int denominador() {
        return den;
    }
}
```

Figura 25. Implementación de la interfaz `Racional`

Obsérvese cómo se ha implementado el constructor de copia. Como el tipo de dato `RacionalImp` no es modificable, la copia ha de compartir la información con el racional a copiar (ver apartado 2.2.4.1). Esto sólo es posible si el racional a copiar también es un objeto de tipo `RacionalImp`, en caso contrario sólo se podrá copiar la información (numerador y denominador). Si el tipo de dato hubiera sido modificable la información habría que copiarla.

Debido a que en Java el tipo `int` es inmutable (no modificable), en este caso en particular, no hay ninguna diferencia entre las dos partes del condicional en la implementación del constructor de copia, que simplemente se podría haber implementado de forma equivalente como:

```
public RacionalImp(Racional r) {
    this(r.numerador(), r.denominador()); // si no se incluye la excepción
}
```

Figura 26. Constructor de copia para el tipo `RacionalImp`

la cual, a su vez, sería la forma correcta de implementar el constructor de copia para un tipo de dato `Racional` modificable. Esta coincidencia y simplificación en la implementación del constructor de copia es más una excepción que una regla y sólo se ha comentado a modo de aclaración, ya que en otro caso al lector le podrían surgir dudas al respecto.

2.3.2.1.1 Método toString

Este método retorna la representación de un objeto como una cadena de caracteres (representación textual del objeto). Como norma general, la implementación de la operación `toString` debe ser concisa, por lo demás no hay una regla que determine como ha de ser la cadena de caracteres resultante, salvo el que dicta la lógica: *si en el lenguaje de programación utilizado hay tipos similares al implementado utilizar el mismo formato y si el tipo de dato tiene una representación textual de uso común utilizar ésta y no otra*.

La especificación del método `toString` en la clase `Object` es la siguiente:

```
/**
 * Retorna la representación textual de este objeto
 * @return la cadena de caracteres construida como
 * getClass().getName() + '@' + Integer.toHexString(hashCode())
 */
public String toString();
```

Figura 27. Especificación del método `toString` *en* `Object`

de forma que, si no se redefine la operación `toString` para el tipo de dato `Racional`, la salida de un programa como el que se indica a continuación:

```
Racional r = new RacionalImp(3, 4);
System.out.println(r.toString()); // o simplemente System.out.println(r)
```

sería similar a la siguiente:

```
estDatos.RacionalImp@15db9742
```

Esta información textual proporcionada es ilegible para un usuario cualquiera, razón por la cual lo más habitual es que el método `toString` se particularice (redefina) para los tipos de datos que se implementen. En el caso del tipo `Racional`, la salida de la operación `toString` está impuesta por la representación textual de uso común para los racionales, una de las dos formas siguientes:

$$\frac{numerador}{denominador} \quad \text{o bien como} \quad numerador/denominador$$

A priori, ambas opciones son válidas, pero la primera ocupa más de una línea y resulta más compleja de generar, por lo que se utilizará la segunda opción. De modo que la implementación de la operación `toString` será la siguiente:

```
@Override
public String toString() {
    return numerador() + "/" + denominador();
}
```

Figura 28. Definición de la operación `toString` *para el tipo* `RacionalImp`

Cuando se indica que la implementación de la operación `toString` debe ser concisa, ha de entenderse que únicamente debe proporcionarse, con cierto formato, la información del objeto. Para el ejemplo que nos ocupa, el numerador y denominador del número racional, pero sin información adicional alguna. En este sentido, un error relativamente común es implementar la operación `toString` de forma análoga a como se muestra en la Figura 29.

```
@Override
public String toString() {
    return "El racional es: " + numerador() + "/" + denominador();
}
```

Figura 29. Incorrecta definición para la operación `toString`

Toda información adicional que se incluya está de más y aunque éste pueda parecer un error sin mayor transcendencia, realmente no es así. Los clientes del tipo de dato (programas que lo usan) deben poder decidir libremente cuándo quieren incorporar algún tipo de información adicional y, si fuera el caso, cuál debe ser ésta.

Por poner un ejemplo, supóngase que se escribe el programa siguiente:

```
Racional r1 = new RacionalImp(2, 3);
Racional r2 = new RacionalImp(1, 2);
System.out.println(r1 + " + " + r2 + " = " + r1.sumar(r2));
```

cuya salida esperada es:

```
2/3 + 1/2 = 7/6
```

Resultaría bastante extraño que tras ejecutar el programa se obtuviera el casi ilegible texto:

```
El racional es: 2/3 + El racional es: 1/2 = El racional es: 7/6
```

resultado de haber implementado la operación `toString` erróneamente (ver Figura 29).

2.3.2.1.2 Método clone

El método `clone` crea y retorna una copia del objeto. Por convenio, el objeto retornado se debe obtener llamando a `super.clone` y debe ser independiente del objeto que lo invoca. Si se copia un objeto mutable es necesario copiar la información del objeto a copiar, pero si éste sólo consta de campos primitivos o referencias a objetos inmutables no es necesario modificar los campos en el objeto retornado por `super.clone`. La clase en la que se incluya la operación `clone` debe implementar la interfaz `Cloneable`, en caso contrario al invocar ésta se lanzará la excepción `CloneNotSupportedException`.

Para el tipo de dato no modificable `RacionaImp` la implementación de la operación `clone` es la siguiente:

```
public Object clone() {
    try {
        RacionalImp r = (RacionalImp) super.clone();
        return r;
    } catch (CloneNotSupportedException e) {
        throw new InternalError(e);
    }
}
```

Figura 30. Definición de la operación clone

2.3.2.1.3 Método equals

El método `equals` indica cuando un objeto es igual a otro y establece una relación de equivalencia sobre objetos de referencias no nulas. La especificación de esta operación para la clase `Object` es la siguiente:

```
/**
 * Indica si el objeto especificado es 'igual' a éste.
 * @param o el objeto a comparar para igualdad
 * @return si x es la referencia de este objeto e y es la referencia
 * del objeto o, retorna cierto si y sólo si x = y (es el mismo objeto).
 * Si alguna de las referencias es nula, retorna falso.
 */
public boolean equals(Object o);
```

Figura 31. Especificación del método `equals` en `Object`

En la clase `Object` el método `equals` establece la relación de equivalencia más discriminante entre objetos, sólo son iguales si son el mismo objeto. Por tanto, en la clase `Object` el método `equals` coincide con el operador ==, pero de manera general esto no será así.

> Para comparar la *igualdad entre objetos* de un mismo tipo debe utilizarse el método `equals` y no el operador ==. En este último caso se compararían las referencias de los objetos y, por tanto, se estaría comprobando si ambos objetos son el mismo.
> El operador == sólo debe utilizarse para comparar la igualdad de expresiones que proporcionen valores de tipos de datos primitivos, salvo que expresamente se estén comparando referencias entre objetos.

Para el tipo de dato `RacionaImp` la implementación de la operación `equals` es la siguiente:

```
/**
 * Indica si el objeto especificado es igual a este racional.
 * @param o el objeto a comparar
 * @return cierto si y sólo si el objeto especificado es un
 * número racional y es igual a éste
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }

    if (! (o instanceof Racional)) {
        return false;
    }

    Racional other = (Racional) o;
    return numerador() * other.denominador()
           == denominador() * other.numerador();
}
```

Figura 32. Definición de la operación `equals` *para el tipo* `RacionalImp`

Dado que el parámetro de la operación `equals` es de tipo `Object`, para hacer la comprobación de igualdad es necesario determinar el tipo del argumento con que se invoca la operación. Esta comprobación de tipo se puede realizar utilizando el operador `instanceof` o bien mediante el método `getClass` de la clase `Object`. El método `getClass` sólo comprueba si los tipos son idénticos, mientras que el operador `instanceof` comprueba si el objeto referenciado en la parte izquierda es una instancia del tipo indicado en la parte derecha o bien de alguno de sus subtipos y, por lo general, es preferible utilizar este último.

2.3.2.1.4 Método hashCode

El método `hashCode` retorna un valor de código *hash* para el objeto. Mientras no se modifique la información del objeto este método debe retornar siempre el mismo valor entero y si dos objetos son iguales, de acuerdo con el método `equals`, al invocar el método `hashCode` con ambos objetos se debe producir el mismo resultado. Por tanto, si en un tipo se redefine el método `equals`, también deberá redefinirse la operación `hashCode` para mantener su contrato.

2.3.2.2 Las interfaces Comparable y Comparator de Java

En ocasiones, puede resultar necesario comparar los objetos de un tipo de dato. No sólo para determinar la igualdad entre éstos, para lo cual ya está el método `equals` heredado de la clase `Object`, si no para establecer cuál de ellos es menor o mayor que otro según cierta *relación de orden*. Esto ocurre, por ejemplo, cuando se quiere definir una función para ordenar una colección de objetos de algún tipo o determinar el mayor o menor de todos ellos. Incluso en los objetos de tipo contenedor podría resultar de interés mantener la información (sus elementos) ordenada de alguna forma.

2.3.2.2.1 El método compareTo

En Java, la interfaz `Comparable` especifica el método `compareTo` (Figura 33), el cual permite comparar objetos de tipos de datos para los cuales ya existe un *orden natural* preestablecido, tal y como ocurre, por ejemplo, para los enteros (tipo `Integer`) y para las cadenas de caracteres (tipo `String`) mediante su orden lexicográfico.

La especificación del método `compareTo` es la siguiente:

```
/**
 * Compara este objeto con el especificado para comprobar el orden.
 * @param o el objeto especificado para comparación
 * @return un entero negativo, cero, o positivo si este objeto es menor,
 * igual o mayor, respectivamente, que el objeto especificado
 * @throws NullPointerException si el objeto especificado es null
 * @throws ClassCastException si el tipo del objeto especificado
 * impide que se compare con este objeto
int compareTo(Objet o);
```

Figura 33. Especificación del método `compareTo` *en la interfaz* `Comparable`

Como el método `compareTo` también permite comprobar la igualdad de dos objetos (cuando el valor retornado es `0`), la recomendación es que se cumpla:

$$\texttt{a.compareTo(b)} = 0 \Leftrightarrow \texttt{a.equals(b)}$$

sin embargo, esto no es obligatorio. Eso sí, cualquier clase que incumpla esta equivalencia debe especificarlo claramente utilizando, preferiblemente, el siguiente lenguaje:

Nota: esta clase tiene un orden natural que es inconsistente con el método equals

Para pode utilizar el método `compareTo`, las interfaces de todos los tipos de datos de Java cuyos objetos tienen un orden natural preestablecido extienden la interfaz `Comparable`. De la misma forma, para definir un nuevo tipo de dato abstracto para el que exista un orden natural entre los objetos, debe extenderse la interfaz `Comparable`.

A modo de ejemplo, ahora se está es disposición de proporcionar la siguiente versión mejorada de la función `max` vista en el apartado 1.1.1. La nueva versión de esta función, definida en la Figura 34, permite calcular la mayor componente de un vector supuesto que el tipo de sus componentes implementa la interfaz `Comparable`.

```
T max(T[] v) {
   T result = v[0];
   for (T e: v) {
      if ((Comparable) result).compareTo(e) < 0) {
         result = e;
      }
   }

   return result;
}
```

Figura 34. Máximo de un vector de elementos de tipo `T` *comparable*

En la versión inicial de la función `max` (Figura 4) se utilizó el operador `<` para comprobar si `result` es menor que `e`, mientras que en esta nueva versión la comparación se realiza mediante el método `compareTo`. El problema del operador `<` es que sólo permite comparar valores de tipos primitivos (`char`, `int`, etc.) y sus correspondientes tipos objeto (`Character`, `Integer`, etc.), pero no está definido para ningún otro tipo de objetos comparables como, por ejemplo, el tipo `String`.

Ahora la función `max` es utilizable con vectores cuyas componentes sean objetos de cualquier tipo `T` que implemente la interfaz `Comparable` y, por ejemplo, el siguiente código cliente es válido:

```
Integer[] vi = {10, 2, -3, 24, 15};
String[] vs = {"esto", "es", "un", "vector", "de", "String"};

System.out.println("Máximo de vi: " + max(vi));
System.out.println("Máximo de vs: " + max(vs));
```

proporcionando la salida:

```
Máximo de vi: 24
Máximo de vs: vector
```

Obsérvese que, en esta última definición de la función `max` (ver Figura 34), el tipo `T` de la variable `result` se convierte de forma explícita (*casting*) en el tipo `Comparable`. Esto es necesario porque el tipo `T` es un parámetro (un tipo cualquiera) y por tanto el compilador desconoce si a sus objetos se les puede aplicar el método `compareTo`. Posteriormente, si durante la ejecución de un programa se invoca a la función con un argumento de tipo `T` que extiende la interfaz `Comparable` (como, por ejemplo, `String`) no hay ningún problema, en caso contrario se produciría un error.

Para finalizar, se especifica e implementa un TDA en el que existe un orden natural. Esto se podría hacer de forma simple y rápida para el TDA `Racional`, sólo habría que decir que su interfaz extiende la interfaz `Comparable` y definir el método `compareTo` para los números racionales. Sin embargo, se especificará e implementará un nuevo TDA para proporcionar otro ejemplo completamente desarrollado, el TDA `Punto`. Los objetos de este TDA son puntos de un plano cartesiano y entre éstos se toma como orden natural el *orden lexicográfico*.

```
public interface Punto extends Comparable {

    /**
     * Retorna la abscisa de este punto.
     * @return la abscisa del punto
     */
    double abscisa();

    /**
     * Retorna la ordenada de este punto.
     * @return la ordenada del punto
     */
    double ordenada();


    /**
     * Retorna la distancia de este punto al origen de coordenadas.
     * @return la distancia del punto al origen de coordenadas. Si
     * el punto tiene de coordenadas (x, y) devuelve el valor
     * <code_HTML> √(x² + y²) </code_HTML>.
     */
    default double distancia() {
        return Math.sqrt(Math.pow(abscisa(), 2)
                         + Math.pow(ordenada(), 2));
    }
```

```
/**
 * Retorna la distancia entre este punto y el especificado.
 * @param p el punto especificado
 * @return la distancia entre este punto y el especificado. Si este
 * punto tiene de coordenadas (x, y) y el especificado (u, v)
 * retorna <code_HTML> √((u − x)² + (v − y)²) </code_HTML>.
 */
default double distancia(Punto p) {
    return Math.sqrt(Math.pow(p.abscisa()- abscisa(), 2)
                   + Math.pow(p.ordenada()- ordenada(), 2));
}

/**
 * Modifica la abcisa de este punto al valor especificado (opcional).
 * @param x el nuevo valor de la abscisa de este punto
 * @throws UnsupportedOperationException si no está soportada
 */
default void setAbscisa(double x) {
    throw new UnsupportedOperationException();
}

/**
 * Modifica la ordenada de este punto al valor especificado (opcional)
 * @param y el nuevo valor de la ordenada de este punto
 * @throws UnsupportedOperationException si esta operación no está
 * soportada
 */
default void setOrdenada(double y) {
    throw new UnsupportedOperationException();
}

/**
 * Compara este punto con el objeto especificado.
 * @param o el objeto a comparar
 * @return un entero negativo, cero o positivo si el orden
 * lexicográfico de este punto es menor, igual o mayor,
 * respectivamente, que el correspondiente al objeto especificado
 * <p>
 * Si (a, b) y (c, d) son las coordenadas de dos puntos p y q,
 * entonces <code_HTML> p ≤ q ⇔ ((a ≤ c) ∨ (a = c) ∧ (b ≤ d)) </code_HTML>
 * </p>
 * @throws NullPointerException si el objeto especificado es null
 * @throws ClassCastException si el tipo del objeto especificado
 * no permite la comparación
 */
@Override
default int compareTo(Object o) {
    if (o == null) {
        throw new NullPointerException();
    }
    if (! (o instanceof Punto)) {
        throw new ClassCastException();
    }
    Punto p = (Punto) o;
```

```
            if (abscisa() != p.abscisa()) {
                return (int) (abscisa() - p.abscisa());
            } else {
                return (int) (ordenada() - p.ordenada());
            }
        }
    }
```

Figura 35. Especificación del TDA `Punto`

Para implementar el TDA `Punto` se utilizará como estructura de datos un par de números reales, uno para la abscisa y otro para la ordenada del punto.

```
public class PuntoImp implements Punto {
    // abscisa del punto
    private double x;
    // ordenada del punto
    private double y;

    /**
     * Crea el punto de coordenadas (0, 0)
     */
    public PuntoImp() {
        this(0, 0);
    }

    /**
     * Crea el punto de abscisa y ordenada especificadas.
     * @param x la abscisa de este punto
     * @param y la ordenada de este punto
     */
    public PuntoImp(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Crea un punto copia del especificado.
     * @param p el punto a copiar
     * @throws NullPointerException si p es null
     */
    public PuntoImp(Punto p) {
        if (p == null) {
            throw new NullPointerException();
        }
        x = p.abscisa();
        y = p.ordenada();
    }

    /**
     * @see estDatos.Punto#abscisa()
     */
    public double abscisa() {
        return x;
    }
```

```
    /**
     * @see estDatos.Punto#ordenada()
     */
    public double ordenada() {
        return y;
    }

    /**
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString() {
        return "(" + abscisa() + ", " + ordenada() + ")";
    }

    /**
     * Compara la igualdad entre este punto y el objeto especificado.
     * @param o el objeto a comparar para igualdad
     * @return cierto si ambos puntos tienen las mismas coordenadas
     * @throws NullPointerException si el objeto especificado es null
     * @throws ClassCastException si el tipo del objeto especificado
     * no permite la comparación
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (! (o instanceof Punto)) {
            return false;
        }

        Punto p = (Punto) o;
        return abscisa() == p.abscisa()
                && ordenada() == p.ordenada();
    }
}
```

Figura 36. Implementación del TDA `Punto`

Para la implementación del TDA `Punto` dado en la Figura 36, la *función de abstracción* que relaciona el tipo de la representación con el tipo abstracto es la siguiente:

$$\begin{aligned} \boldsymbol{Abs}: rep &\rightarrow Punto \\ p &\rightarrow (p.x, p.y) \end{aligned}$$

y el *invariante de la representación*: $verdadero$

2.3.2.2.2 El método compare

En Java, la interfaz `Comparator` especifica el método `compare`, el cual permite comparar objetos de tipos de datos para los cuales no existe un *orden natural* preestablecido, o bien utilizar un criterio de comparación de objetos distinto del *orden natural* en los tipos de datos en los que sí existe éste.

Supóngase, por ejemplo, que la función `max` dada en la Figura 34 se quiere utilizar con un tipo de dato `T` para el que no existe un orden natural preestablecido, habría que suministrar también a la función `max` la función de comparación que permite comparar los objetos de tipo `T`. Más

concretamente, definir el método `compare` para una clase que implemente la interfaz `Comparator` y pasar una instancia de ésta a la función `max`. (para más detalles consúltese el principio del apartado 7.2.1).

El método `compare` es la función de comparación con la que se establece el orden entre los objetos del tipo `T`. La especificación de este método es la siguiente:

```
/**
 * Compara los dos objetos especificados para comprobar el orden.
 * @param o1 el primer objeto a comparar
 * @param o2 el segundo objeto a comparar
 * @return un entero negativo, cero, o positivo si el primer argumento
 * es menor, igual o mayor, respectivamente, que el segundo
 * @throws NullPointerException si algún argumento es null
 * @throws ClassCastException si los tipos de los argumentos a ser
 * comparados impide realizar la comparación
int compare(Objet o1, Object o2);
```

Figura 37. Especificación del método `compare` *de la interfaz* `Comparator`

Por último, para que la función `max` se pueda invocar también con vectores de elementos de un tipo `T` cualquiera, aunque entre los objetos de este tipo no exista un orden natural, únicamente se requiere sobrecargar la función con un parámetro adicional de tipo `Comparator`. En esta definición sobrecargada de la función `max`, la comparación entre objetos se llevará a cabo con el método `compare`, tal y como se indica a continuación:

```
public static <T> T max(T[] v, Comparator cmp) {
   T result = v[0];
   for (T e: v) {
      if (cmp.compare(result, e) < 0) {
         result = e;
      }
   }

   return result;
}
```

Figura 38. Máximo de un vector de elementos de tipo `T`

Para establecer la función de comparación, se puede definir una clase que implemente la interfaz `Comparator`, e invocar la función con un objeto de ésta.

```
public class ComparadorV implements Comparator {
    ComparadorV () { }         // definición por defecto del constructor
    public int compare(Object o1, Object o2) {
        if (o1 == null || o2 == null) {
            throw new NullPointerException();
        }
        if (! (o1 instanceof String && o2 instanceof String)) {
            throw new ClassCastException();
        }
        String s1 = (String) o1;
        String s2 = (String) o2;
        return -s1.compareTo(s2);
    }
}
```

Figura 39. Clase que implementa la interfaz `Comparador`

En la clase `ComparadorV` de la Figura 39 el método `compare` establece un orden inverso al natural para los objetos de un tipo que implemente la interfaz `Comparable`, de forma que el siguiente código cliente permite obtener el mínimo del vector en lugar del máximo:

```
String[] vs = {"esto", "es", "un", "vector", "de", "String"};

System.out.println("Mínimo de vs: " +
                   max(vs, new ComparadorV()));
```

y al ser ejecutado proporciona la salida:

```
Mínimo de vs: String
```

Una segunda posibilidad es utilizar una clase anónima e instanciar el objeto comparador al invocar la función `max` en el cliente, tal y como se muestra a continuación:

```
String[] vs = {"esto", "es", "un", "vector", "de", "String"};

System.out.println("Mínimo de v: " +
    max(v, new Comparator() {  // clase anónima
                public int compare(Object o1, Object o2) {
                    if (o1 == null || o2 == null) {
                        throw new NullPointerException();
                    }
                    if (! (o1 instanceof String
                           && o2 instanceof String)) {
                        throw new ClassCastException();
                    }
                    String s1 = (String) o1;
                    String s2 = (String) o2;
                    return -s1.compareTo(s2);
                }}));
```

En realidad, hay una forma bastante más simple de obtener el mínimo de un vector de cadenas de caracteres. La propia interfaz `Comparator` proporciona varias funciones estáticas que retornan un comparador y, en particular, la función *`Comparator`*`.reverseOrder` ordena en sentido inverso los objetos de un tipo de dato para el que exista un orden natural preestablecido, como es el caso del tipo `String`. De forma que el código previo podría darse simplemente como:

```
String[] v = {"esto", "es", "un", "vector", "de", "String"};

System.out.println("Mínimo de v: " +
                   max(v, Comparator.reverseOrder()));
```

E incluso en el hipotético caso de que la interfaz `Comparator` no dispusiera de esta función estática, desde la versión 8 de Java existe una alternativa mejor y más simple que la invocación con clases anónimas: la invocación con *expresiones lambda*. Éstas se verán más adelante, cuando se introduzcan las ampliaciones de la versión 8 del lenguaje Java que, fundamentalmente, se centraron en incorporar características del paradigma de programación funcional (apartado 7).

3 Abstracción de iteración

Si se tienen nociones generales sobre tipos de datos contenedores (colecciones o agregados) resultará obvio que tarde o temprano surgirá el siguiente problema a solucionar: *cómo procesar todos los elementos de una colección* (o parte de éstos, por ejemplo, en una búsqueda).

Obsérvese que procesar todos los elementos de un *array* `a[0..n-1]` de tamaño `n` (colección ya conocida) es una tarea que se aborda con facilidad en cualquier lenguaje de programación de alto nivel, ya que el acceso al *i-ésimo* elemento se realiza directamente mediante el índice `i` (nombrando al elemento, por lo general, como `a[i]`). Supongamos, por ejemplo, que se quiere aplicar la abstracción funcional `f` a todos los elementos del *array* `a`. Esto se puede hacer fácilmente mediante el código siguiente:

```
for (int i = 0; i < n; i++)
   f(a[i]);
```

Figura 40. Procesamiento de los elementos de un array `a`

Sin embargo, no resulta tan simple realizar un algoritmo equivalente al previo para una colección `c` cualquiera de `n` elementos; es decir, aplicar la función `f` a cada uno de los elementos de la colección `c`.

En primer lugar, no siempre tiene sentido asignar un índice (del rango `[0, n-1]`) a cada uno de los elementos de la colección. A modo de ejemplo, supongamos que la colección es un conjunto, podríamos plantearnos la pregunta siguiente: ¿quién es el `n`-ésimo elemento de un conjunto? Pero, aunque se asignara de forma arbitraria un número correlativo a cada elemento de la colección `c` y se realizaran las abstracciones funcionales siguientes:

- `tamaño(c)`: retorna el número `n` de elementos de la colección `c` especificada.
- `valor(c, i)`: produce el `i`-ésimo elemento de la colección `c`.

el código equivalente de la Figura 41 probablemente resulte ineficiente porque la operación `valor(c, i)` es más específica de lo que realmente se necesita: *obtener todos los elementos de la colección uno a uno*.

