Programación Orientada a Objetos

Bibliografía:

- Programación Orientada a Objetos, anónimo (Curso 2018–2019).
- Java™ Platform, Standard Edition 8, API Specification (https://docs.oracle.com/javase/8/docs/api/).

Capítulo 1

Introducción

La evolución de los lenguajes de programación va ligada al crecimiento de la complejidad en aplicaciones y a la mejora de la capacidad del hardware, y busca acercarse a los conceptos del dominio de la aplicación (**abstracción**). Un **paradigma de programación** es una serie de conceptos que guían la forma de pensar acerca de los programas, construirlos y estructurarlos. Los lenguajes de programación reflejan uno o varios de estos paradigmas.

El paradigma **orientado a objetos** organiza el software como una colección de objetos que se relacionan y proveen la funcionalidad de un sistema. Este paradigma nos permite:

- **Abstracción**: Centrarse en las propiedades de los tipos y no en la implementación.
- **Modularidad**: Descomponer el software en componentes que se combinan entre sí.
- **Encapsulación**: Juntar la estructura de un tipo de datos con sus operaciones.
- **Ocultación de información**: Diferenciar la parte pública de un módulo de la privada.
- **Herencia**: Definir clases a partir de otras.
- **Polimorfismo**: Permitir que una entidad haga referencia a objetos de distintos tipos.

En 1985 se crean el **C++**, extensión orientada a objetos de C muy popular que ayudó a difundir el paradigma, y **Eiffel**, creado por Bertrand Meyer tras un profundo estudio del paradigma pero considerado «teórico» y poco usado.

En 1995, Sun Microsystems (adquirido por Oracle en 2009) crea **Java**, un lenguaje OO puro popularizado por su uso en la web y con numerosas bibliotecas de código, y en el año 2000 Microsoft crea **C#**, que combina C++ y Java, junto con su plataforma **.NET**.

El compilador de Java genera un **código intermedio** o *bytecode*, independiente de la plataforma, e interpretado o compilado en el momento (compilación **JIT**, *Just In Time*) por la máquina virtual de Java (JVM), con lo que el código se puede ejecutar en cualquier máquina que tenga instalado el entorno de ejecución de Java y la máquina virtual, independientemente de la arquitectura del procesador y el sistema operativo.

Java presenta una sintaxis sencilla, parecida a C++ pero eliminando las características complejas, y es robusto al ser fuertemente tipado (el compilador detecta muchos problemas del código), soportar excepciones (para gestión de errores en tiempo de ejecución) y proveer «recogida de basura» (hace innecesaria la gestión de punteros). Aprenderemos Java 1.8.

Capítulo 2

Clases y objetos

En Java los tipos de datos se implementan como **clases**, cada una dentro de un fichero y con nombre normalmente en *CamelCase*, que a su vez se organizan en **paquetes** que agrupan código relacionado, con nombres en *lowerCamelCase*[1], que pueden a su vez estar en otros. Los elementos de la ruta de un paquete, clase, etc. se separan con «`.`». Los ficheros de clase contienen:

- Declaración del paquete en el que está la clase: `package` *`ruta.del.paquete`*`;`.
- Importación de clases de paquetes distintos al de la clase o a `java.lang`, para poder referirnos a ellas sin indicar la ruta completa: `import` *`ruta.de.Clase`*`;`. También se pueden importar constantes
  definidas en clases con `import static ruta.de.Clase.`[`CONSTANTE`|`*`]`;`.
- Definición de la clase: [*`visibilidad`*] [*`mods`*] `class` *`NombreClase`*[<...>] [`extends` *`Clase`*] [`implements` *`Interfaz1`*`,` ...] `{` ... `}`, donde se define la clase. La visibilidad es `public` para que la clase sea visible fuera del paquete; por defecto solo lo es dentro de este.

Los elementos de una clase (dentro de `{` ... `}`) pueden tener visibilidad pública (`public`, accesible fuera del paquete), privada (`private`, accesible solo dentro la clase) o a nivel de paquete (por defecto, accesible solo dentro del paquete). Estos son:

- **Atributos**: Los campos de la estructura: [*`visibilidad`*] [*`mods`*] *`Tipo nombre`* [`=` *`expr`*]`;`. Siempre privados según las buenas prácticas. Cuando una clase A declara un atributo cuyo tipo es otra clase B, decimos que A es **cliente** de B (**relación de clientela**).
- **Métodos**: Operaciones aplicables sobre los objetos:
  [*`visibilidad`*] [*`mods`*] [*`TipoDevuelto`*|`void`] *`nombre`*`(`[`final`] *`Tipo parámetro`*`,` ... [`,` [`final`] *`Tipo`*`...` *`array`*]`)` [`throws` *`Excepción1`*`,` ...] `{` (implementación) `}`. Si se incluye el parámetro *`array`*, su tipo es *`Tipo`*`[]` (ver abajo), y recibe la lista de todos los parámetros después de los obligatorios si son de tipo *`Tipo`*, o el (único) parámetro después de los obligatorios si es de tipo *`Tipo`*`[]`. Puede haber varios métodos con el mismo nombre, siempre que tengan distinto número de parámetros o alguno tenga distinto tipo

[1] Ese término me lo he inventado.

y estos sean disjuntos (**sobrecarga de métodos** u *overloading*).
Para acceder a los atributos se pueden usar **métodos de acceso** (*getters*, normalmente `public` *`Tipo`* `get`*`Atributo`*`()`) y **métodos de modificación** (*setters*, normalmente `public void set`*`Atributo`*`(`*`Tipo atributo`*`)`), lo que permite más tarde cambiar la estructura manteniendo la interfaz compatible. Otras veces se accede a los atributos de otras formas más apropiadas para su significado concreto, o se añaden métodos de acceso para propiedades calculadas.

- **Constructores**: Operaciones encargadas de inicializar correctamente un objeto. Se definen como los métodos pero sin el tipo devuelto, y el nombre es el de la clase. Si no se define ninguno se crea un **constructor por defecto**, que no toma ningún parámetro y deja todos los atributos con su valor por defecto.

Los modificadores (*`mods`*) son:

- `static`: Indica que un atributo es global para todos los objetos de la clase (atributo **de clase**), o que un método es de clase, y por tanto no se llama a través de un objeto y en principio solo puede acceder a atributos de clase.
- `final`: Indica que el valor de un atributo (o variable local) solo puede ser establecido en su declaración o, en su caso, en un constructor. No se les asigna un valor por defecto. Puede combinarse con `static` para definir constantes, que no hace falta hacerlas privadas.

