Empezar a programar usando Java

Natividad Prieto
Assumpció Casanova
Francisco Marqués
Marisa Llorens
Isabel Galiano
Jon Ander Gómez
Jorge González
Carlos Herrero
Carlos Martínez-Hinarejos
Germán Moltó
Javier Piris

3ª edición

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Natividad Prieto (coordinadora)
Assumpció Casanova
Francisco Marqués
Marisa Llorens
Isabel Galiano
Jon Ander Gómez
Jorge González
Carlos Herrero
Carlos Martínez-Hinarejos
Germán Moltó
Javier Piris

Empezar a programar usando Java

3ª edición

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Colección Académica

Para referenciar esta publicación utilice la siguiente cita: PRIETO SÁEZ, N., [et al] (2016) *Empezar a programar usando Java (3ª ed)*. Valencia: Universitat Politècnica

Primera edición, 2012
Segunda edición, 2013
Tercera edición, 2016

Imprime: Byprint Percom, sl.

ISBN: 978-84-9048-542-2
Impreso bajo demanda

Impreso en España

Prólogo

El lector, o más bien usuario, de este libro tiene en sus manos el esfuerzo de un grupo de profesores con amplia experiencia universitaria en la docencia de asignaturas de introducción a la Programación y Estructuras de Datos. Los primeros pasos que dan los estudiantes en estas disciplinas deben estar cuidadosamente guiados para asegurar la atención en lo relevante y la construcción ordenada de los conocimientos, que posteriormente deben aplicar al desarrollo de programas. Otro proceder lleva a la confusión de ideas y a la incertidumbre en su aplicación, ya que las posibilidades que ofrecen los lenguajes de programación son tan amplias que su uso desordenado, o mal aprendido, genera importantes limitaciones en los futuros graduados.

Un nuevo libro de introducción a la Programación es un reto importante en la medida que se requiere seleccionar, ordenar, o crear contenidos propios de la enseñanza de esta materia, de modo que se facilite la capacidad de aprendizaje de los alumnos, a la vez que se cubran todos los objetivos. Todo ello añadiendo aportaciones originales que hagan verdaderamente útil este modo de plantear la enseñanza. Con estas premisas se ha elaborado este libro, dirigido a los profesores y estudiantes de los primeros cursos de Programación. El libro plantea el objetivo de enseñar a programar utilizando Java como lenguaje vehicular. Es cuidadoso en el equilibrio entre enseñar a pensar algoritmos y su correspondiente implementación en un lenguaje. Se ha procurado que la estructura del libro sea clara y con una ordenación de los contenidos que permite una sencilla utilización como libro de texto de una asignatura. Este enfoque, junto a los numerosos ejemplos ilustrativos, lo hacen ideal para su uso en los primeros cursos de la universidad.

No me queda más en este prólogo que agradecer a los autores el trabajo realizado y felicitarles por la capacidad de aunar, filtrar, o componer ideas, venciendo la dificultad que este proceso plantea cuando son varias las personas participantes en un proyecto. Por eso tiene más valor este trabajo que ha generado un texto homogéneo y claro, que seguro que servirá a muchos profesores y estudiantes para el aprendizaje de la Programación en los próximos años.

Emilio Sanchis Arnal
Catedrático de Lenguajes y Sistemas Informáticos
DSIC - UPV

Agradecimientos

Este libro compila una gran cantidad de material docente (apuntes, transparencias, código, ejercicios, etc.) desarrollado a lo largo de muchos años y planes de estudio por los profesores de las primeras asignaturas de Programación de los estudios de Informática de la Universitat Politècnica de València. Así que, de una forma u otra, en este libro se pueden reconocer no solo las aportaciones e ideas de sus autores, sino también las de los distintos compañeros que, durante ese tiempo, han compartido con nosotros la tarea docente de estas asignaturas: el uso del lenguaje de Programación, el enfoque y metodología expositiva seguida en sus temas, los ejercicios y ejemplos en él planteados, ... Por todo ello, los autores no podemos menos que agradecer a estos compañeros su inestimable ayuda y apoyo a la hora de plantear en este libro y en nuestro día a día docente la Programación como una actividad de resolución de problemas por ordenador. También queremos agradecer a los estudiantes que nos han hecho llegar correcciones o sugerencias de mejora sobre el texto inicial.

Nota de los autores

Una publicación docente, como esta, no requiere únicamente el esfuerzo de creación y redacción inicial; tanto o más importante es la revisión y actualización de los contenidos y la bibliografía, máxime en una materia en continua evolución como la Programación. En concreto, en esta tercera edición, las modificaciones con respecto a la edición anterior, son las siguientes:

1. Se ha modificado el código de todos los capítulos para que se adapte a las normas de estilo habituales de Java, así como el código disponible en `http://users.dsic.upv.es/pubdocente/LibroIIPPRG/`.

2. Se han reescrito los capítulos 3 (*Variables y asignación. Tipos de datos elementales. Bloques*) y 4 (*Tipos de datos: clases y referencias*) para unificarlos en un único capítulo dedicado a variables (*Variables y tipos de datos: definición y uso*).

3. Se han introducido ejemplos y problemas adicionales en varios capítulos.

4. Se ha actualizado la bibliografía.

5. Se han corregido las erratas detectadas en diversos capítulos.

Índice general

Capítulo 1

Problemas, algoritmos y programas

Los conceptos que se desarrollarán a continuación son fundamentales en la mecanización del cálculo, objetivo de gran importancia en el desarrollo cultural humano que, además, ha adquirido una relevancia extraordinaria con la aparición y posterior universalización de los computadores. Problemas, algoritmos y programas forman el tronco principal en que se fundamentan los estudios de computación.

Dado un *problema* P, un *algoritmo* A es una secuencia finita de instrucciones, reglas o pasos, que describen de manera precisa cómo resolver P en un tiempo finito.

Aunque "cambiar una rueda pinchada a un coche" es un problema que incluso puede estudiarse y resolverse en el ámbito informático, no es el tipo de problema que habitualmente se resuelve utilizando un computador. Por su misma estructura, y por las unidades de entrada/salida que utilizan, los ordenadores están especializados en el tratamiento de secuencias de información (codificada) como, por ejemplo, series de números, de caracteres, de puntos de una imagen, muestras de una señal, etc. Un ordenador (o computador) puede verse como un mecanismo digital de propósito general que se convierte en un mecanismo para un uso específico cuando procesa un algoritmo o programa determinado.

Ejemplos más habituales de las clases de problemas que se plantearán en el ámbito de la programación a pequeña escala y, por lo tanto, en el de este libro, se pueden encontrar en el campo del cálculo numérico, del tratamiento de textos y de la representación gráfica, entre muchos otros. Algunos ejemplos de ese tipo de problemas son los siguientes:

- Determinar el producto de dos números multidígito a y b.
- Determinar la raíz cuadrada positiva del número 2.