```
n = tamaño(c);
for (int i = 0; i < n; i++)
   f(valor(c, i));
```

Figura 41. Procesamiento de los elementos de una colección `c`

Un enfoque distinto, pero que lleva también a concluir que la solución propuesta para tratar la colección `c` no es válida, sería analizar que tiene que cumplirse para que el código de la Figura 41 fuera equivalente al proporcionado para el *array* (Figura 40). Un simple vistazo a ambos códigos permite concluir que éstos serían equivalentes si la operación `valor(c, i)`, de acceso a los elementos de la colección, se comporta exactamente igual que la operación `a[i]`, de acceso a los elementos del *array*. Es decir, habría que proporcionar acceso aleatorio a los elementos de la colección `c`.

Implementar acceso directo (o aleatorio) a los elementos de una colección cualquiera es una tarea compleja y que, de forma general, sólo tiene una posible solución: tener previamente calculadas las operaciones `valor(c, i)` y almacenadas en un *array*. Por tanto, habría que tener duplicadas las referencias a los elementos de la colección y, aún en el supuesto de que se considerara ésta una opción válida, la iteración sobre los elementos de la colección no sería perezosa. No siempre es necesario recorrer todos los elementos de la colección, por ejemplo, si se desea localizar cierto elemento de ésta. Una vez encontrado el elemento a buscar no es

necesario seguir buscando entre los elementos restantes (evaluación perezosa). Sin embargo, con la solución propuesta siempre se estarían tratando todos los elementos porque las operaciones de acceso ya estarían previamente calculadas para todos ellos.

La **abstracción de iteración** simplifica este procesamiento al no tener en cuenta los detalles sobre cómo se obtienen los elementos de la colección. En consecuencia, *el procesamiento se realiza por igual para cualquier colección de elementos* y, además, éste se lleva a cabo de forma perezosa.

> Un **iterador** (*iterator*) es una clase especial de procedimiento que abstrae el proceso de obtener uno a uno los elementos de un tipo de dato contenedor (colección o agregado) *sin exponer la representación interna de éste*.

Los ítems producidos por el *iterador* se pueden utilizar en otros módulos que especifiquen acciones a realizar por cada ítem, por ejemplo, `f(item)`. Procesar todos los elementos proporcionados por el iterador se puede expresar mediante una sentencia de bucle con una estructura similar al que se muestra en la Figura 42.

```
para cada item producido por el iterador hacer
    f(item);
fin para;
```

Figura 42. Procesamiento de los elementos de una colección `c` con un iterador

Cada iteración del bucle produce un nuevo ítem, sobre el cual actúa entonces el cuerpo del bucle. Obsérvese la separación de responsabilidades, el iterador produce los ítems, mientras que el código del cuerpo del bucle define la acción a realizar con cada uno de ellos. Así, el iterador se puede utilizar en diferentes módulos que realicen diferentes acciones sobre los ítems y se puede implementar de varias formas sin que los cambios afecten a estos módulos.

Como el iterador proporciona los ítems uno a uno, no requiere de una estructura de datos grande como si tuviera que almacenar éstos. Además, si se realiza una búsqueda se puede parar el iterador cuando se localice el ítem de interés (la evaluación es perezosa).

3.1 Abstracción de iteración en LPPOs

La mayoría de los lenguajes de programación (no sólo LPPOs) incluyen un bucle `for` que permite iterar sobre un rango de valores enumerables. De forma análoga, un iterador se utiliza para iterar sobre una abstracción utilizando, normalmente, un bucle de tipo `for`.

Algunos LPPOs (Java, JavaScript, Python) introducen el concepto de *objeto iterable*. Un objeto es **iterable** si define el comportamiento de su iteración; es decir, qué valores se obtienen al aplicar a éste un bucle `for`. Ciertos tipos de datos incorporan esta característica por defecto, por ejemplo, los *arrays* o cualquier otro tipo de dato para el que exista un iterador predefinido.

En los los LPPOs que no ofrecen soporte para la abstracción de iteración un iterador es un objeto de una abstracción de datos que mantiene el estado de la iteración en curso, también conocido como *objeto cursor*. Esta abstracción de datos se basa en el patrón de diseño **iterator** y éste, a su vez, en el patrón de diseño *memento*[3].

[3] *Memento* es un patrón de diseño cuya finalidad es almacenar el estado de un objeto (o del sistema completo) en un momento dado, de manera que se pueda restaurar en ese punto de manera sencilla.

Los lenguajes de programación que ofrecen soporte para la abstracción de iteración (por ejemplo, C#, Python y JavaScript) proporcionan un tipo especial de procedimiento denominado *generador* que permite crear un objeto generador y que es, también, un iterador. Un generador es una clase de corutina que produce uno a uno los ítems a iterar mediante una sentencia `yield` (equivalente del `return`). Cuando la corutina produce un ítem queda suspendida y no se reanuda hasta que el bucle `for` necesita el siguiente ítem, continuando en la sentencia que sigue al `yield`.

Para clarificar el uso del `yield`, supóngase que se define una función generadora, `genPrueba`, en la consola de Python y se invoca ésta para crear un iterador `g`, tal y como se muestra:

```
>>> def genPrueba():
...     print("Se inicia genPrueba")
...     yield 'A'
...     print("Continua genPrueba")
...     yield 'B'
...     print("Finalizando genPrueba")
...
>>> g = genPrueba()  # g es un generador y también un iterador
```

Si ahora se invoca la operación `next` para el iterador `g` varias veces, hasta que finaliza la función `genPrueba` (termine la iteración), en la consola se vería lo siguiente:

```
>>> next(g)
Se inicia genPrueba
'A'
>>> next(g)
Continua genPrueba
'B'
>>> next(g)
Finalizando genPrueba
Traceback (most recent call last):
  ...
StopIteration
>>>
```

La principal ventaja de obtener un iterador mediante una función generadora, es que en este caso no hay que hacer esfuerzo alguno para mantener el estado de la iteración, al contrario de lo que ocurre con un objeto cursor que requiere un ajuste de estado en cada paso de ésta. La propia función generadora se encarga de mantener el estado al quedar suspendida y reactivarse cuando se necesita. A modo de ejemplo, en la Figura 43 se define una función generadora para obtener los términos de la sucesión de Fibonacci y, a continuación, se crea un iterador que se utiliza en un bucle `for` para obtener los primeros veinte términos de la sucesión.

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci()
for i in range(20):
    print(next(fib))
```

Figura 43. Iteración con un iterador creado por invocación de una función generadora

Obsérvese que la función `fibonacci` de la Figura 43 genera una secuencia infinita de ítems y que la condición que establece la terminación de la iteración se proporciona en el bucle `for`. Pero también es posible establecer la condición de terminación en la propia función generadora, tal y como ocurre en la alternativa que se muestra en la Figura 44. En este último caso, no hay una condición explícita de parada en el `for` y se obtienen *todos los ítems* en la iteración invocándose de forma implícita la operación `next`.

```
def fibonacci(n):
    a, b = 0, 1
    while n > 0:
        n = n - 1
        yield a
        a, b = b, a + b

for i in fibonacci(20):
    print(i)
```

Figura 44. Bucle for-each en Python

3.1.1 Iteradores en Java

La biblioteca estándar de Java proporciona las interfaces `Iterable` e `Iterator`. Un objeto de una clase que implementa la interfaz `Iterable` es un *objeto iterable* y puede ser el destino de un bucle `for-each`. Un objeto de una clase que implementa la interfaz `Iterator` es un iterador.

3.1.1.1 La interfaz Iterable

Un bucle `for-each` proporciona una forma simple de iterar sobre todos los ítems de un objeto iterable, para ello se utiliza un iterador de forma implícita (sin exponer éste o sus operaciones). Este bucle no es una construcción propia del lenguaje Java, de forma que el compilador se encarga de transformarlo en otro equivalente haciendo explícita la presencia del iterador, tal y como se verá en el siguiente apartado.

Un *array* (que es un objeto) puede ser el destino de un bucle `for-each` pero, en general, para que un objeto contenedor `c` pueda ser destino del bucle `for-each` su tipo debe implementar la interfaz `Iterable`. En este sentido, un *array* es una excepción porque su tipo no implementa esta interfaz. La iteración permite tratar todos los objetos que contiene `c`, tal y como se muestra en la Figura 45.

```
for (Object o: c) {
    f(o);
}
```

Figura 45. Bucle `for-each` para cualquier contenedor

La interfaz `Iterable` especifica el método `iterator` (Figura 46), el cual retorna un objeto de una clase que implementa la interfaz `Iterator`; es decir, un objeto iterador sobre este contenedor.

```
/**
 * Retorna un iterador externo para este contenedor.
 * @return un iterador externo para este contenedor
 */
public Iterator iterator ();
```

Figura 46. Especificación del método `iterator` en `Iterable`

3.1.1.2 *La interfaz Iterator*

La abstracción de iteración no está directamente soportada en Java y por eso un iterador en Java es un *objeto cursor*. La interfaz `Iterator` específica, entre otros, los métodos `hasNext` y `next` (Figura 47).

```
/**
 * Retorna cierto si la iteración tiene más elementos.
 * @return cierto si la iteración tiene más elementos
 */
public boolean hasNext();

/**
 * Retorna el siguiente elemento en la iteración.
 * @return el siguiente elemento en la iteración
 * @throws NoSuchElementException si la iteración
 * no tiene más elementos
 */
public Object next();
```

Figura 47. Especificación de los métodos `hasNext` *y* `next` *de* `Iterator`

Estos métodos permiten construir fácilmente un bucle `for` o `while` tanto para tratar los ítems que proporciona el iterador como para localizar uno determinado. Por ejemplo, el bucle de la Figura 48 aplica una función `f` a todos los ítems que proporciona el objeto iterador `itr`.

```
while (itr.hasNext()) {
    f(itr.next())
}
```

Figura 48. Bucle para aplicar una función `f` *a los ítems que proporciona el iterador*

En cualquier caso, el bucle que controla la iteración siempre se realiza en el cliente y por eso de este tipo de iterador se dice que es un *iterador externo*.

Como ya se ha indicado en el apartado previo, el bucle `for-each` que permite iterar sobre todos los ítems de un objeto iterable, no es una construcción del lenguaje Java. Pero dado que sobre todo objeto iterable se requiere un iterador, el compilador utiliza éste para reemplazar el bucle `for-each` por un bucle `for` tradicional. Así, por ejemplo, el bucle de la Figura 45 se reemplaza internamente por el siguiente bucle:

```
for (Iterator itr = c.iterator(); itr.hasNext(); ) {
    Object o = itr.next();
    f(o);
}
```

Figura 49. Reemplazo interno del bucle for-each mediante un iterador

3.1.1.2.1 Ejemplo 1. La sucesión de Fibonacci

A continuación, y como un primer ejemplo, se realizará en Java el equivalente del programa de la Figura 44 para obtener los veinte primeros términos de la sucesión de Fibonacci. Para ello es necesario poder crear un objeto iterable que proporcione los términos de la sucesión que se soliciten y, por tanto, una clase (que se nombrará como `Fibonacci`) que implemente la interfaz `Iterable` y en la que se pueda instanciar éste (Figura 50). El objeto iterable únicamente necesita mantener como información (estado) el número de elementos de la sucesión que se han de generar. Y será el objeto de la clase iteradora asociada, el iterador, el que proporcionará uno a uno uno a uno cada elemento de la sucesión.

Lo habitual, es que la clase iteradora asociada se declare como clase interna en la clase de objetos iterables (la clase `Fibonacci`). Esta clase interna puede ser anónima o privada. Si la clase es anónima carece de constructores, en otro caso éstos se declararán como protegidos. La ventaja de hacer interna la clase iteradora es que ésta puede acceder a cualquier campo o método de la clase de objetos iterables, estén o no protegidos.

```
public final class Fibonacci implements Iterable {
    // número de términos a generar
    private int num;
    // número de términos por defecto
    private static final int DEFECTO = 10;

    Fibonacci() {
        this(DEFECTO);
    }

    Fibonacci(int n) {
        num = n;
    }

    public Iterator iterator() {
        return new Iterator() {  // clase anónima
            private long n = Fibonacci.this.num;
            private long a = 0;
            private long b = 1;

            public boolean hasNext() {
                return n > 0;
            }

            public Object next() {
                if (! hasNext()) {
                    throw new NoSuchElementException();
                }
                long current = a;
                a = b;
                b += current;
                n--;
                return current;
            }
        };
    }
}
```

Figura 50. Clase `Fibonacci`*. Instancia objetos iterables que generan la sucesión de Fibonacci*

La implementación de la clase iteradora interna se obtiene de forma inmediata a partir de la función generadora Python de la Figura 44. El estado a mantener es exactamente el mismo, el método `hasNext` se corresponde con la condición de continuación del bucle `while` en la función generadora y el método `next` con el cuerpo de éste que, eso sí, es necesario expresar de forma alternativa. Ahora es fácil escribir en el cliente un bucle `for-each` para obtener los veinte primeros términos de la sucesión de Fibonacci:

```
for (Object item: new Fibonacci(20)) {
    System.out.println(item);
}
```

Figura 51. Bucle `for-each`*. Muestra los 20 primeros términos de la sucesión de Fibonacci*

3.1.1.2.2 Ejemplo 2. Colección de puntos de una ruta

Como segundo ejemplo de uso de iteradores, e implementación de la clase correspondiente, se verá una clase de objetos contenedores. En particular, se utilizará el TDA `Punto` especificado en la Figura 35 para especificar e implementar un nuevo tipo de dato, el TDA `Ruta`. Una ruta será un objeto iterable que consta de una secuencia de puntos del plano cartesiano, la ruta comienza en un punto denominado *origen* y finaliza en otro que se nombrará como *destino*. Se considerará que una ruta no puede contener puntos repetidos.

```
public interface Ruta extends Iterable {

    /**
     * Retorna el número de puntos de la ruta.
     * @return el número de punto de la ruta
     */
    int puntosRuta();

    /**
     * Retorna cierto si esta ruta está vacía.
     * @return cierto si esta ruta está vacía
     */
    default boolean rutaVacia() {
        return puntosRuta() == 0;
    }

    /**
     * Retorna el punto de inicio de esta ruta.
     * @return el punto de inicio de la ruta
     * @throws IllegalStateException si la ruta está vacía
     */
    Punto origen();

    /**
     * Retorna el punto final de esta ruta.
     * @return el punto final de la ruta
     * @throws IllegalStateException si la ruta está vacía
     */
    Punto destino();

    /**
     * Retorna cierto si el punto especificado pertenece
     * a esta ruta.
     * @param p el punto especificado
     * @return cierto si el punto p está en la ruta y falso
     * en caso contrario
     */
    default boolean contiene(Punto p) {
        for (Object o: this) {
            Punto q = (Punto) o;
            if (q.equals(p)) {
              return true;
            }
        }

        return false;
    }
```

```
    /**
     * Añade el punto especificado a esta ruta si no
     * se encuentra en ésta
     * @param p el punto a añadir
     * @throws NullPointerException si p es null
     */
    void agregar(Punto p);

    /**
     * La posición del punto especificado en esta ruta.
     * Si el punto no está en esta ruta retorna -1.
     * @return la posición del punto en esta ruta. Si no
     * están en la ruta retorna -1
     */
    default int posicion(Punto p) {
        int pos = 0;
        for (Object o: this) {
            if (p.equals(o)) {
                return pos;
            }
            pos++;
        }

        return -1;
    }
}
```

Figura 52. TDA `Ruta`

Para implementar el TDA `Ruta` se utilizará como estructura de datos un *array* de puntos, éstos se almacenarán de forma contigua en el *array*. La capacidad inicial del *array* se proporcionará o tomará un valor por defecto. En el caso de que se alcance el número máximo de elementos del *array*, se redimensionará el espacio de almacenamiento. Además, es necesario un índice para conocer el número de puntos que hay en el *array* o, lo que es lo mismo, la primera posición libre de éste que se asignará al siguiente punto que se añada a la ruta.

La clase incluye dos constantes (campos estáticos), la primera establece la capacidad por defecto del *array* de puntos y la segunda es para evitar el desbordamiento en el caso de que se redimensione éste.

```
/**
 * Función de abstracción:
 * Abs: rep → Ruta
 *       r → (r.puntos[0], r.puntos[1], ⋯ , r.puntos[r.numPuntos − 1])
 * Invariante de la representación:
 * (∀i, j: 0 ≤ i < j < r.numPuntos: r.puntos[i] ≠ r.puntos[j]) ∧
 * (0 ≤ r.numPuntos < r.puntos.length)
 */
public class RutaImp implements Ruta {
    // capacidad por defecto del array
    private static final int CAPACIDAD_POR_DEFECTO = 10;
    // capacidad máxima del array
    private static final int MAX_DIM_ARRAY = Integer.MAX_VALUE - 8;
```

```
// array de puntos
private Punto[] puntos;
// número de puntos en el array
private int numPuntos;

public RutaImp() {
    this(CAPACIDAD_POR_DEFECTO);
}

public RutaImp(int capacidad) {
    if (capacidad <= 0) {
        throw new IllegalArgumentException();
    }
    puntos = new Punto[capacidad];
    numPuntos = 0;
}

public RutaImp(Ruta r) {
    if (r == null) {
          new NullPointerException();
    }
    puntos = new Punto[r.puntosRuta() + CAPACIDAD_POR_DEFECTO];
    numPuntos = 0;
    for (Object o: r) {
        agregar((Punto) o);
    }
}

public int puntosRuta() {
    return numPuntos;
}

public Punto origen() {
    if (puntosRuta() == 0) {
        throw new IllegalStateException("ruta vacía");
    }
    return puntos[0];
}

public Punto destino() {
    if (puntosRuta() == 0) {
        throw new IllegalStateException("ruta vacía");
    }
    return puntos[puntosRuta() – 1];
}

private void incrCapacidad() {
    if (puntos.length == MAX_DIM_ARRAY) { // overflow
        throw new OutOfMemoryError();
    }
    int nuevaCapacidad = puntos.length << 1; // duplica
    if (nuevaCapacidad < 0) { // supera Integer.MAX_VALUE
        nuevaCapacidad = MAX_DIM_ARRAY;
    }

    puntos = Arrays.copyOf(puntos, nuevaCapacidad);
}
```

```
public void agregar(Punto p) {
    if (p == null) {
        throw new NullPointerException();
    }
    if (numPuntos == puntos.length) {
          incrCapacidad();
    }
    puntos[numPuntos++] = p;
 }

@Override
public String toString() {
    String s = "[";
    Iterator itr = iterator();
    if (itr.hasNext()) {
        s += itr.next();
    }
    while (itr.hasNext()) {
        s += ", " + itr.next();
    }
    s += "]";
    return s;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + numPuntos;
    result = prime * result + Arrays.hashCode(puntos);
    return result;
}

@Override
public boolean equals(Object o) {
    if (this == o) {
          return true;
    }
    if (! (o instanceof Ruta)) {
          return false;
    }
    Iterator itr1 = iterator();
    Iterator itr2 = ((Ruta) o).iterator();
    while (itr1.hasNext() && itr2.hasNext()) {
        if (! itr1.next().equals(itr2.next())) {
            return false;
        }
    }
    return itr1.hasNext() == itr2.hasNext();
}
```

```
    public Iterator iterator() {
        return new RutaIterator();
    }

    // Clase interna RutaIterator
    private class RutaIterator implements Iterator {
        private int actual; // posición del elemento a iterar

        private RutaIterator() {
            actual = 0;     // posición del primer elemento
        }

        @Override
        public boolean hasNext() {
            return actual < RutaImp.this.numPuntos;
        }

        @Override
        public Object next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
             }
            Punto p = RutaImp.this.puntos[actual++];
            return p;
        }
    }
}
```

Figura 53. Implementación del TDA `Ruta`

Como ya se ha indicado previamente, lo habitual es incluir la clase de objetos iteradores como una clase interna de la clase de objetos iterables. También en este caso se ha hecho así, pero a diferencia del ejemplo previo la clase interna no es anónima. La implementación de la clase es muy simple, ya que los puntos de la ruta se almacenan en posiciones contiguas de un *array*. Por tanto, como estado sólo se requiere mantener el índice (`actual`) de la componente del *array* que tiene el siguiente punto a proporcionar.

Por último, señalar que cuando el estado de un objeto iterable consta de un *array* hay cierta inclinación a realizar *implementaciones incorrectas* cuando hay bucles, accediendo directamente a la estructura en lugar de utilizar un iterador. Por ejemplo, la siguiente implementación de la operación `toString` para la clase `RutaImp` sería incorrecta:

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

    if (! rutaVacia()) {
        s.append(puntos[0]);
    }

    for (int i = 1; i < puntosRuta(); i++) {
        s.append(", ").append(puntos[i]);
    }

    s.append("]");

    return s.toString();
}
```

Figura 54. Operación con bucle mal implementada

En ocasiones, una iteración puede depender de la estructura de la colección de una forma más sutil. Esto ocurre, por ejemplo, cuando en el código utilizado en la iteración no se accede directamente a la representación de la colección, pero alguno de los métodos utilizados no es aplicable (al menos de una forma razonablemente equivalente) a una colección cualquiera.

Por ejemplo, el TDA `Ruta` podría haber incorporado la especificación de la siguiente operación:

```
/**
 * Retorna el punto de ruta que ocupa la posición especificada.
 * @param indice posición del punto a localizar
 * @return el punto de ruta que ocupa la posición especificada
 * @throws IndexOutOfBoundsException si la posición proporcionada
 * está fuera de rango:
 * <code_HTML> (indice < 0) ∨ (indice ≥ puntosRuta()) </code_HTML>
 */
Punto valor (int indice);
```

y entonces, se podría implementar la operación `toString` como:

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

    if (! rutaVacia()) {
        s.append(puntos[0]);
    }

    for (int i = 1; i < puntosRuta(); i++) {
        s.append(", ").append(valor(i));
    }

    s.append("]");

    return s.toString();
}
```

Figura 55. Otra operación con bucle mal implementada

En la implementación de la Figura 55 no se accede directamente a la representación, sólo se utilizan operaciones del tipo de dato, pero al igual que en el bucle de la Figura 54 no se utilizan iteradores y es igualmente incorrecto. En este caso, el bucle `for` utilizado también depende de la representación de la colección en lo que respecta a su tiempo de ejecución. Al inicio del apartado 3, ya se trató esta problemática con un bucle similar y la operación `valor(c, i)`, que es la versión funcional del método `valor`. Ya se indicó entonces, que el método `valor` de acceso por índice tiene poco sentido en la mayor parte de las colecciones y que, en todo caso, no permite acceder de forma directa (o aleatoria) a los elementos de una colección salvo en casos muy concretos

La conclusión que se debe extraer de estos ejemplos es que *las iteraciones sobre objetos iterables deben realizarse siempre utilizando iteradores* (implícitos o explícitos).

Si se realizan algoritmos con bucles para objetos iterables sin utilizar iteradores, podrán obtenerse algoritmos que proporcionen los resultados esperados, pero a efectos del aprendizaje y uso de la abstracción de iteración son incorrectos y así se considerarán a todos los efectos. De una u otra forma, dichos algoritmos establecen una dependencia entre la iteración y la representación de la colección que es, precisamente, lo que se quiere evitar.

4 Jerarquía de tipos

La **jerarquía de tipos** permite definir un conjunto de tipos de datos que tienen un comportamiento similar: *todos ellos tienen ciertas operaciones en común y se comportan de forma análoga como resultado de invocar estas operaciones*. Los tipos pueden diferenciarse por extender o especializar el comportamiento de las operaciones que tienen en común o bien por proporcionar operaciones adicionales.