Tipos de datos en Java:

- Primitivos.
  - Enteros con signo: `byte` (8 bits), `short` (16), `int` (32), `long` (64). Por defecto: `0`.
  - Reales: `float` (IEEE-854, 32 bits), `double` (64). Por defecto: `0`.
  - Caracteres: `char`. Por defecto: `'\u000'`.
  - Booleano: `boolean`. Valores `false` (por defecto) y `true`.
- **Objetos**. **Instancias** de una clase, representadas por una estructura en memoria con un campo por cada atributo, que para tipos primitivos contiene el valor y para el resto contiene un identificador de objeto (*oid*), o el valor `null`, su valor por defecto, que no indica ningún objeto. El **estado** de un objeto es el valor de estos campos.
  Los objetos en Java se manejan por referencia (a través del *oid*). Esto permite la compartición de objetos con integridad referencial, es más eficiente para objetos complejos, permite estructuras recursivas e implica que los objetos se crean cuando se construyen y no en su declaración. El inconveniente es el ***aliasing***, pues al asignar el valor de una variable a otra no se copia el objeto sino su identificador, lo cual es importante tener en cuenta en métodos de acceso a atributos mutables para no comprometer la integridad del objeto.
  - Las cadenas de caracteres son objetos inmutables de clase `String`.
  - Los **enumerados** son clases definidas de forma especial, con la sintaxis [*`visibilidad`*] `enum` *`Nombre { VALOR_1, VALOR_2, ... }`*. Incluyen una constante del propio tipo para cada valor, y estas definen el dominio de valores de la clase, así como métodos `public int ordinal()` que devuelve el índice del valor y `public static` *`Tipo`*`[] values()` que devuelve una lista con todas los valores posibles.

- Los ***arrays*** son objetos que contienen un número fijo de elementos de otro tipo. Su tipo es *`TipoElemento`*`[]...[]` (pueden tener varias dimensiones). Su tamaño puede consultarse mediante el atributo `public final int length`.

El operador == comprueba si dos elementos de un tipo primitivo son iguales o si dos referencias apuntan al mismo objeto (igual *oid*), y en particular si dos elementos de un tipo enumerado tienen el mismo valor (devuelve un booleano).

Los objetos se comunican entre sí mediante **mensajes**, formados por un **objeto receptor** (si no es un método de clase), un **identificador de método** y los **argumentos**. Enviar un mensaje es llamar al método, definido en la clase del objeto, con la sintaxis *`objeto`*`.`*`método`*`(`*`args`*`, ...)` o, si es un método de clase, con *`Clase`*`.`*`método`*`(`*`args`*`, ...)`. Los argumentos se pasan por valor. Este método (si no es de clase) puede acceder al objeto sobre el que se llama (la **instancia actual**) por la palabra reservada `this`. Además, puede referirse a los métodos y atributos de la instancia actual (si no es un método de clase) o de la propia clase sin especificar la instancia actual o la clase, siempre que no haya un parámetro o variable local del mismo nombre. Un constructor puede invocar a otro mediante `this(`*`args`*`, ...);`.

Dentro de un método, la sentencia *`Tipo var`* [`=` *`expr`*]`;` declara una variable local y le asigna opcionalmente un valor, y la expresión `new` *`Clase`*`(`*`arg`*`, ...)` asigna el espacio para la estructura de un objeto, le asigna un identificador, inicializa los campos con su valor por defecto, llama al constructor corrspondiente y devuelve dicho objeto. Para crear un *array* se usa `new` *`TipoElemento`*`[`*`t1`*`]...[`*`tn`*`]`[`{...{`*`valor11`*`, ...}, ...}`], con el número de elementos en cada dimensión, lo que asigna el espacio necesario al *array*, le asigna un identificador e inicializa cada elemento con su valor por defecto o con los valores indicados entre llaves. La destrucción de los objetos no es explícita, sino que el **recolector de basura** (*garbage collector*) periódicamente libera los objetos que no son referenciados. Un objeto puede ser notificado antes de ser eliminado implementando el método `finalize()`, lo que interesa cuando este hace uso de recursos externos. Estructuras de control:

- Sentencias condicionales: `if (`*`cond-expr`*`)` *`then-stmt`* [`else` *`else-stmt`*] (*`cond-expr`* debe evaluar a booleano); `switch (`*`expr`*`) { ... }` (*`expr`* debe evaluar a entero, valor de enumerado o cadena, y es necesario usar `break;` para evitar que la ejecución continúe con el caso siguiente).
- Bucles: `while`, `for`.

Una aplicación OO se organiza como un conjunto de clases relacionadas, siendo una de ellas la **raíz de la aplicación**, que contiene un método (en Java, `public static void main(String[] args)`) encargado de poner en marcha la aplicación. El flujo de ejecución siempre se encuentra aplicando un método sobre un objeto o ejecutando alguna instrucción imperativa, si bien en una aplicación concurrente puede haber varios flujos de ejecución.

Si una clase tiene muchos atributos de tipo primitivo, suele ser necesario separar la funcionalidad definiendo clases con parte de estos atributos. Se usa el patrón **experto en información**: asignar una funcionalidad a la clase que tiene la información necesaria para llevarla a cabo.

Capítulo 3

Herencia

La **herencia** es un mecanismo para definir y utilizar relaciones conceptuales entre clases como de extensión, especialización y combinación, que permite definir una clase a partir de otra, organizando las clases en una **jerarquía** consistente con el sistema de tipos. La **especialización** ocurre cuando una clase es un caso especial o un tipo de otra, mientras que la **generalización**, muy relacionada, ocurre cuando se detectan clases con características en común.

Cuando una clase B **hereda** de una clase A, incorpora automáticamente los atributos y métodos de A (en Java, no incorpora los constructores), pero puede añadir nuevos atributos y métodos y redefinir métodos heredados. Decimos que A es la **superclase** o clase padre de B y B es una **subclase** o clase hija de A. Si C hereda a su vez de B, también hereda de A, pero B es un **descendiente directo** y C es un **descendiente indirecto** de A. Decimos que A y B son **ascendientes** de C. Algunos lenguajes como C++ o Python permiten **herencia múltiple**, de modo que una clase puede heredar directamente de varias, mientras que otros, como Java o C#, solo permiten **herencia simple**.

En Java se indica que una clase deriva de otra indicando `extends` *`Superclase`* detrás del nombre de la clase en su definición. Los descendientes de una clase no pueden ver sus atributos y métodos privados, sino que se proporciona el nivel de visibilidad `protected`, útil para métodos, que da visibilidad a nivel de paquete y a descendientes de la clase.