- Determinar la raíz cuadrada positiva de un número n cualquiera.
- Determinar si el número n, entero mayor que 1, es primo.
- Dada la lista de palabras l, determinar las palabras repetidas.
- Determinar si la palabra p es del idioma castellano.
- Separar silábicamente la palabra p.
- Ordenar y listar alfabéticamente todas las palabras del castellano.
- Dibujar en la pantalla del ordenador un círculo de radio r.

Como se puede observar, en la mayoría de las ocasiones, los problemas se definen de forma general, haciendo uso de identificadores o *parámetros* (en los ejemplos esto es así excepto en el segundo problema, que es un caso particular del tercero). Estos parámetros denotan los datos de entrada o el resultado del problema. Por ejemplo, el número del cuál se desea encontrar su raíz cuadrada o la lista de palabras en las que se desea buscar las repetidas, etc. Las soluciones proporcionadas a esos problemas (algoritmos) tendrán también esa característica.

A veces los problemas están definidos de forma imprecisa puesto que los seres humanos podemos, o bien recabar nueva información sobre ellos, o bien realizar presunciones sobre los mismos. Cuando un problema se encuentra definido de forma imprecisa introduce una ambigüedad indeseable, por ello, siempre que esto ocurra, se deberá precisar el problema, eliminando en lo posible su ambigüedad. Así, por ejemplo, cuando en el problema tercero se desea determinar la raíz cuadrada positiva de un número n, se puede presuponer que dicho número n es real y no negativo, por ello, redefiniremos el problema del modo siguiente: determinar la raíz cuadrada positiva de un número n, entero no negativo, cualquiera. O cuando se desea obtener el vocabulario utilizado en un texto, es necesario definir qué se entiende por palabra, toda secuencia de caracteres separados por blancos, sólo secuencia de letras y dígitos, etc.

Ejemplos de algoritmos pueden encontrarse en las secuencias de reglas aprendidas en nuestra niñez, mediante las cuales realizamos operaciones básicas de números multidígito como, por ejemplo, sumas, restas, productos y divisiones. Son algoritmos ya que definen de forma precisa la resolución en tiempo finito de un problema de índole general.

En general, son características propias de cualquier algoritmo, las siguientes:

- Debe ser finito, esto es, debe realizarse en un tiempo finito; o dicho de otro modo, el algoritmo debe de acabar necesariamente tras un número finito de pasos.
- Debe ser preciso, es decir, debe definirse de forma exacta y precisa, sin ambigüedades.
- Debe ser efectivo, sus reglas o instrucciones se pueden ejecutar.

- Debe ser general, esto significa que debe resolver toda una clase de problemas y no un problema aislado particular.
- Puede tener varias entradas o ninguna; sin embargo, al menos, debe tener una salida, el resultado que se desea obtener.

Como ejemplo adicional, se muestran, a continuación, algunos algoritmos para comprobar si un número es o no primo. Como se recordará un número primo es aquel que sólo es divisible por él mismo o por la unidad. Son muchas las aplicaciones de los números primos; por ejemplo, la clave de seguridad de muchos sistemas, como pueden ser las transacciones secretas en Internet o las comunicaciones por teléfono móvil, se basan en la dificultad de factorizar números de muchas cifras. En concreto, si se toman dos números primos grandes y se multiplican, el número resultante tendrá muchas cifras y la cantidad de operaciones a realizar hará que este sea un problema prácticamente imposible de resolver, ni siquiera utilizando los ordenadores más potentes de hoy en día y los algoritmos de factorización más eficientes. Por ejemplo, dados los dos números primos $p = 999999000001$ y $q = 10009999999999999997$, es posible multiplicarlos pero, al menos por el momento, no factorizar el resultado.

Ejemplo 1.1. Considérese el problema: ¿es n, entero mayor que uno, un número primo?

Un primer algoritmo para resolver este problema es el que se muestra en la figura 1.1; consiste en la descripción de una enumeración de los números anteriores a n comprobando, para cada uno, la divisibilidad del propio n por el número considerado. El detalle de este algoritmo es suficiente para que un humano pueda seguirlo y resolver el problema. Por ejemplo, para constatar que el número 1000003 es primo habría que comprobar que no es divisible ni por 2, ni por 3, ni por 4 y así hasta 1000002. Obsérvese que si el número n es primo el número de comprobaciones a realizar son exactamente $n - 2$.

```
Algoritmo 1.-
Considerar todos los números comprendidos entre 2 y n (excluido).
Para cada número de dicha sucesión comprobar si dicho número
     divide al número n.
Si ningún número divide a n, entonces n es primo.
```

Figura 1.1: Algoritmo 1 para determinar si n es primo.

El algoritmo siguiente, en la figura 1.2, es similar al anterior, ya que la secuencia de cálculos que define para resolver el problema es idéntica a la expresada por el algoritmo primero; sin embargo, se ha escrito utilizando una notación algo más detallada, en la que se han hecho explícitos, enumerándolos, los pasos que se siguen y permitiendo con ello la referencia a un paso determinado del propio algoritmo.

```
Algoritmo 2.- Seguir los pasos siguientes en orden ascendente:
Paso 1. Sea i un número entero de valor igual a 2.
Paso 2. Si i es mayor o igual a n parar, n es primo.
Paso 3. Comprobar si i divide a n, entonces parar, n no es primo.
Paso 4. Reemplazar el valor de i por i + 1, volver al Paso 2.
```

Figura 1.2: Algoritmo 2 para determinar si n es primo.

El tercer algoritmo, en la figura 1.3, mantiene una estrategia similar a la utilizada por los dos primeros: comprobaciones sucesivas de divisibilidad por números anteriores; sin embargo, haciendo uso de propiedades básicas de los números, mejora a los algoritmos anteriores al reducir de forma importante la cantidad de comprobaciones de divisibilidad efectuadas. En concreto, las propiedades tenidas en cuenta son que un número par no es primo (excepto el 2) y que es suficiente con comprobar la divisibilidad hasta $\sqrt{n}$. Por ejemplo, para comprobar la primalidad de 1000003 sólo se comprobaría la divisibilidad por 3, 5, 7, hasta 999; esto es, 499 comparaciones. En general, la primalidad del número n se puede constatar con este algoritmo haciendo sólo $\sqrt{n} - 2/2$ comprobaciones de divisibilidad.

```
Algoritmo 3.- Seguir los pasos siguientes en orden ascendente:
Paso 1. Si n vale 2 entonces parar, n es primo.
Paso 2. Si n es múltiplo de 2 acabar, n no es primo.
Paso 3. Sea i un número entero de valor igual a 3.
Paso 4. Si i es mayor que la raíz cuadrada positiva de n parar,
        n es primo.
Paso 5. Comprobar si i divide a n, entonces parar, n no es primo.
Paso 6. Reemplazar el valor de i por i + 2, volver al Paso 4.
```

Figura 1.3: Algoritmo 3 para determinar si n es primo.

En cualquier caso, como es fácil ver, la descripción o nivel de detalle de la solución de un problema en términos algorítmicos depende de qué o quién debe entenderlo, resolverlo e interpretarlo.

Para facilitar la discusión se introduce el término genérico *procesador*. Se denomina *procesador* a cualquier entidad capaz de interpretar y ejecutar un cierto repertorio de instrucciones.