La raíz de la jerarquía de tipos es un tipo de dato cuya especificación define el comportamiento de las operaciones comunes a todos los tipos de ésta; es decir, establece las signaturas y el comportamiento de las operaciones comunes. Otros miembros de la jerarquía se definen como *subtipos* del tipo raíz, al cual se refieren como *supertipo*.

La jerarquía de tipos se puede extender a más de dos niveles, definiendo subtipos para los subtipos del tipo raíz y así sucesivamente. La extensión de la jerarquía puede deberse a diversas causas:

- Para proporcionar múltiples implementaciones del supertipo.
- Más comúnmente para extender el comportamiento de éste, por ejemplo, proporcionando métodos adicionales.

La utilidad de la jerarquía se basa en la relajación de las normas de enlace de las variables y del paso de argumentos en la invocación de las operaciones comunes de la jerarquía de tipos. Así, una variable declarada de un cierto tipo `T` puede referirse a un valor de alguno de sus subtipos `S` y ésta puede pasarse como argumento a operaciones que esperan una variable de tipo `T`. En una jerarquía de tipos una variable tiene dos tipos: el *aparente* y el *actual*. El *aparente* se corresponde con la declaración de la variable (o de un parámetro en la definción de una función) y el *actual* aquel al que realmente corresponde el valor de la variable (o el argumento en la invocación de la función). Este último siempre será un subtipo del tipo *aparente* (téngase en cuenta que cualquier tipo es subtipo de sí mismo) y el enlace entre el tipo actual y el valor de una variable (objeto) se produce en tiempo de ejecución (*enlace dinámico*).

4.1 Definición de un subtipo

Como ejemplo de extensión del comportamiento de un tipo de dato, se extenderá el tipo `RutaImp` para disponer de métodos adicionales, uno para calcular la longitud total de la ruta y otro para obtener la longitud de ruta entre dos puntos de ésta. El subtipo se nombrará como `RutaDist` (Figura 56).

```
public class RutaDist extends RutaImp {

    public RutaDist() {
        super();
    }

    public RutaDist(int capacidad) {
        super(capacidad);
    }

    public RutaDist(Ruta r) {
        super(r);
    }
```

```java
/**
 * La longitud de la subruta entre las posiciones especificadas.
 * @param posInicial posición de un punto de esta ruta
 * @param posFinal posición de un punto de esta ruta
 * @return la distancia de la subruta entre las posiciones
 * especificadas
 * @throws IndexOutOfBoundsException si las posiciones
 * especificadas están fuera de rango.
 **/
double longitudSubRuta(int posInicial, int posFinal) {
    if (posInicial < 0
            || posInicial > posFinal
            || posFinal >= puntosRuta()) {
        throw new IndexOutOfBoundsException();
    }

    Punto actual = null;
    int pos = -1;
    Iterator itr = iterator();
    double total = 0;

    // avanzar hasta la posición inicial
    while (pos != posInicial) {
        actual = (Punto) itr.next();
        pos++;
    }
    // acumular distancias hasta el punto en la posición final
    while (pos != posFinal) {
        Punto p = (Punto) itr.next();
        total += p.distancia(actual);
        actual = p;
        pos++;
    }

    return total;
}

/**
 * La longitud de la subruta entre los puntos especificados
 * de esta ruta.
 * @param p1 un punto de inicio en esta ruta
 * @param p2 un punto final en esta ruta
 * @return la longitud de ruta entre los puntos p1 y p2
 * @throws IllegalArgumentException si alguno de los puntos, p1
 * o p2, no pertenece a la ruta r.
 **/
double longitudSubRuta(Punto p1, Punto p2) {
    int indice1 = posicion(p1);
    int indice2 = posicion(p2);

    if (indice1 == -1) {
        throw new IllegalArgumentException(p1
                                           + "no está en la ruta");
    }
```

```
        if (indice2 == -1) {
            throw new IllegalArgumentException(p2
                                                + " no está en la ruta");
        }

        return (indice1 <= indice2
                ? longitudSubRuta(indice1, indice2)
                : longitudSubRuta(indice2, indice1));
    }

    /**
     * Retorna la longitud que cubre esta ruta.
     * @return la longitud de esta ruta. Esta longitud es la suma
     * de las distancias entre puntos consecutivos de la ruta
     */
    double longitudRuta() {
        return longitudSubRuta(0, puntosRuta() - 1);
    }
}
```

Figura 56. Extensión del tipo `RutaImp`

5 Abstracciones polimórficas

5.1 Polimorfismo

Ad-Hoc polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type. Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure. (Strachey 1967)

El **polimorfismo** se refiere a la multiplicidad de significados asociados con un nombre. Para ello se utiliza la distinción entre el nombre y un tipo que asocia cada significado diferente con un tipo distinto.

El concepto de polimorfismo fue introducido en la década de los 60. Christopher Strachey (1967) distinguió de modo informal dos clases de polimorfismo:

- **Polimorfismo ad-hoc**. Se produce cuando una función se comporta de forma distinta para diferentes tipos de argumentos.
- **Polimorfismo paramétrico**. Se produce cuando una función se comporta de manera uniforme sobre un rango de tipos (que comparten una estructura común) de argumentos.

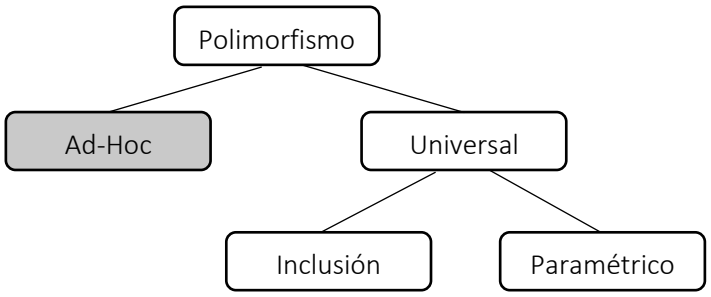

Figura 57. Clases de polimorfismo

Cardelli y Wegner (L. CARDELLI, 1985) redefinieron la clasificación de Strachey para introducir una nueva forma de polimorfismo inherente a los lenguajes de programación orientados a objetos: el **polimorfismo de inclusión**. Ellos establecieron la siguiente clasificación (Figura 57):

- **Polimorfismo ad-hoc**. Se produce cuando una función se comporta de forma diferente para un conjunto finito de tipos distintos no relacionados.
- **Polimorfismo universal**. Se produce cuando una función se comporta de manera uniforme para un conjunto potencialmente infinito de tipos distintos que comparten una estructura común. Este a su vez consta de dos subcategorías: *polimorfismo de inclusión* y *polimorfismo paramétrico*. Ambas subcategorías suelen estar relacionadas, pero son suficientemente distintas en teoría y en la práctica como para denominarlas de forma distinta.

5.1.1 Polimorfismo ad-hoc

El polimorfismo ad-hoc sólo es un *polimorfismo aparente* y un estudio más detallado del mismo hace que el carácter polimórfico desaparezca. Están aquí presentes la **coerción** y la **sobrecarga**.

La **coerción** es una operación semántica que evita un error de tipo. El compilador convierte el tipo de un argumento en una llamada a la función para adecuarlo al tipo de parámetro aceptado

por ésta. La conversión de tipos puede ser implícita (por ejemplo, entre enteros o reales) o explícita (*casting*). En este caso la función permite únicamente un tipo determinado (no varios).

La **sobrecarga** es una abreviación sintáctica que asocia un nombre de función con varias definiciones de funciones (cambia la signatura). Cada función trabaja únicamente sobre su conjunto de tipos de argumentos y el número de conjuntos posibles está limitado al número de definiciones proporcionadas (varias definiciones de funciones con el mismo nombre, pero cada una de ellas admite únicamente su conjunto de tipos de argumentos).

5.1.2 Polimorfismo universal

El polimorfismo universal es el verdadero polimorfismo y la base de las *abstracciones polimórficas*. El rango de tipos aceptados no está restringido y expone un comportamiento uniforme: *la misma función trabaja con tipos diferentes*.

5.1.3 Polimorfismo de inclusión

El **polimorfismo de inclusión** trabaja sobre una jerarquía de tipos (relacionados mediante la herencia) que comparten métodos con una signatura común (en caso contrario estaríamos ante un caso de sobrecarga denominado **redefinición**). Un método invocado sobre un objeto que tiene diferentes tipos de la jerarquía durante su tiempo de vida puede tener definiciones diferentes (no es necesariamente única). Cada definición diferente es una **sobreescritura** (*overriding*) del método en un subtipo. La sobreescritura puede suponer un reemplazo del comportamiento (método independiente del supertipo, igual que la sobrecarga) o un **refinamiento** (método no independiente).

Este tipo de polimorfismo se basa en la distinción entre el tipo estático (o aparente) y el tipo dinámico (o actual) de un objeto. Como un mismo método puede tener distintas definiciones en la jerarquía de tipos, es necesario discriminar el método a invocar cuando se envía un mensaje a un objeto: *el método que se invoca se determina siempre por el tipo dinámico del objeto*.

5.1.3.1 Polimorfismo de inclusión en Java

En los LPPOs el polimorfismo de inclusión permite realizar funciones y abstracciones de datos declarando los objetos como pertenecientes a la raíz de la jerarquía de tipos (relacionados por medio de la herencia).

En el lenguaje Java la raíz de la jerarquía de tipos es la clase `Object` y todos los objetos son de algún subtipo de esta clase. Por lo que podemos realizar abstracciones polimórficas declarando variables o los parámetros de funciones como de tipo `Object`.

5.1.3.1.1 Ejemplo de abstracción polimórfica

A modo de ejemplo realizaremos un tipo de dato, `Bolsa`, que tiene capacidad para guardar hasta diez objetos de cualquier tipo. El área de datos es un *array* de `Object` y como éste es la raíz de la jerarquía de objetos en Java y todos los objetos heredan de `Object`, un objeto de tipo `Bolsa` puede contener objetos de cualquier tipo. Sin embargo, lo más habitual, es que las aplicaciones que requieran el uso del tipo `Bolsa` necesiten limitar el tipo de objetos a uno determinado (o bien a algún subtipo de éste). Restricción ésta que no se puede garantizar mediante el polimorfismo de inclusión.

Por otra parte, el uso de una abstracción de datos como `Bolsa` (Figura 58), requiere realizar conversiones explícitas del tipo de dato del objeto guardado en ella y los errores que se puedan cometer sólo se detectan en tiempo de ejecución.

```
public class Bolsa {
    private static final int CAPACIDAD = 10;
    private Object[] datos;
    private int numDatos;

    public Bolsa() {
        datos = new Object[CAPACIDAD];
        numDatos = 0;
    }

    public void agregar(Object o) {
        if (o == null) {
            throw new NullPointerException();
        }

        if (numDatos == CAPACIDAD) {
            throw new OutOfMemoryError();
        }

        datos[numDatos++] = o;
    }

    public Object sacar(int indice) {
        if (indice < 0 || indice >= numDatos) {
            throw new IndexOutOfBoundsException();
        }

        return datos[indice];
    }

}
```

Figura 58. Ejemplo de polimorfismo de inclusión.

Así, en el programa de ejemplo de la Figura 59 se observa que al recuperar un objeto de la `Bolsa` es necesario realizar un *casting* porque su tipo es `Object`. Además, si la conversión es incorrecta (como así ocurre), el error no se detecta hasta el momento de la ejecución.

```
public class MainBolsa {

    public static void main(String[] args) {
        Bolsa b = new Bolsa();

        b.agregar("Hola");
        b.agregar(100);
        // Error en tiempo de ejecución
        String s = (String) b.sacar(1);
    }

}
```

Figura 59. Programa de prueba para la abstracción `Bolsa`

5.1.4 Polimorfismo paramétrico

El **polimorfismo paramétrico** trabaja sobre funciones que comparten una estructura común mediante un número potencialmente infinito de tipos no relacionados. Una función polimorfa tiene uno o más parámetros de tipo implícito o explícito que determina el tipo del argumento

para cada invocación de la función. Las funciones y abstracciones de datos que exponen polimorfismo paramétrico también se denominan funciones y tipos de datos **genéricos**, respectivamente.

El polimorfismo paramétrico es la forma más pura de polimorfismo: el mismo objeto o la misma función se puede utilizar uniformemente en contextos de tipos diferentes sin cambios, conversiones de tipo, ni necesidad de clase alguna de comprobación en tiempo de ejecución o de codificación especial. Sin embargo, esta uniformidad de comportamiento requiere que todos los datos estén representados, o de algún modo tratados, uniformemente.

5.1.4.1 Polimorfismo paramétrico en Java

Java soporta el polimorfismo paramétrico a partir de su versión 5, en la que se introdujeron los **Generics**. Si bien con algunas particularidades debidas al mantenimiento de la compatibilidad hacia atrás en la máquina virtual Java y que merman algo su potencial.

En Java un tipo de dato polimórfico (o *tipo genérico*), es una clase cuyo nombre incluye como sufijo los *parámetros de tipo* encerrados entres los caracteres '<' y '>' y separados por comas:

```
public class Nombre_de_clase<T1, T2, . . ., Tn> { /* . . . */ }
```

El uso del polimorfismo paramétrico en Java requiere del uso de uno o más tipos dados como parámetros que, posteriormente, se sustituirán por un argumento de tipo (`Object` o uno de sus subtipos).

En las funciones el *parámetro de tipo* se sustituye por el tipo del argumento en el momento de la llamada. En los tipos de datos que son colecciones de elementos del *parámetro de tipo*, éste se establece al instanciar las colecciones en las aplicaciones cliente.

Si en lugar de una clase se define una interfaz genérica se proporciona un tipo de dato abstracto polimórfico.

Por convenio, los nombres de los parámetros son simplemente una letra mayúscula. Los más utilizados son los siguientes:

- `E` – Elemento (de uso frecuente en la biblioteca de colecciones de Java)
- `K` – Clave (*Key*)
- `N` – Número
- `T` – Tipo
- `V` – Valor
- `S`, `U`, `V` – como segundo, tercer y cuarto *parámetro de tipo*

5.1.4.1.1 Ejemplo de abstracción polimórfica

Para poder comparar esté método de polimorfismo con el visto en el apartado 5.1.3.1.1, realizaremos una abstracción de datos equivalente y genérica: `Bolsa<T>`. Donde la información que se almacena en la bolsa será de un tipo `T` genérico. El tipo concreto de éste se establecerá en el programa cliente al instanciar la colección (un objeto de `Bolsa<T>`).

El tipo polimórfico `Bolsa<T>` de la Figura 60 es muy similar a la clase `Bolsa` de la Figura 58, sus diferencias son mínimas:

1. El nombre de la clase, que ahora incluye el *parámetro de tipo* `T` entre los caracteres `<` y `>`.

2. El estado o representación de los objetos es la misma, la única diferencia es que ahora el *array* es de elementos de tipo `T` en lugar de `Object`.
3. Cambia la signatura (o perfil) de algunos métodos. En concreto, los que incluyan parámetros (o tipo de retorno) que esté declarados como de tipo `Object` en la clase `Bolsa`, que ahora se declaran mediante el *parámetro de tipo* `T`. Esto afecta a los métodos `agregar` y `sacar`.

Seguramente lo que puede resultar más extraño es la forma en la que se crea el *array* `datos` en el constructor, que no se corresponde con la forma habitual: `datos = new T[CAPACIDAD]`. Nos referiremos a este punto en el siguiente apartado.

```
public class Bolsa<T> {
    private static final int CAPACIDAD = 10;
    private T[] datos;
    private int numDatos;

    @SuppressWarnings("unchecked")
    public Bolsa() {
        datos = (T[]) new Object[CAPACIDAD];
        numDatos = 0;
    }

    public void agregar(T e) {
        if (e == null) {
            throw new NullPointerException();
        }

        if (numDatos == CAPACIDAD) {
            throw new OutOfMemoryError();
        }

        datos[numDatos++] = e;
    }

    public T sacar(int indice) {
        if (indice < 0 || indice >= numDatos) {
            throw new IndexOutOfBoundsException();
        }

        return datos[indice];
    }
}
```

Figura 60. Ejemplo de polimorfismo paramétrico

Para utilizar el tipo de dato genérico `Bolsa<T>` el cliente debe realizar una invocación a éste, de forma que se reemplaza el parámetro de tipo `T` por un argumento de tipo de objetos. *El tipo del argumento no puede ser un tipo nativo del lenguaje Java* (`int`, `double`, etc.), pero si puede ser a su vez otra invocación de un tipo genérico.

La instanciación de un objeto de tipo genérico requiere el uso del operador `new` con un constructor, en la forma habitual. Lo único es que el constructor se invoca con un nombre que se corresponde con el nombre de la clase y, por consiguiente, también incluye los *argumentos de tipo* encerrados entre los caracteres < y > separados por comas. Ejemplos:

```
Bolsa<Integer> bi;          // que se lee como "Bolsa de Integer"
Bolsa<Bolsa<String>> bs;    // que se lee como "Bolsa de Bolsa de String"
```

Desde la versión 7 de Java, se incluye un mecanismo de inferencia de tipo que permite obviar los *argumentos de tipo* en el constructor (es suficiente con incluir los caracteres delimitadores). Así, por ejemplo, las siguientes instanciaciones son igualmente válidas:

```
Bolsa<Integer> a = new Bolsa<Integer>();
Bolsa<String> b = new Bolsa<>("Hola");
```

Ahora en un programa `MainBolsa` análogo al dado para el polimorfismo de inclusión en la Figura 59, no se requiere conversión explícita de tipos (*casting*) y se puede detectar el error en tiempo de compilación (ver Figura 61).

```
public class MainBolsa {

    public static void main(String[] args) {
        Bolsa<String> b1 = new Bolsa<>();
        Bolsa<Integer> b2 = new Bolsa<>();

        b1.agregar("Hola");
        b2.agregar(100);
        // Error en tiempo de compilación
        String s = b2.sacar(0);
    }

}
```

Figura 61. Programa de prueba para la abstracción `Bolsa<T>`

5.1.4.1.2 Restricciones de los tipos genéricos de Java

Por razones de compatibilidad con versiones previas y para evitar la sobrecarga de tiempo de ejecución de los tipos genéricos, el compilador borra los parámetros y los argumentos de tipo en la máquina virtual Java (JVM) produciendo lo que se denomina el tipo *raw* (`Bolsa`). Es decir; en la JVM todo son clases, interfaces y métodos normales.

Este mecanismo de borrado de tipos también introduce otras medidas para preservar la seguridad de tipos y el polimorfismo de los tipos genéricos, pero más que conocer este mecanismo con detalle, lo que interesa es conocer las restricciones que conlleva. Éstas son las siguientes:

- No se pueden instanciar tipos genéricos con tipos primitivos
- No se pueden crear instancias de *parámetros de tipo*
- No se pueden declarar campos estáticos (`static`) de *parámetros de tipo*
- No se puede invocar el operador `instanceof` con tipos que contengan *parámetros de tipo*
- No se pueden crear *arrays* de tipos que contengan *parámetros de tipo*
- No se pueden crear, capturar o lanzar excepciones de objetos de tipos que contengan *parámetros de tipo*

- No se puede sobrecargar un método cuando los tipos de sus parámetros dan lugar a la misma signatura de tipos *raw*

Una de las limitaciones del borrado de tipo establece que no se pueden crear *arrays* de elementos de *parámetros de tipo*. Esto explica porque en la clase `Bolsa<T>` (Figura 60) no se creó el *array* `datos` mediante la asignación habitual: `datos = new T[CAPACIDAD]`.

Dado que el parámetro de tipo `T` se sustituirá por un tipo de objetos cualquiera (tampoco se pueden instanciar parámetros de tipo con tipos primitivos), la alternativa es utilizar el tipo `Object` para crear el *array*, ya que todos los demás tipos son subtipos de éste:

```
new Object[CAPACIDAD]
```

Pero tampoco es posible asignar directamente `Object[]` al campo `datos` que está declarado como `T[]` (igual que no se puede asignar un objeto de tipo `Object` a, por ejemplo, una variable de tipo `String`). Para poder hacer la asignación es necesario hacer una conversión de tipo explícita (*casting*), en la forma indicada:

```
datos = (T[]) new Object[CAPACIDAD]
```

Por último, debe señalarse que el borrado de tipo en la JVM y el uso de la clase `Object`, que es la raíz de la jerarquía de objetos en Java, nos han permitido tratar y utilizar algunas interfaces parametrizadas como si no lo fueran, pero *a partir de ahora se utilizarán como tales*.

> Las siguientes interfaces de la biblioteca de Java, ya tratadas en apartados previos, son en realidad *tipos de datos abstractos polimórficos*: `Comparable<T>`, `Comparator<T>`, `Iterable<T>` e `Iterator<T>`.

Como norma general, la especificación de los métodos que se vieron para estas interfaces sigue siendo válida, únicamente es necesario sustituir los parámetros o resultados de tipo `Object` por un *parámetro de tipo* (`T`). Por ejemplo, se verá a continuación como sería la clase `Bolsa<T>` de objetos iterables, al implementar la interfaz `Iterable<T>`.

```
public class Bolsa<T> implements Iterable<T> {
    private static final int CAPACIDAD = 10;
    private T[] datos;
    private int numDatos;

    @SuppressWarnings("unchecked")
    public Bolsa() {
        datos = (T[]) new Object[CAPACIDAD];
        numDatos = 0;
    }

    public void agregar(T e) {
        if (e == null) {
            throw new NullPointerException();
        }
        if (numDatos == CAPACIDAD) {
            throw new OutOfMemoryError();
        }
        datos[numDatos++] = e;
    }
```

```
    public T sacar(int indice) {
        if (indice < 0 || indice >= numDatos) {
            throw new IndexOutOfBoundsException();
        }
        return datos[indice];
    }

    public Iterator<T> iterator() {
        return new Iterator<T>() {
            private int actual = 0;

            public boolean hasNext() {
                return actual < Bolsa.this.numDatos;
            }

            public T next() {
                if (! hastNext()) {
                    throw new
                }
                return Bolsa.this.datos[actual++];
            }
        };
    }
}
```

Figura 62. TDA `Bolsa<T>` *de objetos iterables*

Obsérvese que prácticamente no hay diferencias con otras clases de objetos iterables implementadas en apartados previos (por ejemplo, la clase `RutaImp` de la Figura 53). Únicamente se ha añadido el parámetro `T` a los tipos `Iterable<T>` e `Iterator<T>` y ahora el método `next` del iterador en lugar de retornar un objeto de tipo `Object` retorna un objeto de tipo `T`.

5.1.4.1.3 Subtipos de abstracciones polimórficas

Como en Java todos los objetos heredan de `Object` y están relacionados jerárquicamente, cabe preguntarse si dicha relación se puede extender a un contenedor genérico como `Bolsa<T>`, siendo ésta la clase de objetos iterables implementada en la Figura 62.

Si la relación de herencia se pudiera extender al contenedor genérico, se podría realizar la siguiente declaración:

```
Bolsa<Object> b = new Bolsa<Integer>(10);
```

Sin embargo, la línea de código previa provoca un error. En caso contrario, como `b` está declarado como `Bolsa<Object>` se podría hacer lo siguiente:

```
b.agregar(new Object());
Integer n = b.sacar();      // asignación de un Object a un Integer
```

Y se estaría asignando un `Object` a una variable de tipo `Integer`, lo que no es posible.

> En general, si `A` es un subtipo de `B` y `C<T>` es un tipo genérico, `C<A>` no es un subtipo de `C<B>`.

5.1.4.1.4 Comodínes (*wildcards*)

De lo indicado en el apartado previo podría deducirse que no es posible definir abstracciones funcionales que trabajen con colecciones de objetos cualesquiera. Supóngase, por ejemplo, que se desea realizar una función de clase (estática) para imprimir el contenido de cualquier clase de `Bolsa`. Seguramente llegaríamos a una solución similar a la siguiente:

```
public static void imprimirTodo(Bolsa<Object> b) {
    for (Object o: b) {
        System.out.println(o.toString());
    }
}
```

Figura 63. Función sobre objetos de una colección de objetos cualesquiera

Pero como acabamos de ver en el apartado previo, no es posible invocar este método con un argumento que no sea de tipo `Bolsa<Object>`, ya que éste no es un *supertipo* de todas las clases de `Bolsa`. También se nos podría ocurrir definir una función genérica, como la que se indica a continuación:

```
public static <T> void imprimirTodo(Bolsa<T> b) {
    for (Object o: b) {
        System.out.println(o.toString());
    }
}
```

Figura 64. Función genérica sobre objetos de una colección de objetos de tipo `T`

La función genérica de la Figura 64 trabaja para cualquier tipo de `Bolsa`. Sin embargo, el parámetro de tipo `T` no influye (no aparece) en el cuerpo de su definición, por lo que cabe preguntarse si éste se puede obviar, por ejemplo, utilizando el tipo *raw*:

```
public static void imprimirTodo(Bolsa b) {
    for (Object o: b) {
        System.out.println(o.toString());
    }
}
```

Figura 65. Función sobre objetos de una colección de tipo raw

Ciertamente, la función de la Figura 65 trabaja para cualquier tipo de `Bolsa`, no obstante, el uso de tipos *raw* está desaconsejado y el compilador Java como mínimo avisará al respecto (*warning*).