La primera sentencia de un constructor debe ser una llamada a un constructor de la clase padre, que se hace con `super(...);` o a otro constructor de la propia clase, y si se omite el compilador inserta una llamada `super();` al constructor vacío, dando error si este no existe.

El modificador `final` se puede añadir a un método para impedir que sea redefinido en clases descendientes, y en una clase para impedir que se pueda heredar de ella.

3.1. Polimorfismo

Un método es una **redefinición** si tiene la misma signatura (nombre, parámetros y tipo de retorno) que un método de la clase padre. En Java se indica con la **anotación** `@Override` encima de la definición de la función, que es opcional pero señala un fallo si el método no es realmente una redefinición, útil para detectar errores. Esta puede ser de **reemplazo** si se sustituye completamente la implementación o de **refinamiento** si simplemente se añade nueva

funcionalidad, en cuyo caso la sintaxis `super.`*`método`*`(...)` sirve para llamar a la versión del padre de un método redefinido.

El **polimorfismo** es la capacidad de una entidad de referenciar en tiempo de ejecución a objetos de distintas clases, y en lenguajes como Java significa que toda entidad tiene un **tipo estático** (*te*), asociado a su declaración, y uno **dinámico**, que corresponde al objeto al que hace referencia y puede variar. A cada entidad le corresponde un **conjunto de tipos dinámicos** (*ctd*), el de los posibles tipos dinámicos a los que puede hacer referencia y que viene dado por el conjunto de tipos **compatibles** con el tipo estático, es decir, los descendientes de este tipo incluyéndose a sí mismo.

Una asignación polimórfica es válida si el tipo estático de la parte derecha es compatible con el de la parte izquierda. Un paso de parámetros es válido si el tipo estático del parámetro real es compatible con el del parámetro formal. La **ligadura dinámica** significa que la versión de un método que se llama al pasar un mensaje es la asociada al tipo dinámico, que puede ser la del tipo estático o haber sido redefinida.

En Java, la sintaxis `(`*`Tipo`*`)`*`variable`* permite hacer un *casting* a uno de los *`Tipo`*s dinámicos que puede tener la *`variable`*, permitiendo tratar la variable como si este fuese su tipo estático y pudiendo acceder a los métodos de este, si bien esto dará un error en tiempo de ejecución si el tipo dinámico de la variable no es compatible con el que se especifica al hacer el *casting*. La expresión *`variable`* `instanceof` *`Tipo`* devuelve verdadero si el tipo dinámico de la *`variable`* es compatible con el del *`Tipo`* y falso en caso contrario.

Al redefinir un método (al que se tenga acceso desde la clase descendiente) se puede cambiar el tipo de retorno a un tipo descendiente (**regla covariante**) o incrementar su nivel de visibilidad.

Se recomienda situar los atributos y métodos comunes en clases altas de la jerarquía, aplicando herencia si tiene sentido decir que todos los objetos de una clase «lo son» también de otra, usando polimorfismo y ligadura dinámica para evitar análisis de casos. No debe usarse herencia si un método heredado no tiene sentido en la clase hija, y no se debe cambiar la semántica de un método en la clase hija.

3.2. Clases abstractas

Una **clase abstracta** es una que no se puede instanciar, no puede ser `final` y permite definir **métodos abstractos**, sin código, con `;` en vez de la implementación (`{...}`), y que no pueden ser `static` ni `final`. Las clases y métodos abstractos se deben indicar con el modificador `abstract`, y una clase **efectiva** (no abstracta) debe implementar todos los métodos abstractos que hereda. Una clase abstracta puede implementar constructores, pero estos sólo son útiles para su uso por las clases hijas, por lo que se declaran `protected`.

Un **método plantilla** es un método ordinario de una clase abstracta que usa uno o más métodos abstractos, normalmente `protected`, con el fin de evitar repetición de código en las clases descendientes. La generalización por clases abstractas se aplica muchas veces tras la implementación (**refactorización**).

3.3. Interfaces

Una **interfaz** es la definición de un tipo sin la implementación, como una clase totalmente abstracta, y se definen con [*visibilidad*] [*mods*] `interface` *Nombre* [`extends` *Interfaz1*`,` ...] `{ ... }`. Las interfaces no tienen constructores, sus métodos son `public abstract` por defecto (aunque puede haber métodos no abstractos) y sus atributos son `public static final`, y no es necesario indicar estos modificadores.

Una interfaz puede heredar de varias interfaces con `extends`. Una clase puede heredar de varias interfaces (se dice que las **implementa**), y se indica con `implements` *Interfaz1*`,` ....

Las interfaces pueden incluir **métodos por defecto**, con el modificador `default`, que no son abstractos. Una clase puede implementar varias interfaces con distintos métodos por defecto con la misma signatura, en cuyo caso para que una clase compile debe reimplementar este método. La sintaxis *Interfaz*`.super.`*método*`(...)` permite llamar al método por defecto de una interfaz «padre» cuando hay ambigüedad. Las interfaces también pueden incluir métodos `static`.

3.4. La clase `Object`

Cuando una clase no hereda de ninguna, realmente hereda de `Object`. Métodos de `Object`:

1. `public final Class getClass()`. Devuelve la clase de la instancia actual. Se recomienda usar `instanceof` en vez de esto en caso de duda para comprobar compatibilidad.

2. `public boolean equals(Object)`. Indica si un objeto es igual a otro.
   Tipos de igualdad: **superficial**, que compara la igualdad de los campos primitivos y el *oid* de las referencias; **profunda**, que usa `equals` recursivamente, o adaptada a las necesidades de la aplicación. Por defecto la comparación es superficial.
   Es necesario redefinir el método en las clases donde necesitemos igualdad, pero hay que elegir la semántica de igualdad más adecuada. El método heredado en una subclase es correcto si no tiene nuevos atributos o estos no se tienen en cuenta en la igualdad, y en caso contrario, salvo que la versión heredada sea la de `Object`, hay que reutilizarla.

3. `public int hashCode()`. Devuelve el resultado de aplicar alguna función *hash* al objeto; usado en tablas de dispersión. Si *o1*`.equals(`*o2*`)`, debe ser *o1*`.hashCode() ==` *o2*`.hashCode()`. Por defecto se compara la dirección de memoria del objeto. Si se redefine el método `equals` también debe redefinirse `hashCode`.