Un *programa* es un algoritmo escrito con una notación precisa para que pueda ser ejecutado por un procesador. Habitualmente, los procesadores que se utilizarán serán computadores con otros programas para facilitar el manejo de la máquina subyacente.

Cada instrucción al ejecutarse en el procesador supone cierto cambio o transformación, de duración finita, y de resultados definidos y predecibles. Dicho cambio se

produce en los valores de los elementos que manipula el programa. En un instante dado, el conjunto de dichos valores se denomina el *estado del programa*.

Denominamos *cómputo* a la transformación de estado que tiene lugar al ejecutarse una o varias instrucciones de un programa.

1.1 Programas y la actividad de la programación

Como se ve, un programa es la definición precisa de una tarea de computación, siendo el propósito de un programa su ejecución en un procesador, y suponiendo dicha ejecución cierto cómputo o transformación.

Para poder escribir programas de forma precisa y no ambigua es necesario definir reglas que determinen tanto lo que se puede escribir en un programa (y el procesador podrá interpretar) como el resultado de la ejecución de dicho programa por el procesador. Dicha notación, conjunto de reglas y definiciones, es lo que se denomina un *lenguaje de programación*. Más adelante se estudiarán las características de algunos de ellos.

Como es lógico, el propósito principal de la programación consiste en describir la solución computacional (eficiente) de clases de problemas. Aunque hay que destacar que se ha demostrado la existencia de problemas para los que no puede existir solución computacional alguna, lo que implica una limitación importante a las posibilidades de la mecanización del cálculo.

Adicionalmente, los programas son objetos complejos que habitualmente necesitan modificaciones y adaptaciones. De esta complejidad es posible hacerse una idea si se piensa que algunos programas (la antigua iniciativa de defensa estratégica de los EEUU, por ejemplo) pueden contener millones de líneas y que, por otro lado, un error en un único carácter de una sola línea puede suponer el malfuncionamiento de un programa (así, por ejemplo, el Apollo XIII tuvo que cancelar, durante el trayecto, una misión a la luna debido a que en un programa se había sustituido erróneamente una coma por un punto decimal, o al telescopio espacial Hubble se le corrigió de forma indebida las aberraciones de su espejo, al cambiarse en un programa un símbolo `+` por un `-`, con lo que el telescopio acabó `"miope"` y, por ello, inutilizable durante un periodo de tiempo considerable).

La tarea de la programación en aplicaciones reales de cierta envergadura es bastante compleja. Según la complejidad del problema a resolver se habla de:

- *Programación a pequeña escala*: número reducido de líneas de programa, intervención de una sola persona; por ejemplo, un programa de ordenación.
- *Programación a gran escala*: muchas líneas de programa, equipo de programadores; por ejemplo, el desarrollo de un sistema operativo.

El ciclo de existencia de un programa sencillo está formado, a grandes rasgos, por las dos etapas siguientes:

- *Desarrollo*: creación inicial y validación del programa.
- *Mantenimiento*: correcciones y cambios posteriores al desarrollo.

1.2 Lenguajes y modelos de programación

Los orígenes de los lenguajes de programación se encuentran en las máquinas. La llamada máquina original de Von Neumann se diseñó a finales de los años 1940 en Princeton (aunque su diseño coincide en gran medida con el de la máquina creada con elementos exclusivamente mecánicos por Charles Babbage y programada por Ada Byron en Londres hacia 1880).

La mayoría de los ordenadores modernos tienen tanto en común con la máquina original de Von Neumann que se les denomina precisamente máquinas con arquitectura "Von Neumann".

La característica fundamental de dicha arquitectura es la *banalización de la memoria*, esto es, la existencia de un espacio de memoria único y direccionable individualmente, que sirve para mantener tanto datos como instrucciones; existiendo unidades especializadas para el tratamiento de los datos, Unidad Aritmético Lógica (ALU) y de las instrucciones, Unidad de Control (UC). Ésta es también, a grandes rasgos, la estructura del procesador central de casi cualquier computador moderno significativo. Véase la figura 1.4, en la que se puede observar que en el mismo espacio de memoria coexisten tanto datos como instrucciones para la manipulación de los mismos.

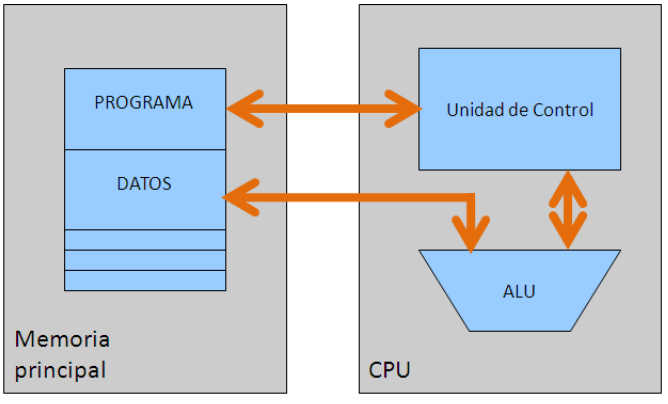

Figura 1.4: Estructura de un procesador con arquitectura Von Neumann.

Al nivel de la máquina, un programa es una sucesión de palabras (compuestas de bits), habitualmente en posiciones consecutivas de memoria que representan instrucciones o datos. El lenguaje con el que se expresa es el *lenguaje máquina*.

Por ejemplo, el fragmento siguiente, muestra en su parte derecha una secuencia de código en lenguaje máquina.

Instrucciones en ensamblador y código máquina

```
Load 24,      # a está en la dir. 24h          10111100 00100100
Multiply 33,  # mult. por b en la dir. 33h     10111111 00110011
Store 3C,     # almacenar en c en la dir. 3Ch  11001110 00111100
```

Obviamente, los programas en lenguaje máquina son ininteligibles, tal y como puede verse en el ejemplo.

Aunque no tanto, también son muy difíciles de entender los denominados *lenguajes ensambladores* (fragmento anterior, columna primera a la izquierda) en los que ya se utilizan mnemónicos e identificadores para las instrucciones y datos.

Estos lenguajes se conocen como de *bajo nivel*. Los problemas principales de dichos lenguajes son el bajo nivel de las operaciones que aportan, así como la posibilidad de efectuar todo tipo de operaciones (de entre las posibles) sobre los datos que manipulan. Así, por ejemplo, es habitual disponer tan solo de operaciones de carácter aritmético, de comparación y de desplazamiento, ello permite interpretar cualquier posición de memoria exclusivamente como un número. Un carácter se representará mediante un código numérico, aunque será visto a nivel máquina como un número (con lo que pueden multiplicarse entre sí, por ejemplo, dos caracteres, lo que posiblemente no tiene sentido).