> El uso del tipo de dato *raw* que surge del borrado de parámetros de tipo en la JVM está desaconsejado y **no se debe utilizar** en los programas que se realicen.

La advertencia previa pone en entredicho algunos apartados iniciales de este documento, en los que se han tratado algunos tipos de datos polimórficos obviando los parámetros de tipo y utilizando el tipo *raw*. Tengáse en cuenta que esto se ha hecho así para poder introducir ciertos conceptos o temas y obviar otros que se han dado posteriormente (como el polimorfismo paramétrico) y que, de otro modo, ofuscarían los primeros. Ahora que ya se han visto todos ellos, ya no hay ningún motivo que justifique utilizar en los programas un tipo *raw* que está desaconsejado.

Finalmente, para definir funciones análogas a la de la Figura 65, pero sin utilizar el tipo *raw*, se incluyó la notación con comodín: `Bolsa<?>`, que indica que los elementos de la colección son de tipo desconocido (o de un tipo cualquiera). Éste es el *supertipo* de todas las clases de `Bolsa`.

Para definir una abstracción funcional sobre una colección `C` de objetos cualesquiera se debe utilizar la notación con comodín: **`C<?>`**.

Ahora se puede reescribir correctamente la función `imprimirTodo` de la Figura 63 como:

```
public void imprimirTodo(Bolsa<?> b) {
    for (Object o: b) {
        System.out.println(o.toString());
    }
}
```

Figura 66. Función sobre tipos de bolsa cualesquiera (solución)

Cuando se utiliza el tipo comodín se debe tener presente que éste representa un tipo de dato desconocido (o cualquier tipo de dato) y que no se puede añadir elementos a la colección si no se sabe de qué tipo son éstos. Por tanto, el comodín sólo se puede utilizar para realizar funciones de recuperación de información como, por ejemplo, la función de la Figura 66. También se puede utilizar en la declaración de instancias porque el tipo queda determinado por la llamada al constructor.

5.1.4.1.4.1 Comodínes delimitados

Aunque el comodín, `'?'`, representa un tipo de dato cualquiera, en ocasiones puede resultar útil restringir los tipos válidos a una jerarquía determinada.

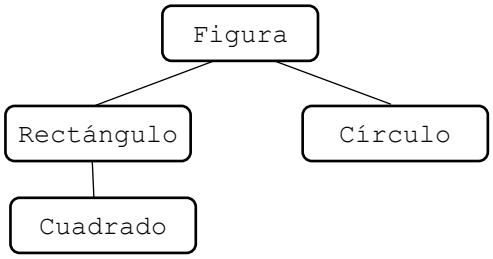

Figura 67. Jerarquía de figuras geométricas

Supóngase, por ejemplo, que estamos realizando una aplicación sobre figuras geométricas que están relacionadas jerárquicamente según se muestra en la Figura 67. Se podría, por ejemplo, querer realizar una función para obtener el área total de todas las figuras pertenecientes a una colección genérica como `Bolsa<T>`. Es decir, para uno cualquiera de los siguientes tipos: `Bolsa<Figura>`, `Bolsa<Rectángulo>`, `Bolsa<Cuadrado>` y `Bolsa<Círculo>`. Y de no ser uno de estos que se produzca un error. Si el TDA `Figura` especifica los métodos siguientes:

```
public interface Figura {
    /**
     * Retorna el centro de esta figura.
     * @return el centro de esta figura
     */
    Punto centro();
     /**
     * Retorna el área de esta figura.
     * @return el área de esta figura
     */
    double area();
}
```

Figura 68. TDA `Figura`

Se puede utilizar el comodín, tal y como se vio en el apartado previo, y obtener fácilmente una función para calcular el área total de las figuras de una bolsa. Por ejemplo, mediante la función que se muestra en la Figura 69. El problema de esta función es que se puede invocar con una bolsa de cualquier tipo de objeto y, para varios de éstos carece de sentido. Así, para una bolsa de tipo `Bolsa<String>` se produce un error en la ejecución porque no se puede realizar la conversión de `String` a `Figura`. Por otra parte, el método `area` no es aplicable a objetos que no sean figuras, mismamente, a las cadenas de caracteres.

```
public static double areaTotal(Bolsa<?> b) {
    double total = 0;
    for (Object o: b) {
        Figura f = (Figura) o;
        total += f.area();
    }
    return total;
}
```

Figura 69. Área total de las figuras de una bolsa

En estos casos, se puede delimitar el comodín para restringir la función a la jerarquía de objetos de interés (las figuras). La limitación se puede aplicar a un tipo y sus subtipos mediante la palabra reservada **`extends`** y a un tipo y sus supertipos mediante la palabra reservada **`super`**.

```
public static double areaTotal(Bolsa<? extends Figura> b) {
    double total = 0;
    for (Figura f: b) {
        total += f.area();
    }
    return total;
}
```

Figura 70. Función con comodín delimitado

En el caso de la función `areaTotal` interesa delimitar el comodín al tipo `Figura` y sus subtipos, tal y como se muestra en la Figura 70. Además, al delimitar el comodín en la forma indicada, ya no es necesario realizar la conversión de tipo porque en el bucle `for-each` se puede utilizar directamente un objeto de tipo `Figura`, en lugar de tipo `Object`. Obsérvese, que los tipos `Bolsa<?>` y `Bolsa<? extends Object>` son equivalentes, dado que el tipo `Object` es la raíz de la jerarquía de objetos en Java.

En el caso previo, el límite al comodín se ha impuesto por la parte superior. Como ya se ha indicado, también se puede establecer un límite por la parte inferior utilizando la notación `Bolsa<? super B>`. Incluso es posible delimitar simultáneamente ambos extremos (superior e inferior) mediante la notación `Bolsa<A extends B>`, que restringe los tipos válidos para los elementos de la colección a los subtipos de `A` que, además, son supertipos de `B`.

5.2 Clases de abstracciones polimórficas

El polimorfismo generaliza las abstracciones de modo que puedan trabajar para varios tipos. Éste se puede utilizar en tres de las clases de abstracciones: *abstracción funcional*, *abstracción de datos* y *abstracción de iteración*.

Una abstracción funcional o de iteración puede ser polimórfica con respecto a los tipos de uno o más de sus argumentos. Una abstracción de datos puede ser polimórfica con respecto a los tipos de sus elementos.

6 La biblioteca estándar de Java

En este apartado se describirá el *framework* de colecciones (o contenedores) incluido en la biblioteca estándar de Java: *The Java Collections Framework*. Ésta es una arquitectura unificada para representar y manipular colecciones que consta de los siguientes componentes:

- **Interfaces**. Tipos de datos abstractos jerarquizados que representan colecciones.
- **Implementaciones**. Implementaciones concretas o parciales de los tipos de datos abstractos (interfaces). Esencialmente, son estructuras de datos reutilizables.
- **Algoritmos**. Operaciones genéricas útiles para colecciones (búsqueda, ordenación, etc.).

Las abstracciones proporcionadas por la librería utilizan el polimorfismo paramétrico: los TDAs son colecciones de elementos de un tipo cualquiera y los algoritmos son abstracciones funcionales reutilizables.

El uso del *framework* reduce el esfuerzo de programación, aumenta la velocidad y la calidad del programa, permite la interoperabilidad entre las APIs relacionadas y fomenta la reutilización del software.

6.1 Interfaces

El núcleo de la biblioteca consta de interfaces que encapsulan diferentes tipos de colecciones organizadas jerárquicamente (ver Figura 71). Contienen un conjunto o multiconjunto[4] de objetos que, en algunos casos, pueden mantener un cierto orden. Algunas de estas colecciones están indexadas de forma asociativa; es decir, son estructuras de búsqueda que actúan como funciones aplicando valores índices (*claves*) en otros datos (`Map`).

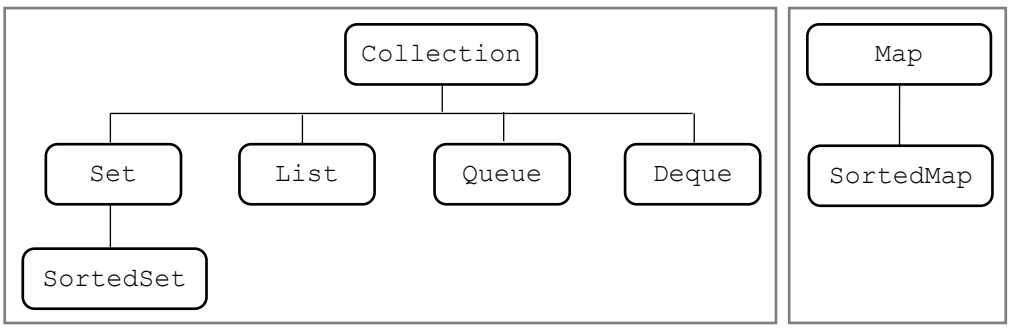

Figura 71. Interfaces de colecciones

La jerarquía se compone de dos árboles distintos porque un `Map` no es realmente una colección. Además, se recuerda que las interfaces son genéricas. Por ejemplo, esta es la declaración de la interfaz `Collection`:

```
public interface Collection<E>...
```

donde `E` es el tipo genérico (o parámetro de tipo) de los elementos de la colección.

Para mantener un número manejable de interfaces, Java no proporciona interfaces separadas para cada variante del tipo de objeto contenido en la colección (tipos de datos modificables o no modificables, de tamaño fijo, etc.). A tal fin, las operaciones modificadoras de la interfaz se designan como **opcionales** y las implementaciones de una interfaz pueden optar por soportar o

[4] Un multiconjunto (*bag*) es como un conjunto, pero en el que los elementos se pueden repetir. Es decir, cada miembro del multiconjunto tiene una multiplicidad: *el número de veces que se repite*.

no tales operaciones. Éstas últimas son las responsables de documentar que operaciones opcionales soporta. Si un cliente invoca una operación opcional no soportada la colección lanza la excepción `UnsupportedOperationException`.

Se describe a continuación brevemente el núcleo de las interfaces de colecciones:

- `Collection`. Es la raíz de la jerarquía de colecciones. Una colección representa un grupo de objetos denominados *elementos*. No se proporciona una implementación directa de esta interfaz, pero si de subinterfaces más específicas (como, por ejemplo, `Set` y `List`).
- `Set`. Es una colección que no puede contener elementos duplicados; es decir, se utiliza para representar conjuntos.
- `List`. Es una colección ordenada que puede contener elementos duplicados, también conocida como secuencia. Los elementos de la colección tienen un orden específico que es independiente de su valor. Se pueden realizar operaciones de inserción, extracción y consulta para una posición o índice dado.
- `Queue`. Colección de elementos que ordena los elementos para su posterior procesamiento. Las colas habitualmente ordenan los elementos de forma FIFO (*first-in-first-out*) pero hay excepciones. Por ejemplo, las colas de prioridad ordenan los elementos con respecto a sus valores.
- `Deque`. Colección lineal que soporta operaciones de inserción, extracción y consulta en ambos extremos, por lo que se pueden utilizar como colas FIFO (*first-in-first-out*) y pilas LIFO (*last-in-first-out*).
- `Map`. Un objeto que mapea claves a valores. Las claves no se pueden duplicar y a una clave se le puede asignar a lo sumo un valor.
- `SortedSet`. Un `Set` que mantiene ordenados sus elementos en orden ascendente. En este caso el orden si depende del valor de los elementos.
- `SortedMap`. Un `Map` que mantiene ordenadas sus asignaciones clave-valor por orden ascendente de la clave.

6.1.1 La interfaz `Collection<E>`

Es la raíz de la jerarquía de colecciones y extiende la interfaz `Iterable<E>`. Una colección representa un grupo de objetos iterables denominados *elementos*. Algunas colecciones permiten elementos duplicados y otras no. No se proporciona una implementación directa de esta interfaz, pero si se proporciona para otras más específicas como `Set<E>` y `List<E>`.

Esta interfaz se utiliza para pasar colecciones y manipularlas cuando se quiere la máxima generalidad. Por ejemplo, por convenio, todas las implementaciones de colecciones de propósito general incluyen un *constructor de conversión* que toma como argumento un objeto de tipo `Collection`. Este constructor crea e inicializa una nueva colección que contiene todos los elementos de la colección especificada. El tipo dinámico de la nueva colección puede ser el mismo o no que el correspondiente al argumento, si es el mismo se realiza una copia de éste (*constructor de copia*) y si no se realiza una conversión entre subtipos de `Collection` (*constructor de conversión*).

Como es sabido las interfaces no tienen constructores, ya que son tipos de datos abstractos y, por tanto, no representan tipos específicos de objetos concretos. En consecuencia, no se pueden crear directamente instancias de `Collection<E>`, pero si se pueden escribir funciones utilizando las operaciones especificadas en la interfaz. Estas funciones trabajarán para todas las implementaciones de `Collection<E>`.

Resumen de métodos	
`boolean`	`add(E e)` Asegura que la colección contiene el elemento especificado (*opcional*).
`boolean`	`addAll(Collection<? extends E> c)` Añade a la colección todos los elementos de la colección especificada (*opcional*).
`void`	`clear()` Elimina todos los elementos de la colección (*opcional*).
`boolean`	`contains(Object o)` Retorna `true` si la colección contiene el elemento especificado.
`boolean`	`containsAll(Collection<?> c)` Retorna `true` si la colección contiene todos los elementos de la colección especificada.
`boolean`	`equals(Object o)` Compara si el objeto especificado y la colección son iguales.
`int`	`hashCode()` Retorna el valor del código *hash* para la colección.
`boolean`	`isEmpty()` Retorna `true` si la colección no tiene elementos.
`Iterator<E>`	`iterator()` Retorna un iterador sobre los elementos de la colección.
`boolean`	`remove(Object o)` Elimina de la colección el elemento especificado (*opcional*).
`boolean`	`removeAll(Collection<?> c)` Elimina de la colección todos los elementos que también estén contenidos en la colección especificada (*opcional*).
`boolean`	`retainAll(Collection<?> c)` Elimina de la colección todos los elementos que no estén contenidos en la colección especificada (*opcional*).
`int`	`size()` Retorna el número de elementos de la colección.
`Object[]`	`toArray()` Retorna un *array* con todos los elementos de la colección.
`<T> T[]`	`toArray(T[] a)` Retorna un *array* con todos los elementos de la colección; el tipo del array retornado es el del *array* especificado. Si la colección cabe en el array especificado, ésta se ubica en él. En otro caso se retorna un nuevo *array* del tamaño de la colección.

Tabla 1. Interfaz `java.util.Collection`

La Tabla 1 recoge una parte significativa de los métodos aplicables al TDA `Collection<E>`, junto con una breve descripción de su comportamiento. Se pueden entonces utilizar éstos para, por ejemplo, escribir la función de la Figura 72 que determina si todos los elementos de una colección se encuentran presentes en otra (se ignoran las repeticiones de elementos).

Obsérvese que los objetos `C0` y `C1` son colecciones de tipo desconocido, incluso podrían ser instancias de implementaciones de `Collection<E>` completamente diferentes. Pero la función está realizada sólo en base a la especificación de la interfaz y siempre trabajará, es más no es necesario reescribirla para cada implementación de `Collection<E>`. Sólo sería necesario reescribir la función si una implementación permitiera alguna mejora, por ejemplo, mediante un algoritmo más eficiente.

```
/**
 * Retorna cierto si la primera colección especificada es
 * un subconjunto de la segunda, ignorándose las posibles
 * repeticiones de elementos.
 * @param c0 la colección a comprobar la inclusión
 * @param c1 la colección que incluye
 * @return cierto si c0 está incluida en c1, se ignoran las
 * repeticiones
 */
public static boolean subsetOf(Collection<?> c0, Collection<?> c1) {
    for (Object x: c0) {
        if (! c1.contains(x)) {
            return false;
        }
    }

    return true;
}
```

Figura 72. Ejemplo de función sobre `Collection<E>`

6.2 Clases abstractas

Para ayudar a los diseñadores a desarrollar nuevas implementaciones a partir de las interfaces de colecciones, la biblioteca estándar de Java proporciona una serie de clases abstractas paralelas a éstas con una serie de métodos ya realizados (implementan parcialmente las interfaces), concretamente todos aquellos para los que existe un algoritmo independiente de la estructura de datos con la que se vayan a representar los objetos concretos. Así, un diseñador de una nueva implementación sólo tiene que extender la correspondiente clase abstracta, elegir la estructura de datos e implementar todas las operaciones que dependen de ésta (por ejemplo, los constructores).

En la biblioteca, las clases abstractas con implementaciones parciales siempre se nombran anteponiendo la palabra `Abstract` al nombre de la interfaz. Así, por ejemplo, para la interfaz `Collection<E>` existe una clase abstracta que implementa parcialmente alguno de sus métodos y que se denomina `AbstractCollection<E>`.

La idea de utilizar implementaciones parciales de esta forma, surge de aplicar el patrón de diseño denominado método plantilla (***template method***)[5]. Las clases abstractas se utilizan como plantillas para las implementaciones reales y el programador sólo necesita cubrir:

- los constructores. La recomendación es que al menos haya dos constructores: un *constructor por defecto* (sin parámetros) y el *constructor de conversión*
- los métodos abstractos de la plantilla (que son los que la plantilla no implementa)

Además, si se implementa una clase con operaciones de modificación de acceso público, se requiere reescribir algunas de las operaciones opcionales (recuérdese que éstas, por defecto, se implementan como operaciones no soportadas).

[5] El patrón de diseño *template method* define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura. No debe confundirse este término con ciertas construcciones de algunos lenguajes de programación, como por ejemplo C++, que utilizan las plantillas como mecanismo para implementar el polimorfismo paramétrico.

6.2.1 La clase `AbstractCollection<E>`

Las clases abstractas están diseñadas para utilizarse exclusivamente en la cláusula **`extends`** y facilitar a los programadores la creación de nuevas clases `Collection`, pero no pueden utilizarse para especificar el tipo de un parámetro, variable local o campo. Esto explica porque el constructor para la clase `AbstractCollection<E>` está declarado como protegido (**`protected`**) en la Tabla 2, resumen de las operaciones de la clase.

Resumen de constructores	
`protected`	`AbstractCollection()` Constructor exclusivo.
Resumen de métodos	
`boolean`	`add(E e)` Asegura que la colección contiene el elemento especificado (*opcional*).
`boolean`	`addAll(Collection<? extends E> c)` Añade a la colección todos los elementos de la colección especificada (*opcional*).
`void`	`clear()` Elimina todos los elementos de la colección (*opcional*).
`boolean`	`contains(Object o)` Retorna `true` si la colección contiene el elemento especificado.
`boolean`	`containsAll(Collection<?> c)` Retorna `true` si la colección contiene todos los elementos de la colección especificada.
`boolean`	`equals(Object o)` Compara si el objeto especificado y la colección son iguales.
`int`	`hashCode()` Retorna el valor del código *hash* para la colección.
`boolean`	`isEmpty()` Retorna `true` si la colección no tiene elementos.
`abstract` `Iterator<E>`	`iterator()` Retorna un iterador sobre los elementos de la colección.
`boolean`	`remove(Object o)` Elimina de la colección el elemento especificado si está presente (*opcional*).
`boolean`	`removeAll(Collection<?> c)` Elimina de la colección todos los elementos que también estén contenidos en la colección especificada (*opcional*).
`boolean`	`retainAll(Collection<?> c)` Elimina de la colección todos los elementos que no estén contenidos en la colección especificada (*opcional*).
`abstract` `int`	`size()` Retorna el número de elementos de la colección.
`Object[]`	`toArray()` Retorna un *array* con todos los elementos de la colección.
`<T> T[]`	`toArray(T[] a)` Retorna un *array* con todos los elementos de la colección; el tipo del array retornado es el del *array* especificado. Si la colección cabe en el array especificado, ésta se ubica en él. En otro caso se retorna un nuevo *array* del tamaño de la colección.
`String`	`toString()` Retorna una cadena de caracteres que representa la colección.

Tabla 2. Operaciones de la clase `java.util.AbstractCollection`

Esta clase proporciona una plantilla para realizar implementaciones reales de la interfaz `Collection`, minimizando el esfuerzo requerido para implementar ésta.

Si se implementa una colección no modificable, únicamente se requiere extender esta clase y proporcionar implementaciones para los métodos abstractos `iterator` y `size`.

Si la colección que se implementa es una colección modificable se debe sobrecargar el método `add` (que en otro caso lanzaría la excepción `UnsupportedOperationException`) y el iterador retornado por la operación `iterator` debe implementar el método `remove`.

En lo que se refiere a los constructores, lo habitual es proporcionar un constructor por defecto (sin parámetros) y el constructor de conversión al que se le pasa una colección cualquiera (`Collection<? extends E>`), según se recomienda en la especificación de la interfaz `Collection<E>`.

6.2.2 Ejemplo de implementación

A continuación, a modo de ejemplo, se hará una implementación del tipo modificable `Bolsa<T>` de la Figura 62 extendiendo la clase `AbstractCollection<T>`. En este caso se utilizará la misma estructura de datos (representación) para implementar la clase, la única diferencia es que aquí se redimensionará el *array* `datos` en caso de que se agote su capacidad.

```
public class Bolsa<T> extends AbstractCollection<T> {
    // capacidad de una bolsa por defecto
    private static final int CAPACIDAD = 10;
    // capacidad máxima del array
    private static final int MAX_DIM_ARRAY = Integer.MAX_VALUE - 8;
    // espacio para objetos de tipo T en la bolsa
    protected T[] datos;
    // número de objetos en la bolsa
    protected int numDatos;

    /**
     * Crea una bolsa de capacidad por defecto.
     */
    public Bolsa() {
        this(CAPACIDAD);
    }

    /**
     * Crea una bolsa de capacidad especificada.
     * @param capacidad la capacidad de la bolsa
     * @throws IllegalArgumentException si la capacidad
     * no es positiva
     */
    @SuppressWarnings("unchecked")
    public Bolsa(int capacidad) {
        if (capacidad <= 0) {
            throw new IllegalArgumentException();
        }
        datos = (T[]) new Object[capacidad];
        numDatos = 0;
    }
```

```java
/**
 * Crea una bolsa con los elementos de la
 * colección especificada.
 * @param c la colección cuyos elementos son
 * añadidos a esta colección
 * @throws NullPointerException si la colección
 * especificada es null
 */
@SuppressWarnings("unchecked")
public Bolsa(Collection<? extends T> c) {
    if (c == null) {
        throw new NullPointerException();
    }
    datos = (T[]) new Object[c.size() + CAPACIDAD];
    numDatos = 0;
    this.addAll(c);
}

/**
 * @see java.util.AbstractCollection#iterator()
 */
@Override
public Iterator<T> iterator() {
    return new BolsaIterator<T>();
}

/**
 * @see java.util.AbstractCollection#size()
 */
@Override
public int size() {
    return numDatos;
}

/**
 * Asigna el doble de espacio de memoria al campo datos.
 * @throws OutOfMemoryError si datos tiene la máxima dimensión
 */
private void incrCapacidad() {
    if (datos.length == MAX_DIM_ARRAY) { // overflow
        throw new OutOfMemoryError();
    }

    int nuevaCapacidad = datos.length << 1;  // duplica
    if (nuevaCapacidad < 0) { // supera Integer.MAX_VALUE
          nuevaCapacidad = MAX_DIM_ARRAY;
    }
    datos = Arrays.copyOf(datos, nuevaCapacidad);
}
```

```
    /**
     * Añade el elemento especificado a esta bolsa.
     * @param e el elemento a añadir
     * @return cierto si la colección se modifica
     */
    public boolean add(T e) {
        if (e == null) {
            throw new NullPointerException();
        }
        if (numDatos == CAPACIDAD) {
            incrCapacidad();
        }
        datos[numDatos++] = e;
        return true;
    }

    @SuppressWarnings("hiding")
    public class BolsaIterator<T> implements Iterator<T> {
        private int actual;
        private boolean opNext;

        public BolsaIterator() {
            actual = 0;
            opNext = false;
        }

        public boolean hasNext() {
              return actual < Bolsa.this.size();
        }

        @SuppressWarnings("unchecked")
        public T next() {
            if (! hasNext()) {
                throw new NoSuchElementException();
            }
            opNext = true;
            return (T) Bolsa.this.datos[actual++];
        }

        public void remove() {
            if (! opNext) {
                throw new IllegalStateException();
            }
            opNext = false;
            System.arraycopy(datos, actual, datos,
                             actual - 1, Bolsa.this.size() - actual);
            actual--;
            numDatos--;
        }
    }
}
```

Figura 73. Tipo de dato `Bolsa<T>` *extendiendo* `AbstractCollection<T>`

7 Programación funcional

El paradigma de programación funcional es un paradigma de programación declarativa, en el que los programas se realizan con expresiones o declaraciones en lugar de sentencias. En programación declarativa un algoritmo describe el problema que se quiere solucionar, pero no las instrucciones necesarias para solucionarlo. Mientras que en la programación imperativa un algoritmo consta de todos los pasos requeridos para solucionar el problema.