4. `public String toString()`. Devuelve una representación textual de un objeto. Por defecto se imprime el nombre de la clase y el *oid*. Se recomienda usar `getClass().getName()`, que devuelve la clase de la instancia actual, permitiendo que la implementación sea heredable. Debe ser redefinido en una subclase si añade nuevos atributos.

5. `protected Object clone() throws CloneNotSupportedException`. Devuelve una copia del objeto.
   Tipos de copia: **superficial**, que copia los campos primitivos y los *oid*; **profunda**, que aplica `clone` recursivamente, y adaptada a las necesidades de la aplicación. Por defecto la copia es superficial.
   Al redefinir el método, hay que cambiar la visibilidad a `public`, cambiar el tipo de retorno

al tipo propio y ocultar la excepción (el `throws ...`). Esta excepción se lanza si la clase del objeto no implementa la interfaz `Cloneable`, que no implementa ningún método pero marca que los objetos de una clase (y todas las clases descendientes) pueden ser clonadas. Al redefinir este método, para que funcione con la herencia, es conveniente llamar a la versión heredada y, si el tipo padre no es `Cloneable`, manejar la excepción mediante `try {` (código que llame a `super.clone()`) `} catch (CloneNotSupportedException e) {` (manejo del error) `}` de modo que si salta la excepción se salte directamente al código de manejo del error, lo que no debería ocurrir.

Además, para cada tipo primitivo existe una clase (`Integer`, `Float`, `Double`, `Character`, `Boolean`, etc.) correspondiente que actúa como **objeto envoltorio** y hereda de `Object`. Java permite convertir automáticamente entre el tipo primitivo y el envoltorio (***autoboxing***).

Capítulo 4

Genericidad

Una **clase genérica** es aquella que en su declaración tiene uno o varios tipos variables (**parámetros**). Se indican como *NombreClase*`<`T_1 [`extends` K_{11}`[&` ...]][`,` ...]`>`, y dentro de la clase se pueden usar los tipos $\texttt{T}_i$ como otros tipos cualesquiera, sobre los que se pueden usar los métodos de los tipos no primitivos $\texttt{K}_{ij}$, o de `Object` si no se especifican, salvo constructores. El uso de tipos $\texttt{K}_{ij}$ permite **genericidad restringida**.

También existen las interfaces genéricas, con la misma sintaxis. Una clase genérica puede heredar de otra e implementar interfaces genéricas. En la herencia se establecen los parámetros de la clase padre e interfaces implementadas, que pueden ser parámetros de la propia clase hija o tipos concretos.

La parametrización de una clase genérica se realiza en la declaración de una variable (incluyendo parámetro, etc.) y en la construcción de objetos, con la forma *NombreClase*`<`T_1`, ...>`, donde los $\texttt{T}_i$ son los tipos parámetros y deben ser compatibles con todos los $\texttt{K}_{ij}$, si bien en la construcción se pueden omitir los parámetros con la sintaxis *NombreClase*`<>`. En la declaración de variables los T_i se pueden cambiar por `?` `extends` E_i (**tipo comodín**) para indicar que se permite cualquier tipo compatible con E_i, o `?` `super` S_i para cualquier tipo con el que S_i sea compatible.

También podemos declarar un tipo genérico sin especificar «`<...>`» con los parámetros de tipo (**tipo puro**), en cuyo caso equivale a parametrizarlo a `Object` (salvo genericidad restringida), si bien esto no es recomendable y el compilador lo marca como un aviso. La información de los tipos parámetro se pierde en tiempo de ejecución, pues no se almacenan en el código compilado, por lo que solo podemos comprobar si el tipo dinámico de una variable es o no del tipo puro (o compatible). El compilador marca una conversión a un tipo genérico como un *warning*, pues no se puede comprobar el tipo utilizado en la parametrización, pero podemos suprimirlo escribiendo la línea `@SuppressWarnings("unchecked")` encima de la sentencia que realiza la conversión.

Un **método genérico** es el que en su declaración acepta uno o varios parámetros de tipo, independientes de los de la clase (si los hay), que pueden aparecer en la signatura y el cuerpo del método y que se indican poniendo `<...>` (sintaxis como en las clases genéricas) antes del tipo de retorno (o de `void`). Se llaman como un método normal, y los tipos se infieren a partir de sus parámetros o la variable a la que se asigna el resultado.

4.1. Ordenación

Las clases e interfaces que se definen a continuación están todas en `java.util`, salvo si se indica lo contrario. Se omite cualquier declaración, o parte de declaración, que no es relevante.

`public interface java.lang.Comparable<T>` Una clase es **comparable** si implementa esta interfaz, y si dice que tiene un **orden natural**.

- `int compareTo(T)` Devuelve un entero positivo si el objeto receptor es mayor, negativo si es menor y 0 si es igual al parámetro.

`@FunctionalInterface public interface Comparator<T>`

- `int compare(T, T)` Devuelve un entero positivo si el primer parámetro es mayor, negativo si es menor y 0 si es igual al segundo.

4.2. Iteradores

Permiten recorrer una secuencia de elementos.

`public interface Iterator<E>`

- `boolean hasNext()` Indica si quedan elementos en la iteración.
- `E next()` Devuelve (y «consume») el siguiente elemento en la iteración.
- `default void remove()` Elimina de la colección el último elemento devuelto por el iterador, y sólo puede ser llamado una vez después de cada llamada a `next()`. Si la colección se modifica mientras el iterador está en progreso, su comportamiento es no especificado.

`public interface java.lang.Iterable<T>`

- `Iterator<T> iterator()` Devuelve un iterador sobre elementos de tipo `T`.

Dado un objeto `c` que implementa la interfaz `Iterable<`*`T`*`>`, la sintaxis `for (`*`T e : c`*`)` *`stmt`* equivale a:

```
Iterator _it = c.iterator();
while (_it.hasNext()) {
        T e = _it.next();
        stmt
}
```

Esto también se puede hacer si `c` es de tipo *`T`*`[]`, en cuyo caso se hace recorrido secuencial.

4.3. Colecciones

`public interface Collection<E> extends Iterable<E>` Colección de elementos.

- `int size()` Devuelve el número de elementos en la colección, si cabe en un entero.
- `boolean isEmpty()` Indica si la colección está vacía.

`boolean contains(Object)` Indica si la colección contiene el elemento dado. Es un error común pasar un elemento de tipo no compatible con `E`.

`boolean add(E)` Se asegura de que la colección contenga el elemento especificado, indicando si la colección ha cambiado como resultado de la llamada.