Hacia finales de la década de los años 50 aparecieron lenguajes de programación orientados a hacer los programas más potentes, inteligibles y seguros; estos lenguajes serían denominados, en contraposición a los anteriores, *lenguajes de alto nivel*. En ellos, un segmento como el anterior, para multiplicar ciertos valores a y b, dando como resultado c, podría ser simplemente `c = a * b;` que, además de más legible, es bastante más seguro puesto que implica que para poderse ejecutar, típicamente se comprueba que los datos implicados deben de ser numéricos. Por ejemplo, si a, b o c se hubiesen definido previamente como caracteres, la operación anterior puede no tener sentido y el programa detenerse antes de su ejecución, advirtiendo de ello al programador, que podrá subsanar el error.

Así, por ejemplo, la motivación fundamental del primer lenguaje de alto nivel, el FORTRAN (FORmula TRANslator), desarrollado en 1957, era la de disponer de un lenguaje conciso para poder escribir programas de índole numérica y traducirlos automáticamente a lenguaje máquina.

Esta forma de trabajo es la utilizada hoy en día de forma habitual. A los programas que traducen las instrucciones de un lenguaje de alto nivel a un lenguaje máquina se les denomina *compiladores* e *intérpretes*. Un intérprete traduce a lenguaje máquina cada instrucción del lenguaje de alto nivel, una a una, en tiempo de ejecución. Un compilador traduce mediante todas las instrucciones del programa a lenguaje máquina, previamente a su ejecución.

En la figura 1.5 se muestra el proceso seguido para poder compilar y ejecutar un programa en cualquier lenguaje de alto nivel.

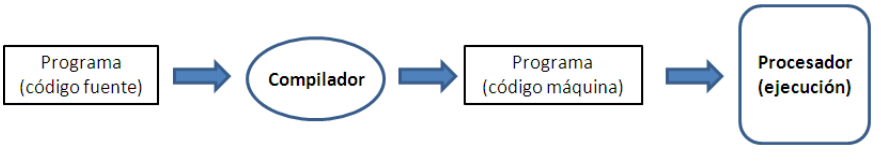

Figura 1.5: Proceso de compilación y ejecución de un programa.

Otros lenguajes de programación que aparecieron en la década de los 60, poco tiempo después del FORTRAN son el APL, el Algol, el Cobol, el LISP, el Basic y el PL1.

Algunas características comunes a todos ellos y, en general, a todos los lenguajes de alto nivel son:

- Tienen operadores y estructuras más cercanas a las utilizadas por las personas.
- Son más seguros que el código máquina y protegen de errores evidentes.
- El código que proporcionan es transportable y, por lo tanto, independiente de la máquina en que se tenga que ejecutar.
- El código que proporcionan es más legible.

En la década de los 70, como reacción a la falta de claridad y de estructuración introducida en los programas por los abusos que permitían los primeros lenguajes de programación, se originó la, así denominada, *programación estructurada*, que consiste en el uso de un conjunto de modos de declaración y constructores en los lenguajes, reducido para que sea fácilmente abarcable y, al mismo tiempo, suficiente para expresar la solución algorítmica de cualquier problema resoluble.

Ejemplos de dichos lenguajes son los conocidos Pascal, C y Módula-2.

El modelo introducido por la *programación estructurada* tiene aún hoy en día una gran importancia para el desarrollo de programas. De hecho, se asumirá de forma implícita a lo largo del libro aunque, como se verá, enmarcándolo dentro de la programación orientada a objetos.

Otro aspecto significativo de los lenguajes de programación de alto nivel que hay que destacar es el de que los mismos representan un procesador o *máquina extendida*: esto es, aquélla que puede ejecutar las instrucciones de dicho lenguaje. Consideraremos, en general, que un lenguaje de programación es una extensión de la máquina en que se apoya, del mismo modo que un programa es una extensión del lenguaje de programación en que se construye.

Un lenguaje de programación proporciona un *modelo de computación* que no tiene por que ser igual al de la máquina que lo sustenta, pudiendo ser de hecho completamente diferente. Por ejemplo, un lenguaje puede hacer parecer que un programa se está ejecutando en varias máquinas distintas, aun cuando sólo existe una; o, por el contrario, puede hacer parecer que se está ejecutando en una sola máquina (muy rápidamente) cuando realmente ha subdividido la computación que realiza entre varias máquinas diferentes.

A lo largo de la historia los seres humanos hemos desarrollado varios modelos de computación posibles (unos basados en una máquina universal, otros en las funciones recursivas, otros en la noción de inferencia, etc). Se ha demostrado que todos estos modelos son computacionalmente equivalentes, esto es: si existe una solución algorítmica para un problema utilizando uno de los modelos, también existe una solución utilizando cualquiera de los otros.

El modelo más extendido de computación hace uso de una máquina universal bastante similar en su esencia a los procesadores actuales denominada, en honor a su inventor, *Máquina de Turing*. En este modelo, una computación es una transformación de estados y un programa representa una sucesión de computaciones, o transformaciones, del estado inicial del problema al final o solución del mismo. Este modelo es el que seguiremos a lo largo del presente libro. En él, la solución de un problema se define dando una secuencia de pasos que indican la secuencia de computaciones para resolverlo. Este modelo de programación recibe el nombre de *modelo* o *paradigma imperativo*.

Diagramas y listas bastante completos con la evolución de los lenguajes, pueden encontrarse, si se efectúa una búsqueda, en muchas *URLs*; entre ellas `http://www.levenez.com/lang/`

1.3 La programación orientada a objetos. El lenguaje Java

Aunque la *programación orientada a objetos* tuvo sus inicios en la década de los 70, es sólo más recientemente cuando ha adquirido relevancia, siendo en la actualidad uno de los modelos de desarrollo de programas predominante. Así, presenta mejoras para el desarrollo de programas en comparación a lo que aporta la programación estructurada que, como se ha mencionado, fue el modelo de desarrollo fundamental durante la década de los 70.

El elemento central de un programa orientado a objetos es la *clase*. Una clase determina completamente el comportamiento y las características propias de sus componentes. A los casos particulares de una clase se les denomina *objetos*. Un programa se entiende como un conjunto de objetos que interactúan entre sí.

Una de las principales ventajas de la programación orientada a objetos es que facilita la reutilización del código ya realizado (*reusabilidad*), al tiempo que permite ocultar detalles (*ocultación*) no relevantes (*abstracción*), aspectos fundamentales en la gestión de proyectos de programación complejos.

El lenguaje Java (1991) es un lenguaje orientado a objetos, de aparición relativamente reciente. En ese sentido, un programa en Java consta de una o más clases interdependientes. Las clases permiten describir las propiedades y habilidades de los objetos de la vida real con los que el programa tiene que tratar.

El lenguaje Java presenta, además, algunas características que lo diferencian, a veces significativamente, de otros lenguajes. En particular está diseñado para facilitar el trabajo en la `WWW`, mediante el uso de los programas navegadores de uso completamente difundido hoy en día. Los programas de Java que se ejecutan a través de la red se denominan *applets* (aplicación pequeña).

Otras de sus características son: la inclusión en el lenguaje de un entorno para la programación gráfica (`AWT` y `Swing`) y el hecho de que su ejecución es *independiente de la plataforma*, lo que significa que un mismo programa se ejecutará exactamente igual en diferentes sistemas.