En código funcional, el resultado de una función únicamente depende de los argumentos que se pasan a ésta. Por tanto, cualquier expresión del lenguaje se puede sustituir por otra de igual valor sin modificar la semántica de un programa. Esto es lo que se conoce como **transparencia referencial**. Obsérvese que esto no ocurre con el código imperativo, donde se producen de forma habitual efectos colaterales por asignaciones destructivas que modifican el estado.

La transparencia referencial facilita la modificación de un programa, porque las modificaciones que se realicen en una parte del mismo no afecta a los resultados de otras partes de éste. Adicionalmente, hace que los programas funcionales sean más seguros en entornos multihilo porque evitan el estado modificable compartido.

A medida que el software se vuelve más y más complejo, es cada vez más importante estructurarlo bien. El software bien estructurado es fácil de escribir y depurar, y proporciona una colección de módulos que pueden ser reutilizados para reducir los costes de mantenimiento. Las características de los lenguajes funcionales en particular, las funciones de orden superior y la evaluación perezosa, pueden contribuir de forma significativa a la modularidad. Esta es una certeza al alza desde hace unos años, aunque la base de la programación funcional surgió entre 1930 y 1940 (el *lambda-calculus*) y el primer lenguaje de programación funcional (el Lisp) se diseñó en 1958. Tanto es así, que hoy no hay un lenguaje de programación orientado a objetos e imperativo de uso común que no haya adoptado estas características (C++, C#, PHP, Java y Python, aunque este último ya las incluyó desde un principio).

7.1 Funciones de orden superior y expresiones lambda

Los programas funcionales están únicamente constituidos por definiciones de funciones e invocación a éstas u otras funciones predefinidas y tanto la composición de funciones como las funciones de orden superior juegan un papel trascendental.

> Una **función de orden superior** es una función que toma como argumento (o entrada) otra función, o bien, retorna una función.

Para que un lenguaje de programación soporte *abstracciones funcionales de orden superior* es necesario que el lenguaje trate las funciones como *valores de primera clase*; es decir, se deben poder manipular como cualquier otro valor de los tipos integrados del lenguaje. La forma habitual de proporcionar esta característica adicional de las funciones es utilizar *expresiones lambda* (del *Lambda calculus*).

En el ámbito de los lenguajes de programación, una expresión lambda es la declaración de una función anónima que se puede utilizar como argumento para otra función o almacenar en una variable.

7.2 Programación funcional en Java

El lenguaje de programación Java hasta la aparición de la versión 8, siempre ha sido un lenguaje sólo orientado a objetos imperativo y carecía de características propias de los lenguajes funcionales como son las expresiones lambda. Esto no quiere decir que en versiones previas de Java no se pudieran realizar abstracciones funcionales de orden superior, pero sí que el lenguaje no ofrecía un buen soporte para su construcción. En consecuencia, la implementación de estas abstracciones requería de algún artificio adicional, dando lugar a un código más complejo o, cuando menos, más extenso.

7.2.1 Expresiones lambda e interfaces funcionales en Java

En las versiones previas de Java, la forma habitual de realizar las abstracciones funcionales de orden superior se ha basado en pasar como argumento un objeto de una clase que implementa un TDA (interfaz) con un único método abstracto. De forma que:

1. Para definir una función de orden superior `fos(f)`, el parámetro funcional `f` de su definición se reemplaza por una interfaz y en la definición de la función `fos` el método abstracto sustituye al parámetro funcional `f`.
2. La definición del método abstracto se ha de corresponder con la definición del argumento funcional `f` con el que se invoca a la función `fos`.
3. Para invocar la función de orden superior `fos(f)`, el argumento funcional `f` de su invocación se reemplaza por un objeto de una clase que implementa la interfaz.

Este ha sido, por ejemplo, el modo habitual de proporcionar una función de comparación entre objetos, mediante un objeto de una clase que implementa la interfaz `Comparable<T>`. Esta clase implementa el método abstracto `compare`, que es la función de comparación (ver apartado 2.3.2.2.2).

Un caso relativamente frecuente en el que resulta conveniente el uso de abstracciones funcionales de orden superior, es cuando se dispone de una colección de datos que se quieren filtrar por distintos criterios. En lugar de realizar una función para cada uno de los criterios de filtrado, es más simple (más abstracto) realizar una única función que reciba como argumentos la colección de datos y el criterio por el que se filtran éstos; es decir, una función que determine si un dato cumple o no el criterio de filtrado (retorna un booleano). Por ejemplo, la función `imprimirCadenas` de la Figura 74, muestra en la salida estándar los elementos de una colección de cadenas de caracteres que cumplen un criterio dado.

```
public static void imprimirCadenas(Collection<? extends String> c,
                                   Condicion cond) {
    for (String s: c) {
        if (cond.test(s)) {
            System.out.println(s);
        }
    }
}
```

Figura 74. Imprime cadenas de una colección que cumplan una condición dada

En la función `imprimirCadenas`, el criterio (o condición) de filtrado se proporciona mediante el parámetro de tipo `Condicion`, y éste especifica un método abstracto (`test`) que es la función de filtrado. Así pues, `Condicion` es la interfaz que se muestra en la Figura 75. Las interfaces que tienen un único método abstracto y se utilizan con el objetivo indicado reciben el nombre de **interfaces funcionales**. Finalmente, debe implementarse la interfaz (su método abstracto), esto se puede hacer directamente en la invocación de la función de orden superior,

`imprimirCadenas`, mediante una clase anónima. Por ejemplo, así se ha hecho en la invocación de la Figura 76 con el objetivo de imprimir todas las cadenas de una colección `a`.

```
public interface Condicion {
    boolean test(String s);
}
```

Figura 75. Interfaz funcional

```
imprimirCadenas(a, new Condicion() {
    public boolean test(String s) {
        return true;
    }
});
```

Figura 76. Imprime todas las cadenas de `a`

Otro ejemplo es el que se muestra en la Figura 77, en este caso sólo se imprimirán las cadenas de caracteres cuya longitud esté comprendida entre tres y siete (ambos incluidos) y no contengan el carácter '`a`'.

```
imprimirCadenas(a, new Condicion() {
    public boolean test(String s) {
        return s.length() >= 3 && s.length() <= 7 && ! s.contains("a");
    }
});
```

Figura 77. Imprime las cadenas de `a` *que cumplen una condición*

7.2.1.1 Sintaxis de las expresiones lambda

En Java una expresión lambda es la declaración de una función anónima que se puede utilizar como argumento para otra función o almacenarse en una variable. En la declaración se utiliza el símbolo de aplicación o mapeo, →, para separar los argumentos de la función de la expresión que permite calcular su imagen. Por ejemplo, la función suma $f(x,y) = x + y$ se puede expresar como $(x,y) \rightarrow x + y$. El símbolo de mapeo se representa mediante el par de caracteres `->`. El cuerpo de la función se expresa igual que en una función normal: un bloque de sentencias entre llaves con alguna sentencia `return` para indicar el valor de retorno (valor de la imagen), si es necesario. Sin embargo, se pueden aplicar varias simplificaciones sintácticas dependiendo de cómo sea la expresión lambda:

1. En primer lugar, lo más habitual es prescindir del tipo de los parámetros y que éstos sean inferidos. En este caso, si la expresión lambda tiene un único parámetro también se puede prescindir de los paréntesis.
2. Si el cuerpo de la función se reduce a una única expresión entonces se puede prescindir de las llaves y de la sentencia `return`. Téngase en cuenta que incluir la sentencia de retorno ya conlleva de por sí especificar un bloque de sentencias entre llaves.

Así, por ejemplo, la expresión lambda equivalente a la función $f(x,y) = x + y$, en Java se expresaría de una de las dos formas siguientes:

1. `(x, y) -> {return x + y;}`
2. `(x, y) -> x + y`

Volviendo al ejemplo de la colección `a` de cadenas de caracteres, la función que permite imprimir las cadenas de `a` que cumplen un determinado criterio sigue siendo la indicada en la Figura 74, lo que cambia es su invocación que ahora simplemente se haría en la forma:

```
imprimirCadenas(a, s -> s.length() >= 3 && s.length() <= 7
                   && ! s.contains("a"));
```

Figura 78. Imprime las cadenas de `a` *que cumplen la condición dada por una λ-expresión*

La invocación de una función de orden superior utilizando expresiones lambda en general es bastante más simple que la invocación mediante las correspondientes funciones anónimas (aunque en el ejemplo de la Figura 78 no se aprecie demasiado). Por otra parte, una función puede retornar una expresión lambda, pero no puede retornar una clase sea ésta anónima o no. De modo, que las expresiones lambda (o funciones anónimas) se pueden obtener en tiempo de ejecución, mientras cualquier otro tipo de función únicamente se puede definir de forma estática.

En un lenguaje de programación fuertemente tipado, como lo es Java, toda expresión debe tener un tipo, así que una expresión lambda también debe de tener un cierto tipo.

> En Java el tipo de dato de una *expresión lambda* es una **interfaz funcional**. Ésta es una interfaz con un único método abstracto (adicionalmente puede incluir métodos por defecto o funciones estáticas).

Así, la interfaz funcional `Condicion`, tampoco cambia, sigue siendo la que se muestra en la Figura 75. Si bien, ahora puede ir precedida de la anotación opcional `@FunctionalInterface` (ver Figura 79) que informa de que la interfaz es una *interfaz funcional* y, por tanto, debe contener un único método abstracto. En caso contrario, el compilador generaría un error.

```
@FunctionalInterface
public interface Condicion {
    boolean test(String s);
}
```

Figura 79. Interfaz funcional en Java 8

7.2.1.2 Referencias a métodos

En algunas ocasiones las expresiones lambda sólo se utilizan para invocar un método existente. En estos casos, es preferible referirse al método existente por el nombre. Para ello se antepone el operador *doble punto* '`::`' y un prefijo que depende de la clase de método:

Clase de método	*Ejemplo*
Referencia a un método estático	`clase::método`
Referencia a un método de un objeto concreto	`objeto::método`
Referencia a un método de un objeto arbitrario de cierto tipo	`tipo::método`
Referencia a un constructor	`clase::new`

Tabla 3. Clases de referencias a métodos

Por ejemplo, si se define en la clase `Prueba` el método estático, `lengthEven(s)`, que retorna cierto si la longitud del `String s` especificado es par y falso en caso contrario. Se pueden imprimir todas las cadenas de longitud par de la colección `a`, en la forma:

```
imprimirCadenas(a, Prueba::lengthEven);
```

Figura 80. Referencia a un método estático

Si `legthEven` fuera un método de instancias, entonces se podrían imprimir todas las cadenas de longitud par de la colección `a`, en la forma:

```
Prueba p = new Prueba();
imprimirCadenas(a, p::lengthEven);
```

Figura 81. Referencia a un método de un objeto

7.2.1.3 Interfaces funcionales

La biblioteca estándar de Java incorpora varias interfaces funcionales de uso común en el package `java.util.function`. En la Tabla 4 se muestran las más relevantes, éstas aparecen de forma relativamente habitual como tipo de parámetro (o de resultado) de algunos métodos en bastantes interfaces de la biblioteca.

Interfaz funcional	*Descripción y método funcional*
`Predicate<T>`	Determina si la entrada de tipo `T` verifica algún criterio.
	`boolean test(T t)` Evalúa este predicado para el argumento dado
`Supplier<T>`	Proporciona un objeto de tipo `T` (productor).
	`T get()` Produce un valor de tipo T
`Consumer<T>`	Acepta una entrada de tipo `T` y no retorna un resultado (consumidor).
	`void accept(T t)` Realiza la operación dada sobre el argumento proporcionado
`Function<T,R>`	Aplica una función a una entrada de tipo T, generando un resultado de tipo R.
	`R apply(T t)` Aplica esta función al argumento dado

Tabla 4. Interfaces funcionales genéricas (package `java.util.function`)

Las interfaces funcionales predefinidas de la biblioteca Java se utilizan exactamente de la misma forma que las interfaces funcionales definidas por el programador. En la práctica, estas interfaces predefinidas cubren un amplio abanico de '*tipos de funciones*' y no suele ser necesario que el programador defina interfaces funcionales adicionales.

Así, por ejemplo, se puede reescribir la función `imprimirCadenas` de la Figura 74 utilizando la interfaz funcional `Predicate<? super String>`:

```
public static void imprimirCadenas(Collection<? extends String> c,
                                   Predicate<? super String> cond) {
    for (String s: c) {
        if (cond.test(s)) {
            System.out.println(s);
        }
    }
}
```

Figura 82. Uso de interfaces funcionales predefinidas

De forma similar, se podría aplicar cualquier acción a las cadenas seleccionadas. Para ello únicamente es necesario reescribir la función de orden superior `imprimirCadenas` utilizando también la interfaz funcional `Consumer<? super String>`. La nueva función se denominará `procesarCadenas` y está definida en la Figura 83.

De esta forma, para procesar todas las cadenas de longitud tres pertenecientes a la colección `a`, se invocará la función `procesarCadenas` en la forma:

```
procesarCadenas(a, s -> s.length() == 3, s -> System.out.println(s))
```

o bien

```
procesarCadenas(a, s::length() == 3, System.out::println)
```

```
public static void procesarCadenas(Collection<? extends String> c,
                                   Predicate<? super String> cond,
                                   Consumer<? super String> cons) {
    for (String s: a) {
        if (cond.test(s)) {
            cons.accept(s);
        }
    }
}
```

Figura 83. Selección y procesamiento de las cadenas de una colección

Adicionalmente, también se pueden definir abstracciones funcionales polimórficas de orden superior. Por ejemplo, una función para procesar los elementos seleccionados en una colección `c` iterable (ver apartado 3.1.1.1) cualquiera:

```
public static<T> void procesarElementos(Iterable<T> c,
                                        Predicate<T> tester,
                                        Consumer<T> block) {
    for (T item: c) {
        if (tester.test(item)) {
            block.accept(item);
        }
    }
}
```

Figura 84. Selección y procesamiento de los elementos de una colección

Si `a` es una colección de cadenas de caracteres, se puede invocar la función de la Figura 84 para imprimir todas las cadenas de ésta con longitud mayor que tres:

```
procesarElementos(a, a::length() > 3, System.out::println)
```

7.2.2 Iteradores internos

Mientras que en los *iteradores externos* el control de la iteración lo tiene el cliente (ver apartado 3.1.1.2), en los **iteradores internos** el control de la iteración lo tiene el propio iterador y el programa cliente debe proporcionar a éste la abstracción funcional a aplicar. En este caso, el iterador ofrecerá al programa cliente operaciones para controlar la ejecución (p.e., sobre todos los elementos de la colección, sobre aquellos que cumplan una condición, etc.).

En varios lenguajes de programación únicamente se ofrece una operación para iterar sobre todos los elementos de la colección. Por ejemplo, en Java la interfaz `Iterable<T>` incluye (desde la versión 8) la especificación del método `forEach` dada en la Figura 85.

```
/**
 * Realiza la acción dada para cada elemento del Iterable hasta que
 * todos los elementos se hayan procesado o la acción lance una excepción.
 * @param action la acción a realizar con cada elemento
 * @throws NullPointerException si la acción especificada es null
 */
default void forEach(Consumer<? super T> action)
```

Figura 85. Especificación del método `forEach` en `Iterable`

La implementación por defecto del método `forEach` es equivalente a:

```
for (T t : this) {
    action.accept(t);
}
```

De forma que, por ejemplo, escribir en la salida estándar todos los elementos de una colección `a` iterable, se reduce a:

```
a.forEach(System.out::println);
```

Figura 86. Imprimir todos los elementos de una colección `a` *iterable*

Y si los elementos de la colección `a` son comparables (su clase implementa la interfaz `Comparable<T>`), escribir en la salida estándar todos los elementos de la colección que son estrictamente mayores que otro elemento `e` dado, sería:

```
a.forEach(item -> {
    if (compare(item, e) > 0) {
        System.out.println(item));
    }
});
```

Figura 87. Imprimir todos los elementos de una colección `a` *que son mayores que otro* `e` *dado*

7.2.2.1 Comparación con los iteradores externos

Dado que en los *iteradores externos* el control de la iteración lo realiza el programa cliente, éstos son más flexibles que los *iteradores internos*. Con iteradores externos el programador construye el bucle como mejor crea conveniente, mientras que con los iteradores internos la forma en la que se itera está impuesta.

Como contrapartida los *iteradores internos* son más seguros y simples de utilizar porque la iteración está encapsulada y no hay intervención del cliente. De esta forma, se evitan los posibles errores que si se podrían producir con un *iterador externo* al ser el cliente el responsable de construir la iteración.

Por otra parte, un *iterador externo* únicamente permite tratar los elementos de una colección de forma estrictamente secuencial. Los *iteradores internos* no tienen esta limitación y pueden aprovechar las ventajas de la *programación paralela* siempre que sea posible dividir el problema en subproblemas que se puedan resolver de forma simultánea y las soluciones de éstos se puedan combinar para resolver el problema original.

7.2.3 Evaluación perezosa

Algunos lenguajes de programación son **perezosos** (*lazy*), mientras que otros son **impacientes** (*strict*). En los primeros, al contrario de lo que ocurre con estos últimos, la evaluación de expresiones se retrasa hasta que su valor es necesario. Así, por ejemplo, en la invocación de una función un lenguaje con evaluación impaciente evaluará todos sus argumentos antes de iniciar la ejecución de ésta, mientras que un lenguaje con evaluación perezosa iniciará la ejecución de la función y sus argumentos sólo serán evaluados cuando se necesiten.

La evaluación perezosa tiene ciertas ventajas para algunos problemas específicos, tales como:

1. Componer estructuras de datos infinitas
2. Evaluar condiciones de error
3. Definir estructuras de control como abstracciones, en lugar de operaciones primitivas

Permitiendo reducir el consumo de memoria de una aplicación, ya que los valores se crean sólo cuando se necesitan. Y también permite describir la solución de un problema en términos de secuencias infinitas, algo relativamente frecuente, al separar la parte de la generación (o producción) de resultados de la parte de su consumo.

Por ejemplo, se puede dar el siguiente algoritmo para obtener los `n` primeros números primos:

```
Generar la secuencia de números naturales;
Filtrar los números primos;
Coger los n primeros primos;
```

Figura 88. Algoritmo para obtener la secuencia de los `n` primeros números primos

que expresado en forma funcional podría ser una composición de funciones similar a la siguiente:

```
tomar(n, filtrar(esPrimo, secuenciaNaturales()))
```

Figura 89. Expresión funcional del algoritmo de la Figura 88

Sin entrar ahora en excesivos detalles sobre cómo sería exactamente cada línea de código del algoritmo de la Figura 88, se pueden extraer del mismo algunas observaciones interesantes. Las dos primeras líneas del algoritmo pueden proporcionar secuencias infinitas, la primera línea genera (o produce) la secuencia de números naturales y la segunda es una operación intermedia de filtrado que produce la secuencia de números primos. Este filtrado es una función de orden superior que require como argumento otra función para chequear si un número natural dado es o no primo, la función `esPrimo`. A pesar de que ambas líneas podrían generar secuencias infinitas, si la evaluación es perezosa sus términos sólo se evalúan cuando se requieren. Esto sólo ocurre con los `n` primeros términos cuando se toman (o consumen) y los infinitos términos restantes no se evalúan porque no se necesitan.

Por otra parte, el algoritmo de la Figura 88 describe la solución del problema y resulta más corto y fácil de entender que un algoritmo imperativo porque para éste último hay que indicar cada paso a realizar (prueba tú mismo a propocionar una versión imperativa equivalente).

En programación funcional, este tipo de problemas en el que intervienen secuencias infinitas, por lo general, se solucionan mediante una estructura especial: *una secuencia perezosa, llamada* `Stream`.

7.2.3.1 La evaluación en Java

Java es un lenguaje de programación impaciente, todo se evalúa de forma inmediata. Por ejemplo, los argumentos de una función se pasan por valor; es decir, primero se evalúan y después se pasa a la función el valor evaluado.

Hay lenguajes perezosos (Miranda, Haskell), otros son impacientes (perezosos) por defecto y opcionalmente perezosos (impacientes), como Scheme, y la mayor parte son impacientes, como Java. Sin embargo, Java no siempre es impaciente, al igual que el resto de lenguajes, y tiene algunas construcciones que se evalúan de forma perezosa:

- Los operadores `||` y `&&`
- El operador ternario `?:`
- La sentencia altenativa `if-else`
- Los bucles `for` y `while`
- Los bloques `try-catch`
- Los `Stream` de Java 8

Por ejemplo, los operadores de conjunción y disyunción no evalúan su parte derecha si no es necesario. Pero la programación perozosa en Java está limitada a poco más que el uso del tipo `Stream`. Éste se verá en el próximo apartado.

Si bien en Java no es posible evitar la evaluación de los argumentos de un método (o función) cuando no se necesitan, se puede minimizar el efecto de estas evaluaciones innecesarias. Para

ello se requiere modificar la signatura (o perfil) del método y utilizar la interfaz funcional `Supplier<T>` tal y como se verá a continuación.

En la clase que se proporciona en la Figura 90 la evaluación de la función `f` es impaciente y, sin embargo, la evaluación del segundo argumento no es necesaria si la longitud de la cadena correspondiente al primer argumento es mayor que cinco.

```
public class ClaseEjemplo1 {
    private String a;
    private String b;

    public ClaseEjemplo1(String a, String b) {
        this.a = a;
        this.b = b;
    }

    public String getLeft() {
        System.out.println("método getLeft");
        return a;
    }

    public String getRight() {
        System.out.println("método getRight");
        return b;
    }

   /**
    * Retorna la primera cadena especificada si su longitud es
    * mayor que cinco. En caso contrario, retorna la concatenación
    * de las dos cadenas especificadas.
    * @param a la primera cadena proporcionada
    * @param b la segunda cadena proporcionada
    * @return la cadena a si su longitud es mayor que cinco y
    * la concatenación a + b en caso contrario
    */
    public static String f(String a, String b) {
        return a.length() > 5 ? a : a + b;
    }

    public static void main(String[] args) {
        ClaseEjemplo1 p = new ClaseEjemplo1("abcdef", "otra");
        System.out.println(f(p.getLeft(), p.getRight()));
    }
}
```

Figura 90. Clase con una función `f` de evaluación impaciente

Así, al ejecutar el código de la función `main` se obtiene el siguiente resultado:

```
método getLeft
método getRight
abcdef
```

ya que al evaluarse el segundo argumento se invoca el método `getRight`. Sin embargo, el valor calculado para este segundo argumento no influye en el resultado de la función `f`, al ser mayor que cinco la lontigud de la cadena correspondiente al primer argumento.