`boolean remove(Object)` Elimina una única instancia del elemento especificado de la colección, si esta presente, indicando si la colección ha cambiado como resultado de la llamada.

`boolean addAll(Collection<? extends E>)` Añade a esta colección todos los elementos de otra, indicando si ha cambiado como resultado de la llamada.

`boolean removeAll(Collection<?>)` Elimina de essta colección todos los elementos en común con otra, indicando si ha cambiado como resultado de la llamada.

`void clear()` Elimina todos los elementos de la colección, que quedará vacía.

`default Stream<E> stream()` Devuelve un flujo secuencial con esta colección como fuente.

`public abstract class AbstractCollection<E> implements Collection<E>`

`public String toString()` Devuelve una representación como cadena de caracteres de la colección como una lista de sus elementos.

`public interface List<E> extends Collection<E>` Lista, colección ordenada o secuencia.

`boolean add(E)` Añade un elemento al final de la lista y devuelve `true`.

`default void sort(Comparator<? super E>)` Ordena una lista de acuerdo a un comparador.

`boolean equals(Object)` Indica si el otro objeto es una lista del mismo tamaño y con los mismos elementos en las mismas posiciones.

`E get(int)` Devuelve el elemento en una determinada posición en el rango $[0, \texttt{size}())$.

`E set(int, E)` Sustituye el elemento en la posición dada, en el rango $[0, \texttt{size}())$, por el elemento dado.

`void add(int, E)` Inserta el elemento dado en la posición dada, en el rango $[0, \texttt{size}()]$, moviendo todos los elementos posteriores una posición a la derecha.

`E remove(int)` Elimina el elemento en la posición especificada, en el rango $[0, \texttt{size}())$, moviendo todos los elementos posteriores una posición a la izquierda.

`public abstract class AbstractList<E>extends AbstractCollection<E>implements List<E>`

`public abstract class AbstractSequentialList<E> extends AbstractList<E>`

`public class LinkedList<E>extends AbstractSequentialList<E>implements List<E>, Cloneable`
Lista doblemente enlazada, con la que podemos gestionar pilas y colas.

`LinkedList()` Construye una lista vacía.

`LinkedList(Collection<? extends E>)` Construye una lista con los elementos de la colección dada, en el orden en que los devuelve el iterador.

`public E getFirst() throws NoSuchElementException` Devuelve el primer elemento.

`public E getLast() throws NoSuchElementException` Devuelve el último elemento.

`public E removeFirst() throws NoSuchElementException` Elimina y devuelve el primer elemento.

`public E removeLast() throws NoSuchElementException` Elimina y devuelve el último elemento.

`public void addFirst(E)` Inserta un elemento al principio.

`public void addLast(E)` Inserta un elemento al final.

`public Object clone()` Devuelve una copia superficial (no se copian los elementos).

`public class ArrayList<E> expands AbstractList<E> implements List<E>, Cloneable` Lista contigua redimensionable.

`ArrayList()` Construye una lista vacía.

`ArrayList(Collection<? extends E>)` Construye una lista con los elementos de la colección dada, en el orden en que los devuelve el iterador.

`public Object clone()` Devuelve una copia superficial (no se copian los elementos).

`public interface Set<E> extends Collection<E>` Conjunto o colección sin elementos duplicados.

`boolean add(E)` Añade un elemento si no estaba ya presente, indicando si se ha añadido (si no estaba ya presente).

`boolean equals(Object)` Indica si el objeto dado es un conjunto de igual longitud que este y todos los miembros del conjunto dado están contenidos en este (o, equivalentemente, todos los de este están en el otro).

`public abstract class AbstractSet<E>extends AbstractCollection<E>implements Set<E>`

`public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable` Conjunto implementado por tabla de dispersión.

`HashSet()` Construye un conjunto vacío.

`HashSet(Collection<? extends E>)` Construye un conjunto con los elementos de la colección dada.

`public Object clone()` Devuelve una copia superficial (no se copian los elementos).

`public interface SortedSet<E> extends Set<E>` Conjunto totalmente ordenado, por el orden natural de `E` o por un `Comparator<? super E>`. El iterador devuelve los elementos en orden ascendente.

`SortedSet<E> headSet(E)` Devuelve una vista de los elementos estrictamente menores que el dado, y que depende del original. Una vista es un objeto que permite acceder a otro con una interfaz algo distinta.

`SortedSet<E> tailSet(E)` Devuelve una vista de los elementos estrictamente mayores que el dado, y que depende del original.

`E first()` Devuelve el menor elemento del conjunto.

`E last()` Devuelve el mayor elemento del conjunto.

`public interface NavigableSet<E> extends SortedSet<E>`

`public class TreeSet<E>extends AbstractSet<E>implements NavigableSet<E>, Cloneable`
Conjunto implementado por árbol rojo-negro.

`TreeSet()` Construye un conjunto vacío.

`TreeSet(Collection<? extends E>)` Construye un conjunto con los elementos de la colección dada.

`TreeSet(Comparator<? super E>)` Construye un conjunto vacío con el comparador dado.

`public Object clone()` Devuelve una copia superficial (no se copian los elementos).

`public class Collections` Utilidades estáticas para colecciones.

`public static <T extends Comparable<? super T>> void sort(List<T>)`
Ordena una lista según el orden natural de sus elementos.

`public static <T> void sort(List<T>, Comparator<? super T>)`
Ordena una lista según un comparador.

`@SafeVarargs public static <T> boolean addAll(Collection<? super T>, T...)`
Añade todos los elementos a la colección dada, indicando si la colección ha cambiado como resultado de la llamada.

`public static <T> Set<T> unmodifiableSet(Set<? extends T>)` Devuelve una vista no modificable del conjunto dado. Para dar acceso de solo lectura a un conjunto interno a una clase sin necesidad de hacer una copia.

`public static <T> List<T> unmodifiableList(List<? extends T>)` Devuelve una vista no modificable de la lista dada.

`public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V>)`
Devuelve una copia no modificable del mapa dado.

`public class Arrays` Utilidades estáticas para *arrays*.

`@SafeVarargs public static <T> List<T> asList(T...)` Devuelve una vista de un *array* como lista de tamaño fijo.

4.4. Mapas

`public interface Map<K,V>` Mapa o diccionario clave-valor.