Para la consecución de las características anteriores, el Java hace uso de lo que se denomina *Máquina Virtual Java* (Java Virtual Machine, *JVM*). La *JVM* es una extensión (mediante un programa) del sistema real en el que se trabaja, que permite ejecutar el código resultante de un programa Java ya compilado independientemente de la plataforma en que se esté utilizando. En particular, todo navegador dispone (o puede disponer) de una *JVM*; de ahí la universalidad de su uso.

El procedimiento necesario para la ejecución un programa en Java puede verse, de forma resumida, en la figura 1.6.

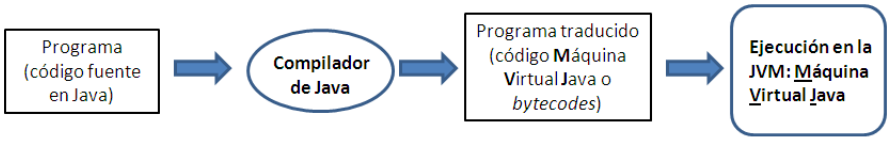

Figura 1.6: Proceso de compilación y ejecución de un programa en Java.

Es interesante comparar dicho proceso con el que aparece en la figura 1.5, donde se muestra un proceso similar pero para un programa escrito en otros lenguajes de programación. La diferencia, como puede observarse, consiste en el uso de la, ya mencionada, máquina virtual, en el caso del Java (JVM).

Una de las ventajas de este modelo, es que permite utilizar el mismo código Java virtual, ya compilado, siempre que en el sistema se disponga de una máquina virtual Java.

Uno de los inconvenientes de un modelo así, estriba en que puede penalizar el tiempo de ejecución del programa final ya que introduce un elemento intermedio, la máquina virtual, para permitir la ejecución.

1.4 Un mismo ejemplo en diferentes lenguajes

Como ejemplo final de este capítulo, se muestra a continuación el algoritmo ya visto para determinar si un número n entero y positivo es o no un número primo (Algoritmo 3, figura 1.3), implementado en diferentes lenguajes de programación:

- `Pascal`, en la figura 1.7.
- `C/C++`, en la figura 1.8.
- `Python`, en la figura 1.9.
- `Java`, en la figura 1.10.
- `C#`, en la figura 1.11.

La similitud que se puede observar en los ejemplos, entre los distintos lenguajes, se debe principalmente a que en la evolución de los mismos, muchos de ellos heredan, mejorándolas, características de los lenguajes anteriores.

En particular, el lenguaje `C++` es una ampliación del `C` hacia la Programación Orientada a Objetos, mientras que el `Java` es una evolución de los dos anteriores, que presenta mejoras con respecto a ellos en cuanto a la gestión de la memoria, así como un modelo de ejecución, diferente, basado, como ya se ha mencionado, en una máquina virtual.

También están basados en un modelo de máquina virtual el `C#` y el `Python`. Se puede decir que el `C#` es un heredero directo del `Java`; mientras que el `Python`, aunque toma características de los anteriores, presenta también bastantes elementos innovadores.

```
function es_primo(n: integer): boolean;
var i: integer; primo: boolean; raiz: real;
begin
  if n = 2 then primo := true
  else if n mod 2 = 0 then primo := false
  else begin
         primo :=t rue;
         i := 3; raiz := sqrt(n);
         while (i <= raiz) and primo do
         begin
           primo := ((n mod i) <> 0);
           i := i + 2;
         end;
       end;
  es_primo := primo;
end;
```

Figura 1.7: ¿Es n primo? Algoritmo 3, versión en `Pascal`.

```
int es_primo(int n) {
  int i, primo; float raiz;
  if (n == 2) primo = 0;
  else if (n % 2) primo = 1;
  else {
         i = 3; raiz = sqrt(n);
         while ((i <= raiz) && !(n % i)) {i += 2;}
         primo = !(n % i);
  }
  return primo;
}
```

Figura 1.8: ¿Es n primo? Algoritmo 3, versión en `C/C++`.

```
from math import sqrt
def es_primo(n):
    if n == 2: primo = True
    elif n % 2 == 0: primo = False
    else:
        i = 3
        raíz = sqrt(n)
        while i <= raíz and n%i != 0: i += 2
        primo = (n % i != 0)
    return primo
```

Figura 1.9: ¿Es n primo? Algoritmo 3, versión en `Python`.

```
public static boolean es_primo(int n) {
    int i; double raíz; boolean primo;
    if (n == 2) { primo = true; }
    else if (n % 2 == 0) { primo = false; }
    else {
        i = 3; raíz = Math.sqrt(n);
        while ((i <= raíz) && (n % i != 0)) { i += 2; }
        primo = (n % i != 0);
    }
    return primo;
}
```

Figura 1.10: ¿Es n primo? Algoritmo 3, versión en `Java`.

```
static bool es_primo(int n) {
  int i; double raíz; bool primo;
  if (n == 2) primo = true;
  else if (n % 2 == 0) primo = false;
  else {
         i = 3; raíz = Math.Sqrt(n);
         while ((i <= raíz) && (n % i != 0)) {i += 2;}
         primo = (n % i != 0);
  }
  return primo;
}
```

Figura 1.11: ¿Es n primo? Algoritmo 3, versión en `C#`.

Más información

[Pyl75] Z.W. (selec.) Pylyshyn. *Perspectivas de la revolución de los computadores/Selec., comentarios e introd. de Z.W. Pylyshyn; tr. por Luis García Llorente; rev. de Eva Sánchez.* Alianza, 1975. Incluye textos de H. Aiken, Ch. Babbage, J. von Neumann, C. Shannon, A.M. Turing y otros.

[Tra77] B.A. Trajtenbrot. *Los algoritmos y la resolución automática de problemas.* MIR, 1977.

Capítulo 2

Objetos, clases y programas

La *Programación Orientada a Objetos* (POO) es el modelo de construcción de programas predominante en la actualidad debido a que presenta un sistema basado fuertemente en la representación de la realidad y que, al mismo tiempo, refuerza el uso de buenos criterios aplicables al desarrollo de programas, como son la *abstracción*, la *ocultación de información* y la *reusabilidad*, entre otros.

El objetivo de este capítulo es introducir de manera superficial las nociones básicas de la POO. Un estudio en profundidad de todas ellas se abordará en el resto de capítulos del libro.

El elemento fundamental en la POO es, por supuesto, el *objeto*. Un *objeto* se puede definir como una agrupación o colección de datos y operaciones que poseen determinada estructura y mediante los cuales se modelan aspectos relevantes de un problema.

Los objetos que comparten cierto comportamiento se pueden agrupar en diferentes categorías llamadas clases. Una *clase* es, por lo tanto, una descripción de cuál es el comportamiento de cada uno de los objetos de la clase. Se dice entonces que el *objeto* es una *instancia* de la *clase*.

El lenguaje Java es un lenguaje orientado a objetos por lo que, a grosso modo, se puede decir que programar en Java consiste en *escribir las definiciones de las clases y utilizar esas clases para crear objetos* de forma que, mediante los mismos, se represente adecuadamente el problema que se desea resolver.