Para que la evaluación de la función `f` sea perezosa se va sustituir el tipo `String` de sus parámetros por `Supplier<String>`. De esta forma únicamente se generará una cadenada de caracteres, mediante el método `get` de `Supplier`, cuando realmente se necesite (Figura 91).

```
public class ClaseEjemplo2 {
    private String a;
    private String b;

    public ClaseEjemplo2(String a, String b) {
        this.a = a;
        this.b = b;
    }

    public String getLeft() {
        System.out.println("método getLeft");
        return a;
    }

    public String getRight() {
        System.out.println("método getRight");
        return b;
    }

    public static String f(Supplier<String> a, Supplier<String> b) {
        String s = a.get();
        return s.length() > 5 ? s : s + b.get();
    }

    public static void main(String[] args) {
        ClaseEjemplo2 p = new ClaseEjemplo2("abcdef", "otra");
        System.out.println(f(p::getLeft, p::getRight));
    }
}
```

Figura 91. Clase con una función `f` de evaluación perezosa

En este caso, al ejecutar el código del programa de la Figura 91, se obtiene la siguiente salida:

```
método getLeft
abcdef
```

Obsérvese que ahora los argumentos de la función `f` serán, por lo general, expresiones lambda. En este caso en particular, ambos argumentos se reducen a invocar directamente un método del objeto `p`, por lo que se han utilizado directamente sus referencias. Aquí también se evalúa el segundo argumento de la función `f` (como no puede ser de otra forma), pero lo que no se evalúa es su método `get` (no se genera la segunda cadena) y, por tanto, no se accede al método `getRight` de la clase.

7.2.4 La interface `Stream<T>`

Como ya se indicó al final del apartado 7.2.3, en programación funcional los problemas que tratan secuencias infinitas normalmente se solucionan mediante una secuencia perezosa denominada `Stream`.

Los **stream** de Java se utilizan de la misma forma, aunque tienen algunas diferencias con los stream de los lenguajes funcionales. En primer lugar, se diseñaron para permitir la paralelización

automática, lo que obliga a limitar los métodos funcionales aplicables. Además, tienen estado, de forma que una vez utilizado con operaciones que cambian su estado ya no es utilizable.

Resumen de algunos métodos	
`<R,A> R`	`collect(Collector<? super T,A,R> collector)` Realiza una operación modificable de reducción sobre los elementos de este `Stream` utilizando un `Collector`.
`long`	`count()` Retorna el número de elementos en este `Stream`.
`Stream<T>`	`distinct()` Retorna un nuevo `Stream` que consta de los elementos distintos (según `Object.equals(Object)`) de este `Stream`.
`static <T> Stream<T>`	`empty()` Retorna un `Stream` vacío.
`Stream<T>`	`filter(Predicate<? super T> predicate)` Retorna un nuevo `Stream` con los elementos de este `Stream` que verifican el predicado especificado.
`Optional<T>`	`findAny()` Retorna un `Optional` que describe algún elemento de este `Stream`, o un `Optional` vacío.
`Optional<T>`	`findFirst()` Retorna un `Optional` que describe el primer elemento de este `Stream`, o un `Optional` vacío.
`void`	`forEach(Consumer<? super T> action)` Realiza la acción especificada para cada elemento de este `Stream`.
`static <T> Stream<T>`	`generate(Supplier<T> s)` Retorna un `Stream` en el que cada elemento es generado por el productor proporcionado.
`<R> Stream<R>`	`map(Function<? super T,? extends R> mapper)` Retorna el `Stream` resultante de aplicar la función dada a todos los elementos de este `Stream`.
`Stream<T>`	`limit(long maxSize)` Retorna un Stream a partir de los elementos de éste, limitado al número máximo de elementos especificado.
`Optional<T>`	`max(Comparator<? super T> comparator)` Retorna el elemento máximo de este `Stream` de acuerdo con el `Comparator` proporcionado.
`static <T> Stream<T>`	`of(T...values)` Retorna un `Stream` cuyos elementos son los valores especificados.
`Optional<T>`	`reduce(BinaryOperator<T> accumulator)` Realiza una reducción en los elementos de este flujo, utilizando una función de acumulación asociativa, y devuelve un `Optional` que describe el valor reducido, si lo hay.
`Stream<T>`	`sorted()` Retorna un `Stream` formado por los elementos de este `Stream` ordenados de acuerdo al orden natural.
`Stream<T>`	`sorted(Comparator<? super T> comparator)` Retorna un `Stream` formado por los elementos de este `Stream` ordenados de acuerdo al `Comparator` proporcionado.
`Object[]`	`toArray()` Retorna un *array* conteniendo los elementos de este `Stream`.

Tabla 5. Parte de la interfaz `java.util.stream.Stream<T>`

Un **stream** es una secuencia de elementos con soporte de operaciones de agregados secuenciales y paralelas (a pesar de la similitud de nombre, no tiene nada que ver con los *streams* de Java I/O: `InputStream` , `OutputStream`, `PrintStream`, etc.). En la Tabla 5 se incluyen, resumidos, algunos de los métodos de la interfaz `Stream<T>`.

Los *streams* no almacenan datos (no son estructuras de datos) sino que llevan los valores de una fuente a una tubería de operaciones para obtener un único resultado, o efecto secundario, sin modificar la fuente. Proporcionan una descripción declarativa de su fuente y de las operaciones computacionales que se pueden realizar sobre ésta.

7.2.4.1 ¿Cómo trabajan los streams?

Para realizar un cálculo, la secuencia de operaciones se compone en una tubería (*stream pipeline*). Cada tubería consta de:

1. *Una fuente*. Puede ser una colección, un *array*, una función generadora, o un canal de entrada/salida.
2. Cero o más *operaciones intermedias*. Una operación intermedia, como por ejemplo `filter`, produce un nuevo `Stream`.
3. Una *operación terminal*. Esta operación produce un resultado que no es de tipo `Stream` (por ejemplo, un valor primitivo o una colección) o bien un efecto secundario.

Debe de tenerse en cuenta que hay operaciones sólo intermedias y sólo terminales, pero también hay operaciones que se pueden comportar de ambas formas.

Los `Stream` son perezosos, el cálculo sobre los datos de origen sólo se realiza cuando se inicia la operación terminal y los elementos de la fuente se consumen sólo cuando se necesitan. Además, el cálculo se puede realizar de forma secuencial o en paralelo.

Por tanto, operaciones que puede parecer recorren varias veces el `Stream` lo atraviesan una única vez. Además, si no es necesario, no se recorren todos los elementos del `Stream`, sólo los que se necesitan para completar la operación terminal.

La mayor parte de las operaciones de `Stream` aceptan argumentos que describen comportamiento especificado por el usuario. Para preservar un comportamiento correcto, estos argumentos de comportamiento deben:

1. No interferir (no modificar el origen de datos).
2. Por lo general, carecer de estado (su resultado no debe depender de ningún estado que pueda cambiar durante la ejecución de la tubería).

Estos argumentos son siempre instancias de *interfaces funcionales* que, normalmente, serán expresiones lambda o referencias a métodos.

7.2.4.2 Clasificación de métodos por categorías

Los métodos del `Stream` pueden ser *intermedios* o *terminales*. Adicionalmente, algunos de estos métodos se denominan como métodos *corto-circuito*. Las siguientes operaciones corresponden a cada una de las categorías:

- *Métodos intermedios*: `map` (y operaciones relacionadas, `mapToInt`, `flatMap`, etc.), `filter`, `distinct`, `sorted`, `peek`, `limit`, `skip`, `parallel`, `sequential` y `unordered`.
- *Métodos terminales*: `forEach`, `forEachOrdered`, `toArray`, `reduce`, `collect`, `min`, `max`, `count`, `anyMatch`, `allMatch`, `noneMatch`, `findFirst`, `findAny` e `iterator`.

- *Métodos corto-circuito*: `anyMatch`, `allMatch`, `noneMatch`, `findFirst`, `findAny`, `limit` y `skip`.

Las operaciones intermedias producen otros `Stream` y no se procesan hasta que se invoca algún método terminal. Estas pueden ser a su vez con o sin estado. Ciertas operaciones (como `filter`) no retienen el estado de elementos previamente vistos, cada elemento se procesa de forma independiente. Otras operaciones (como `sorted`) pueden incorporar estado de los elementos vistos previamente para procesar un nuevo elemento y, por tanto, puede que necesiten procesar toda la entrada antes de producir un resultado.

Por tanto, si la tubería de operaciones (*pipeline*) contiene operaciones intermedias con estado, el procesamiento en paralelo puede requerir múltiples pasadas sobre los datos o requerir un espacio temporal de memoria significativo. Sólo las tuberías que contienen exclusivamente operaciones intermedias sin estado se pueden procesar en un único paso, ya sea de forma secuencial o en paralelo, con un almacenamiento intermedio de datos mínimo.

Cuando se invoca una operación terminal se considera consumido el `Stream` y no se pueden realizar otras operaciones. Las operaciones terminales pueden producir un efecto lateral (por ejemplo, `forEach`) o producir un valor (por ejemplo, `findFirst`).

Una operación intermedia es *corto-circuito* si para una entrada infinita, puede producir como resultado un *stream* finito. Una operación terminal es *corto-circuito* si para una entrada infinita puede terminar en un tiempo finito. Así, tener alguna operación corto-circuito en la tubería de operaciones es una condición necesaria, aunque no suficiente, para procesar un *stream* infinito y terminar en un tiempo finito.

Una operación corto-circuito provoca que las operaciones intermedias previas de la tubería de operaciones se procesen sólo hasta que la operación corto-circuito se evalúa.

7.2.4.3 Algunas operaciones de streams

Para obtener un `Stream` inicial para una colección, está disponible el método `stream` de la interfaz `Collection<E>` válido, por tanto, para cualquier colección. La especificación de esta operación es la siguiente:

```
/**
 * Retorna un stream secuencial con esta colección como fuente.
 * @return un stream secuencial sobre los elementos en esta colección
 */
default Stream<E> stream();
```

Figura 92. Especificación de la operación `stream` *de* `Collection`

También se puede obtener un `Stream` desde un *array* mediante los dos siguientes métodos estáticos de la clase `Arrays`:

```
/**
 * Retorna un stream secuencial con fuente el array especificado.
 * @param a el array especificado
 * @return un stream secuencial de fuente el array especificado
static <T> Stream<T> stream(T[] a);
```

Figura 93. Especificación de la operación `stream` *para arrays*

```
/**
 * Retorna un stream secuencial con fuente el rango [start, end)
 * del array especificado
 * @param a el array especificado
 * @param start el primer índice a cubrir (incluido)
 * @param end el último índice a cubrir (excluido)
 * @return un stream secuencial de fuente el array especificado
 * desde la componente de índice start hasta end excluida
static <T> Stream<T> stream(T[] a, int start, int end);
```

Figura 94. Especificación de la operación `stream` *para arrays*

Adicionalmente, la forma más común de inicializar un `Stream` es mediante los métodos estáticos de `Stream`: `of` y `generate` (ver Tabla 5). El primero admite como argumento un *array* (`Stream` sobre los elementos de éste), o la secuencia de elementos (seperados por comas) sobre los que actúa el `Stream`. El segundo requiere como argumento un generador (o productor) de tipo `Supplier<T>`.

7.2.4.3.1 Método `forEach`

Realiza un bucle sobre los elementos de un `Stream`. Es una operación terminal que recibe un consumidor (`Consumer<? super T>`).

Código de ejemplo:

```
String[] vs = {"esto", "es", "un", "vector", "de", "String"};
Stream.of(vs).forEach(System.out::println);
```

7.2.4.3.2 Método `map`

Produce un nuevo `Stream` que resulta de aplicar una función (`Function<T,R>`) a cada elemento del `Stream` original.

Código de ejemplo:

```
Integer[] vnum = {1, 2, 3, 4, 5, 6};
Stream.of(vnum).map(n -> n * n).forEach(System.out::println);
```

7.2.4.3.3 Método `filter`

Produce un nuevo `Stream` que contiene únicamente los elementos del `Stream` original que cumplen un criterio dado (`Predicate<? super T>`).

Código de ejemplo:

```
String[] vs = {"esto", "es", "un", "vector", "de", "String"};
Stream.of(vs).filter(s -> s.length() > 3).forEach(System.out::println);
```

7.2.4.3.4 Método `findFirst`

Produce un `Optional` que describe el primer elemento del `Stream`. Como suele ser habitual que los `Stream` se filtren, éste puede estar vacío.

La clase `Optional<T>` es una nueva clase de Java 8 que extiende la clase `Object` y es un contenedor de un objeto (*singleton*) que puede contener o no un valor no nulo. El método `isPresent` permite comprobar si tiene valor y, si es así, el método `get` recuperarlo.

El método `findAny` es similar, pero describe un elemento cualquiera del `Stream`, por lo que esta operación es más rápida para `Stream` que se procesan en paralelo.

Código de ejemplo:

```
String[] vs = {"esto", "es", "un", "vector", "de", "String"};
Stream<String> svs = Stream.of(vs);
Optional<String> os = svs.filter(s -> s.length() > 3).findFirst();
try {
    System.out.println(os.get());
}
catch (NoSuchElementException e) {
    e.printStackTrace();
}
```

7.2.4.3.5 Método `limit`

Este método de `Stream` es una operación *corto-circuito* que limita la fuente de datos de un *stream*, a lo sumo, al número de ítems que se especifique. Si este número es negativo se lanza la excepción `IllegalArgumentException`. Este método es especialmente útil para limitar fuentes de datos infinitas.

Aunque Java no soporta generadores, como se vió en el apartado 3.1, el tipo `Stream<T>` posibilita la generación de secuencias infinitas. Para ello, es necesario que la fuente de datos de un *stream* sea una clase que implemente la interfaz `Supplier<T>` (un productor). Por ejemplo, la clase `RandomGenerator` de la Figura 95 genera un número entero aleatorio cuando se invoca su método `get`.

```
public class RandomGenerator implements Supplier<Long> {
    private Random ran;

    private long getLongSeed() {
        SecureRandom sec = new SecureRandom();
        byte[] sbuf = sec.generateSeed(8);
        ByteBuffer bb = ByteBuffer.wrap(sbuf);
        return bb.getLong();
    }

    public RandomGenerator() {
        ran = new Random(getLongSeed());
    }

    @Override
    public Long get() {
        return ran.nextLong();
    }
}
```

Figura 95. Productor de números enteros aleatorios

La operación `Stream.generator(new RandomGenerator())` proporciona un *stream* de infinitos números enteros aleatorios. El siguiente código mostraría en la salida estándar los 100 primeros:

```
Stream<Long> longs = Stream.generator(new RandomGenerator());
longs.filter(100).forEach(System.out::println);
```

7.2.4.3.6 Operaciones de reducción

Una operación de reducción combina una secuencia de elementos de entrada en un único resultado aplicando de forma repetitiva una operación de combinación. Por ejemplo, determinar la suma o el máximo de un conjunto de números, o de elementos de una lista.

Los Stream disponen de múltiples operaciones de reducción, generales como `reduce` y `collect`, o más especializadas como `max` o `count`.

Una función de reducción es asociativa y carece de estado por lo que es inherentemente paralela. Si la operación de reducción es modificadora, como por ejemplo `collect`, ésta se realiza en un contenedor modificable como, `Collection` o `StringBuilder`.

Código de ejemplo:

```
Integer[] vnum = {1, 2, 3, 4, 5, 6};
System.out.println(Stream.of(vnum).count());
System.out.println(Stream.of(vnum).reduce((x, y) -> x + y));

Stream<Integer> svi = Stream.of(vnum);
Collection<Integer> a = svi.collect(
    Collectors.toCollection(LinkedList::new));
```

7.2.4.4 Ejemplo de uso de Streams

Se verá aquí como se implementa en Java el algoritmo de la Figura 88 para obtener los `n` primeros números primos.

En primer lugar, se definirá la clase `NaturalGenerator` (Figura 96) que implementa la interfaz `Supplier<Long>` para generar un *stream* de infinitos números naturales consecutivos.

```
public class NaturalGenerator implements Supplier<Long> {
    private long current;

    public NaturalGenerator() {
        current = 0;
    }

    public NaturalGenerator(long initial) {
        if (initial <= 0) {
            throw new IllegalArgumentException();
        }

        current = initial - 1;
    }

    @Override
    public Long get() {
        return ++current;
    }
}
```

Figura 96. Productor de números naturales consecutivos

Supuesto que se dispone de la función `esPrimo(n)`, para comprobar si el número natural que se especifica es un número primo, el siguiente código de la función `main` muestra en la salida estándar los `num` primeros números primos que se indiquen como parámetro:

```
    try {
        int num = Integer.parseInt(args[0]);
        Stream<Long> naturals = Stream.generate(new NaturalGenerator());

        System.out.println("Números primos: ");
        naturals.filter(n -> esPrimo(n))
                        .limit(num).forEach(System.out::println);
    }
    catch (IndexOutOfBoundsException e) {
        System.out.println("Falta parámetro con la cantidad de "
                           + "números primos a generar.");
    }
```

8 Referencias

A. AHO, J. E. (1974). *The Design and Analysys of Computer Algorithms.* Addison-Wesley.

A. AHO, J. E. (1983). *Data Structures and Algorithms.* Addison-Wesley.

B. LISKOV, J. G. (1986). *Abstraction and Specification in Program Development.* The MIT Press.

B. LISKOV, J. G. (2000). *Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Professional.

G. BRASSARD, P. B. (1990). *Algorítmica. Concepción y Análisis.* Masson.

Hilfinger, P. (2007). *EECS Instructional and Electronics Support.* Recuperado el 11 de Septiembre de 2014, de http://inst.eecs.berkeley.edu/~cs61b/fa07/book2/data-structures.pdf

L. CARDELLI, P. W. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys, Vol. 17 Nº 4*, 471-523.

MARÍ, R. P. (1998). *Diseño de Programas. Formalismo y Abstracción.* Prentice-Hall.

O. J. DHAL, E. W. (1972). *Structured Programming.* Academic Press.

Weiss, M. A. (2010). *Data Structures & Problem Solving Using Java, Fourth Edition.* Pearson.

ANEXO I. Manejo de errores utilizando excepciones[i]

La documentación que aquí se recoge es una versión traducida y revisada del tutorial Essential Java Classes, Lesson: Exceptions de la documentación Java de Oracle disponible en línea.

Durante la ejecución de los programas se pueden producir errores. Pero ¿qué sucede realmente después de que ha ocurrido el error? ¿Cómo se maneja el error? ¿Quién lo maneja?, ¿Puede recuperarlo el programa?

El lenguaje Java utiliza excepciones para proporcionar capacidades de manejo de errores. En este anexo se muestra qué es una excepción, cómo lanzar y capturar excepciones, qué hacer con una excepción una vez capturada, y cómo hacer un mejor uso de las excepciones heredadas de las clases proporcionadas por el entorno de desarrollo de Java.

9.1 ¿Qué es una excepción?

El término **excepción** es una forma corta de la frase "*suceso excepcional*" y puede definirse de la siguiente forma.

Definición: Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.

Muchas clases de errores pueden utilizar excepciones -- desde serios problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un *array* fuera de sus límites. Cuando dicho error ocurre dentro de un método Java, el método crea un objeto `exception` y lo maneja fuera, en el sistema de ejecución. Este objeto contiene información sobre la excepción, incluyendo su tipo y el estado del programa cuando ocurrió el error. El sistema de ejecución es el responsable de buscar algún código para manejar el error. En terminología java, crear un objeto `exception` y manejarlo por el sistema de ejecución se llama **lanzar una excepción** (`throw`).

Después de que un método lance una excepción, el sistema de ejecución entra en acción para buscar el manejador de la excepción. El conjunto de algunos métodos posibles para manejar la excepción es el conjunto de métodos de la pila de llamadas del método donde ocurrió el error. El sistema de ejecución busca hacia atrás en la pila de llamadas, empezando por el método en el que ocurrió el error, hasta que encuentra un método que contiene el **manejador de excepción** adecuado.

Un manejador de excepción es considerado adecuado si el tipo de la excepción lanzada es el mismo que el de la excepción tratada por el manejador. Así, la excepción sube sobre la pila de llamadas hasta que encuentra el manejador apropiado y una de las llamadas a métodos maneja la excepción, se dice que el manejador de excepción elegido **captura la excepción** (`catch`).

Si el sistema de ejecución busca exhaustivamente por todos los métodos de la pila de llamadas sin encontrar el manejador de excepción adecuado, el sistema de ejecución finaliza (y, en consecuencia, el programa Java también).

9.2 Ventajas de utilizar excepciones

Mediante el uso de excepciones para manejar errores, los programas Java tienen las siguientes ventajas frente a las técnicas de manejo de errores tradicionales:

1. Separar el *manejo de errores* del *código normal*.
2. Propagar los errores sobre la pila de llamadas.
3. Agrupación de errores y diferenciación.

9.2.1 Separar el manejo de errores del código normal

En la programación tradicional, la detección, el informe y el manejo de errores dan lugar a un código largo y caótico. Por ejemplo, supongamos que tenemos una función que lee un fichero completo dentro de la memoria. El algoritmo en pseudo-código podría ser similar al siguiente:

```
1 acción leerFichero () {
2     abrir el fichero;
3     determinar su tamaño;
4     asignar suficiente memoria;
5     volcar el fichero a la memoria;
6     cerrar el fichero;
7 fin acción;
```

A primera vista esta función parece bastante sencilla, pero ignora todos los posibles errores que podrían producirse:

1. Que el fichero no se pueda abrir.
2. Que no se pueda determinar la longitud del fichero.
3. Que no haya suficiente memoria libre para asignar al tamaño del fichero.
4. Que falle la lectura del fichero.
5. Que el fichero no se pueda cerrar.

Para tratar todos estos errores en la función, habría que añadir bastante código para la detección y el manejo de errores. El aspecto final de la función sería algo parecido al que se indica a continuación:

```
 1 función leerFichero () → Entero
 2     codigoError ← 0; // 0, sin error
 3     abrir el fichero;
 4     si ficheroAbierto entonces
 5         determinar la longitud del fichero;
 6         si obtenerLongitudDelFichero entonces
 7            asignar suficiente memoria;
 8             si obtenerSuficienteMemoria entonces
 9                 volcar el fichero a memoria;
10                 si falloDeLectura entonces
11                     codigoError ← -1;
12                 fin si;
13             sino
14                 codigoError ← -2;
15             fin si;
16         sino
17             codigoError ← -3;
18         fin si;
19         cerrar el fichero;
20         si ficheroNoCerrado entonces
21             codigoError ← codigoError - 4;
22         fin si;
23     sino
24         codigoError = -8;
25     fin si;
26     retornar codigodeError;
27 fin función;
```

Con la detección de errores, las 7 líneas originales se han convertido en 27 líneas de código, aumentando éste casi un 400%. Además, ahora la detección de errores y el retorno del código de error están completamente mezclados con el código original de la función, dando lugar a un código caótico y bastante ilegible.

Java proporciona una solución elegante al problema del tratamiento de errores: **las excepciones**. Las excepciones permiten escribir el flujo principal del código y tratar los casos excepcionales en otro lugar. Si la función `leerFichero` utilizara excepciones en lugar de las técnicas de manejo de errores tradicionales su código podría ser parecido al siguiente:

```
1  function void leerFichero() {
2      try {
3          abrir el fichero;
4          determinar su tamaño;
5          asignar suficiente memoria;
6          volcar el fichero a la memoria;
7      } catch (falloAbrirFichero) {
8          tratar error 1;
9      } catch (falloDeterminarTamaño) {
10         tratar error 2;
11     } catch (falloAsignarMemoria) {
12         tratar error 3;
13     } catch (falloLeerFichero) {
14         tratar error 4;
15     } catch (falloCerrarFichero) {
16         tratar error 5;
17     }
18 }
```

Figura 97. Ejemplo de uso de excepciones en Java

Las excepciones no evitan el esfuerzo de detectar, informar y tratar los errores, pero sí que permiten separar el código principal del código correspondiente a los casos excepcionales haciendo el código legible y, probablemente, más corto.

9.2.2 Propagar los errores sobre la pila de llamadas

Una segunda ventaja de las excepciones es la posibilidad del propagar el error encontrado sobre la pila de llamadas a métodos. Supóngase que el método `leerFichero` es el cuarto método en una serie de llamadas anidadas a métodos realizadas desde un programa principal: `metodo1` llama a `metodo2`, que llama a `metodo3`, que finalmente llama a `leerFichero`.