- `int size()` Devuelve el total de asociaciones clave-valor.
- `boolean isEmpty()` Indica si el mapa es vacío.
- `boolean containsKey(Object)` Indica si el mapa contiene una asociación para la clave dada.
- `boolean containsValue(Object)` Indica si el mapa contiene una asociación con el valor dado.
- `V get(Object)` Obtiene el valor asociado a la clave dada, o `null` si el mapa no tiene asociación para la clave.
- `V put(K, V)` Asocia el valor dado a la clave dada, sustituyendo la asociación anterior para esta clave de haber alguna.
- `V remove(Object)` Elimina la asociación para la clave dada, si existe.
- `void putAll(Map<? extends K, ? extends V>)` Copia todas las asociaciones del mapa dado al objeto receptor.
- `void clear()` Elimina todas las asociaciones del mapa, dejándolo vacío.
- `Set<K> keySet()` Devuelve una vista de las claves del mapa, que depende de este. Si el mapa se modifica mientras se itera la vista, el comportamiento es no especificado. Eliminar un elemento del conjunto elimina la asociación correspondiente del mapa.
- `Collection<V> values()` Devuelve una vista de los valores del mapa, que depende de este. Si el mapa se modifica mientras se itera la vista, el comportamiento es no especificado. Eliminar un elemento de la colección elimina la asociación correspondiente del mapa.
- `default V getOrDefault(Object, V)` Devuelve el valor asociado a la clave dada, o el que se le pasa como parámetro si no hay ninguno asociado.
- `default V putIfAbsent(K, V)` Si la clave dada no está asociada ningún valor (o está asociada a `null`), la asocia al valor dado, devolviendo `null` si este es el caso o su valor actual en caso contrario.

`public class AbstractMap<K,V> implements Map<K,V>`

`public class HashMap<K,V>extends AbstractMap<K,V>implements Map<K,V>, Cloneable`
Diccionario implementado por tabla de dispersión.

- `HashMap()` Construye un mapa vacío.
- `HashMap(Map<? extends K,? extends V>)` Construye un mapa con las asociaciones del mapa dado.

`public interface SortedMap<K,V> extends Map<K,V>` El iterador de `keySet()` devuelve las claves en orden ascendente.

`public interface NavigableMap<K,V> extends SortedMap<K,V>`

`public class TreeMap<K,V>extends AbstractMap<K,V>implements NavigableMap<K,V>, Cloneable`

Diccionario implementado por árbol rojo-negro.

`TreeMap()` Construye un mapa vacío.

`TreeMap(Map<? extends K,? extends V>)` Construye un mapa con las asociaciones del mapa dado.

`TreeMap(Comparator<? super K>)` Construye un mapa vacío con el comparador dado.

4.5. Recomendaciones

1. En constructores y métodos públicos, especificar el tipo de retorno y tipo de los parámetros mediante la interfaz, y no la clase.

2. Evitar el uso de *arrays*, pues tienen tamaño fijo, no definen las operaciones fundamentales de `Object` y es necesario usar `java.util.Arrays` para estas.

Capítulo 5

Programación funcional

En Java, la sintaxis `new` *`Clase`*`[<`*`T1`*`, ...>](`*`Arg1`*`, ...) { ... }` se puede usar para crear una clase anónima que herede de *`Clase`*. Una **interfaz funcional** es una que se declara con la anotación (opcional) `@FunctionalInterface` encima de la definición y que contienen un sólo método abstracto.

Las **expresiones lambda** tienen sintaxis `(([`*`Tipo`*`] `*`var`*`, ...)|`*`var`*`) ->({ `*`cuerpo`*` }|`*`expresión`*`)`, donde *`cuerpo`* es el cuerpo de una función, *`expresión`* es una única expresión que equivale a `{ return` *`expresión`*`; }` y el *`Tipo`* de los argumentos normalmente se puede inferir. Internamente son clases compatibles con las interfaces funcionales que se requieran siempre que su signatura sea compatible con la de su único método abstracto. Si la expresión lambda se refiere a variables de su entorno léxico (variables locales o elementos de la clase), las instancias de su clase asociada se denominan **clausuras** (*closures*).

Salvo que se indique lo contrario, las siguientes interfaces se definen en `java.util.function`.

```
@FunctionalInterface public interface Predicate<T>

    boolean test(T)

@FunctionalInterface public interface Function<T,R>

    R apply(T)

@FunctionalInterface public interface Supplier<T>

    T get()

@FunctionalInterface public interface Consumer<T>

    void accept(T)
```

El paquete `java.util.stream` contiene clases para soportar operaciones funcionales en flujos de elementos. Las operaciones con flujos se dividen en operaciones **intermedias** y **terminales**. Las operaciones intermedias devuelven un nuevo flujo. Las operaciones terminales pueden atravesar el flujo para producir un resultado o un efecto colateral.

`public interface java.util.stream.Stream<T>` Secuencia de elementos que soporta operaciones secuenciales y paralelas en agregado.

`Stream<T> filter(Predicate<? super T>)` Devuelve un flujo con los elementos de este para los que se cumple el predicado.

`<R> Stream<R> map(Function<? super T,? extends R>)` Devuelve un flujo con el resultado de aplicar la función dada a los elementos de este.

`Stream<T> sorted()` Devuelve un flujo con los elementos de este, ordenados según su orden natural.

`Stream<T> sorted(Comparator<? super T>)` Devuelve un flujo con los elementos de este, ordenados según el comparador dado.

`void forEach(Consumer<? super T>)` Realiza una acción en cada elemento.

`long count()` Devuelve el número de elementos en el flujo.

`boolean anyMatch(Predicate<? super T>)` Indica si algún elemento de este flujo cumple el predicado.

`boolean allMatch(Predicate<? super T>)` Indica si todos los elementos de este flujo cumplen el predicado.

Donde sea aceptable una expresión lambda también es aceptable una referencia a un método:

- *`Clase`*`::`*`métodoDeClase`* equivale a `e ->`*`Clase`*`.`*`métodoDeClase`*`(e)`.
- *`Clase`*`::`*`métodoDeInstancia`* equivale a `(`*`Clase`*` e) ->e.`*`métodoDeInstancia`*`()`.
- *`objeto`*`::`*`método`* equivale a `e ->`*`objeto`*`.`*`método`*`(e)`.
- *`Clase`*`::new` equivale a `e ->new `*`Clase`*`(e)`.

Capítulo 6

Excepciones

La **corrección** es la capacidad del software de cumplir con su especificación, de modo que el incumplimiento de los requisitos de uso implica la finalización de la ejecución. La **robustez** es la capacidad del software de reaccionar adecuadamente a situaciones excepcionales (fallos de disco, red, etc.). En Java, para esto se usan excepciones.