El lenguaje Java posee un gran número de clases predefinidas, por lo que no es necesario reinventarlas, basta con utilizarlas cuando se necesiten.

Atendiendo a la estructura de la clase y al uso que se va a hacer de ella se pueden distinguir tres tipos básicos de clases:

Clase Tipo de Dato: es aquella que define los elementos que componen, o que tiene, cualquier objeto del tipo y, en base a estos, las operaciones que se le pueden aplicar.

Clase Programa: es aquella que, realmente, inicia la ejecución del código de una aplicación, por lo que se suele decir que la "lanza".

Clase de Utilidades: es un repositorio de operaciones que pueden utilizarse desde otras clases.

En la figura 2.1 se muestra, como ejemplo, el código completo en Java de la Clase Tipo de Dato `Circulo`; antes de estudiarlo con detalle, conviene señalar que todas las líneas precedidas por los símbolos `//` o enmarcadas en un bloque `/** ...*/` son comentarios Java, líneas no ejecutables cuyo único fin es documentar la clase.

En esencia, la clase `Circulo` define un tipo que, como ilustran las líneas de la 8 a la 10 y la 14 y 18 de su código, se compone de los siguientes elementos: un `radio`, que puede tomar como valor cualquier número real (`double`); un `color`, que puede ser el nombre (`String`) de cualquier color (negro, rojo, etc.); las cooordenadas `centroX` y `centroY` de su centro, que pueden tomar valores de tipo entero (`int`). Al mismo tiempo, en base a los elementos que tiene, la clase define las operaciones que se pueden aplicar a cualquier objeto del tipo que define: obtener su `radio` (`getRadio` en línea 22) o modificarlo (`setRadio` en línea 31), calcular su área (`area` en línea 46) o su perímetro (`perimetro` en línea 48), etc.

La clase (gráfica) `Pizarra` es otro ejemplo de Clase Tipo de Dato. Como el código asociado a esta clase queda fuera de los propósitos del libro, en la figura 2.2 aparece única y exclusivamente la documentación esencial asociada a esta clase; como se verá en un apartado posterior, para usar esta clase no es necesario conocer los detalles de cómo está implementada sino que basta con que su documentación esté disponible.

En concreto, de la documentación de la clase `Pizarra` se tiene que una `Pizarra` se puede construir (veáse la parte denominada `Constructor Summary`) dándole, si se desea, un título y un tamaño inicial. Además, sobre cualquier objeto de tipo `Pizarra` se pueden realizar dos operaciones (veáse la parte `Method Summary`): `add`, que añade un objeto (de tipo `Circulo` o `Rectangulo`) a la `Pizarra` y lo dibuja, y `dibujaTodo` que, para actualizar el estado de la `Pizarra`, vuelve a dibujar todos los objetos que contiene en un momento dado. Por supuesto, una clase puede utilizar tantos objetos de tipo `Pizarra` como se desee.

```java
/**
 * Clase Circulo: define un círculo de un determinado radio, color y
 * posición de su centro, con la funcionalidad que aparece a continuación.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class Circulo {
    private double radio;
    private String color;
    private int centroX, centroY;

    /** Crea un Circulo de radio r, color c y centro en (px, py). */
    public Circulo(double r, String c, int px, int py) {
        radio = r;  color = c; centroX = px; centroY = py;
    }
    /** Crea un Circulo de radio 50, negro y centro en (100, 100). */
    public Circulo() {
        radio = 50; color = "negro"; centroX = 100;  centroY = 100;
    }

    /** Devuelve el radio del Circulo. */
    public double getRadio() { return radio; }
    /** Devuelve el color del Circulo. */
    public String getColor() { return color; }
    /** Devuelve la abscisa del centro del Circulo. */
    public int getCentroX() { return centroX; }
    /** Devuelve la ordenada del centro del Circulo. */
    public int getCentroY() { return centroY; }

    /** Actualiza el radio del Circulo a nuevoRadio. */
    public void setRadio(double nuevoRadio) { radio = nuevoRadio; }
    /** Actualiza el color del Circulo a nuevoColor. */
    public void setColor(String nuevoColor) { color = nuevoColor; }
    /** Actualiza el centro del Circulo a (nuevoCentroX, nuevoCentroY). */
    public void setCentro(int nuevoCentroX, int nuevoCentroY) {
        centroX = nuevoCentroX; centroY = nuevoCentroY;
    }
    /** Desplaza el centro del Circulo a la derecha 10 unidades. */
    public void aLaDerecha() { centroX = centroX + 10; }
    /** Incrementa un 30% el radio del Circulo. */
    public void crece() { radio = radio * 1.3; }
    /** Decrementa un 30% el radio del Circulo. */
    public void decrece() { radio = radio / 1.3; }

    /** Devuelve el área del Circulo. */
    public double area() { return 3.14 * radio * radio; }
    /** Devuelve el perímetro del Circulo. */
    public double perimetro() { return 2 * 3.14 * radio; }

    /** Devuelve un String con las componentes del Circulo en el
     *  siguiente formato: Circulo de radio r, color c y centro (x, y). */
    public String toString() {
        return "Circulo de radio " + radio + ", color " + color
            + " y centro (" + centroX + ", " + centroY + ")";
    }
}
```

Figura 2.1: Clase `Circulo`.

Class Pizarra

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
                  └javax.swing.JFrame
                      └Pizarra
```

```
public class Pizarra
extends javax.swing.JFrame
```

Clase Pizarra: define una pizarra de determinados tamaño y titulo, en la que se pueden dibujar objetos de tipo Circulo y Rectangulo.
Como un cuadrado es un rectángulo con la misma base que altura, en una pizarra de esta clase se pueden dibujar no solo circulos y rectángulos sino también cuadrados.

Version:
2016
Author:
Libro IIP-PRG

Constructor Summary

`Pizarra()` Crea una Pizarra por defecto (estándar), i.e. de titulo "La pizarra por defecto", tamaño 200 x 200 (pixeles) y vacía, sin dibujos.
`Pizarra(java.lang.String t, int ancho, int alto)` Crea una Pizarra de titulo t, tamaño ancho x alto (pixeles) y vacia, sin dibujos.

Method Summary

void	`add(java.lang.Object o)` Añade el objeto o a la Pizarra y lo dibuja.
void	`dibujaTodo()` Vuelve a dibujar todos los objetos que hay en la Pizarra, actualizando su estado.

Figura 2.2: Extracto de la documentación de la clase `Pizarra`.