Supóngase también que `metodo1` es el único método interesado en el error que ocurre dentro de `leerFichero`. Tradicionalmente las técnicas de notificación del error forzarían a `metodo2` y `metodo3` a propagar el código de error retornado por `leerFichero` sobre la pila de llamadas hasta que el código de error llegue finalmente a `metodo1`, que es el único método que está interesado en él. Esto obligaría a codificar los métodos de forma similar a la que se indica:

```
1 acción metodo1 ()
2     error ← metodo2();
3     si error ≠ 0 entonces
4         tratar el error;
5     sino
6         proceder con método1;
7 fin acción;
```

```
 8 función metodo2 () → Entero
 9     error ← metodo3();
10     si error ≠ 0 entonces
11         retornar error;
12     sino
13         proceder con método2;
14 fin función;
15 función metodo3 () → Entero
16     error ← leerFichero();
17     si error ≠ 0 entonces
18         retornar error;
19     sino
20         proceder con método3;
21     fin si;
22 fin función;
```

El sistema de ejecución Java busca hacia atrás en la pila de llamadas para encontrar cualquier método que esté interesado en manejar una excepción particular. Un método Java puede obviar cualquier excepción lanzada dentro de él, por lo tanto, permite que otros métodos que estén por encima de él en la pila de llamadas puedan capturar la excepción. De tal forma que sólo los métodos interesados en el error se deben preocupar de detectar éste.

```
 1 function void metodo1() {
 2     try {
 3         metodo2();
 4     } catch (excepcion) {
 5         tratar el error;
 6     }
 7 }
 8 function metodo2() throws excepcion {
 9     metodo3();
10 }
11 function metodo3() throws excepcion {
12     leerFichero();
13 }
```

Sin embargo, como se puede apreciar en el pseudocódigo, requiere cierto esfuerzo por parte de los métodos centrales. Cualquier *excepción comprobada* que se pueda lanzar dentro de un método, forma parte de la interfaz de programación público del método y se debe especificar en la cláusula `throws` de éste. Así el método informa a su llamador sobre las excepciones que puede lanzar, para que el llamador pueda decidir qué hacer con esa excepción.

Obsérvese de nuevo la diferencia del factor de aumento de código y el factor de ofuscación entre las dos técnicas de manejo de errores. El código que utiliza excepciones es más compacto y más fácil de entender.

9.2.3 Agrupación de errores y diferenciación

Las excepciones se dividen en categorías o grupos. Como todas las excepciones lanzadas dentro de los programas Java son objetos de primera clase, la forma natural de agrupar o categorizar las excepciones es mediante jerarquía de clases. Las excepciones Java deben ser ejemplares de la clase `Throwable`, o de cualquier descendiente de ésta. Como de las otras clases Java, se pueden crear subclases de la clase `Throwable` y subclases de estas subclases; es decir, una jerarquía de excepciones. Cada hoja de la jerarquía (una clase sin subclases) representa un tipo

específico de excepción y cada nodo interno (una clase con una o más subclases) representa un grupo de excepciones relacionadas.

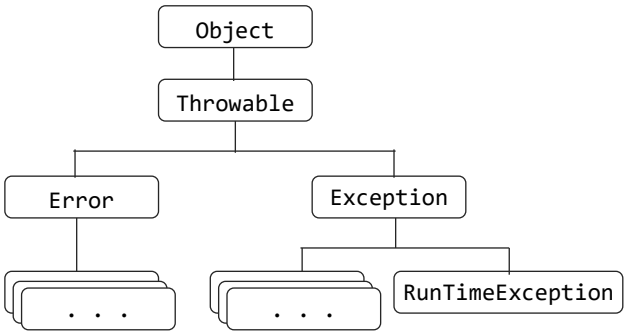

Figura 98. Jerarquía de excepciones en Java

Por ejemplo, en la plataforma Java, `IOException` es un grupo de excepciones relacionadas para operaciones de entrada/salida erróneas o que han sido interrumpidas. Además, es la clase más general para este tipo de excepciones y sus descendientes representan errores más específicos. Por ejemplo, `FileNotFoundException` indica que un fichero no se puede localizar en el disco y es una clase de excepción específica (hoja en la jerarquía).

Un método puede escribir manejadores específicos para manejar excepciones específicas (como `FileNotFoundException`), pero también se pueden capturar excepciones más generales (un grupo de excepciones, como `IOException`) y obtener más detalles sobre la excepción consultando el argumento pasado al manejador de excepciones:

```
catch (FileNotFoundException e) {
    ...
}
```

```
catch (IOException e) {
    // salida a System.err
    e.printStackTrace();
    // salida a stdout
    e.printStackTrace(System.out);
}
```

Incluso se puede configurar un manejador de excepciones para cualquier excepción mediante la clase `Exception`, que es la superclase de todas las clases de excepciones. Las excepciones se controlarán de esta forma sólo si lo único que se pretende hacer es informar del error al usuario y terminar la ejecución del programa:

```
catch (Exception e) {
    ...
}
```

Sin embargo, en la mayor parte de las situaciones, se tratará la excepción más específica posible. La razón es que la primera cosa que debe hacer un manejador es determinar qué tipo de excepción se produjo para, a posteriori, decidir la mejor estrategia de recuperación. Los manejadores de excepciones que son demasiado generales pueden hacer que el código sea más propenso a errores por la captura y manejo de errores que no se hayan anticipado y que, por tanto, no se tratan correctamente en el manejador.

9.3 Requerimientos para capturar o especificar excepciones en Java

El lenguaje de programación Java requiere que un método capture o especifique todas las *excepciones comprobadas* que se puedan lanzar dentro de su ámbito. Esto significa que el código que lanza estas excepciones debe cerrarse de alguna de las dos formas siguientes:

- Una sentencia `try` que captura la excepción. El `try` debe proporcionar un manejador de excepción, tal y como se describe en la Captura y manejo de excepciones.
- Un método que especifica que puede lanzar una excepción. El método debe proporcionar una cláusula `throws` que lista esta excepción, tal y como se describe en Especificar las excepciones lanzadas por un método.

No todas las excepciones están sujetas a la captura o especificación de requisitos. Para comprender por qué, se necesitan distinguir tres categorías básicas de excepciones, de las cuales sólo una está sujeta al requisito.

9.3.1 Categorías de excepciones

Se pueden distinguir tres categorías o clases de excepciones: las *excepciones comprobadas*, los *errores* y las *excepciones en tiempo de ejecución* (*`runtime exceptions`*).

La primera clase de excepciones son las *excepciones comprobadas*. Estas son las condiciones excepcionales para las que una aplicación bien escrita se ha de anticipar y de las que se debe poder recuperar. Por ejemplo, una aplicación que solicita al usuario un nombre de archivo de entrada para, a continuación, abrir el archivo pasando éste al constructor `java.io.FileReader`. Si el usuario proporciona un nombre de archivo que no existe, falla la construcción del objeto `FileReader` y el constructor lanza la excepción `java.io.FileNotFoundException`. Una aplicación bien escrita debería recoger esta excepción y notificar al usuario el error, solicitándole un nuevo nombre de archivo.

Las *excepciones comprobadas* están sujetas a la captura o especificación. Todas las excepciones son excepciones comprobadas, excepto las determinadas por las clases `Error` y `RuntimeException`, así como por sus subclases.

La segunda clase de excepciones es el *error*. Estas son las condiciones excepcionales que son externas a la aplicación y, por lo general, ni se pueden anticipar ni se pueden recuperar. Por ejemplo, una aplicación abre correctamente un archivo de entrada, pero no lo puede leer debido a un fallo del hardware o del sistema operativo. La lectura infructuosa lanzará la excepción `java.io.IOError`. Una aplicación puede optar por capturar esta excepción, con el fin de notificar al usuario del problema, pero también podría tener sentido imprimir un seguimiento de la pila y salir.

Los *errores* no están sujetos a la captura o especificación. Son excepciones de la clase `Error` o de sus subclases.

La tercera clase de excepción son las *excepciones en tiempo de ejecución*. Estas son las condiciones excepcionales internas de la aplicación y para las que la aplicación, en general, no se puede anticipar ni recuperar. Por lo general, indican errores de programación, tales como errores de lógica o de uso indebido de una API. Volviendo al ejemplo inicial, si un error de lógica hace que se pase el valor `null` al constructor, éste lanzará la excepción `NullPointerException`. La aplicación puede detectar esta excepción, pero probablemente tiene más sentido eliminar el error de programación que provoca la excepción.

Las *excepciones en tiempo de ejecución* no están sujetas a captura o especificación. Son excepciones de la clase `RuntimeException` y sus subclases.

Los *errores* y las *excepciones en tiempo de ejecución* se conocen, colectivamente, como *excepciones no comprobadas*.

9.4 Captura y manejo de excepciones

En este apartado se describirá cómo utilizar los tres componentes de un manejador de excepciones: los bloques `try`, `catch` y `finally`. Un bloque `try` es especialmente adecuado en situaciones en que se utilizan recursos que se pueden cerrar (`Closeable`), tales como los `stream`.

El siguiente ejemplo define e implementa la clase `ListOfNumbers`. Cuando se crea una instancia de esta clase se crea un `ArrayList` que contiene diez elementos enteros con valores secuenciales del `0` al `9`. La clase también define un método llamado `writeList`, que escribe la lista de números en un archivo de texto llamado `OutFile.txt`. El ejemplo utiliza clases de salida definidas en `java.io`.

```
1 // Nota: Esta clase no compilará
2 import java.io.*;
3 import java.util.List;
4 import java.util.ArrayList;
5
6 public class ListOfNumbers {
7     private List<Integer> list;
8     private static final int SIZE = 10;
9
10    public ListOfNumbers() {
11        list = new ArrayList<Integer>(SIZE);
12        for (int i = 0; i < SIZE; i++) {
13            list.add(new Integer(i));
14        }
15    }
16
17    public void writeList() {
18        // El constructor FileWriter lanza la excepción
19        // IOException, que debe ser capturada.
20        PrintWriter out = new PrintWriter(
21            new FileWriter("OutFile.txt"));
22
23        for (int i = 0; i < SIZE; i++) {
24            // El método get(int) lanza la excepción
25            // IndexOutOfBoundsException, que debe ser capturada.
26            out.println("Value at: " + i + " = " + this.list.get(i));
27        }
28        out.close();
29    }
30 }
```

La llamada al constructor `FileWriter` en la línea *21* inicializa un `stream` de salida sobre un archivo. Si el fichero no se puede abrir, el constructor lanza la excepción `IOException`. Por otra parte, la invocación al método `get` de la clase `ArrayList` en la línea *26* lanza la excepción `IndexOutOfBoundsException` si el valor de su argumento es menor que `0` o mayor o igual que el número de elementos de la lista.

Si se intenta compilar la clase `ListOfNumbers`, el compilador imprime un mensaje de error en la línea *21*: "`Unhandled exception type IOException`" . Sin embargo, no muestra mensaje de error alguno en la línea *26*. La razón es que la primera es una *excepción comprobada*, mientras que la segunda es una *excepción no comprobada* (*`runtime exception`*).

A continuación, se escribirán los manejadores necesarios para capturar y tratar las excepciones.

9.4.1 El bloque `try`

El primer paso en la construcción de un manejador de excepción es encerrar el código que podría lanzar la excepción dentro de un bloque `try`, de la forma siguiente:

```
try {
    código
}
bloques catch y finally
```

El segmento en el ejemplo etiquetado como `código` contiene una o más líneas de código que podría lanzar una excepción. (Los bloques `catch` y `finally` se explican en las dos subsecciones siguientes).

Para construir un manejador de excepción para el método `writeList` de la clase `ListOfNumbers`, se encierran las sentencias del método `writeList` que lanzan las excepciones dentro de un bloque `try`. Esto se puede hacer de varias formas. Se puede poner cada línea de código que lanza una excepción dentro de su propio bloque `try` y proporcionar manejadores de excepciones separados para cada uno. O bien, se puede poner todo el código dentro de un simple bloque `try` y asociar a éste múltiples manejadores:

```
17      public void writeList() {
18          PrintWriter out = null;
19
20          try {
21              out = new PrintWriter(new FileWriter("OutFile.txt"));
22              for (int i = 0; i < SIZE; i++) {
23                out.println("Value at: " + i + " = " + this.list.get(i));
24              }
25          }
 .          bloques catch y finally . . .
 .
 .      }
```

Si dentro del bloque `try` se produce una excepción, ésta es tratada por el manejador asociado con él. Para asociar un manejador de excepción con un bloque `try` se debe poner después un bloque `catch`.

9.4.2 Los bloques `catch`

Para asociar manejadores de excepción con un bloque `try` se deben proporcionar uno o más bloques `catch` directamente después del bucle `try`. Entre el final del bloque `try` y el comienzo del primer bloque de captura no puede ubicarse código alguno.

Cada bloque `catch` es un manejador de excepción que trata el tipo de excepción indicado por su argumento. El tipo de la excepción debe ser el nombre de una clase que herede de la clase `Throwable`.

El bloque `catch` contiene código que se ejecuta cuando se invoca el manejador de excepción y el sistema de ejecución invoca al manejador de excepción que es el primero en la pila de llamadas cuyo tipo del argumento coincide con el tipo de excepción lanzada.

Ejemplo de dos manejadores de excepción para el método `writeList`:

```
try {
  . . .
} catch (IndexOutOfBoundsException e) {
    System.err.println("IndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

Los manejadores de excepciones pueden hacer algo más que imprimir mensajes de error o detener el programa. Pueden recuperarse del error, pedir al usuario que tome una decisión, o propagar el error hasta un manejador superior mediante excepciones encadenadas.

9.4.2.1 Captura de más de un tipo de excepción con un manejador

Desde la versión 7 de Java, un simple bloque `catch` puede tratar más de un tipo de excepción. Esta característica permite reducir la duplicación de código y reduce el impulso de utilizar un tipo de excepción más general.

En la cláusula `catch`, se especifican los tipos de excepciones que manejara el bloque, separando cada tipo de excepción con una barra vertical '`|`':

```
catch (IOException | SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Nota: Si un bloque `catch` maneja más de un tipo de excepción, entonces el parámetro del `catch` es implícitamente `final`. En este ejemplo, el parámetro `ex` es `final` y por consiguiente no se le puede asignar un valor dentro del bloque `catch`.

9.4.3 El bloque `finally`

El bloque `finally` siempre se ejecuta cuando existe el bloque `try`. Esto asegura que el bloque `finally` se ejecutará incluso si se produce una excepción. Pero `finally`, es útil para algo más que el manejo de excepciones. Permite al programador disponer de un código de limpieza que siempre se ejecuta, independientemente de que la ejecución del código se rompa accidentalmente o se produzca algún cambio que modifique el flujo de ejecución. Poner código de limpieza en un bloque `finally` siempre es una buena práctica, incluso cuando no se prevean excepciones.

Nota: Si la MVJ finaliza mientras se ejecuta el código `try` o `catch`, el bloque `finally` no se podrá ejecutar. De la misma forma, si el hilo de ejecución del código `try` o `catch` se interrumpe o muere, el último bloque no se podrá ejecutar aunque la aplicación en su conjunto seguirá ejecutándose.

Por ejemplo, el bloque `try` del método `writeList` de la clase `ListOfNumber` abre un `PrintWriter` que se debe cerrar antes de salir del método `writeList`. Esto plantea un problema porque el bloque `try` de `writeList` puede terminar de tres formas:

1. El constructor `FileWriter` falla y lanza la excepción `IOException`.
2. La sentencia `list.get(i)` falla y lanza la excepción `IndexOutOfBoundsException`
3. Todo se ejecuta con éxito y el bloque `try` sale normalmente.

El sistema de ejecución siempre ejecuta las declaraciones dentro del bloque `finally`, independientemente de lo que suceda dentro del bloque `try`. Así que es el lugar perfecto para limpiar, para el ejemplo, cerrar el `PrintWriter`.

El siguiente bloque `finally` para el método `writeList` limpia, cerrando el `PrinWriter`:

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

Importante: El bloque `finally` es una herramienta clave para prevenir la pérdida de recursos. Cuando cierre un archivo o recupere otros recursos, situé el código en un bloque `finally` para asegurar que siempre se recupera el recurso.

9.4.4 La sentencia `try`-con-recursos

La sentencia `try`-con-recursos es una sentencia `try` que declara uno o más recursos. Un recurso es un objeto que se debe de cerrar una vez que el programa ha terminado con él. La sentencia `try`-con-recursos asegura que cada recurso se cierra al final de la declaración. Cualquier objeto que implementa `java.lang.AutoCloseable`, que incluye todos los objetos que implementan `java.io.Closeable`, se pueden utilizar como recurso.

El siguiente ejemplo lee la primera línea de un archivo. Para leer los datos del archivo se utiliza una instancia de `BufferedReader`. Esta instancia debe cerrarse después de que el programa haya terminado con ella:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader (new FileReader(path)) {
        return br.readLine();
    }
}
```

En este ejemplo, el recurso declarado en la sentencia `try`-con-recursos es un `BufferedReader`. La declaración del recurso aparece entre paréntesis inmediatamente después de la palabra clave `try`. La clase `BufferedReader`, en Java 7 y versiones posteriores, implementa la interfaz `java.lang.AutoCloseable`. Dado que la instancia `BufferedReader` se declara en una sentencia `try`-con-recurso, ésta se cerrará con independencia de que la sentencia `try` se complete con normalidad o bruscamente (como resultado de que el método `BufferedReader.readLine` lance la excepción `IOException`).

Para versiones previas a la siete, se puede utilizar el bloque `finally` para asegurar que un recurso se cierra independientemente de si la sentencia `try` se completa con normalidad o de forma abrupta. El ejemplo previo utiliza un bloque `finally` en lugar de una sentencia `try`-con-recursos.

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                                    throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Sin embargo, en este ejemplo, si los dos métodos `readLine` y `close` lanzan una excepción, entonces el método `readFirstLineFromFileWithFinallyBlock` lanza una excepción desde el bloque `finally` y se suprime la excepción lanzada desde el bloque `try`. Por el contrario, en el método `readFirstLineFromFile` se lanza la excepción desde el bloque `try` y se suprime la excepción lanzada desde el bloque `try`-con-recursos. En Java 7 y versiones posteriores, se pueden recuperar las excepciones suprimidas.

En una sentencia `try`-con-recursos se pueden declarar varios recursos. El siguiente ejemplo recupera los nombres de los archivos empaquetados en un archivo zip y crea un archivo de texto que contiene los nombres de estos archivos:

```
public static void writeToFileZipFileContents(String zipFileName,
                                               String outputFileName)
                                               throws java.io.IOException {

    java.nio.charset.Charset charset =
         java.nio.charset.StandardCharsets.US_ASCII;
    java.nio.file.Path outputFilePath =
         java.nio.file.Paths.get(outputFileName);

    // Open zip file and create output file with
    // try-with-resources statement

    try (
        java.util.zip.ZipFile zf =
             new java.util.zip.ZipFile(zipFileName);
        java.io.BufferedWriter writer =
            java.nio.file.Files.newBufferedWriter(outputFilePath,
                                                  charset)
    ) {
        // Enumerate each entry
        for (java.util.Enumeration entries =
                                zf.entries();
                                entries.hasMoreElements();) {
            // Get the entry name and write it to the output file
            String newLine = System.getProperty("line.separator");
            String zipEntryName =
               ((java.util.zip.ZipEntry)entries.nextElement()).getName()
               + newLine;
            writer.write(zipEntryName, 0, zipEntryName.length());
        }
    }
}
```

En este ejemplo, la sentencia `try`-con-recursos contiene dos declaraciones que están separadas por punto y coma: `ZipFile` y `BufferedWriter`. Cuando el bloque que sigue directamente a las declaraciones termina, de forma normal o como consecuencia de una excepción, se llama automáticamente a los métodos `close` de los objetos `BufferedWriter` y `ZipFile` en este orden. Obsérvese que los métodos `close` de los recursos se llaman en orden inverso a como éstos fueron creados.

El siguiente ejemplo utiliza una sentencia `try`-con-recursos para cerrar automáticamente un objeto `java.sql.Statement`:

```
public static void viewTable(Connection con) throws SQLException {
    String query = "select COF_NAME, SUP_ID, PRICE,
                    SALES, TOTAL from COFFEES";

    try (Statement stmt = con.createStatement()) {
        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");

            System.out.println(coffeeName + ", " + supplierID + ", "
                               + price + ", " + sales + ", " + total);
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
}
```

El recurso `java.sql.Statement` que se utiliza en el ejemplo es parte de la API JDBC 4.1 y posterior.

Nota: Una sentencia `try`-con-recursos puede tener bloques `catch` y `finally` exactamente igual que una sentencia `try` ordinaria. Estos bloques se ejecutan después de que los recursos declarados se han cerrado.

9.4.5 Juntando los bloques `try-catch-finally`

En las secciones previas se describe como construir los bloques `try`, `catch` y `finally` para el método `writeList` de la clase `ListOfNumbers`. Ahora veremos cómo queda finalmente el código de este método y se analizará lo que puede suceder durante su ejecución.

Como se ha mencionado previamente, el bloque `try` de este método tiene tres formas posibles de ejecutarse; que aquí se reducirán a dos:

1. El código de la sentencia `try` falla y lanza una excepción. Esta puede ser una `IOException` causada por la llamada al constructor `FileWriter` o una `IndexOutBoundException` causada por un índice incorrecto del `for` para la operación `get` de acceso a la lista.
2. Todo se ejecuta con éxito y la sentencia `try` termina normalmente.

```
public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Caught IndexOutOfBoundsException: "
                           +  e.getMessage());

    } catch (IOException e) {
        System.err.println("Caught IOException: " +  e.getMessage());

    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

Figura 99. Ejemplo de excepciones. Bloques `try-catch-finally`

9.4.5.1 Caso 1. Se produce una excepción

La sentencia que crea un `FileWriter` puede fallar por varias razones. Por ejemplo, porque el constructor lance la excepción `IOException` si el programa no puede crear o escribir en el archivo indicado.

Cuando `FileWriter` lanza una `IOException`, el sistema de ejecución detiene inmediatamente la ejecución del bloque `try`; y las llamadas a métodos en ejecución no se han completado. El sistema de ejecución comenzará a buscar el manejador apropiado en la parte superior de la pila de llamadas. Para este ejemplo, cuando se produce la `IOException`, el constructor `FileWriter` está en la parte superior de la pila de llamadas. Sin embargo, este constructor no tiene un manejador de excepción apropiado, por lo que el sistema de ejecución comprueba el siguiente método en la pila de llamadas: el método `writeList`. Este método tiene dos manejadores de excepciones: uno para `IOException` y otro para `IndexOutOfBoundsException`.

El sistema de ejecución comprueba los manejadores de `writeList` según el orden en el que aparecen tras la sentencia `try`. El argumento del primer manejador de excepción es `IndexOutOfBoundsException` que no coincide con el tipo de excepción lanzada. Por tanto, el sistema de ejecución comprueba el siguiente manejador de excepciones, `IOException`. Este coincide con el tipo de excepción lanzada, por lo que el sistema de ejecución finaliza la búsqueda de un manejador de excepción apropiado y se ejecuta el código de ese bloque `catch` …

Después de ejecutarse el manejador de excepciones, el sistema de ejecución pasa el control al bloque `finally`. El código de este bloque se ejecuta independientemente de la excepción capturada por encima de ella. En este escenario, el `FileWriter` no llego a abrirse y, por tanto,

no necesita cerrarse. Después de que el bloque `finally` termina de ejecutarse, el programa continúa con la primera sentencia después de este bloque.

En estas condiciones la salida del programa es la que se muestra a continuación:

```
Entering try statement
Caught IOException: OutFile.txt
PrintWriter not open
```

9.4.5.2 Caso 2. El bloque try termina normalmente

En este escenario, se ejecutan con éxito todas las sentencias dentro del ámbito del bloque `try` y no se lanzan excepciones. La ejecución llega al final del bloque `try` y el sistema pasa el control al bloque `finally`. Como todo se ha realizado correctamente, el `PrintWriter` está abierto cuando el control alcanza el bloque `finally`, y éste cierra el `PrintWriter`. Una vez más, cuando el bloque `finally` termina de ejecutarse, el programa continúa con la primera sentencia después de este bloque.