Una excepción es un objeto de clase compatible con `java.lang.Exception`, que podemos lanzar con la cláusula `throw` *`objeto`*`;`. Para capturar una excepción usamos la construcción `try`–`catch`, con la sintaxis:

`try {` *`stmt`*`... }`
`catch (`*`Tipo var`*`) {` *`stmt`*`... }` ...
[`finally {` *`stmt`*`... }`]

Si se lanza una excepción en el bloque `try`, se comprueba el *`Tipo`* en cada bloque `catch` (que debería ser compatible con `java.lang.Exception`) en orden y, si se encuentra uno compatible con el tipo dinámico de la excepción, se ejecuta el interior de dicho bloque con la variable *`var`* (de tipo estático *`Tipo`*) asociada al objeto excepción, y decimos que la excepción ha sido capturada. El bloque `finally`, si existe, se ejecuta tanto si la excepción es capturada como si no. Finalmente, si la excepción no es capturada, el resultado es como si la propia construcción lanzara la excepción.

Las siguientes clases se definen en `java.lang`.

`public class Throwable` Superclase de todos los errores y excepciones en el lenguaje. Sólo instancias de esta clase (o compatibles) son lanzadas por la máquina virtual de Java o la cláusula `throw`. Solo esta clase o compatibles pueden ser el tipo de argumento de una cláusula `catch`. `Throwable` y cualquier clase compatible con `Throwable` que no lo es también con `RuntimeException` o `Error` se consideran excepciones comprobadas.

- `public String getMessage()` Deuvuelve la cadena de mensaje de este lanzable.
- `public void printStackTrace()` Imprime este lanzable y su *backtrace* (estado de la pila de llamadas en el momento de crear la excepción) al flujo de error estándar.

`public class Exception` Forma de `Throwable` que indica condiciones que una aplicación razonable querría capturar.

- `public Exception(String)` Construye una excepción con el mensaje especificado.

Si se lanza una excepción fuera de un bloque `try`, se termina la ejecución del método y el resultado es como si la llamada a dicho método lanzara la excepción, y decimos que el método produce la excepción. Si un método puede producir una excepción comprobada, debemos escribir `throws` *`Tipo1`*`,` ... detrás de la lista de parámetros (`(...)`) y antes de la implementación (`{...}`) o el `;` para métodos abstractos, siendo *`Tipo1`*`,` ... una lista de excepciones tales que cualquier excepción comprobada que pueda producir el método sea de tipo compatible con uno de la lista. Entonces se considera que el método puede producir un error de cualquier tipo de la lista. Si el método `main` de la aplicación produce una excepción, el programa termina con error.

Si bien las excepciones permiten aportar robustez, también permiten controlar el uso correcto de operaciones, notificando de errores de programación. Para ello se usan las **excepciones en tiempo de ejecución**, compatibles con `java.lang.RuntimeException`, y que en general no se tratan. La máquina virtual también puede lanzar excepciones de este tipo.

`public class RuntimeException extends Exception` Superclase de las excepciones que pueden ser lanzadas durante la operación normal de la máquina virtual de Java.

`public class NullPointerException extends RuntimeException` Lanzada cuando una aplicación intenta usar `null` cuando se requiere un objeto. Las aplicaciones deberían lanzar instancias de esta clase para otros usos no permitidos de `null`.

`public class IllegalArgumentException extends RuntimeException` Lanzada para indicar que a un método se le ha pasado un argumento inapropiado.

`public class IllegalStateException extends RuntimeException` Señala que el entorno o la aplicación no está en un estado apropiado para la operación solicitada.

`public class NoSuchElementException extends RuntimeException` Lanzada por algunos métodos accesores para indicar que el elemento solicitado no existe.

`public class ClassCastException extends RuntimeException` Lanzada para indicar que el código ha intentado hacer *casting* a un objeto hacia una subclase a la que no pertenece.

`public class IndexOutOfBoundsException extends RuntimeException`

`public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException` Lanzada para indicar que se ha accedido a un *array* con un índice no permitido.

El **diseño por contrato** es un modo de proceder ante los errores de programación basado en que las operaciones tienen requisitos de uso o **precondiciones** con los que se considera un error no cumplir, y si no se cumplen el programa no debe continuar.

Apéndice A

Algo más de la API

`java.io`

`public interface Closeable` Fuente o destino de datos que puede ser cerrado.

- `void close()` Cierra este flujo y libera cualquier recurso asociado con él.

`public class FileNotFoundException extends IOException` Indica que ha fallado un intento de abrir el fichero con nombre especificado.

`public class IOException extends Exception` Señaliza una excepción de E/S.

- `public IOException(String)` Construye una `IOException` con el mensaje especificado.

`public abstract class InputStream implements Closeable` Representa un flujo de entrada de bytes.

`public class PrintStream` Añade a otro flujo de salida la habilidad de imprimir representaciones de datos convenientemente.

- `public void println(String)` Imprime una cadena y termina la línea.

`public abstract class PrintWriter` Imprime representaciones de objetos con formato a un flujo de salida de texto.

- `public PrintWriter(String) throws FileNotFoundException` Construye un nuevo `PrintWriter` con el nombre de fichero indicado.
- `public boolean checkError()` Comprueba su estado de error.
- `public void println(String)` Imprime una cadena y termina la línea.

java.lang

`public final class Integer`

- `public static String toHexString(int)` Devuelve una representación textual del argumento entero como entero sin signo en base 16, una cadena de dígitos ASCII en hexadecimal sin `0`s extra a la izquierda.

`public final class Math`

- `public static final double PI` El valor de doble precisión más cerca que cualquier otro a π.
- `public static double sqrt(double)` Devuelve la raíz cuadrada positiva de un valor correctamente redondeada.
- `public static double pow(double, double)` Devuelve el primer argumento elevado a la potencia del segundo.

`public final class String implements CharSequence` Representa cadenas inmutables de caracteres. Todos los literales de cadena en programas Java, como `"abc"`, son instancias de esta clase. El lenguaje proporciona soporte especial para el operador de concatenación (+), y para conversión de otros objetos a cadenas al concatenar. Las conversiones se implementan mediante el método `toString`.

- `public boolean contains(CharSequence)` Indica si esta cadena contiene la secuencia de caracteres especificada.
- `public int length()` Devuelve el número de unidades de código Unicode en la cadena.

`public final class System`

- `public static final InputStream in` El flujo de entrada estándar.
- `public static final PrintStream out` El flujo de salida estándar.

java.net

`public class MalformedURLException extends IOException` Indica una URL mal formada.