```
/**
 * Clase PrimerPrograma: dibuja en una cierta pizarra un circulo y dos
 * rectangulos de características dadas; para ello, usa las clases
 * Pizarra, Circulo y Rectangulo.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class PrimerPrograma {
    public static void main(String[] args) {
        // Crear la Pizarra miPizarra, de título "ESPACIO DIBUJO" y
        // tamaño 300 x 300 píxeles
        Pizarra miPizarra = new Pizarra("ESPACIO DIBUJO", 300, 300);
        // Crear un Circulo c1 de radio 50, amarillo y con centro
        // en (100, 100)
        Circulo c1 = new Circulo(50, "amarillo", 100, 100);
        // Crear un Rectangulo r1 30 x 30, azul y con centro
        // en (125, 125)
        Rectangulo r1 = new Rectangulo(30, 30, "azul", 125, 125);
        // Crear un Rectangulo r2 100 x 10, rojo y con centro
        // en (50, 155)
        Rectangulo r2 = new Rectangulo(100, 10, "rojo", 50, 155);

        // Añadir c1 a miPizarra, dibujándolo
        miPizarra.add(c1);
        // Añadir r1 a miPizarra, dibujándolo
        miPizarra.add(r1);
        // Añadir r2 a miPizarra, dibujándolo
        miPizarra.add(r2);
    }
}
```

Figura 2.3: Clase `PrimerPrograma`.

La figura 2.3 muestra el código de la Clase Programa `PrimerPrograma`, que usa las clases `Pizarra`, `Circulo` y `Rectangulo`; en sus líneas de código aparecen las instrucciones que se irán ejecutando en secuencia, una tras otra en el orden en el que aparecen escritas, a medida que lo haga el propio programa:

1. Crear una `Pizarra miPizarra` de título "ESPACIO DIBUJO" y tamaño (en píxeles) 300 x 300.
2. Crear un `Circulo c` de radio 50, amarillo y con centro en (100, 100).
3. Crear un `Rectangulo r1` de base y altura iguales a 30, azul y con centro en (125, 125).
4. Crear un `Rectangulo r2` de base 100, altura 10, color rojo y con centro en (50, 155).
5. Añadir `c` a `miPizarra`, dibujándolo.
6. Añadir `r1` a `miPizarra`, dibujándolo; nótese que, en realidad, lo que se dibuja es un cuadrado.
7. Añadir `r2` a `miPizarra`, dibujándolo.

El resultado de la ejecución de esta Clase Programa se muestra en la figura 2.4.

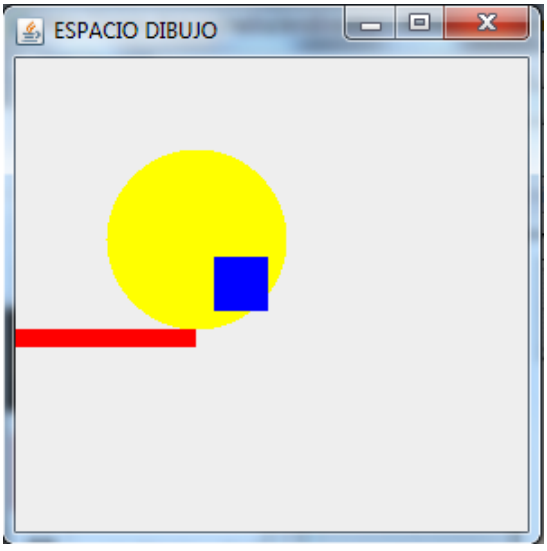

Figura 2.4: Resultado de la ejecución de la clase `PrimerPrograma`.

2.1 Definición de una clase: atributos y métodos

La definición de una clase Java se escribe en un fichero cuya extensión es `.java` y su estructura general es la que se muestra a continuación:

```
[modificadores] class NombreDeLaClase [extends OtraClase] {
    [[modificadores] tipo nomVar1;
     [modificadores] tipo nomVar2;
         ...          ...   ...
     [modificadores] tipo nomVarN; ]

    [[modificadores] tipo nomMetodo1 ([listaParams]) { cuerpo }
     [modificadores] tipo nomMetodo2 ([listaParams]) { cuerpo }
         ...          ...       ...
     [modificadores] tipo nomMetodoM ([listaParams]) { cuerpo } ]
}
```

Este esquema, donde los ítems que aparecen entre corchetes tienen un carácter opcional, permite observar claramente que la definición de una clase consta de dos partes bien diferenciadas: en la primera, se describen en Java (o declaran) los denominados *atributos* de la clase (`nomVar1`, ..., `nomVarN`), esto es, los elementos

distintivos de los que está compuesto (o que tiene) cualquier objeto de la clase; en la segunda, se describen en Java (o declaran) los denominados *métodos* de la clase (`nomMetodo1`, ..., `nomMetodoM`), es decir, las operaciones que se pueden aplicar a cualquier objeto de la clase. Nótese que es posible que una clase, bien no tenga atributos, bien no tenga métodos.

Con respecto a los `modificadores` que aparecen como opcionales en este esquema, cabe señalar que los únicos que se utilizarán por el momento son el modificador especial `static` y los modificadores de acceso o visibilidad `private` y `public`; también por el momento, sólo se explicará el papel que tienen estos últimos.

En términos generales, un modificador de acceso o visibilidad permite que el programador establezca en qué otras clases Java se puede usar, o no, el ítem a cuya declaración preceden (clase, atributo o método). Más concretamente, se tiene que todo ítem (clase, atributo o método) que se declare como...

- `private` es accesible únicamente desde dentro de la clase y, por tanto, inaccesible desde fuera de ella, la propia clase incluida; así, por ejemplo, cualquier intento de acceso a los atributos privados `radio` o `color` de la clase `Circulo` que se realice desde la clase `PrimerPrograma` dará lugar a un error de compilación.
- `public` es accesible tanto desde dicha clase como desde fuera de ella, la propia clase incluida; así, por ejemplo, los métodos públicos `getRadio` o `area` de la clase `Circulo` pueden ser usados desde cualquier otra clase Java, como `PrimerPrograma` o `Pizarra`.

Atributos

Los *atributos* de una clase Java representan las componentes distintivas que tiene cualquier objeto de dicha clase. Se declaran, habitualmente, como `private` y, obligatoriamente, de un cierto *tipo de datos*; este tipo puede ser primitivo o una clase Java pero, en cualquier caso, define los valores que el atributo puede tomar y, en consecuencia, las operaciones que sobre él se pueden realizar. Para ilustrar lo dicho, se utilizarán ahora los tipos de los atributos de la clase `Circulo`; a saber:

1. `radio` de tipo real (`double` en Java), que puede tomar valores reales como 2.57 y formar parte de expresiones aritméticas como la del cálculo del perímetro de un círculo (`2 * 3.14 * radio`).
2. `color` de tipo `String`, clase del estándar de Java cuyos valores posibles son las cadenas de caracteres que se pueden formar con los símbolos aceptados en el lenguaje.[1]

[1] Aunque el lenguaje Java tiene maneras más precisas de representar el color de un objeto, en este ejemplo introductorio se ha optado por está versión tan sencilla como limitada.

3. `centroX` y `centroY` de tipo entero (`int` en Java), que pueden tomar valores enteros y formar parte, al igual que los valores de otros tipos numéricos, de expresiones aritméticas. Y esto es necesariamente así porque ambos atributos representan la posición del centro de un círculo en una pantalla, un espacio bidimensional de píxeles cuyo origen `(0, 0)` se sitúa en la esquina superior izquierda de esta.

Finalmente, mencionar tan sólo que hay dos tipos de atributos: de instancia y de clase. Los primeros, mucho más frecuentes que los segundos, mantienen información propia y exclusiva de cada objeto de una clase, como puede ser el valor del radio que tiene un objeto dado de tipo `Circulo` a lo largo de la ejecución de un programa; en contraposición, los segundos mantienen información común e idéntica a todos los objetos de la clase -de ahí su nombre- y se distinguen de los primeros sintácticamente usando el modificador `static` en su declaración.

Métodos

Los *métodos* de una clase Java definen las operaciones que se pueden aplicar a cualquiera de sus objetos. Se declaran, habitualmente, como `public` y, obligatoriamente, su definición consta de los siguientes ítems:

1. *Cabecera* o *perfil*, en la que se detalla el nombre del método, el tipo del resultado que devuelve (*return*) y la lista de parámetros que pudiera requerir llevar a cabo la operación que representa. Así, por ejemplo, el método de nombre `perimetro` de la clase `Circulo` devuelve un resultado de tipo `double` y no tiene parámetros.

   Es importante señalar ahora que un método Java puede no devolver resultado alguno, pues su labor es exclusivamente la de modificar el estado del objeto sobre el que se aplica. Para señalar esta circunstancia, Java obliga a declarar en su cabecera que el tipo de su resultado es `void`; este es el caso, por ejemplo, del método `setRadio` de la clase `Circulo`.

2. *Cuerpo*, en el que se detalla el código a ejecutar para obtener el resultado del método. Pueden formar parte de este código cualquier instrucción incluida en el repertorio del lenguaje, cuyo estudio se deja para capítulos sucesivos: asignación, composición, condicional, de repetición (bucle) y cualquier combinación de las anteriores.

   Destacar también que, a menos que el tipo de su resultado sea `void`, la instrucción `return` debe aparecer obligatoriamente en el cuerpo de un método, pues su efecto es devolver su resultado. Por ejemplo, el cuerpo del método `perimetro` de la clase `Circulo` tiene una única instrucción que es `return 2 * 3.14 * radio`.

A grosso modo, los métodos de una clase Java se pueden clasificar como sigue:

- *Constructores*: los que se usan para crear un objeto. Como ilustra la declaración de los constructores de la clase `Circulo`, se distinguen del resto de los métodos de una clase por dos motivos: en su cabecera no existe el ítem tipo de resultado y, además, su nombre debe ser igual que el de la clase; en su cuerpo deben figurar las instrucciones de inicialización de los atributos de la clase, resultando más que recomendable que no exista ninguna más.
- *Modificadores*: los que, como su nombre indica, se usan para modificar el estado de un objeto (el valor de, al menos, uno de sus atributos). Se distinguen del resto de los métodos de una clase porque en su cabecera el tipo del resultado que devuelven es siempre `void`, como se puede observar en los métodos de la clase `Circulo` cuyo nombre empieza por `set`.
- *Consultores*: los que, como su nombre indica, se usan para consultar, sin modificar, el estado del objeto. Se distinguen del resto de los métodos de una clase porque en su cabecera el tipo del resultado que devuelven es siempre distinto de `void`, como se puede observar en los métodos de la clase `Circulo` cuyo nombre empieza por `get`.
- *El método* `main`: es tan singular, en el sentido estricto del término, que puede considerarse un tipo en sí mismo: es el único método Java que se puede usar para indicar cuál es el punto de inicio de la ejecución del código de una aplicación, por lo que su aparición en la definición de una clase convierte a esta, automáticamente, en una clase Programa; precisamente por ello, como se aprecia en la clase `PrimerPrograma`, su cabecera es, obligatoriamente:

```
public static void main(String[] args)
```

Comentar también que, el lenguaje Java permite la *sobrecarga de métodos*, o declarar en la misma clase dos o más métodos del mismo nombre pero con listas de parámetros distintas; nótese que el hecho de que el número, tipo u orden de sus parámetros difiera evita la ambigüedad que provocaría que sean homónimos a la hora de usarlos. Un tipo frecuente de métodos *sobrecargados* son los constructores de una clase; la ventaja que ello aporta es proporcionar al usuario formas alternativas de hacer lo mismo, i.e. la de inicializar los atributos de una clase a unos valores u otros según prefiera. Por ejemplo, supóngase que se quiere dibujar un círculo en una pizarra; como la clase `Circulo` define los siguientes constructores, el usuario puede decidir qué círculo en concreto se dibuja invocando a uno u otro.

```
public Circulo(double r, String c, int px, int py) {
    radio = r;  color = c; centroX = px; centroY = py;
}
public Circulo() {
    radio = 50; color = "negro"; centroX = 100;  centroY = 100;
}
```

Así, usará el constructor con parámetros si desea dibujar un círculo de, por ejemplo, radio 3.0, color amarillo y centro en (50, 155); sin embargo, si prefiere dibujar un círculo estándar, cuyos valores del radio, color y centro son los establecidos por defecto en la aplicación (50, negro y (100, 100)) deberá usar el *constructor por defecto*, sin parámetros, de la clase para crearlo.

Señalar finalmente que Java permite la sobrecarga de algunos operadores como, por ejemplo, el operador binario `+`, que según el contexto en el que aparezca suma valores de tipo numérico o concatena objetos de tipo `String`. Y cabe mencionarlo ahora, aún sin entrar en detalles, porque el uso de este operador es abrumadoramente común en Java; comparando los métodos `aLaDerecha` y `toString` de la clase `Circulo`, que figuran a continuación, permite ilustrar el significado y utilidad de la sobrecarga del operador `+` a la hora de programar en Java.

```
/** Desplaza el centro del Circulo a la derecha 10 unidades. */
public void aLaDerecha() { centroX = centroX + 10; }

/** Devuelve un String con las componentes del Circulo en el
 *  siguiente formato: Circulo de radio r, color c y centro (x, y). */
public String toString() {
    return "Circulo de radio " + radio + ", color " + color
        + " y centro (" + centroX + ", " + centroY + ")";
}
```

2.2 Uso de una clase: creación y uso de objetos mediante los operadores `new` y `.`

Usar una clase significa crear y manipular objetos de esta en el cuerpo de un método de otra clase. Así, por ejemplo, la clase `PrimerPrograma` (figura 2.3) usa las clases `Pizarra`, `Circulo` y `Rectangulo` ya que en el cuerpo de su método `main`...

- Se crean objetos de estas clases: la pizarra `miPizarra`, el círculo `c` y los rectángulos `r1` y `r2`.
- Se usa el método `add` de la clase `Pizarra` para manipular `miPizarra`, esto es, para modificar su estado inicial y dibujar en ella, en este orden, los objetos `c`, `r1` y `r2`; dado lo sencillo del programa, no se manipula ningún objeto más en el cuerpo del `main`.

Más específicamente, y como ilustra el siguiente grupo de instrucciones del `main` de `PrimerPrograma` (líneas 12, 14, 16 y 18), para crear un objeto de una clase Java (`Pizarra`, `Circulo` y `Rectangulo` en el ejemplo) se debe usar el operador `new` seguido del nombre de un constructor de la clase con sus parámetros instanciados a los valores correspondientes.

Para seguir leyendo haga click aquí