La salida de `ListOfNumber` cuando no se producen excepciones es la siguiente:

```
Entering try statement
Closing PrintWriter
```

9.5 Especificar las excepciones lanzadas por un método

En la sección anterior se mostró como escribir un manejador de excepciones para el método `writeList` de la clase `ListOfNumbers`. A veces, resulta adecuado para el código capturar las excepciones que pudieran producirse al ejecutar éste. Sin embargo, en otras ocasiones, es mejor que otro método más arriba en la pila de llamadas maneje la excepción. Por ejemplo, si la clase `ListOfNumbers` forma parte de un paquete de clases, seguramente no es factible anticipar como se utilizará éste por todos los usuarios. En este caso, es mejor no capturar la excepción y permitir que un método más arriba en la pila de llamadas maneje la excepción.

Si el método `writeList` no captura las *excepciones comprobadas* que puedan ocurrir en su código, éste debe especificar que puede lanzar estas excepciones. Para ello, es necesario agregar la cláusula `throws` en la cabecera del método, esta cláusula lista las *excepciones comprobadas* separando éstas mediante el carácter ‘,’:

```
public void writeList() throws IOException {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + this.list.get(i));
    }
    out.close();
}
```

Aunque la ejecución del método `writeList` puede provocar dos excepciones: `IOException` e `IndexOufBoundsException`, esta última es una excepción no comprobada por lo que su especificación en la cláusula `throws` no es obligatoria.

9.6 Cómo lanzar excepciones

Antes de poder capturar una excepción, algo de código en algún lugar debe lanzar ésta. Para lanzar una excepción se utiliza la sentencia `throw`. La plataforma Java ofrece numerosas clases de excepción. Todas las clases son descendientes de la clase `Throwable`.

Todos los programas permiten diferenciar entre los distintos tipos de excepciones que se puede producir durante su ejecución. Adicionalmente, se pueden crear clases de excepciones propias para representar los problemas de las clases que se creen. También se pueden crear excepciones encadenadas.

9.6.1 La sentencia throw

Todos los métodos utilizan la sentencia `throw` para lanzar una excepción. La sentencia `throw` requiere un solo argumento: un objeto de tipo excepción a lanzar. El objeto será una instancia de cualquier subclase de la clase `Throwable`.

Por ejemplo, en un método pop de una clase que implementa otra cuyas instancias son pilas comunes (*stack*). El método elimina el elemento superior de la pila y retorna éste:

```
public Object pop() {
    Object obj

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

Figura 100. Lanzando excepciones en Java. Sentencia `throw`

El método `pop` comprueba si los elementos están en la pila. Si la pila está vacía (su tamaño es igual a `0`), `pop` instancia un nuevo objeto `EmptyStackException` (un miembro de `java.util`) y lo lanza. La sección ….explica cómo crear clases de excepciones propias.

Téngase en cuenta que las declaraciones del método pop no contienen una cláusula `throws` porque `EmptyStackException` no es una *excepción comprobada*.

9.6.2 Excepciones encadenadas

Se dice que una excepción está encadenada cuando ésta se lanza en el manejador de otra excepción (bloque `catch`).

Los siguientes métodos y constructores de `Throwable` soportan excepciones encadenadas:

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

El argumento `Throwable` de `initCause` y de los constructores es la excepción que provoca la excepción encadenada. El método `getCause` retorna la primera de éstas.

ANEXO II. Análisis de algoritmos

En el apartado 2.2 se indicó que un tipo de dato abstracto podía tener varias implementaciones y que la representación elegida podía favorecer unas u otras operaciones. Ahora bien, cómo se pueden comparar distintos algoritmos correspondientes a una misma operación.

Una solución factible para comparar la rapidez de los algoritmos sería mediante un procedimiento empírico, programando los algoritmos (en algún lenguaje de programación) y ejecutándolos para medir el tiempo que tarda en ejecutarse cada algoritmo para diferentes entradas (tamaño y casos distintos). Sin embargo, esta aproximación empírica tiene ciertos inconvenientes, no sólo depende del tamaño del problema (tamaño o talla de los datos de entrada), también depende de cómo se elijan los datos de entrada (casos seleccionados para realizar el experimento) y de factores externos que no aportan información sobre los algoritmos como, por ejemplo, el lenguaje de programación elegido y la máquina en la que se realice la ejecución.

En este capítulo se presentarán criterios de comparación de algoritmos basados en la eficiencia, es decir, en el mejor aprovechamiento de los recursos computacionales. Se estudiaran dos factores:

- El *coste* o *complejidad espacial*, es decir, la cantidad de memoria que consume la estructura de datos.
- El *coste* o *complejidad temporal*, es decir, el tiempo que necesita el algoritmo para resolver un problema.

Ambos determinan el **coste** o **complejidad computacional**[6]. No siempre coincidirán consumo espacial óptimo con mínimo coste temporal; es más, por lo general ambos factores entrarán en competencia, siendo necesario adoptar un compromiso razonable entre ambos costes.

9.1 Complejidad temporal

Una alternativa para calcular el tiempo de ejecución de un algoritmo en función del tamaño `n` del problema, es contar el número de instrucciones a ejecutar y multiplicarlo por el tiempo requerido para cada instrucción. Considérese, por ejemplo, el siguiente algoritmo para calcular el valor $n!$:

```
i ← 0;
k ← 1;
mientras i < n hacer
  i ← i + 1;
  k ← k * i;
fin mientras
```

Figura 101. Algoritmo para calcular `n!`

Los valores que tomaría la variable `i` durante la ejecución serían: `0`, `1`, `2`, …, `n`. Las sentencias del bucle se ejecutarían `n` veces y, así, tendríamos la siguiente tabla que indica el número de veces que se ejecuta cada instrucción:

Asignaciones	Sumas	Productos	Comparaciones
$2n + 2$	n	n	$n + 1$

Tabla 6. Conteo del número de instrucciones en la ejecución del algoritmo `n!`

[6] Un estudio más detallado de la complejidad computacional, y de los métodos para realizar su cálculo, será uno de los primeros objetivos de la asignatura Algoritmia.

Si, además, se quiere independizar el cálculo del tiempo de ejecución de un algoritmo de factores externos (lenguaje de programación, máquina de ejecución, etc.), no es relevante el coste concreto de cada operación elemental. Bastará con saber cuántas operaciones elementales ejecuta un programa y cómo depende ese número de la talla del problema a resolver. Siendo una *operación elemental* (o *paso*) un segmento de código cuyo tiempo de proceso no depende de la talla del problema considerado y está acotado por alguna constante.

Entre las operaciones que se consideran pasos están las siguientes: *operaciones aritméticas*, *operaciones lógicas*, *asignaciones*, *acceso o asignación a elementos de un array* y *operaciones de entrada/salida de valores de tipos de datos simples* (entero, real, carácter o lógico).

El coste de las operaciones elementales que no son pasos se expresa en función del número de pasos con el que podrían efectuarse y el **coste computacional temporal** de un algoritmo se define como el número de pasos expresado en función de la talla del problema. Por ejemplo, el coste computacional temporal del algoritmo para calcular $n!$ es, $t(n) = 5n + 3$ (que es la suma de los conteos correspondientes de la tabla). No obstante, los factores de cada término en la expresión previa no son relevantes, sustituyéndose por constantes arbitrarias $t(n) = c_1 n + c_2$. Obsérvese que, con la introducción del concepto de paso, el coste computacional temporal del algoritmo previo también se podría haber calculado de la forma siguiente:

```
i ← 0;                          1 paso
k ← 1;                          n + 1 pasos
mientras i < n hacer
  i ← i + 1;
  k ← k * i;                    1 paso, n veces
fin mientras
```

$2n + 2$

Figura 102. Coste computacional temporal del cálculo de `n`!

> *Cualquier secuencia de pasos cuya longitud no depende de la talla del problema cuenta como una cantidad constante de pasos*.

9.1.1 Mejor caso, peor caso y caso promedio

En la práctica, la mayor parte de los algoritmos incluyen alguna sentencia condicional y, en consecuencia, el coste computacional temporal además de depender de la talla del problema también va a depender de los datos concretos que se le presenten (casos). Esto hace que más que calcular un valor $t(n)$ del coste computacional temporal haya que calcular un rango de valores para el mismo

$$t_{min}(n) \leq t(n) \leq t_{max}(n)$$

los valores extremos de este rango se conocen habitualmente como **mejor caso** y **peor caso**. Y, entre ambos, se hallará algún **caso promedio** o más frecuente.

Considérese, por ejemplo, el cuerpo de la función que busca un valor `x` en una *array* `a[0..n-1]` de `n` elementos (Figura 103). Dando por hecho que la construcción *`retorna`* fuerza la terminación de la función y produce un resultado (en nuestro caso, `cierto` o `falso`), para este algoritmo no resulta tan obvio determinar los valores que va a tomar la variable `i` durante la ejecución y el número de veces que se ejecuta el bucle `mientras`. La razón es que éstos van a depender de los datos de entrada (casos), concretamente de si el *array* contiene o no el valor `x` buscado y, en el caso de que así se sea, la posición del *array* en que este valor se encuentre.

```
i ← 0;
mientras i < n hacer
  si a[i] = x
  entonces retorna cierto;
  i ← i + 1;
fin mientras;
retorna falso;
```

Figura 103. Algoritmo de búsqueda de un valor en un array

Cuando se habla del mejor de los casos nos referimos a los datos de entrada que, para cada valor particular de la talla `n` del problema, se resuelven más rápidamente con el algoritmo. Para el ejemplo, cuando el elemento `x` buscado se encuentra en la primera posición del *array*. En este caso, el bucle se ejecuta una única vez (y no completamente) y el algoritmo termina con una secuencia constante de pasos que no depende de la talla del problema; es decir

$$t_{min}(n) = c_1$$

El peor de los casos lo determinan los datos de entrada que, para cada valor particular de `n`, hacen que el algoritmo se ejecute con el mayor número posible de pasos. Para el ejemplo, el peor caso es cuando el algoritmo termina retornando `falso`; es decir, cuando el valor `x` buscado no se encuentra en el *array*. Si este el caso, el bucle se ejecuta `n` veces una cantidad constante de pasos; es decir

$$t_{max}(n) = c_2 n + c_3$$

El cálculo analítico del coste computacional temporal para el caso promedio (o más frecuente) es algo más complejo y sobrepasa los objetivos que se pretenden alcanzar aquí y, además, para comparar el coste temporal de los algoritmos, por lo general, será suficiente con comparar el peor de los casos.

9.2 Notación asintótica

En general, el análisis de coste de los algoritmos es especialmente relevante cuando estos se aplican a problemas de gran tamaño (o talla). Casi siempre los problemas pequeños se pueden resolver de cualquier forma y las limitaciones aparecen al tratar problemas grandes. En todo caso, debe tenerse en cuenta que cualquier técnica de ingeniería, si funciona, acaba aplicándose al problema más grande que sea posible: las tecnologías de éxito, antes o después, acaban llevándose al límite de sus posibilidades.

Estas consideraciones llevan a estudiar el comportamiento de un algoritmo cuando se fuerza el tamaño `n` del problema al que se aplica. Matemáticamente hablando, cuando `n` tiende a infinito, es decir, su comportamiento asintótico (sin considerar constantes). Se introducirán, por tanto, ciertas herramientas matemáticas fundamentales que simplifican notablemente el análisis de costes y permiten expresar de forma muy concisa los resultados. Aprenderemos a caracterizar el coste mediante funciones simples que acoten superior e inferiormente el coste de todas las posibles entradas para tallas suficientemente grandes. Para ello se necesitan definir familias de cotas.

9.2.1 Orden y omega

Dada una función $f: N \to R^+$ la clase $O(f(n))$, que ha de leerse **del orden de** $f(n)$, es el conjunto de todas las funciones de N en R^+ acotadas superiormente por un múltiplo real positivo de $f(n)$ para valores de n suficientemente grandes.

$$O(f(n) = \{g: N \to R^+ \mid \exists c \in R^+,\ \exists n_0 \epsilon N: \forall n \geq n_0,\ g(n) \leq c \cdot f(n)\}$$

Intuitivamente, la definición previa refleja el hecho de que el crecimiento asintótico de las funciones g es como mucho proporcional al de la función f. Informalmente puede decirse que la tasa de crecimiento de la función f es una cota superior para las tasas de crecimiento de g.

La notación $O(f(n))$ es útil para estimar una cota superior del tiempo de ejecución de un algoritmo para entradas de talla n.

La siguiente figura muestra qué significa que una función $g(n)$ sea $O(f\ (n))$[7].

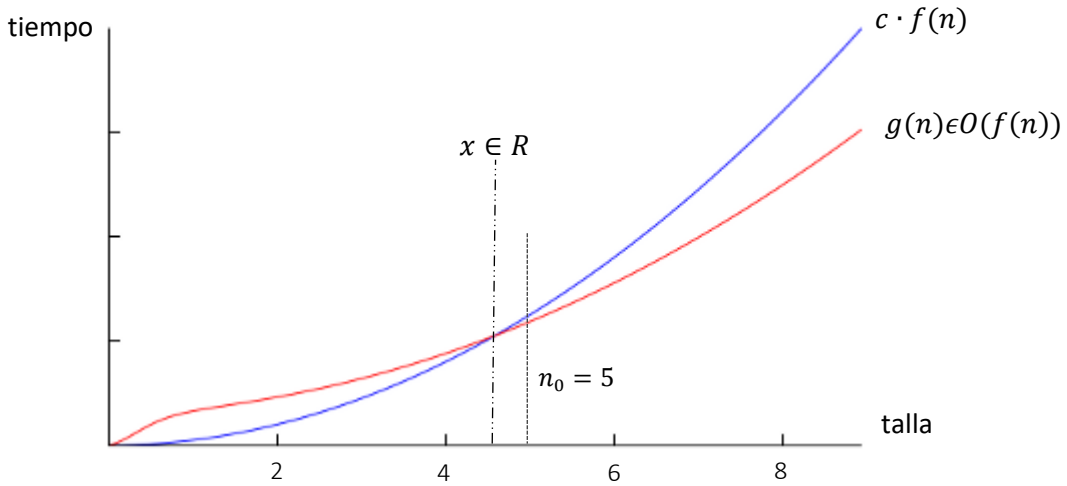

Figura 104. La función `g(n)` *"es del orden de"* `f(n)`

Es interesante poder estimar también una cota inferior del tiempo de ejecución de un algoritmo para entradas de talla n. La siguiente definición tiene este objetivo:

$$\Omega(f(n) = \{g: N \to R^+ \mid \exists c \in R^+,\ \exists n_0 \epsilon N: \forall n \geq n_0,\ g(n) \geq c \cdot f(n)\}$$

Es decir, $\Omega(f(n))$ que se lee **omega de** $f(n)$, es el conjunto de todas las funciones de N en R^+ acotadas inferiormente por un múltiplo real positivo de $f(n)$ para valores de n suficientemente grandes.

9.2.1.1 Propiedades básicas de la notación O

Sean $f: N \to R^+$, $g: N \to R^+$ y $h: N \to R^+$ funciones de N en R^+. Entonces los siguientes enunciados son ciertos:

1. Si $\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$ entonces $g(n)\ \epsilon\ O(f(n))$.
2. $f(n)\ \in\ O(f(n)),\ \forall f: N \to R^+$ (*reflexividad*).
3. Si $h(n)\ \in\ O(g(n))$ y $g(n)\ \in\ O(f(n))$ entonces $h(n)\ \epsilon\ O(f(n))$ (*transitividad*).
4. $O(f(n))\ =\ O(c \cdot f(n)),\ \forall c\ \in\ R^+$.

La última propiedad justifica la preferencia por omitir factores constantes. En el caso de funciones logarítmicas se omite la base porque

$$\log_a n = \frac{1}{\log a} \log n,\ \text{ con } \frac{1}{\log a} \epsilon R^+ \text{ si } a > 1$$

Los enunciados previos son las propiedades básicas de la notación O. Adicionalmente, se enumeran otras proposiciones que resultan de utilidad:

[7] Aunque $O(f\ (n))$ es una clase de funciones (un conjunto), también podemos encontrar de forma usual la notación $g = O(f(n))$ en lugar de la notación utilizada aquí, $g \in O(f\ (n))$. Esta igualdad orientada (nunca se escribe $O(f(n)) = g$) perdura por razones históricas. Es la que hace que digamos que el tiempo de un algoritmo es del orden de $f(n)$ en vez de lo que sería más correcto, está en el orden de $f(n)$.

5. Si $g(n) \epsilon O(f\ (n))$, entonces $c \cdot g(n) \epsilon O(f(n))\ \ \forall c \epsilon R^{+}$.
6. Si $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ entonces $O(g(n)) \subset O(f(n))$.
7. $O(f(n) + g(n)) = O(max(f(n), g(n)))$.

9.2.2 Orden exacto

El análisis asintótico de un algoritmo resultaría más satisfactorio si pudiéramos acotar, a la vez superior e inferiormente, su tiempo de ejecución por una misma función $f(n)$. Para ello se introduce una última notación

$$\Theta\big(f(n)\big) = \mathrm{O}\big(f(n)\big) \cap \Omega(f(n))$$

que se lee **del orden exacto de** $f(n)$.

9.2.3 Órdenes de complejidad

Se dice que $O(f(n))$ define un orden de complejidad. Para representar los distintos órdenes se utiliza la función $f: N \to R^{+}$ más sencilla de cada uno de ellos. Además, existe una jerarquía de distintos órdenes dada por la relación de inclusión entre éstos (según la sexta proposición del apartado 9.2.1.1). Esta jerarquía para los órdenes más habituales es la siguiente:

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!)$$

Al haber una relación de inclusión entre diferentes órdenes de complejidad una función $g(n) \epsilon O(f(n))$ también pertenece a cualquiera de los órdenes superiores a $O(f(n))$. Por ello, para que la notación O sea significativa *debe darse siempre la menor de las cotas superiores*; esto es, la función $g(n) \epsilon O(f(n))$ de forma significativa si:

$$0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

De forma análoga se establece la siguiente jerarquía entre diferentes omegas:

$$\Omega(1) \supset \Omega(\log n) \supset \Omega(\sqrt{n}) \supset \Omega(n) \supset \Omega(n \log n) \supset \Omega(n^2) \supset \Omega(n^3) \supset \Omega(2^n) \supset \Omega(n!)$$

y para que la notación Ω sea significativa *debe darse siempre la mayor de las cotas inferiores*; es decir, la función $g(n) \epsilon \Omega(f(n))$ de forma significativa si:

$$0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

Las funciones que pertenecen a cada orden tienen un adjetivo que las identifican:

Sublineales		*Constantes*	$O(1)$
		Logarítmicas	$O(\log n)$
			$O(\sqrt{n})$
Lineales			$O(n)$
Superlineales			$O(n \log n)$
	Polinómicas	*Cuadráticas*	$O(n^2)$
		Cúbicas	$O(n^3)$
	Exponencial		$O(2^n)$
	Factorial		$O(n!)$

Tabla 7. Nombres de las funciones pertenecientes a distintos órdenes

Se dice que el coste temporal de un algoritmo es lineal cuando es de $O(n)$ y logarítmico cuando es de $O(\log n)$, etc. En las siguientes figuras se comparan los órdenes sublineales (Figura 105) y superlineales (Figura 106) con el orden lineal, respectivamente.

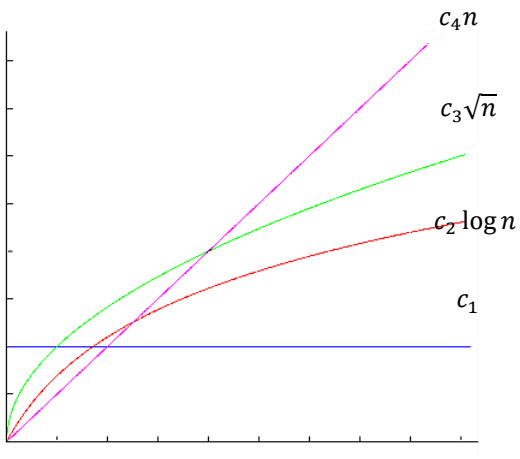

Figura 105. Órdenes sublineales y lineal

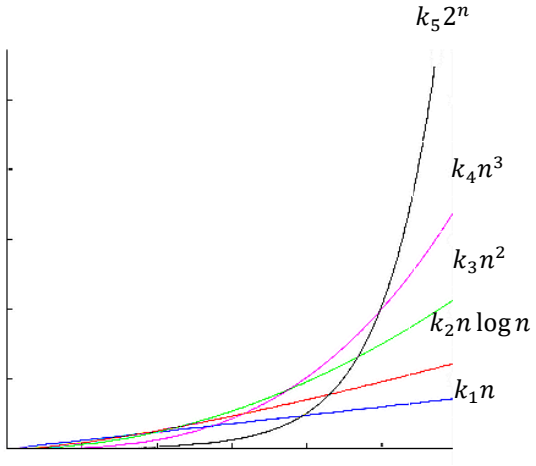

Figura 106. Órdenes lineal y superlineales

9.2.3.1 Impacto práctico

Para hacernos una idea de la importancia de los órdenes de complejidad se presentan en una tabla los tiempos utilizados por las funciones de complejidad para resolver un problema de talla n.

Talla	*Funciones de complejidad*						
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
5	3	5	12	25	125	32	120
10	4	10	33	100	1.000	1.024	$3{,}63\ 10^6$
100	7	100	664	10^4	10^6	$1{,}27\ 10^{30}$	$> 10^{100}$
200	8	200	1.529	$4\ 10^4$	$8\ 10^6$	$1{,}6\ 10^{60}$	$> 10^{100}$
1.000	10	1.000	9.965	10^6	10^9	$> 10^{100}$	$> 10^{100}$
2.000	11	2.000	$2{,}2\ 10^4$	$4\ 10^6$	$8\ 10^9$	$> 10^{100}$	$> 10^{100}$
10.000	14	10^4	$1{,}33\ 10^5$	10^8	10^{12}	$> 10^{100}$	$> 10^{100}$

Tabla 8. Tiempo en función de la talla para las funciones de complejidad más comunes

Se observa que los algoritmos de complejidad $O(n)$ y $O(n \log n)$ son los que muestran un comportamiento más natural, al doblar el número de datos procesados se duplica el tiempo necesario para procesarlos.

Los algoritmos de complejidad logarítmica crecen muy lentamente conforme n crece. Necesitan poco más tiempo para procesar el doble de datos.

Los algoritmos de complejidad polinómica presentan dificultades a medida que crece la talla, la práctica viene a decirnos que son el límite de lo tratable. Los cuadráticos dejan de ser útiles para tallas medias o grandes y los cúbicos sólo son útiles para problemas pequeños, complejidades polinómicas de mayor potencia prácticamente son inaceptables.

Cualquier algoritmo por encima de una complejidad polinómica se dice intratable y sólo será aplicable a problemas muy pequeños.

El comportamiento de las funciones de complejidad a medida que crece la talla n del problema expuesto en la tabla, explica el porqué de la búsqueda de algoritmos de complejidad lineal incluso, si es factible y con algo de suerte, de complejidad logarítmica. De no encontrarse un algoritmo de complejidad lineal, un algoritmo de complejidad $O(n \log n)$ no es mala alternativa. Si se encuentran soluciones polinomiales, se puede tratar con ellas a pesar de las limitaciones que tienen para problemas de talla media y grande; pero ante soluciones de complejidad exponencial más vale seguir buscando.

9.3 Complejidad espacial

La complejidad espacial es el estudio de la eficiencia de los algoritmos en lo que respecta a su consumo de memoria.

Todo lo expuesto para la complejidad temporal es igualmente válido para el estudio de la complejidad espacial, sustituyendo tiempo empleado por espacio de memoria utilizado. Un razonamiento similar al seguido con el coste temporal lleva a considerar únicamente la evolución del consumo de memoria con la talla del problema. En el estudio asintótico no es relevante si un algoritmo consume la mitad o el doble que otro, pero sí que utilice una cantidad de memoria que crece con el cuadrado de la talla n cuando otro sólo requiere una cantidad de memoria constante, por ejemplo. El concepto de paso también es utilizable aquí, asociado al concepto de celda de memoria, no importa el número de bytes que ocupa una variable, sólo importa si su tamaño es o no función (y de qué orden) de n, la talla del problema.

Índice de Tablas

Índice de Figuras

[i] Traducción de Lesson: Exceptions en The Java™ Tutorials