`public final class URL` Representa una URL.

- `public URL(String) throws MalformedURLException` Construye una URL a partir de su representación como cadena.
- `public final InputStream openStream() throws IOException` Abre una conexión a la URL y devuelve un `InputStream` para leer de esta.

java.time

`public final class LocalTime implements ChronoLocalDate` Fecha sin zona horaria. Inmutable.

- `public static LocalDate now()` Obtiene la fecha actual del reloj del sistema en la zona horaria por defecto.
- `public static LocalDate of(int, int, int)` Obtiene una instancia de `LocalDate` desde un año, mes y día, respectivamente.

java.time.chrono

`public interface ChronoLocalDate`

- `default boolean isAfter(ChronoLocalDate)` Comprueba si esta fecha es posterior a la especificada.
- `default boolean isBefore(ChronoLocalDate)` Comprueba si es anterior.
- `default boolean isEqual(ChronoLocalDate)` Comprueba si es igual.

java.util

`public class Random` Genera números pseudoaleatorios.

- `public Random()` Construye un generador de números aleatorios.
- `public int nextInt(int)` Devuelve un valor entero pseudoaleatorio distribuido uniformemente entre 0 (inclusive) y el valor especificado (exclusive), extraído de la secuencia del generador.

`public final class Scanner implements Closeable` Escáner de texto simple.

- `public Scanner(InputStream)` Construye un escáner que produce valores del flujo de entrada especificado.
- `public String nextLine()` Devuelve el resto de la línea actual, excluyendo cualquier separador de línea al final. La posición se establece al principio de la siguiente línea.

`public final class UUID` Representa unn identificador único universal.

- `public static UUID randomUUID()` Obtiene un UUID usando un generador de números pseudoaleatorios.

Apéndice B

Despliegue de software Java

En Eclipse, todo el código fuente o compilado se encuentra en un directorio llamado **espacio de trabajo**. Para importar un proyecto, movemos su directorio al espacio de trabajo y, desde Eclipse, usamos «Archivo→Importar→General→Proyectos existentes en el espacio de trabajo», donde nos pide el directorio raíz introducimos la ruta del espacio de trabajo, pulsamos «Examinar» y, en el panel inferior «Proyectos», seleccionamos el que deseamos importar, y pulsamos «Finalizar».

El código fuente Java se guarda en ficheros *`Clase`*`.java` que, al compilarlos, se convierten en ficheros compilados o de *bytecode* *`Clase`*`.class`, uno por clase. Para ejecutar un programa usamos el comando `java [`*`OPCIÓN`*`]...` *`CLASE`*, donde *`CLASE`* es la ruta completa de la clase principal. Las clases se buscan según un *classpath*, una lista de directorios (o ficheros comprimidos) separados por `;`, de forma que en uno de ellos debe haber un fichero de *bytecode* cuya ruta relativa al directorio debe ser el resultado de cambiar los separadores `.` en la ruta completa de la clase por `/` y añadir al final la extensión `.class`. El *classpath* se indica al comando con la opción `-cp` *`CLASSPATH`*; si no aparece, se toma de la variable de entorno `CLASSPATH`, y si esta tampoco existe, se asume que es «`.`».

Un fichero JAR es un comprimido con extensión (normalmente) `.jar` que contiene los ficheros compilados y un **manifiesto** que describe el contenido. El manifiesto se encuentra en `META-INF/MANIFEST.MF` dentro del comprimido, y suele tener el siguiente aspecto:

`Class-Path:` *`ELEMENTO`* ...
`Main-Class:` *`RUTA_CLASE`*

Donde la entrada `Class-Path` indica el *classpath* como lista separada por espacios en la que `.` indica el propio fichero y las rutas relativas que no comienzan con `.` lo son respecto al directorio donde se encuentra el comprimido. `Main-Class` indica el nombre completo de la clase principal. Esto permite ejecutar un programa en Java con el comando `java -jar` *`fichero.jar`*, o abriendo el fichero desde un explorador de ficheros.

Si tenemos una biblioteca en un fichero JAR que queremos usar en nuestro proyecto, creamos dentro una carpeta `lib`, copiamos el comprimido dentro junto con su documentación JavaDoc (descomprimida) y, dentro de Eclipse, pulsamos F5 para actualizar el contenido de `lib`. Pulsamos el fichero de la biblioteca con el botón derecho del ratón, usamos «Vía de acceso de construcción→Añadir a la vía de construcción» y aparece una sección «Librerías referidas» con la biblioteca añadida. Pulsamos aquí con el botón derecho del ratón, en «Propiedades→Ubicación del JavaDoc→Vía de acceso de ubicación JavaDoc→Examinar» se-

leccionamos el directorio con la documentación, y pulsamos «Validar».

Para generar un archivo JAR de nuestro proyecto, usamos «Archivo→Exportar→Java» y podemos elegir «Archivo JAR» para empaquetar una biblioteca (sin clase principal), «Archivo JAR ejecutable» para un ejecutable o «JavaDoc» para extraer la documentación JavaDoc.

Si elegimos la segunda opción, tenemos las opciones:

- *Launch configuration*: Para elegir la clase principal de entre las posibles.
- *Export destination*: Dónde queremos que se guarde el fichero JAR.
- *Library handling*: «*Extract required libraries into generated JAR*» descomprime los ficheros JAR de las bibliotecas e introduce las clases en el JAR de salida. «*Package required libraries into generated JAR*» guarda los propios ficheros JAR de las bibliotecas dentro del nuestro. «*Copy required libraries into a sub-folder next to the generated JAR*» crea un directorio junto al fichero JAR donde mete las bibliotecas, que añade como dependencias en el `Class-Path` del manifiesto.

JavaDoc es una forma de generar documentación de código Java a partir de comentarios. Para ello, encima de cada clase, interfaz, etc., método o incluso paquete, creamos un comentario como el siguiente:

```
/**
 * Descripción.
 *
 * @param nombre Descripción del parámetro «nombre».
 * ...
 * @return Valor devuelto.
 * @throws Tipo Significado si el método lanza una excepción del Tipo dado.
 * ...
 * @seeAlso Clase,método,etc.
 */
```

Cualquier parte puede omitirse (de hecho muchas son específicas para métodos), y puede usarse código HTML o marcadores propios de JavaDoc (por ejemplo, para enlazar a otra clase o método). Si un método redefine a otro sin cambiar la semántica, podemos usar `/** {@inheritDoc} */` para heredar la documentación.